SPECIFICATIONS AND RATIONALE FOR TELOS, A PASCAL-BASED ARTIFICIAL INTELLIGENCE PROGRAMMING LANGUAGE

by

RICHARD JOSEPH LEBLANC JR.

Computer Sciences Technical Report #309
MACC Technical Report #49

December 1977

SPECIFICATIONS AND RATIONALE FOR TELOS, A PASCAL BASED ARTIFICIAL INTELLIGENCE PROGRAMMING LANGUAGE

BY

RICHARD JOSEPH LEBLANC JR.

A thesis submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

1977

ABSTRACT

A new programming language called TELOS is introduced through a complete specification of its syntax and an informal description of its semantics. The design of TELOS is discussed in view of a set of language design goals and the needs of artificial intelligence (AI) programming.

TELOS is an attempt to provide powerful abstraction mechanisms and other structuring facilities within a language that provides the special capabilities needed for AI research. Like most other AI languages, TELOS includes facilities needed for experimentation with large stores of general knowledge, tentatively modifiable and associatively referencable, and with various planning and reasoning strategies. However, in contrast to other AI languages whose design has focused on building in certain powerful high-level constructs, the design of TELOS has focused on building in powerful abstraction mechanisms with which these particular high-level constructs, as well as numerous others, can be defined and implemented with reasonable ease.

Recently, programming languages have begun to appear with features specifically designed to facilitate abstraction of the several different kinds needed in the programming process, in particular, data abstraction and control abstraction as well as procedural abstraction. TELOS implements a set of data, control, and procedural abstraction

mechanisms specifically tailored to AI requirements. By emphasizing abstraction mechanisms rather than high-level constructs, it has been possible to minimize theoretical bias in the language, making TELOS potentially usable for investigation of competing theories.

The data abstraction capabilities provided in TELOS add to the already powerful data type extension capabilities of PASCAL. A programmer may define a problem-specific data type by including details of representation and implementation within a definitional scoping called a "capsule". The procedures and functions which realize possible primitive operations on objects of the type being defined are an integral part of the definition, and the objects are characterized and used in terms of these defining operations.

The control abstraction capabilities provided in TELOS enable convenient programmer definition of the novel kinds of control regimes which are investigated in AI research, that is, those which realize alternative problem-solving strategies. Just as TELOS capsules localize data representation details, TELOS "overseers" localize interprocess control-transfer and communication details needed to realize desired control regimes.

Besides its abstraction mechanisms, TELOS contains other facilities which can contribute to the building of well-structured programs, starting with the rich set of program structuring facilities already available in PASCAL. If used

correctly, these facilities can result in modularized, hierarchical programs with reasonably comprehensible control and data flows.

The design of TELOS is based on many of the same goals as the design of PASCAL, though TELOS reflects a different relative emphasis among the goals. There are conflicts between the need to include certain language capabilities seen as necessary to support effective AI programming and the goals of simplicity and minimality. The TELOS design attempts to mediate the conflicts with compromises intended to provide support for programming effectiveness, ease of debugging, program comprehensibility and evolutionary program development.

Several benefits should accrue for AI research from such a language design:

(1) AI programs are highly complex. TELOS provides many aids to managing and containing program complexity.

(2) Much of AI programming involves putting preliminary ideas and hypotheses into programs and then changing the programs as suggested by experience with them. The improved program comprehensibility possible with TELOS can make this kind of evolutionary programming easier and more efficient.

(3) Improved AI programmer productivity, as is possible with the features TELOS provides, will mean improved AI research productivity.

ACKNOWLEDGEMENTS

I would like to thank Professor Larry Travis for his considerable involvement with the development of TELOS over the past several years. His support and enthusiasm for the work have been among his most significant contributions.

Acknowledgement is due to a number of my fellow graduate students for many useful discussions. In particular, Steve Zeigler and Marty Honda have had a considerable influence on the final design of TELOS.

Professor Travis, in his former capacity as Director of the Madison Academic Computing Center, and Professor Tad Pinkerton, as his successor, deserve thanks for providing some financial support for the development of TELOS. Additionally, my work with Professor Charles Fischer on the UW-PASCAL compiler (also supported by MACC) has been an invaluable experience.

I wish to thank my parents for all of the encouragement they have given me for so many years. My love of learning is among their greatest gifts to me.

Finally, a special thanks to my wife Kathy. Her patience and understanding have made this possible and her devotion has made it especially worthwhile.

TABLE OF CONTENTS

CHAPTER 1

Requirements for an AI Programming Environment

A major consideration in the design of a programming language is the type of programs for which the language will be used. The characterizing features of a language are those which a designer feels are fundamental and thus will be used frequently. Such features are implemented as part of the language to allow the programmer to ignore details and concentrate on algorithms. Certain specialized features have been seen as useful for programs in the area of artificial intelligence and consequently have been included in some recently developed languages for AI programming.

Important aspects of a programming environment for the development of artificial intelligence programs include capabilities for programming complex environment structures, facilities for sophisticated data manipulation, and system support for interactive incremental model building. List processing languages were an important first step toward providing an environment to support the development of programs that manipulate complex symbolic data with varying structure. The evolution of highly interactive LISP [MAEHL62] systems that provide advanced testing and editing components was the next important step. Finally, several specialized programming languages have been developed, usually as extensions to existing languages intended for more

general uses; e.g., PLANNER [Hew72] and CONNIVER [MS72] from LISP.

Associative Retrieval

Associative retrieval capabilities are one of the most important components of these latest AI languages. Their run-time environments generally include a data base or associative net that is accessed only by way of content-based references. The content of this data base is usually interpreted as the "state of the world" as known to the program (in a robot problem solving program, for instance) or as a knowledge base from which to draw facts or inferences (e.g., in a question answering program). The data base essentially provides an alternate way to reference data objects, quite different in both form and utility from the traditional approach of naming data objects by binding them as the value of a variable.

The basic mechanism of associative retrieval is to reference a complex data object (an object with structure rather than a single atomic value) through partial specification of its value. Thus the entire object is referenced through the content of the specified part or parts. Clearly, such a reference is not necessarily unique. This lack of uniqueness is a source of considerable power in providing data manipulation facilities, but can cause both conceptual and efficiency problems due to its potential generality.

Under the most general implementation of such a retrieval mechanism, one could attempt to retrieve any complex object with arbitrary structure that includes a particular atomic value any place within the structure. Depending upon the implementation, this request could be expensive. Further, it is not obvious that a program could meaningfully process various results of such requests, since the results could have different and unrelated structures. A language designer might well be wary of providing facilities that make such a retrieval appear as simple to a programmer as a more completely specified reference that includes some structural information. Further, an implementor might well question whether the retrieval mechanisms necessary to honor efficiently such a general request should be implemented at the expense of making more likely operations less efficient than they might otherwise be.

A second aspect of associative retrieval included in the latest AI programming languages is the invocation of procedures by matching a "goal pattern" rather than by naming them directly. This feature is an extension of the idea of generic procedures found in general purpose programming languages. A generic procedure is really a set of related procedures, all called by the same name, but differing in the type(s) of the parameters on which they operate and the values they return. The compiler identifies the procedure to be called and generates the appropriate code.

Pattern directed invocation eliminates the requirement that the caller designate the name of a procedure to be called. Rather, a goal to be achieved is specified, allowing any procedure whose goal pattern matches the goal specification to be activated. This matching is done at run time, allowing the use of procedures added to the system after the calling procedure was defined or compiled. As in the case of associative data retrieval, there exists a possible ambiguity in matching the goal specification against goal patterns of procedures. Again, this is a source of both power and difficulties. The ambiguity is typically resolved by invoking the procedures that match in succession until one indicates that it has successfully achieved the goal. (The associated concept of failure and backtracking will be discussed below.) This sequencing results in a depth-first search for a solution.

While depth-first searching is adequate for some problems, it is frequently inefficient and even unnatural as a search strategy. CONNIVER includes some features that give a programmer more control of the searching technique used. A language might go farther along this line by allowing procedures to be retrieved according to their goal pattern, much like data objects, and giving complete control of invocation to the programmer. This flexibility requires the system to provide the programmer with the ability to suspend the execution of a procedure and to restart it later, along

with the capability to manipulate these suspended procedures as data objects, including functions to examine the state of their executions as a basis for scheduling.

Contexts

An important tool for programming many AI applications is a facility for saving and restoring data contexts. A context consists principally of the state of the associative data base, but may also include the values of some or all global variables. Contexts are crucial for the implementation of any sort of mechanism for attempting alternative solution methods, such as backtracking.

Previous mention has been made of the use of the data base as a world model. Frequently procedures attempting to achieve some goal, modeled by the the establishment of some new state of the data base, will be designed to manipulate the data base as intermediate goals are processed. The programmer could write procedures to keep an account of all changes they make so the changes might be undone, if necessary, but it would be a tedious and often inefficient task. Having a mechanism for saving and restoring contexts built into the language allows considerable simplification in implementing a program that may attempt alternative solution strategies.

The simplest mechanism of the kind just described provides the programmer with a simple backtracking environment;

that is, the ability to save the context at a decision point and to allow processing to continue with a selected strategy until it succeeds or fails. Success allows processing to continue on to the next step while failure results in a restoration of the context at the decision point and the invocation of an alternative strategy. Such are the mechanisms provided in PLANNER and FUZZY [LeF74]. A stack of contexts is available to programmers in these languages.

A number of uses can be seen for a more general context scheme. Backtracking is useful only for implementation of depth-first searching and, even if it happens that such a strategy is well-suited to a problem, a programmer might prefer to preserve some information from attempts that fail. Such preservation requires the retention of the various contexts used in a search for later reference, creating a tree of contexts that branches (potentially) at every decision point. This approach, of course, has wider applications than merely providing more information to a depth-first attempt at a solution. It can be used to implement a breadth-first search or any more sophisticated scheme, such as a "multitracking" approach, where a number of strategies are attempted in parallel. These strategies may be implemented through use of parallel multiprocessing hardware or through use of software to provide pseudo-parallel processing, perhaps using dynamic evaluations of progress for scheduling decisions.

Another important use of contexts is in the area of planning. Plans that must be designed in the absence of complete information or depending on future events must include options for the various situations that might occur. An obvious way to implement a program that creates conditional plans is to add information to the data base to represent each of the possibilities in an unknown case. Having a context mechanism simplifies such an approach, particularly when segments of a plan may depend on any number of conditional assumptions.

The Concept of Failure

An ubiquitous feature of AI programming languages is the concept of failure, previously mentioned in the discussion of contexts. Failure is a special kind of return from a procedure, which signifies that a procedure was unable to achieve a goal or otherwise fulfill its role in a search. Having the concept of failure built into a language does not really add any new capabilities, but like many other features of AI languages, it enables the programmer to concentrate more on his algorithm and less on coding details.

Failure indication is the basis of the deductive mechanisms that are an integral part of several AI languages. Failure initiates backtracking in PLANNER, QLISP [Wil76] and FUZZY. Since CONNIVER offers more general control statements, failure in that language is not an automatic trigger

but will certainly play an important part in any search strategy a programmer designs. Failure is also a part of the less powerful control mechanisms of SAIL [Van73], where it is the negative return of a match function that controls a FOR EACH statement. (This statement provides the ability to process, in turn, each value that a match function generates.)

Demons

Another language feature unique to AI systems is the demon. A demon is a procedure activated by a conditional interrupt, usually when some "sensitive" data is altered or accessed. After execution of a demon, control returns to the procedure that was executing when the event that triggered the demon occurred (as is typical for an interrupt).

The semantics of demon invocation are highly dependent upon how the programmer uses demons to alter global variables and the data base. The most common use of demons is a solution to the "frame problem" [MH69] that occurs in deductive problem solving formalisms. Simply stated, this involves maintaining the logical consistency of a data base as it is used to represent successive states in a solution space. Demons may be invoked as assertions are added to or removed from the data base, making other appropriate changes in a state to insure its consistency.

Generators

Generators are a control construct introduced in the list processing language IPL-V [NTFGM64] and carried over to some later AI languages, particularly CONNIVER, FUZZY and SAIL. More recently, general-purpose languages such as ALPHARD [SWL77] and CLU [LSAS77] have included generators. Generators are special purpose coroutines that may be reentered, providing a new value each time they are entered. They may be used to scan a list structure, return the results of successive deductions or pattern matches, or return at each step the result of any iterative general computation implemented by a programmer. To implement generators as truly independent of the procedures to which they provide values, it is necessary that they have the ability to operate in an independent data context (maintained during suspensions), thus requiring the context tree implementation discussed above, rather than the simple context stack. A tree is necessary so that a procedure using a generator is not restricted in its use of contexts in between invocations of the generator.

Incremental Model Building

The construction of complex heuristic programs can be greatly aided by interactive program development tools. In contrast to the development of more well-defined algorithmic programs, experimental execution and subsequent modification

is frequently part of the design effort for AI programs. Thus a system that provides editing and interactive debugging packages is strongly desired. The most advanced developments in this area are found in the work of Teitelman [Tei69,Tei77].

Such tools are only a part of the desired capabilities, however. It is also important that a system be procedure oriented so that an entire program need not be recompiled or reprocessed as the result of modifying a limited number of existing procedures or adding new ones. In an interpretive LISP-based system, this feature is available automatically since LISP is inherently function oriented. It is something that must consciously be developed in any compiled system, particularly one based on a general-purpose programming language rather than LISP.

The associative retrieval of procedures makes it possible for existing procedures to interact with newly defined ones without the necessity that they be "aware" of one another (in the sense of being able to name one another explicitly). This capability has been stressed as one of the main strengths of pattern-directed procedure invocation (or retrieval), simplifying the execute-and-modify cycle described above as a useful development technique and permitting the addition of new heuristics to achieve a particular goal without requiring modification of the procedures that

generate that goal, or even without detailed knowledge of what other heuristics are available.

CHAPTER 2

Existing Artificial Intelligence Languages

The recognition of a number of inadequacies shared by some or all of the existing artificial intelligence programming languages has led to the design of TELOS. One of the major drawbacks in existing languages is the lack of a capability to work with large amounts of data as is necessary for "real world" problem solving, particularly due to efficiency problems. Other problems with these languages include context mechanisms, local and global control primitives, facilities for "non-AI" processing, general efficiency and transportability.

Data Base Size

The attention of AI researchers has of late turned away from demonstrating solutions to "toy" problems toward "real world" problems. (Examples of such systems include the Hearsay speech understanding system [REFN73] and MYCIN [Sho76], a medical diagnosis program.) The ability to handle a large data base is crucial to developing a problem solving system that can function in a complex environment. The LISP-based AI languages are not used sucessfully for such work, largely due to the lack of efficiency in available implementations of these languages. Part of the reason for this difficulty is that, being rooted in LISP, they lose ef-

ficiency when forced to manipulate list structures (their only primitive data type) to perform tasks for which they are not well suited. SAIL has a more efficient data base mechanism, at the expense of only handling ordered triples (though they may be nested).

The existing AI languages do not give programmers ways to help a compiler or interpreter execute a program as efficiently as possible. An important aid in the manipulation of large data bases would be some way to specify logical segmentation of the data base. Segmentation would allow requests for searches with respect to one or several of these logical segments with the implication that a system might maintain separate indices of the objects in the various segments if such a scheme is found most efficient by the implementor.

The existing languages are also not well suited for work with large data bases because they allow no interaction with secondary storage other than perhaps depending upon a demand paging mechanism that might be part of the operating system under which a system runs. Another advantage of logical segmentation is that a system might use it for storage management purposes. Objects in the same logical domain are natural candidates for storage together so as to minimize swapping, and if the associative index of the data base is also segmented, even it need not all be kept in working memory. A system can even use the segmentation as a basis for

anticipating storage requests so as to fetch information from secondary storage before it is actually required.

Once an object is located in the data base, all of the LISP-based languages except QLISP require that a working space copy be made before the object can be used. (QLISP does not separate the data base and working space.) As the data base and the objects in it grow more complex, this copying becomes particularly expensive. A mechanism for referencing objects in the data base, possible with some restrictions on how the reference may be used to change an object, can be very useful in minimizing the cost of interacting with such objects.

TELOS is designed as a set of extensions to PASCAL. This helps to eliminate some of the data base efficiency problems found with LISP-based languages, since the more flexible data structures of PASCAL can be used to implement the data base mechanism. Since PASCAL is a strongly typed language, objects in the data base that have different structures are recognized to be of different types. These type distinctions provide a natural segmentation of the data base, yielding the advantages discussed above. Finally, while TELOS does separate the data base from the general working space (PASCAL's heap storage for dynamically allocated objects), pointers to data base objects are included in the language. Data base pointers eliminate the ineffi-

ciency of unnecessarily copying data base objects into the working space in order to examine them.

Contexts

PLANNER and FUZZY offer a context mechanism that allows the programmer a stack of contexts. In MICRO-PLANNER, a new context is created at each backtrack point and released when the alternatives at that point have been exhausted. In FUZZY, a new context is created each time a procedure is entered and released upon exit (with appropriate effects when the procedure succeeds). FUZZY also allows the programmer to add new contexts explicitly to the stack and creates contexts in association with the FOR constructs available for looping. As previously noted, such a stack mechanism is only useful for programming searches using a depth-first strategy. Since many problems do not lend themselves to a depth-first search and since such a strategy is often inefficient even if appropriate, a more powerful mechanism is clearly desired.

This particular problem was a principal motivation for the development of CONNIVER [SM72] after early experience with MICRO-PLANNER revealed the weakness. The approach taken by Sussman and McDermott in developing CONNIVER was to allow the programmer to treat context frames as data objects and build them into any desired structure. This has the appeal of great generality, but suffers from the difficulty of

requiring too much involvement by the programmer with the details of an implementation.

The ability to manipulate the data structures used by a system to control a program is a general problem with CONNIVER, not limited to the context mechanism. While such power is sometimes useful, it also encourages the use of "clever" programming tricks that are highly dependent on implementation details, which tends to make programs hard to understand or debug. The available documentation for CONNIVER is, in fact, a unique blend of language description and technical documentation.

The contexts provided in SAIL are quite different, reflecting the fact that SAIL is rooted in ALGOL, a block structured language. Contexts have scope based on the block in which they are declared. Variables must be explicitly stored into and retrieved from a context. Consequently, contexts have no inherent ordering or relationship to one another. Given this context mechanism, it is possible for a context to include a value for a variable that has been deallocated, causing an error if an attempt is made to restore such a variable.

The most crucial problem with the SAIL context mechanism is that it does not include the associative data base. Having contexts for variables is helpful, but as can be seen from the earlier discussion of the use of contexts, the ca-

pability to restore states of a data base is especially useful.

The context mechanism available with QA4 and QLISP offers the best combination of usefulness and usability. It includes primitives for creating a tree of contexts, independent of any control structures used by a program. The importance of this arrangement is that it allows a programmer to maintain a more general hierarchy than a stack so that searches other than depth-first might be conveniently performed. At the same time, the independence from control structures prevents the system from forcing the use of more contexts than are necessary. The net result of these features is a useful, well-defined context mechanism that never need be programmed around and that allows an efficient use of contexts.

TELOS includes a context mechanism much like that provided in QA4 and QLISP. An important addition to this mechanism in TELOS is that a routine may only reference contexts that it can name, and their descendants. There is no system provided routine that allows a routine to discover the name of any ancestor of a context. Thus routines can protect local versions of the data base from routines they call.

Control Primitives

Innovations in control structures in AI languages have been mostly in the area of global strategy control. The two

main contributions of PLANNER in this area were pattern-directed function invocation and automatic backtracking. As discussed previously, automatic backtracking has been found to be quite useful in some cases but a hindrance in others. CONNIVER took the step of offering the programmer the tools to construct any desired control mechanism (backtracking included). SAIL offers a different approach to this specification of global control, because it is not dependent on pattern-directed invocation. The SAIL programmer must explicitly specify saving and restoration of variables in contexts and call procedures explicitly, either by naming them or using procedure variables.

The concept of associatively retrieving a procedure for execution is powerful, but is more controllable used in conjunction with procedure variables as found in SAIL. Procedure variables allow a programmer maximum flexibility in creating control strategies. This situation is an example of a case where all the desired features appear to exist somewhere but are unfortunately not combined in any one previous language.

TELOS allows associative retrieval of routines through the general data base mechanism and the use of routine reference variables. Any criterion (including a goal pattern) may be used to find objects in the data base that contain routine references. There is no automatic invocation of referenced routines; rather, invocation is left to the dis-

cretion of the routine initiating the data base search. This separation of retrieval from invocation, the availability of routine reference variables, and the tree-structured context mechanism provide the basis for programming a wide variety of control strategies. Special routines called overseers are included in TELOS to encapsulate the implementation details of such strategies and thus to present the strategies to other parts of the program as control abstractions.

Programming in the LISP-based AI languages is frequently made difficult by the lack of any local control structures other than COND. While this branching construct is powerful, it isn't very useful for programming iterations. It is unfortunate to be forced to use recursive calls when a simple local WHILE loop would solve the problem. Use of COND and GO to gain this efficiency sacrifices a considerable amount of elegance and clarity. (The result has been the introduction of certain ALGOL-like control structures into some LISP implementations.) SAIL, being ALGOL-based, naturally avoids this problem, having the complete set of ALGOL control constructs to draw upon. TELOS similarly avoids the problem, having available the variety of control constructs included in PASCAL.

In addition to the LISP primitives, FUZZY offers an interesting and useful set of FOR loops. Unfortunately these are tied too closely to the context and backtracking mecha-

nisms, leading to some difficult problems in trying to avoid what the system does automatically.

Facilities for General Processing

SAIL is strong in this area, having all of ALGOL as its base as well as some useful extensions, especially a facility for creating and controlling independent processes. The LISP-based languages are weak here, essentially offering only list processing and recursion. Simply trying to do any arithmetic (not an uncommon need, even in AI programming!) can be difficult in LISP. Of course, a language rooted in an advanced LISP system (as QLISP in INTERLISP [Tei75]) has a less serious problem, since such LISPs include syntactic extensions and operations that relieve many such problems. Nevertheless, a programmer wanting to use one of the AI languages must choose, if such alternatives are available, between the general processing capabilities of SAIL or the more sophisticated data base mechanisms of the LISP-based languages. The fact that the associative retrieval capabilities of SAIL are not linked to the context mechanisms and handle only ordered triples is a severe limitation. Again, there is clearly the need to draw the most useful aspects of the various languages together.

Recent work in the programming languages field has focused on facilities for defining new data types, particularly structures or records consisting of existing types. The

existing AI languages have typically provided a small set of composite types that take atomic values as their component parts. It is more useful to provide for user-defined types in a language and to allow such types to be treated on a par with built-in types in the construction of composite types.

A related research theme in programming languages is data abstraction. Proposals in this area suggest that including basic operations on a data type within its definition provides a number of advantages. A data type can be considered as an abstraction, using the operations so defined on objects of the type without consideration of the implementation of the operations or of the type itself. Such a feature is clearly useful in the construction of large AI systems where sheer complexity is one of the usual problems encountered.

TELOS, having the features of PASCAL included, has no problems with features for implementing the lowest levels of algorithmic detail (such as arithmetic). It also has the powerful type definition capabilities of PASCAL, in particular for defining record and pointer types. Like SAIL, it includes facilities for creating and controlling suspendable processes (though there is currently no provision for parallel execution). Additionally, TELOS includes an encapsulation mechanism intended for the definition of abstract data types.

Effeciency

Experience with the implementations of MICRO-PLANNER and FUZZY on the UNIVAC 1110 has shown them to be unreasonably inefficient for programming other than small problems, even though 1110 LISP is one of the fastest available and the interpreters for these two languages are compiled. The problem seems to be the dynamic nature of LISP, which requires, among other things, resolving all non-local variable references at run-time. This problem seems to indicate the necessity of either a dedicated computer or an AI language that uses a compiled language as its base.

SAIL is the primary example of an AI language based upon a compiled language (ALGOL) and does not suffer from many of the efficiency problems found with the LISP-based languages. As noted previously, the fact that SAIL has ALGOL as a base gives it strong advantages in general processing facilities over the LISP-based languages. The major weaknesses of SAIL are in the AI processing features it lacks, while the weaknesses of the other languages derive to a great extent from their base in LISP. These observations seem to indicate that adding the desired AI features to a general-purpose algorithmic language should turn out to be a fruitful step, exactly what has been done in the design of TELOS.

Transportability

The lack of easy dissemination of most AI languages beyond the research centers at which they were developed poses a difficult problem for the community of AI researchers. (MICRO-PLANNER is something of an exception to this statement.) The lack of general availability of most AI languages makes it impossible for researchers to understand and evaluate new developments, much less take advantage of them and build upon them. The main cause of this problem has been that implementations have been heavily dependent on local LISP features or operating systems. While this does contribute to greater efficiency, some effort must be made to consider the transportability problem.

Comments by McDermott [McD76] eloquently indicate the difficulties that occur in an attempt to extend the work of other AI researchers due to only "preliminary" versions of reported programs being implemented. Only marginal benefits will result from the development of programs complete as reported, if it is impossible to transport them to systems other than the one on which they are developed. It is hoped that TELOS can be made somewhat transportable through the use of PASCAL as its implementation language as well as its basis.

Language Level

According to the developers of most of the languages considered above, far less use is currently being made of these languages (with the possible exception of SAIL) than was anticipated when they were first introduced [SIGART/SIGPLAN77]. The consensus judgement seems to be that the languages are too "high level." They include features that implement particular problem-solving strategies considered necessary when the languages were designed, rather than providing more generally useful utilities. The need seems to be for a language that provides tools for the easy implementation of strategies without being restrictive about their nature. TELOS has been designed explicitly for the provision of such tools, particularly through its abstraction features.

2.PAK

One recent effort toward the design of an AI language has been based on many of the same goals as TELOS. The 2.PAK language has as its main objectives the provision of a good set of primitives suitable for AI applications and the inclusion of ideas obtained from research into programming languages in general [Mel74]. The first of these objectives includes a recognition that the primitives provided by the several languages modelled after PLANNER are sometimes too high-level and are difficult for a programmer to control as

precisely as is required. The second objective leads to such general goals as minimality, readability, simplicity, natural syntax, efficiency and good abstraction capabilities.

The specific features of 2.PAK developed from these goals include its basis as a block structured language, a variety of standard data types typical of general purpose languages with the use of strongly typed variables, user-defined record structures, global control primitives that deal with transfers to and from coroutines, and local sequencing statements that include if, while and case. The AI features include generalized pattern matching with user-defined functions and data types, a context mechanism completely independent of any control primitives and an associative data base in the form of a directed graph with labeled nodes and edges.

The goals of 2.PAK and the resulting features seem intelligently selected from the experiences of programmers with the early AI languages. The main weakness that appears is the restricted nature of the provided data base, which allows nothing but strings as labels, thus eliminating the use of records in the data base. This data base is actually an extension of the base language, and other schemes may be implemented by any programmer. Since the goal of an AI language is to provide such tools, requiring a programmer to

develop an associative retrieval system independently is not the most desirable situation.

The fact that 2.PAK is not based on any well-known programming language also reduces its attractiveness (and probably its transportability). There is considerable resistance by programmers to the adoption of a completely new language. It is hoped that TELOS will be attractive to programmers, since it adds a number of useful features to a well-known and widely available language.

CHAPTER 3

TELOS Specifications

TELOS is an extension of the programming language PASCAL. Although TELOS is designed with the intention of providing features of use specifically in artificial intelligence programming, such programming is sufficiently general that most of the features will be useful in a much wider range of applications. A summary of the main features follows.

As with most other AI languages, a TELOS-compiled program includes an associative data base. TELOS provides a context-saving mechanism that allows alternative versions of the data base to co-exist, and pattern mechanisms used for associative referencing of data base objects. The data base consists of pointer-linked objects constructed from the structured types provided by PASCAL and TELOS. These data base objects have identity across contexts, i.e., they may take on different values in different contexts with these different values being identified as versions of the same object. Data base objects may be referenced associatively but they may also be referenced directly with data base object pointers. Given a pointer to a particular data base object, all of the different context-specific values of the object are potentially available. Data base objects may have data base object pointers as components, enabling ex-

plicit cross-referencing within the data base. Associative cross-referencing is also possible, resulting from data base objects having patterns as components. Data base component pointers are also available; such a pointer references directly the value of a data base object in a particular context, or a component of such a context-specific value.

The TELOS data context mechanism enables the creation of a tree of contexts, with the root context existing at the beginning of program execution. All other contexts must be explicitly created by the programmer. In addition to the system-provided data base, variables may be explicitly declared as context-relative. An independent instance of each such relative variable is provided when a new context is created.

The patterns used in associative retrieval are constructed as "calls" to (i.e., records containing actual parameter assignments for) a special kind of routine called a matchroutine. TELOS provides three standard matchroutines: MatchAny, MatchValue and MatchObject. MatchAny successfully matches any object; MatchValue matches only objects exactly equal to its single parameter; and MatchObject uses other matchroutine parameter records provided as its parameters to match structured objects component by component. Matchroutines that satisfy any other desired specification may be provided by the programmer.

More generally useful features in TELOS include routines that may be executed as suspendable processes; a mechanism for building encapsulated, abstract data types, possibly including suspended processes as components; an event mechanism that integrates exception handling, process intercommunication and control, and user-specified conditional interrupts (the "demons" of earlier AI languages); routine reference variables, parameter records ("calls") and generalized routine invocation; and extensions to the pointer and array types provided in PASCAL.

Coroutines are routines much like procedures, differing in that they are executed as processes rather than being simply called as procedures are. That is, coroutines may return control to the routines which invoke them without terminating. They may be used to define iteration-controlling generators or they may be used by control-regime-defining overseers for implementation of any control scheme desired. Overseers are routines that serve as process schedulers and inter-process message handlers, encapsulating the details of control regimes (e.g. complex problem-solving strategies) implemented using processes. An overseer serves to implement a strategy as a control abstraction, much as encapsulated data types implement data abstractions.

The data encapsulation mechanism of TELOS provides for the definition of capsules, which include a data structure

definition along with routines defining the operations possible on the structure. Capsules may be parameterized by types, thus allowing the generation of a number of distinct capsule types based on a single capsule definition. The abstraction capabilities provided by overseers and capsules will be useful in managing the complexity that arises in the design of large programs (e.g., AI programs).

The event mechanism of TELOS provides for the definition of three kinds of events: escape events for exception handling, suspend events for process control and intercommunication, and signal events for a conditional-interrupt capability. Event handlers, which are invoked when events occur, may be activated on routine calls and within compound statements. Related handlers may be grouped together as teams.

Parameter forms may be defined in TELOS, distinguished by the number and types of parameters required by a routine. These forms may be used to declare variables that reference routines and to pass such references as parameters, providing a complete specification of the parameters required by a formal routine. Parameter records defined in terms of these parameter forms, i.e., records containing fields corresponding to the parameter structure of a routine, provide a capability in TELOS for creating and manipulating expressions as data objects. The capability to include routine reference variables and parameter-describing patterns (or other kinds

of routine descriptions) within objects stored in the data base provides the basis for a programmer to implement pattern-directed (or other kinds of generalized) routine invocation.

Pointers in TELOS may be declared to reference objects of more than one type. Such multi-type pointers are restricted to use within typecase statements, which guarantee the type of the referenced object before it is used and enable the retention in the language of the advantages of strict typing of pointers.

Array declarations are extended in TELOS to allow some type compatibility between arrays of different size but with the same component type and with index types which are subranges of the same type. These extensions allow differently sized arrays to be passed as parameters corresponding to a single formal parameter type, and it also allows creation within dynamic storage space of arrays whose size is determined at run-time.

The following specifications describe the extensions to the syntax and informal semantics of PASCAL that comprise TELOS. They are based on the syntax and notation of the revised report as printed in [JW75]. The complete syntax of PASCAL is included here, but only the TELOS extensions are discussed. Where syntax is presented without discussion, it is unchanged from the PASCAL report.

1. Notation

Throughout this document the symbol "->" is substituted for the "up arrow" symbol used in pointer declarations and operations in PASCAL. The style of capitalization adopted for reserved words below does not imply that capitalization must be significant to a TELOS compiler.

<letter> ::=

A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|

a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<special symbol> ::= + | - | * | / | = | <> | < | > | <= |

>= | (|) | [|] | { | } | := | . | , | ; | : | ' |

-> | [! | !] | [? | ?] | [* | *] | $ | And | Any |

Array | Begin | BeginX | Capsule | Case | Clone | Const

| ContextRef | Continue | Coroutine | DB | DBC | Declarations | Div | Do | Downto | Else | End | Environment

| Escape | Event | Except | Exports | Extend | File |

For | Form | ForStep | Function | Goto | Header | If |

In | Incontext | Index | Indexing | Label | Matching |

MatchRoutine | Mod | NewProcess | Nil | Not | Of | Or |

Otherwise | Overseer | Packed | Parameters | Pattern |

Procedure | ProcessRef | Program | ReadOnly | Record |

Relative | Repeat | RoutineRef | Routines | Sequencer |

Set | Signal | Suspend | Team | Then | To | Type |

Typecase | Unstored | Until | Var | While | With

1.1 Identifiers

Identifiers are defined as

<identifier> ::= <letter> {<letter, digit or underscore>}

<letter, digit or underscore> ::= <letter> | <digit> | _

1.2 Numeric and string constants

Integer and real constants are described by

<unsigned number> ::= <unsigned integer> | <unsigned real>

<unsigned integer> ::= <digit sequence>

<unsigned real> ::= <unsigned integer>.<digit sequence> |
    <unsigned integer>.<digit sequence>E<scale factor> |
    <unsigned integer>E<scale factor>

<digit sequence> ::= <digit>{<digit>}

<scale factor> ::= <unsigned integer> |
    <sign><unsigned integer>

<sign> ::= + | -

Strings are defined as

<string> ::= '<character>{<character>}'

Strings consisting of a single character are the constants of the standard type Char, as in PASCAL.

2. Data type definitions

<type> ::= <simple type> | <pointer type> | <reference type> |
    <capsule type> | <structured type>

<type definition> ::= <identifer> = <type>

2.1 Simple types

<simple type> ::= <scalar type> | <subrange type> |
<type identifier>

<scalar type> ::= (<identifier> {,<identifier>})

<subrange type> ::= <const> .. <const>

Standard simple types are Integer, Real, Boolean and Char.

2.2 Pointer types

<pointer type> ::= <working space pointer type> |
<data base pointer type>

2.2.1 Working space pointer types

<working space pointer type> ::= -> Any |
-> <type identifier> {Or <type identifier>}

TELOS includes pointers that may reference dynamically allocated objects of more than one type. A pointer type declared with a sequence of type identifers separated by Or (rather than a single identifier, as in PASCAL) allows a pointer of this type to reference objects of any of the types included in the specified set of type alternatives.

Use of the Any specification allows a pointer to reference objects of any type. Pointer variables that may reference more than one type are called multi-type pointers. The pointer in PASCAL is simply the special case of a TELOS working space pointer which may point to objects of only a single type.

There are restrictions placed on the use of multi-type pointers to guarantee type security. Type-dependent uses of referenced variables (<pointer variable>->) based on multi-type pointers may occur only within a typecase statement (see Section 7.2.2.1). The value of a multi-type pointer may be assigned to another pointer in general only if the second may reference any type referenced by the first. Assignment to a more restricted pointer may take place only within a typecase alternative that guarantees that an object of an allowable type is being assigned to the more restricted pointer. Any two pointers that may reference at least one common type may be compared for equality or type equivalence (see SameType function, Section 8.8) at any time.

2.2.2 Data base pointer types

<data base pointer type> ::= <data base object pointer type>
    | <data base component pointer type>

2.2.2.1 Data base object pointers

<data base object pointer type> ::= DB-> Any |
DB-> <type identifier> {Or <type identifier>}

Objects in the data base may be referenced using data base object pointers (DBOP's) as well as through the associative referencing mechanism (described in Section 3.6). The value of an object (a DBO) referenced by a DBOP is context dependent, being the value for the current context if a context prefix is not specified when the DBOP is used. DBOP's are read-only references. They cannot be used to directly alter the value of a data base object with a simple assignment statement. System functions are provided for changing DBO values through DBOP's. Note that DBOP's may be components of objects stored in the data base, so that DBO's may cross-reference one another. Multi-type data base pointers are subject to the same restrictions as multi-type working space pointers.

2.2.2.2 Data base component pointers

<data base component pointer type> ::= DBC-> Any |
DBC-> <type identifier> {Or <type identifier>}

Components of objects in the data base may be referenced using data base component pointers. Like DBOP's, DBCP's are read-only references. The objects they reference

may not be changed with an assignment statement or passed as Var parameters. However, component pointer references are context-specific. DBCP's are used to reference a component of a DBO value for a particular context. If the structure of the DBO value into which the DBCP references is changed, use of the DBCP becomes an invalid pointer reference.

2.3 Reference types

<reference type> ::= <routine ref type> | <process ref type> |
<context ref type>

Assignment of a reference variable value to a second reference variable does not cause a copy of the referenced object to be made, but rather results in two variables referencing the same object.

2.3.1 Routine reference types

To improve on the capability to pass routines as parameters and to allow variables that reference routines, routine reference types are definable in TELOS. Routine reference types are defined in terms of parameter forms, which in turn are defined in terms of parameter and return type specifications.

<parameter form> ::=

    <proc routine kind> <parameter spec> ¦

    Function <parameter spec> : <type identifier> ¦

    MatchRoutine <parameter spec> Matching <type identifier>

<proc routine kind> ::= Procedure ¦ Coroutine ¦ Overseer

<parameter spec> ::= (<formal parameter section>

    {,<formal parameter section>}) ¦ <empty>

<parameter form definition> ::=

    <identifier> = <parameter form>

Routine reference types can be defined from parameter forms using:

<routine ref type> ::= RoutineRef Any ¦

    RoutineRef <parameter form identifier>

      {Or <parameter form identifier>}

<parameter form identifier> ::= <identifier>

A routine reference type defined with only one parameter form identifier specified can be used to declare variables that may reference routines with only one parameter structure. An empty <parameter spec> in a parameter form declaration indicates that the parameter form defined has no parameters.

A routine reference type defined with Any as the form specification or with a sequence of parameter form identifi-

ers separated by Or can be used to declare a multi-type routine reference variable. In the former case, a variable of the type may be used to reference any routine; in the latter, it may take on values from a restricted set of parameter forms. Multi-type routine reference variables are restricted in much the same way that multi-type pointers are. They can be used to call a routine only in a typecase statement. The value of a multi-type routine variable may be assigned to another routine reference variable in general only if the second may take values of any form that may be a value of the first. Assignment to a more restricted variable may take place only within a typecase alternative which guarantees that a routine of the right form is being assigned to the restricted variable. Routine reference variables may be compared for equality at any time. They may take as values only routines declared at the outermost level (not routines declared within other routines).

2.3.2 Process reference types

One of the routine kinds introduced in TELOS, the coroutine (see Section 5.2), is executed as a suspendable process. Variables used in the manipulation of such processes are process reference variables. Process reference types are defined using:

<process ref type> ::= ProcessRef Any |

ProcessRef <parameter form identifier>

{Or <parameter form identifier>}

A process reference variable can reference processes created from coroutines with any of the parameter forms specified in its process reference type declaration. Of course, a single parameter form is an alternative. If a process reference variable can reference processes created from more than one parameter form, any type-dependent use of the variable must take place within a typecase statement. The restrictions on multi-type process reference variables are analogous to those on pointers and routine reference variables.

An overseer or a routine within a capsule (in a restricted way) may use a process reference variable to access or change the parameter values of a suspended process. The form of such a reference is:

<internal variable> ::=

<process ref variable>.<parameter identifier>

where <parameter identifier> is one of those appearing in the parameter form declaration of the process archetype.

2.3.3 Context reference types

To support the data context mechanism of TELOS, a built-in type called ContextRef is provided. Variables of this type are used to name the contexts created and manipulated by the context operations described in Section 8.4. A standard variable

CurrentContext : ContextRef

may be used in any program to access (though not to change) the currently active context.

2.4 Capsule types

An encapsulation facility is provided for the definition of data abstractions. It enables the definition of data objects whose structure may only be accessed and manipulated by routines that are part of the defining capsule. Unless explicitly exported, representation details of capsule objects are not available. Capsule definitions may be parameterized by types. A capsule type is defined using a capsule name and appropriate actual parameter type names. Parameters other than types are not possible; in particular, TELOS does not consider size to be a defining characteristic of capsule data types.

<capsule type> ::= <capsule identifier> |
    <capsule identifier> (<type identifier>
      {, <type identifier>})

2.4.1 The capsule specification

<capsule definition> ::=
    <identifier> = <capsule specification>
<capsule specification> ::= Capsule <type parm list>
    <exports part> <constant definition part>
    <type definition part> <header part>
    <event declaration part> <routine heading list> End

<type parm list> ::= <empty> |
    (<formal type parm> {;<formal type parm>})
<formal type parm> ::=
    <identifier> [<routine spec> {;<routine spec>}]
<routine spec> ::= <standard op> |
    <identifier> : <parameter form>
<standard op> ::= = | <> | < | > | <= | >= | :=

The type parameter descriptions must include specifications of all operations that are required to be defined for allowable actual type parameters. The syntax provides a set of standard operators that may be required of formal type parameters. When an actual parameter used in a capsule type definition is itself a capsule type, the relational opera-

tors stand for functions within the parameter capsule named EQ, NE, LT, GT, LE and GE, respectively, which take two operands of the type as ReadOnly parameters (see Section 5.1) and return a Boolean value. The assignment operator stands for a procedure named ASSIGN that takes two parameters of the same type, the first a Var parameter and the second a ReadOnly parameter. These standard ops are only an abbreviation. The same routines could be imported using the second form of specification. The second form includes the specification of a routine name and its parameter form. Any actual parameter (type) must include an operator that matches this specification.

When the actual type parameter is a standard PASCAL type, the meanings of the standard operators are as defined in PASCAL. (All are not necessarily defined; recall in particular that comparisons are undefined for record types in PASCAL.) When the actual parameter type is a capsule type, routines with the required names and parameter forms must have appeared in the capsule specification used to define the parameter type.

The exports list has the following form:

<exports part> ::= Exports <export item> {,<export item>} ;
<export item> ::= <identifier> |

<identifier> (<export item> {,<export item>})

The various forms of export items serve several different purposes. The first form, simply an identifier, is used to export any identifier defined within the capsule. This includes constants, types, events, and variables in the header part. (Note that capsule operations and their parameter forms are indicated to be externally available not in the <exports part> but rather in the <routine heading list>; see below.) The internal structure of any record or capsule so exported is not accessible outside of the capsule, and header variables may be used only to reference values, not to change them. (Header variables of process reference types may not be exported.) The second export item form is used for exporting capsules and other types defined within the capsule, allowing external reference only to the particular operations or internal details of the exported type as specified with the subordinate exports list included within parentheses. Names of routines within an inner capsule must appear in this list to be accessible outside of the outer capsule.

<header part> ::=

Header <record section> {; <record section>} End

The <header part> defines what is in effect a record (without variants) that serves as the top-level portion of the data structure defined by the capsule. An instance of this "record" is allocated when an object of this type is created. At later stages of processing, the entire object may consist of this header and other data objects that can be reached by following pointers, starting with any contained in the header. The rest of the structure is built by routines defined in the capsule. Thus the total structure of a capsule object is defined only implicitly at the point of capsule definition.

<routine heading list> ::=

    <routine heading> {, <routine heading>}

<routine heading> ::= <procedure heading> ¦

    <function heading> ¦ <overseer heading> ¦

    <coroutine heading> ¦ <matchroutine heading>

The routine heading list of the capsule specification defines the externally accessible routines that are part of the capsule. Only the headings of the routines are part of the specification of the capsule. Their bodies will be part of the capsule body, which appears in the routine declaration section of the program.

<constant definition part> and <type definition part> denote the same syntax here as in the standard PASCAL definition. See section 2.5.2 for the definition of <record section>.

From outside of the capsule, references to names in the exports list that are fields of an instance of the header part are analogous to references to PASCAL record fields:

<capsule variable>.<identifier>

The routines appearing in the routine heading list are the operators defined for objects of a capsule type. All of these operators must have their operand objects passed as explicit parameters. In general, capsule names should be distinguished from type names, because capsules may be parameterized. In the case of these particular operand objects, their "type" in the formal parameter specification of the operations may be specified with the name of the capsule within which the operations are being defined. These operations are not associated with a specific instance of the type, but rather are allowed to use and reference non-exported items defined in the capsule because of their inclusion in the capsule definition. References to externally accessible routines and to types and constants in the exports list from outside of the capsule take the form:

<capsule type identifier>$<identifier>

2.4.2 The capsule body

The details of the implementation of a capsule are contained within the capsule body, which appears in the routine declaration section of the program.

<routine declaration> ::= <capsule body>

<capsule body> ::= Capsule <capsule identifier>
<constant definition part> <type definition part>
<header part> <event declaration part>
<routine declaration part> End

The <capsule identifier> associates the capsule body with the capsule specification previously detailed for that identifier (in the type definition part of the program). Any constants, types or header variables that appear in the body are additions to those that appeared in the specification and are accessible only within the body itself. Any such header variables may be considered to be appended to those from the specification part to form the entire header "record". No definitions from the specification part may be repeated in the body.

All routines listed in the specification must be declared fully in the routine declaration part. The headings of these routines are repeated in the body, with the number and types of their parameters required to be the same. Oth-

er routines may also be declared in the body. These routines are only accessible within the body itself.

2.5 Structured types

<structured type> ::= <unpacked structured type> ¦
    Packed <unpacked structured type>
<unpacked structured type> ::= <array type> ¦ <record type> ¦
    <set type> ¦ <file type> ¦ <parm record type>

Set types and file types are the same in TELOS as in PASCAL. Array types and record types are extended as described in the following sections and parameter record types are added as a new structured type.

2.5.1 Array types

<array type> ::=
  Array [<index type> {,<index type>}] Of <component type> ¦
  <array type identifier> [<index type> {,<index type>}]

Array declaration syntax is extended so that the index type range specified in an array type declaration determines allowable index values for an array of that type, but it does not necessarily determine array size. If a variable is declared to be of an array type simply using a type identifier (or an explicit in-line array declaration), its size is

determined by the maximum allowable index values (just as in PASCAL). If the second declaration form above is used, the size is determined by the ranges of the provided index types, which must be the same as or subranges of the corresponding index types in the original declaration. The number of index types provided must be the same as in the original declaration. Array variables declared using the same array type identifier but with different index ranges may be provided as actual parameters for a formal parameter of the array type from which they were defined (see Section 5.1).

Two standard functions are introduced for use with arrays. HiBound and LoBound produce the upper and lower bound values when provided with an array reference and an integer dimension as parameters. Thus HiBound(A,2) returns as its value the upper bound of the second dimension of array A.

TELOS provides extensions of the standard procedure New that allow the dynamic allocation of arrays with size determined at run-time (described in Section 8.2). Array assignments can only take place when HiBound and LoBound return the same values for the source and target arrays. Thus assignment can never change the size of an array object.

2.5.2 Record types

<record type> ::= Record <field list> End

<field list> ::= <fixed part> | <variant part> |
<fixed part> ; <variant part>
<fixed part> ::= <record section> {;<record section>}
<record section> ::= <empty> |
<field identifier> {,<field identifier>} : <type>
<field attributes>
<variant part> ::= Case <tag field> <type identifier> Of
<variant> {;<variant>}
<variant> ::= <empty> | <case label list> : (<field list>)
<case label list> ::= <case label> {,<case label>}
<case label> ::= <constant>
<tag field> ::= <identifier> : | <empty>

The standard PASCAL record declaration syntax is extended only by the addition of <field attributes> to the declaration of field names and their type in a <record section>. The various attributes that may be specified by this extension provide instructions to the data base storage mechanism when objects of the type being declared are stored in the data base.

<field attribute> ::= <empty> | <indexing attribute> |
Unstored
<indexing attribute> ::= Index <exception list> | Sequencer
<exception list> ::= <empty> |
Except (<constant> {, <constant>})

The indexing attribute Index may be attached to any field, signifying that the values in or pointed to by that field are to be usable for associative retrieval from the data base of objects of the record type being declared. Data base objects consist of pointer-linked structures with a single record, array or capsule header designated as the "top object" when such a structure is stored. For the Index attribute on fields in other than the top object to cause those fields to be indexed, all pointers followed to reach the object from the top must have Index attached to them. An exception list may be attached to any field which is of a simple type, specifying certain constant values of that type for which indexing is not to be performed. The Sequencer attribute may be substituted for Index on any field which is of an enumerated type, real type or string type (packed array of Char). This indicates that the order in which objects are retrieved by Find and FindEach (see Section 8.6) is to be based on the field. The ordering is descending for a field of an enumeration or real type and lexicographic for strings. Only one sequencer per record is allowed, and only the sequencer in the top object of a data base object is used by the retrieval functions.

Unstored may be attached to any field, meaning that the field will not be included when an object of the type containing the field is stored in the data base. No space is

provided for it in a data base object and the field is thus inaccessible using data base pointers. Any Unstored fields are left uninitialized when objects containing them are copied from the data base to working space.

2.5.3 Parameter record types

Records with fields corresponding to the parameter structures declared for parameter forms may be declared as parameter record types. Parameter records may be used to build the actual parameter specification of a routine call at run-time.

<parm record type> ::=

    Parameters Of <parameter form identifier>

The first field of a parameter record is named Routine and its type is RoutineRef <routine form identifier>, using the form specified in the type declaration. The record defined contains additional fields named and typed by the parameter names in the corresponding parameter form declaration.

Parameter fields corresponding to value parameters may be given values by simple assignment of a value of an allowable type to the field. Var and ReadOnly (see Section 5.1) parameter fields contain a reference to an object rather

then a value. This reference may be to any global variable not in a variant, to a field in the same record or capsule header as the parameter record, or to a dynamically allocated variable (i.e., an object or component of an object pointed to using a working space pointer). Var and ReadOnly parameter references are established using the Bind routine, described in Section 8.3.

2.5.3.1 Patterns

<working space pointer type> ::=

    Pattern Matching <type identifier>

The "pattern" type definition syntax above defines a multi-type pointer that may point to any parameter record for a matchroutine that will match <type identifier>. Thus the range of this pointer type is implicitly defined by the matchroutine declarations in the program. Matchroutine parameter records are used as data descriptions ("patterns") for such purposes as associative retrieval from the data base. A number of system-defined matchroutines and special syntactic conventions, described below, are available to facilitate use of patterns.

3. Variable declaration and denotation

<variable declaration> ::=

    <identifier> {,<identifier>} : <type>

<variable> ::= <entire variable> ¦ <referenced variable> ¦

    <component variable> ¦ <function designator>

<entire variable> ::= <variable identifier>

<variable identifier> ::= <identifier>

<referenced variable> ::= <pointer variable> ->

<pointer variable> ::= <variable>

3.1 Component variables

<component variable> ::= <indexed variable> ¦ <file buffer> ¦

    <prefixed variable>

<indexed variable> ::=

    <array variable> [<expression> {,<expression>}]

<array variable> ::= <variable>

<file buffer> ::= <file variable> ->

<file variable> ::= <variable>

3.1.1 Prefixed variables

<prefixed variable> ::= <variable> . <identifier>

The prefixed variable notation has a number of different interpretations in TELOS. If the variable names a record or a parameter record then the identifier is interpreted as a field name. If the variable is a process reference variable, then the identifier is interpreted as a parameter name of the process archetype parameter form. Finally, if the variable is a context reference variable, then the identifier must be a relative variable (see Section 9.1) or a data base object pointer. In either case, the prefixed variable refers to the context-relative value of the designated object.

3.2 Function designators

A function designator is considered to be a <factor> (see the <expression> syntax in Section 6) in PASCAL. TELOS includes it as a variable so that if a function returns an appropriate value, it may be used in a component variable denotation.

<function designator> ::= <function>
    <actual parameters> <event handler clause>
<function> ::= <function identifier> |
    <routine reference variable>

<actual parameters> and <event handler clause> are defined in Sections 7.1.3 and 7.1.3.1, respectively.

4. Event declarations

The event mechanism in TELOS is available for handling faults that occur during execution, for monitoring certain system-defined events, for monitoring events defined by the programmer, and for communication and control transfers between processes and their controlling overseers. Events are defined using the following syntax:

<event declaration> ::=

<event kind> Event <event identifier> <parameter spec>

<event kind> ::= Escape | Signal | Suspend

All three kinds of events may have parameters, declared like routine parameters. <parameter spec> is defined below in Section 5.1. The three event kinds are distinguished by the following properties. An escape event is regarded as a termination of the computation that caused the event. An event handler (see Section 7.1.3.1) must be active to handle any escape event that is raised or the program will terminate. The handler for an escape event may not return control to the point where the event occurred.

A signal event is informational in nature and need not be handled. Entry and exit of every routine are among the system-defined signal events. If a handler is invoked by a signal event, it may return control to the point in the com-

putation where the event occurred or it may cause that computation to be terminated (simply by not returning control).

A suspend event is used to transfer control from a process back to its controlling overseer (see Section 5.4). All suspend events must be handled by the controlling overseer. The overseer may choose not to immediately return control to a process when a suspend event occurs without causing termination of the process.

5. Routine declarations

<routine declaration> ::= <procedure declaration> ¦
    <function declaration> ¦ <coroutine declaration> ¦
    <overseer declaration> ¦ <matchroutine declaration> ¦
    <team declaration>

To the procedures and functions in PASCAL, TELOS adds coroutines, overseers, matchroutines and teams.

5.1 Procedure declarations

<procedure declaration> ::= <procedure heading> ; <block>

<procedure heading> ::=

    Procedure <identifier> <parameter spec> <form spec>

<form spec> ::= <empty> ¦ Form <parameter form identifier>

<parameter spec> ::= <empty> ¦

    (<formal parameter section>

{; <formal parameter section>})

<formal parameter section> ::= <parameter group> ¦

Var <parameter group> ¦ ReadOnly <parameter group>

<parameter group> ::=

<identifier> {, <identifier>}

: <type identifier> <array indicator>

<array indicator> ::= * ¦ <empty>

<form spec> at the end of a procedure heading may be used to indicate that the procedure has some previously declared form as its parameter form. Even when <form spec> is not null, the formal parameters are included in the declaration. The same type names must be used to declare the parameters as in the form declaration, though renaming of tha parameters is allowed. Declaring a routine without a form name results in an implicit declaration of a form with the same name as the routine.

The presence of a non-empty array indicator after an array type identifier means that actual parameter arrays of any size within the range allowed by <type identifier> may be used. Such formal parameters are called variable-size array parameters.

TELOS adds the ReadOnly parameter mode to those available in PASCAL. Neither a ReadOnly parameter nor any object

referenced through it may appear on the left-hand side of an assignment or be passed as a Var parameter to another routine. Thus the parameter acts like a named constant within the body.

PASCAL allows passing procedures and functions as parameters without any indication of the parameters these routines require. This form of parameter specification is not included in TELOS, and thus TELOS is not upward compatible with PASCAL in this respect. Routines are passed as parameters simply by passing routine references, so no special routine parameter syntax is needed (and a simple routine name is not a valid actual parameter). If the parameter type can accommodate more than one parameter form, it can only be used to call a routine within a typecase statement, as is usual with multi-type objects. Compile-time determination of the correctness of parameters for calls using routine reference parameters is possible using this mechanism.

```
<block> ::= <label declaration part>
        <constant definition part> <type definition part>
        <variable declaration part> <event declaration part>
        <routine declaration part> <statement part>
<label declaration part> ::= <empty> |
        Label <label> {, <label>} ;
<constant definition part> ::= <empty> |
```

Const <constant definition> {; <constant definition>} ;

<type definition part> ::= <empty> ¦

Type <type part definition> {; <type part definition>};

<type part definition> ::= <type definition> ¦

<capsule definition> ¦ <parameter form definition>

<variable declaration part> ::= <empty> ¦

Var <variable declaration> {; <variable declaration>} ;

<event declaration part> ::= {<event declaration>;}

<routine declaration part> ::= {<routine declaration>;}

<statement part> ::= <compound statement>

<label> ::= <unsigned integer> ¦ <identifier>

TELOS allows identifiers as well as integers to be used as labels.

<constant definition> ::= <identifier> = <constant>

<constant> ::= <unsigned number> ¦ <sign><unsigned number> ¦

<constant identifier> ¦ <sign><constant identifier> ¦

<string>

<constant identifier> ::= <identifier>

5.2 Function declarations

<function declaration> ::= <function heading> ; <block>

<function heading> ::= Function <identifier>

<parameter spec> : <result type> <form spec>

<result type> ::= <type identifier>

Any type for which assignment is defined may be the result type of a function, rather than only scalar, subrange and pointer types as in PASCAL. Thus only files and capsule types which don't have an exported ASSIGN procedure are excluded as return values. The return value of a function is specified by assignment to the function identifier within the body. Any other use of the identifier within the body is interpreted as a recursive call of the function.

5.3 Coroutine declarations

<coroutine declaration> ::= <coroutine heading> ; <block>

<coroutine heading> ::=

Coroutine <identifier> <parameter spec> <form spec>

Coroutines are syntactically like procedures, with the substitution of Coroutine for Procedure in the declaration. Coroutines are executed as processes; that is, they may return control to a calling routine without terminating, using an event statement that causes a suspend event. Such event statements may only appear in coroutines and overseers, and in procedures and functions declared within these two kinds of routines. After execution of the last statement in a coroutine, the suspend event EOP (end of process) automatically occurs.

Within capsules coroutines may not be declared within other routines, in order to ensure that the environment of a suspended process is always accessible when it is restarted.

5.4 Overseer declarations

<overseer declaration> ::= <overseer heading> ; <block>

<overseer heading> ::=

Overseer <identifier> <parameter spec> <form spec>

Overseers are special-purpose procedures available for the implementation of user-specified control regimes and problem-solving strategies. They are distinct from standard procedures to serve the purpose of encapsulating the details of such implementations. Overseers are the only routines within which use may be made of TELOS features for manipulating processes, except for routines within capsules which may manipulate processes attached to capsule instances. (Any routine may use the step statement, which executes a "generator" coroutine; see Section 7.3.2.1.) These features include the statements and standard routines that create, copy and terminate processes; the continue statement that invokes (restarts) a suspended process; and the capability to reference the parameters of a suspended process. (Any routine declared within an overseer may also use these features.)

As noted in the previous section, overseers may themselves cause suspend events, suspending the process in which they are executing. (This enables the construction of hierarchical, multi-level control regimes.) Suspend event handlers may appear on overseer calls in order to handle events caused within the overseer, but it is not required that the event be handled at the point of call, except in the case of an overseer call from the main process (the process corresponding to the main program). In the case of such a call, if control is not returned to the suspended overseer from within the handler, the overseer is terminated (analogously to what happens for signal events). Suspend event handlers also appear within overseers, as part of the continue statement that causes a process to be invoked. Since all suspend events caused by a process must be handled in the continue statement used to invoke it, these are the only places that suspend event handlers may appear.

5.5 MatchRoutine declarations

<matchroutine declaration> ::=

    <matchroutine heading> ; <matchroutine block>

<matchroutine heading> ::=

    MatchRoutine <identifier> <parameter spec>

    Matching <identifier> : <match type> <form spec>

<match type> ::= <type identifier>

<matchroutine block> ::= <label declaration part>
<constant definition part> <type definition part>
<variable declaration part> <routine declaration part>
<index part> <compound statement>
<index part> ::= <empty> | Indexing <compound statement> ;

Matchroutines allow the programmer to specify data descriptions other than those available using the system-provided matchroutines MatchPattern, MatchValue and MatchAny. They may have parameters like any other routine and they always have one additional implicit parameter: the object being matched. A name and type for this object are provided after the keyword Matching. The parameter mode for this matching parameter is ReadOnly. If the object to be matched satisfies the conditions of the matchroutine, the routine indicates success by causing the parameterless escape event Matched, which terminates execution of the routine. Failure is indicated by exiting the routine without an explicit success indication. Matchroutines are not executed by explicit call as are other routines. They may only be called by the Match function or by the various data base associative retrieval routines (described in Section 8.6).

Matchroutines have an optional index part, used to limit associative retrieval cost by narrowing the set of candidate objects against which the routine will be applied. The

index part is also used to determine the index entries under which parameter records for matchroutines are listed when they are stored in the data base. Within the <compound statement> that is the body of the index part, values may be provided for indexing by executing

Index (<expression> {,<expression>})

Any number of calls to the procedure Index may occur within the statement. Such a call specifies that the value(s) of the expression(s) in those calls which are executed must be somewhere within any object the matchroutine will attempt to match. This index part is executed only once for a given retrieval operation, before the retrieval mechanism selects the candidates to provide to the matchroutine.

5.6 Team declarations

<team declaration> ::= <team heading> ; <team block>

<team heading> ::= Team <team id> <parameter spec>

<team block> ::= <label declaration part>
<constant definition part> <type definition part>
<variable declaration part> <routine declaration part>
Begin <handler item> {; <handler item>} End

Teams are not executable routines, but rather serve to name and parameterize event handlers, and to enable collection of a group of event handlers into one body. This entire group of handlers can then be used as a "team" by in-

clusion of a single specification in a handler list. Values for the parameters are provided whenever the team identifier is so used. The names defined in the declaration sections of the team may be used within the included handlers. These handlers are, of course, also parameterized by the parameters of the individual events to which they respectively apply. <handler item> is defined in Section 7.1.3.1.

6. Expressions

<unsigned constant> ::= <unsigned number> | <string> |
    <constant identifier> | Nil | Any

<factor> ::= <variable> | <unsigned constant> | <set> |
    (<expression>) | Not <factor>

<set> ::= [<element list>]

<element list> ::= <empty> | <element> {,<element>}

<element> ::= <expression> | <expression>..<expression>

<term> ::= <factor> | <term> <multiplying operator> <factor>

<simple expression> ::= <term> | <adding operator> <term> |
    <simple expression> <adding operator> <term>

<expression> ::= <simple expression> |
    <constructor expression> | <process expression> |
    <simple expression><relational operator>
      <simple expression>

<multiplying operator> ::= * | / | div | mod | and

<adding operator> ::= + | - | or

<relational operator> ::= = | <> | < | <= | >= | > | in

There are several ways in which the expression syntax of TELOS differs from that of PASCAL. As noted in Section 3.2, <function designator> is a <variable> rather than a <factor>. <constructor expression> and <process expression> are added as forms of <expression>. Finally, the constant Any is added, to be used in the construction of matchroutine parameter records. Note that the constant Nil is the null value for reference variables as well as for working space and data base pointers.

6.1 Constructor expressions

<constructor expression> ::=

    <object constructor> | <pattern constructor>

6.1.1 Object constructors

<object constructor> ::=

    <type desc> [! <field value list> !]

<type desc> ::= <type id> | -> <type id>

<field value list> ::= <labeled field value list> |

    <positional field value list> | <array field value list>

<labeled field value list> ::=

    <labeled field value> {, <labeled field value>}

<labeled field value> ::=

    <field identifier> : <field value>

<field value> ::= <expression>

<positional field value list> ::=

    <field value> {,<field value>}

<array field value list> ::=

    : <expression> : <positional field value list>

Record, capsule and parameter-record objects may be constructed using the labeled field value list form. The field identifiers designate the destinations of their corresponding values, which must match the declared types of the fields. The fields need not be named in any particular order, and it is not required that all fields be assigned values. Any fields for which values are not provided are considered to be uninitialized. When a parameter record is constructed with a labeled field value constructor, it is required that the first labeled field value provide a value for the Routine field. Since only routines within capsules have write-access to capsule header variables, capsule constructors are meaningful only within capsules.

The positional field value form of constructor is an alternate mechanism that can be used to construct record, capsule and parameter-record objects. The first value on the list is assigned to the first field of the object being constructed (which is the Routine field in the parameter record case), the second value to the second field, and so

on for as many values as are provided. Type agreement between values and fields is, of course, required. It is not required that all fields be assigned values. Any not assigned are considered to be uninitialized. An error occurs if too many values are included in the constructor.

The positional field value list can additionally be used to construct array objects. If the <type identifier> in <type desc> names an array type, it determines the type of the elements required in the list and its maximum index range determines the size of the array constructed. The first value in the list is assigned to the first element of the array, the second value to the second element, and so on for as many values as are provided. It is not required that all elements be assigned values. All values in the list must be of the component type required by <type identifier>, unless that type is a pointer type. If the component type required is a pointer type, the values may be of any type which the pointer type may reference. Objects are allocated in the heap to store each value and the corresponding pointers are stored in the appropriate elements of the constructed array.

The array field value list is used to construct an array for which the lower bound is to be determined by the expression bracketed by colons and for which the size is to be

determined by the number of values in the list. The value of the expression must be of the index type of <type identifier> and the upper bound resulting from the constructor must be no greater than the maximum value of the index type.

If the pointer form of <type desc> is used, the value of <constructor expression> is a pointer to the object constructed. Otherwise, the constructed object is the value of the expression.

6.1.2 Pattern constructors

<pattern constructor> ::=

<type identifier> [? <field value list> ?]

A pattern constructor can be used to construct a parameter record for a matchroutine that matches objects of the type determined by <type identifier>. The matchroutine will always be the system-defined routine MatchObject, which can match record, array or capsule types. The value of <pattern constructor> will be a pointer to the constructed matchroutine parameter record.

If <type identifier> names a record type, then MatchObject will have parameters with the same names as each of the fields of the record type. The type of each of these parameters is a pattern matching the type of the correspond-

ing field. (MatchObject cannot be used to match variant records; a user-defined matchroutine must be used instead.) Either a labeled field value list or a positional list may be used to construct the parameter record for MatchObject matching a record. Since all of the parameter fields in the object being constructed are themselves patterns (i.e., pointers to parameter records), all of the expressions in the value list must evaluate to pointers to parameter records or values that can be converted to parameter records according to the following rules:

(1) If the expression is Any, then a parameter record is created for a call to MatchAny, a system-defined matchroutine with no parameters which matches any value.

(2) If the expression evaluates to a value of the type of the field to be matched, a parameter record is created for a call to MatchValue, with the value of the expression as its single parameter (named Value). This system-defined matchroutine matches an object of the same type as and exactly equal to Value, for any type for which equality is defined.

An occurrence of

<matchroutine> <actual parameters>

which looks like a function designator works differently for matchroutines; it is an expression that directs that a parameter record be constructed for the indicated matchroutine with the provided parameters, with a pointer to this parameter record being the value of the expression. Only the system function Match can actually call matchroutines.

<matchroutine> ::= <identifier> |
    <matchroutine reference variable>

Association of values from the value list with fields of the parameter record operates in a manner similar to that for object constructors for both labeled and positional lists. Any parameters to MatchObject for which patterns are not provided are set as parameter records for MatchAny. When MatchObject is applied against a particular object, it will succeed if each of its parameters successfully matches the corresponding field in the object.

If <type identifier> names an array type, only a single value may appear within the brackets. For array types, MatchObject has a single parameter named Component, which is a pattern for the component type of the array being matched. MatchObject will successfully match a particular array when the component pattern successfully matches every element of the array. The conversions listed above for the record case

will be applied to a value provided for Component, if necessary.

If <type identifier> names a capsule type, exported header fields of the capsule are considered as determining a record whose field names are used as parameter names for MatchObject. Construction of the parameter record for MatchObject for a capsule type works like such construction for a record type except that only the labeled field value form of constructor may be used. More complex patterns for capsule objects can be constructed using matchroutines other than MatchObject, defined as part of capsule definitions.

6.2 Process expressions

<process expression> ::=
    NewProcess <coroutine specification> |
    Clone <process ref variable>
<coroutine specification> ::= <coroutine> <actual parameters>
    | <coroutine parameter record variable>
<coroutine> ::= <coroutine identifier> |
    <coroutine reference variable>

Process expressions may be used by overseers and routines within capsules to create processes. When the NewProcess form is used, the coroutine specified and the actual parameters provided are used to create a process, but

no execution of the new process occurs. A reference to the created process is the value of the expression. If the process expression is within a capsule routine, the coroutine must be declared within the same capsule and the process reference which is the value of the expression may be assigned only to a process reference variable included in the header of an instance of the capsule.

When the Clone form is used, a new process is created by making an exact copy of the process referenced by <process ref variable>. The Bind routine (described in Section 8.3) may be used to make changes in the Var and ReadOnly parameter bindings of the newly created process. Simple assignments may be used to change value parameters. (Recall that overseers may use

<process ref variable>.<parameter name>

to reference parameters of processes they control.) If the reference variable is multi-type, it can be used in a clone expression only within a typecase statement that establishes its archetype form. Again, if the expression is within a capsule routine, the value of the expression can only be assigned to a process reference variable in the header of a capsule instance, so that the process can only be manipulated by operations defined within that same capsule.

If a process is created by an overseer, it is attached to that overseer. Only that overseer may invoke, terminate or copy the process and that the process is automatically terminated when the overseer terminates. When a process is created within a capsule routine, it becomes attached to the capsule instance whose header contains the reference variable to which the value of the expression is assigned. It is automatically terminated if that capsule instance is returned to free storage.

The base execution context of the newly created process is the value of CurrentContext when either form of process expression is executed. (An incontext statement can, of course, be used to make CurrentContext distinct from the main context of the creating routine; see Section 7.2.5.) Note that if a cloned process was executing within one or more incontext statements (thus causing a stack of current context values to be created) when it suspended, the corresponding current context stack of the clone process will be the same as the one associated with the cloned process except possibly for the bottom context of the stack. That is, only the base context of the clone may be different, depending on the context in which the clone expression is evaluated.

7. Statements

<statement> ::= <unlabeled statement> ¦
    <label> : <unlabeled statement>

<unlabeled statement> ::= <simple statement> ¦
    <structured statement>

<label> ::= <unsigned integer> ¦ <identifier>

Identifiers are allowed as labels in TELOS, in addition to the unsigned integer labels of PASCAL.

7.1 Simple statements

<simple statement> ::=
    <assignment statement> ¦ <goto statement> ¦
    <routine statement> ¦ <event statement> ¦
    <continue statement> ¦ <empty statement>

7.1.1 Assignment statements

<assignment statement> ::= <variable> := <expression> ¦
    <function identifier> := <expression>

7.1.1.1 Pattern assignment

If the type of the variable on the left-hand side of an assignment is Pattern Matching <type>, the same type conversions that take place within pattern constructors (see Section 6.1.2) will be applied to the value of the right-hand side expression. No conversion will take place if the value

is already a pointer to a parameter record for a matchroutine matching <type>. Such an object can be created by a pattern constructor, by an appropriately valued function or by an occurrence of

<matchroutine> <actual parameters>

7.1.1.2 Routine reference assignment

Assignment to routine reference type variables requires that the right-hand side be a routine reference, not a simple routine name. The system function

Ref(<routine identifier>)

returns as its value a reference to the routine whose name is provided as its parameter.

7.1.2 Goto statements

<goto statement> ::= Goto <label>

TELOS restricts the goto statement so that Goto's out of routines are not allowed. That is, the label referenced in a goto statement must be declared and defined in the same routine in which the statement appears. TELOS is not upward compatible with standard PASCAL in this respect.

7.1.3 Routine statements

<routine statement> ::= <procedure or overseer>

<actual parameters> <event handler clause>

<procedure or overseer> ::= <identifier> |
    <routine reference variable>
<routine reference variable> ::= <variable>
<actual parameters> ::= <empty> | (<parameter list>) |
    (<parameter list> ; <parameter list>)
<parameter list> ::=
    <actual parameter> {, <actual parameter> }
<actual parameter> ::= <expression>

The routine statement is used to cause execution of a procedure or an overseer. The routine to be called may be explicitly named or specified using a routine reference variable. Value and variable parameters operate in TELOS just as in PASCAL. TELOS also includes a ReadOnly parameter mode. The actual parameter corresponding to a ReadOnly parameter may be any expression other than one designating a component of a packed structure (as for Var parameters). ReadOnly parameters act like variable parameters except that, within the routine, assignment to the parameter or any variable referenced through the parameter is forbidden; also, within the routine, the parameter or any variable referenced through it may not be bound to a Var parameter.

A variable number of parameters may be provided to a routine if the last or only formal parameter is a variable-size array ReadOnly parameter. If the array is the

only formal parameter, the entire parameter list is used to construct an array to be used as the single actual parameter. If there are other formal parameters and the array is the last, a semicolon separates the rest of the actual parameters from those used to construct the array. The lower bound of the constructed array is the minimum of the index type of the parameter type; its size is determined by the number of values in the sequence. The types of the values must match the component type of the array formal parameter. If this component type is a pattern type, the type conversions described for pattern constructors (see Section 6.1.2) will be applied to the expressions used to construct the array. If the component type is a pointer type, these expressions may be of any type that may be referenced by pointers of that type. Objects are allocated in the heap to store each value and the corresponding pointers are stored in the appropriate elements of the constructed array. In any of the above cases, the size of the array constructed must be within the bounds allowed by the type of the array formal parameter.

7.1.3.1 Event Handlers

<event handler clause> ::=

    [* <handler item> {; <handler item>} *]

<handler item> ::= <event identifier> : <event handler> |

    <team identifier> <actual parameters>

<event handler> ::= <statement>

During the execution of the routine to which a handler clause is attached, the handlers for the events listed explicitly or within any named teams are active. If one of these events occurs within the routine, control is transferred to the corresponding handler. Within a handler, the parameters of the corresponding event may be referenced simply using the parameter identifiers.

Handlers for suspend or signal events may cause control to return to the place where the event occurred by executing the standard procedure Resume. Resume may not be used in escape event handlers because escape events cause termination of the routine or statement where they occur. If a handler for a signal event does not execute Resume, the routine causing the event is terminated just as if the event had been an escape event. If a handler neither causes another event nor returns control by using Resume, then control passes to the statement following the routine statement to which the handler is attached. Such a control flow is not allowed if the handler is attached to a function call (see Section 3.2). Any handler associated with a function call must either execute a Resume (for a signal event), cause another event to occur, or execute Yield(<expression>) to provide a value for the function evaluation.

7.1.4 Event statements

<event statement> ::=

    <event kind> <event identifier> <actual parameters>

Event statements are executed to cause the occurrence of a user-defined event. Actual parameters must be provided for all parameters specified in the event declaration. They behave like parameters to routines and have the same restrictions placed upon them. Suspend events may only be caused from within Coroutines and Overseers. All suspend events within a coroutine must be handled within a continue statement used to invoke a process created from the coroutine, so handlers for suspend events in coroutines may only appear in continue statements. (This ensures that an overseer maintains control of the processes attached to it.) Handlers for suspend events in overseers may be attached to routine statements that call overseers. (Such handlers are required to be on the overseer call if it occurs in the main process. The main process is created when execution begins, with the main program as its archetype. All routines are executed in the main process if a program does no explicit process creation.)

7.1.5 Continue statements

<continue statement> ::= Continue <process reference>

    Until <handler item> {; <handler item>} End

The continue statement is used to continue (resume) the execution of a suspended process (or start execution of one that has just been created). The continue statement may only appear in an overseer or within a routine in a capsule. If the process invoked by the continue statement has previously returned via an EOP suspend event, such an event occurs again immediately upon reinvocation of the process. If the process causes any suspend event for which a handler is not provided in the continue statement, a RunError escape event occurs. Handlers for escape and signal events may also be included in the handler list of a continue statement.

7.2 Structured statements

<structured statement> ::= <compound statement> ¦
    <conditional statement> ¦ <repetitive statement> ¦
    <with statement> ¦ <incontext statement>

7.2.1 Compound statements

<compound statement> ::=
    Begin <statement> {; <statement>} End ¦
    BeginX <statement> {; <statement>}
        Except <handler item> {; <handler item>} End

The second form of compound statement activates the handlers for the list of events specified in the handler

items during the execution of the statement list and any routines called therein.

7.2.2 Conditional statements

<conditional statement> ::= <if statement> ¦
    <case statement> ¦ <typecase statement>
<if statement> ::= If <expression> Then <statement> ¦
    If <expression> Then <statement> Else <statement>
<case statement> ::= Case <expression> Of
    <case list element> {; <case list element>}
    <otherwise clause> End
<case list element> ::= <empty> ¦
    <case label> {, <case label>} : <statement>
<case label> ::= <constant>
<otherwise clause> ::= <empty> ¦ Otherwise <statement>

The case statement of PASCAL is extended to include an optional Otherwise clause in TELOS.

7.2.2.1 Typecase statements

<typecase statement> ::= Typecase <multi-type variable> Of
    <typecase element> {; <typecase element>}
    <otherwise clause> End
<multi-type variable> ::= <variable>
<typecase element> ::= <empty> ¦
    <type or form identifier> : <statement>

<type or form identifier> ::= <identifier>

The type of the object referenced by the multi-type variable is used to select the statement to be executed by matching with the type or parameter form names used as labels. (Parameter forms rather than types are used if the multi-type variable in question is a routine or process reference.) If none of the labels match the type (or form) of the variable, the otherwise clause is executed. If none is present, an error occurs. If a type-labeled statement is executed, the type of the object referenced by the multi-type variable is known, so the variable may be used like a uniquely typed variable in this syntactic context. No assignment may be made to the multi-type variable within a <typecase element> that could change its type (and could thus result in its value contradicting the label of the statement). Within the <overwise clause> nothing is known about the type of the multi-type variable, and thus no type-dependent use may be made of it, and the only restrictions on assignment to it are those that exist for multi-type variables in general.

7.2.3 Repetitive statements

<repetitive statement> ::= <while statement> |
<for statement> | <repeat statement> | <step statement>

<while statement> ::= While <expression> Do <statement>

<repeat statement> ::=

Repeat <statement> {; <statement>} Until <expression>

<for statement> ::=

For <control variable> := <for list> Do <statement>

<for list> ::= <initial value> To <final value> |

<initial value> Downto <final value>

<control variable> ::= <identifier>

<initial value> ::= <expression>

<final value> ::= <expression>

7.2.3.1 Step statements

<step statement> ::=

ForStep <coroutine specification> <event handler clause>

Do <statement>

Overseers and routines within capsules may create processes and manipulate them as described above. Any routine may use the step statement, which takes advantage of the suspension capability of processes to provide a "generator" feature. The coroutine in the heading is activated as a process. This process is executed until it causes the system suspend event Step. Control then passes to the step statement and the contained <statement> is executed once. A Var parameter is used to communicate the value provided by the generator coroutine for use within <stqtement>. This sequence is repeated until the process finishes execution of

the coroutine body, resulting in the occurrence of the suspend event EOP. If the process causes any other suspend events, an error occurs, resulting in a RunError escape event. The event handler on the coroutine call may handle escape and signal events.

7.2.4 With statements

<with statement> ::=

    With <record variable list> Do <statement> |

    With <multi-type variable list> Do <statement>

<record variable list> ::= <record variable> <tag constants>

    {, <record variable> <tag constants>}

<tag constants> ::= <empty> | (<constant> {, <constant>})

Tag values may be specified for a record variable in a with statement. They are used to indicate that certain variants of the record must be active when the with statement is executed. The first constant must correspond to the value of the first tag, the second to the second tag, and so on. A tag error will occur if this is not the case. In the scope of the With, these tags may not be changed and thus access to fields of the specified variants is guaranteed to be correct without further run-time checking.

<multi-type variable list> ::=

<multi-type variable> : <type or form identifier>

{, <multi-type variable> : <type or form identifier>}

This version of the with statement can be used to make type-dependent references to a multi-type object, much like in a typecase statement. If the type identifier does not agree with the type of the object at run-time, a type error occurs. In the scope of this With, field or parameter names that designate fields or parameters of the object named by <multi-type variable> may be referenced without the variable prefix (like record fields in a standard with statement). In this scope, no assignment may be made that changes the type of the multi-type variable.

7.2.5 Incontext statements

Execution of a TELOS program begins in a context created by the system; this context becomes the root of the context tree. No other contexts are ever created implicitly; new contexts are only created upon explicit command (using CreateContext). Routines are executed in the context that is current for their calling routine when the call takes place. An incontext statement may be used as a means to call routines in a separate context.

Changes to the database and to relative variables may be made with respect to any context in the context tree. Assigning a new value to a relative variable in a particular context has no effect on the value of that variable in any other context, including any current descendants of the changed context. Changing an existing database object has more complex implications that will be discussed in Section 8.6.

<incontext statement> ::=

Incontext <context reference> Do <statement>

The incontext statement sets the current context to the one referenced by <context reference> for the execution of the <statement>. At the end of this execution, the current context is switched back to the starting context, i.e., to the context within which the incontext statement was executed. The current context of a routine can be changed by no other means. In particular, an assignment to CurrentContext is not allowed. The execution context of the program may, however, change when an overseer invokes a process since a process resumes execution in the context in which it was executing when it last suspended.

The "context blocks" imposed by the incontext statement help to keep track of when the implicitly referenced context

is being changed. References to other contexts are possible at any time, of course, but only explicitly through use of context variable notation. The only contexts that can be referenced by a routine are its initial current context (with its descendants) and those which the routine can explicitly name.

The incontext statement is analogous to the with statement in PASCAL. It sets up a run-time scope much like the scope created by the with statement at compile time. An important difference is that the incontext scope is inherited by called routines while the with scope is not.

8. Standard routines

The routines described below are additions to the set of standard routines provided by PASCAL. Many of them take parameters of more than one type. In such cases the parameter type is replaced in these specifications by a meta-variable enclosed by '<' and '>' that describes the parameter.

8.1 Array bounds functions

HiBound (ReadOnly Arr : <array variable>; Dim : Integer)
: <index type of this dimension of Arr>

When provided with an array and an integer dimension as parameters, HiBound returns as its value the upper

bound of the dimension of the array selected by the second parameter. A RunError event occurs if Dim does not select a valid dimension.

LoBound (ReadOnly Arr : <array variable>; Dim : Integer)
: <index type of this dimension of Arr>

When provided with an array and an integer dimension as parameters, LoBound returns as its value the lower bound of the dimension of the array selected by the second parameter. A RunError event occurs if Dim does not select a valid dimension.

8.2 Dynamic allocation and de-allocation procedures

Extensions of PASCAL's New allow the dynamic allocation of arrays with size determined at run-time, as well as the use of New in connection with multi-type pointers. A call to New takes the form:

New (<subpool> <pointer variable> <allocation parameters>)

<subpool> ::= <integer> , ¦ <empty>
<allocation parameters> ::= {, <tag constant>} ¦
{, <bound setting>} ¦ , <type identifier> ¦ <empty>
<bound setting> ::= <type identifier> ¦
<expression> .. <expression>

The subpool parameter to New is used to logically group heap allocations into groups. The Dispose procedure may be used to free, in one operation, all heap allocations in the same subpool. If <subpool> is <empty>, then the object is allocated from subpool 0.

The first case of the allocation parameters is the syntax for allocating records with variants. The tag constants supplied are used to allocate the smallest space necessary for the corresponding variants and these values are assigned to the tags. The second is for allocating arrays. As in the case for declaring array variables, lack of any bounds settings will result in the maximum size being allocated. The third form of allocation parameter is required if <pointer variable> is a multi-type pointer.

CopyStructure (<subpool> <pointer variable>)

: <-> same type as pointer>

CopyStructure copies the object pointed to by the pointer and the entire pointer closure from that object. It returns a pointer to the top object of the newly created structure. <subpool> has the same meaning as for New.

Dispose (<integer>) or Dispose (<pointer variable>)

The first form of Dispose returns to free storage all objects allocated as part of the subpool designated by the <integer>. The second form deallocates the object referenced by the pointer.

DisposeStructure (<pointer variable>)

This procedure is an extension of Dispose, causing the object referenced by the parameter pointer to be de-allocated, along with all objects in the pointer closure from the object. The pointer closure from a given object includes all objects that can be reached by following pointers starting from the given object.

8.3 Parameter record routines

Bind (Var Field : <param record field>; Var V : <variable>)

Bind (Var Parm : <process parameter> ; Var V : <variable>)

A procedure used to set Var and ReadOnly parameter fields in parameter records, and to change the bindings of such parameters for processes created by cloning existing processes. V must be a global variable; a field in a record or capsule header containing the parameter record as a component (for the first form of Bind); a field in a capsule header containing a reference to the process (for the second form); or an object allocated from the heap or part of such an object. It cannot be

a local variable of any routine, or a component of such a variable. V may not be within a variant and, of course, it must be of the type required by the parameter field. Binding to an object allocated in the heap results in a validity check when a parameter record is used to call a routine.

Execute (ReadOnly PR : <procedure or overseer param record>)

A procedure used to cause the execution of a procedure or an overseer specified using a parameter record. The routine is called with the parameter values and bindings provided in the record. A list of event handlers to be active during the execution may be attached to Execute, as on any other procedure call. A parameter record may be reused by Execute any number of times.

Eval (ReadOnly PR : <function param record>)

: <return type of PR.Routine>

A function used to cause the execution of a function specified using a parameter record. The type of the value it returns is determined by the return type of the function referenced in the parameter record. A parameter record may be reused by Eval any number of times.

8.4 Context routines

CreateContext (Parent : ContextRef) : ContextRef

A function which creates a new context that descends from the context referenced by Parent and which returns a reference to the new context as its value. CreateContext does not change CurrentContext.

For example, CreateContext (CurrentContext) creates a descendant of the current context which is initially identical to it.

Release (Cxt : ContextRef)

Cxt and all its descendants are removed from the context tree. A context error escape event results if this procedure is executed when CurrentContext has Cxt or any of its descendants as its value. (Among other things, this restriction prevents Release from ever being legally applied to the root of the context tree.) A context error may later result if an attempt is made to use a context that has been released.

Subcontexts (Parent : ContextRef; Var Child : ContextRef)

A system-defined coroutine that provides as its values references to each of the subcontexts (immediate descendants in the context tree) of the context referenced by Parent, for use in a step statement. That is, it suspends using the system event Step, with Child set

to a different one of the descendants on successive invocations, with no particular order implied.

Finalize (C1, C2 : ContextRef)

Causes the context C1 to become identical to C2 and then to be stripped of all its descendants. C2 must be a descendant of C1. All values established for objects in C2 by way of explicit changes in contexts between C1 and C2 will be included in the final constitution of C1.

8.5 Event routines

All event routines may be used only within handlers.

Resume

A procedure that returns control from an event handler for a signal or suspend event to the location where the event occurred. Resume is not allowed in escape event handlers because escape events cause termination of the statement or routine in which they occur.

Yield (Value : <expression>)

Provides a value for a function call from a handler attached to the call. The type of <expression> must match the return type of the function.

Enable (<team identifier> <actual parameters>)

A procedure that may be used within a handler to add the handlers in the named team to the list of active handlers of which it is a part. An instance of the team is created with the provided parameters.

Disable (Tm : <team identifier>) or Disable

Disable is a procedure that removes handlers from the active list. If it has a parameter, it removes all of the handlers in Tm. If it has no parameter, it deactivates the handler in which it occurs.

8.6 Data base routines

The TELOS data base consists of pointer-linked structures created from record, array and capsule objects. A particular data base object (DBO) may take on different values (of the same type) with respect to different contexts. If a DBO has not been explicitly given a value in a particular context, it will inherit its value for that context from the parent context. A DBO does not necessarily have a value in every context. If no value has been explicitly given in some context and no ancestor of that context has a value for that DBO, then the DBO is said to be undefined for that context. The value of a DBO in any context for which it is defined is that structure that can be reached by following normal pointers from a designated "top object" (a record,

array or capsule header). Only objects of types defined at the outermost level of the program may be stored in the data base, either as a top object or as a component of a DBO. Records containing untagged variants may not be stored in the data base; any attempt to store such a record will result in a RunError.

Defined (ReadOnly Ptr : <DBOP>; Cxt : ContextRef)

: Boolean

A function that returns true if Ptr references a defined value in Cxt.

Store (ReadOnly Obj : <object expression>)

: <DB-> type of stored object>

The parameter Obj must be a record, array or capsule object. Store creates a new data base object (DBO) with the pointer closure from Obj copied into the data base as the value of the new DBO in the current context. The copy of Obj itself is called the top object of the new DBO. The type of Obj determines the type of the DBO, restricting the type of values taken on by the DBO in any other context to be of the same type. The values of any indexed fields of the top object are used to create associative references to the DBO. If a pointer field is indexed, indexing is applied to the object it references. If a matchroutine parameter rec-

ord is contained in or pointed to from an indexed field of the top object, the index part of the routine it references is executed to obtain index values for retrieval by the PatternGet function (defined later in this section). Store returns a data base object pointer (DBOP) pointing to the newly created DBO.

Change (Var Obj : <DBO component> ; NewValue : <expression>)

Change is used to alter values within an existing DBO. (Simple assignment may not be used.) Obj must reference a portion of a DBO value that is neither a working space pointer nor a structured object containing any such pointer (other than Unstored pointers). (Replace is used to change a DBO when its structure is changed.) NewValue must be of the proper type to be assigned as the value of the designated component. If the Change is performed in a context other than the one in which the active DBO value was created, then a new DBO value is created in the current context. This new value of the DBO will be active in the current context and all of its descendants unless a Change or Replace has previously been performed on the DBO in one or more of the descendant contexts. For each such case, a DBOConflict signal event is caused so that the programmer may resolve the conflict. A default handler that results in a RunError event is invoked if the program provides no

other for DBOConflict. The DBO component changed may be specified using a component pointer, as well as through a DBOP.

Replace (Ptr : <DBOP>; ReadOnly Obj : <object expression>)

The type of Obj must be the same as the type of the DBO referenced by Ptr. This DBO is given a value in the current context equivalent to that which would result from Store(Obj). If the DBO has previously been explicitly given a value in the current context, that value is removed from the data base and all component pointers referencing into it are invalidated. DBO values for contexts other than the current context and its descendants are unaffected, with DBOConflict events occurring under the same conditions and with the same results as for Change.

Delete (Var Obj : <data base object>)

The procedure Delete is used to remove changes to data base objects from the data base. If Obj has been given a value explicitly in the current context (by use of Store, Change or Replace), that value is deleted, with the value of the object in the current context reverting to an inherited value. If it is the only existing value for Obj, the object is also removed from the data base, in which case any DBOP's referencing it become

invalid pointers. Delete has no effect if there is no explicit value for Obj in the current context. DBOConflict events can result from Delete under the same conditions and with the same results as for Change.

Find (ReadOnly Ptn : <pattern>) : <DB-> type pattern matches>

Find searches the data base for a DBO whose value in the current context is matched by Ptn. If more than one match is found, a sequencer field in the top object will be used to determine which to return, selecting the DBO with the value ranking highest in the ordering on that field. If no sequencer is specified, or if there is not a unique highest value, the most recently created DBO is selected (from those with the highest value in the latter case). The data base is locked during this operation: no Stores or Changes may be done by matchroutines activated during the match.

FindEach (Ptn : <pattern>; Var Matched : <DBOP variable>)

A coroutine that will return pointers to all DBO's that match Ptn, upon successive invocations. It uses the suspend event Step so that it can be used as a generator by a step statement. The DBOP parameter, Matched, is a Var parameter that receives the return values; it must be a DBOP for the type of object that Ptn matches.

The ordering of the DBOP's returned is determined like the selection process in Find. Between invocations of a process created from FindEach, Stores and Changes may be executed that will change the generation sequence. For example, if an object is stored which Ptn matches, it will be returned by the process. If its sequencer value is greater than or equal to that of any DBO yet to be produced, a DBOP to the new object will be the value of the next invocation of the process.

FindContext (Ptn : <pattern>; Var Cxt : ContextRef)

A coroutine that returns all contexts in the subtree rooted at CurrentContext that contain an object that Ptn matches. It uses the suspend event Step with a different context assigned to Cxt each time it suspends.

PatternGet (ReadOnly Obj : <any type>;
Var WSP : <working space pointer>;
Var DBP : <data base object pointer>)

PatternGet is a coroutine which searches the data dase for objects with top level records containing or pointing to a matchroutine parameter record which matches Obj. The coroutine uses the suspend event Step, with the pointers referencing a new object each time it suspends. The two pointer variables provided as actual

parameters must be pointers to the same type of objects or to the same set of types, if they are multi-type. The types of objects searched in the data base for an acceptable pattern are limited to the types allowed for the parameter pointers. When PatternGet suspends, WSP references a working space copy resulting from the match and DBP references the DBO from which the copy was made. Providing a working space copy as one of the products of a successful match makes possible binding Var parameters of the matchroutine to components of the object of which the parameter record is a component. After matching, the (modified) working space copy is available for inspection of or further operation on these components.

Match (ReadOnly Ptn : <pattern>;
ReadOnly Obj : <data object>) : Boolean

A function that returns true if Ptn matches Obj. This function is the matching component used by the functions that search the data base.

Fetch (ReadOnly Obj : <DBO component>) : <-> type of Obj>

A function which copies a DBO value or part of a DBO from the data base into the working space and returns a pointer to the copy. The entire pointer closure from Obj is copied. Fetch may have a subpool specified, as

with New. It is an optional first parameter, preceeding Obj.

8.7 MatchRoutines

MatchAny

A matchroutine requiring no parameters which matches any object.

MatchValue (Value : <value>)

A matchroutine that matches an object of the same type as and exactly equal to Value (for any type for which equality is declared).

MatchObject (<parameters depend on type being matched>)

A description of MatchObject is given in Section 6.1.2.

8.8 Miscellaneous routines

SameType (P1, P2 : <ptr or ref variable>) : Boolean

A function that tells if two RoutineRef variables, ProcessRef variables or pointer variables reference objects of the same type. (In the case of pointers, both must be of the same kind -- working space, data base object or data base component.) The value of the function is false if either or both of the variables do not reference valid objects. One or both of the variables may be multi-type.

Terminate (Pref : ProcessRef)

A procedure used to de-activate, i.e., to remove from the system, suspended processes attached to either an overseer or capsule instance. It may be used only by overseers and capsule routines.

VarHash (Var V : <variable>) : Integer

A function that provides a unique integer value corresponding to each object local to a routine or in the working space and to each data base object. (VarHash may not be applied to components of data base objects.) This value does not vary with the content of the object. When applied to Var or ReadOnly parameters or to such parameter fields in a parameter record, the value returned is that corresponding to the object bound as the parameter.

Ref (Routine : <routine identifier>) : <routine reference>

Ref returns a reference to the routine named as its parameter.

9. Programs

<program> ::= <program heading> <program block> .

<program heading> ::=

Program <identifier> (<program parameters>)

<program parameters> ::=

<file identifier> {, <file identifier>}

<program block> ::=

<label declaration part> <constant definition part>

<type definition part> <variable declaration part>

<relative declaration part> <event declaration part>

<routine declaration part> <statement part>

9.1 Relative variable declarations

<relative declaration part> ::= <empty> |

Relative <variable declaration>

{; <variable declaration>} ;

The relative variable declaration part in the main program is used to create a set of context-relative variables. When a new context is created, space is allocated for each of the relative variables in a block attached to the context. The initial values for these variables in this context are the values the variables have in the parent context. After that initialization the values of the variables in different contexts are independent; changing a relative variable in any context affects the value in no other context. A relative variable identifier stands for the version of the variable associated with the current context. It may be prefixed by a context reference to obtain a value from another context (see Section 3.2.1). Data base object

pointers may not be relative variables, since the objects they reference are already context-relative. It is to be noted that the dynamic allocation space (heap) is not context-relative and that a relative variable that is a heap pointer may reference the same heap object from its value in more than one context. The initialization of relative variables in a new context does not include making copies of the objects any relative pointers reference.

9.2 Modular Compilation Features

TELOS includes extensions that allow the creation of a global environment, separate compilation of routines using that environment, and additions to the environment without requiring recompilation of existing routines and declarations. Four kinds of modules are recognized to implement these features. The syntax of file identifiers and module headings may be implementation dependent, in order to interface with existing file systems as efficiently as possible.

9.2.1 Declaration modules

Declaration modules are used to create an environment.

<declaration module> ::=

    <declaration heading> <declaration block> .

<declaration heading> ::= Declarations (<file identifier>) ;

<declaration block> ::= <constant definition part>
<type definition part> <variable declaration part>
<relative declaration part> <event declaration part>
<routine heading list>

<routine heading list> ::= <empty> |
<routine heading> {, <routine heading>}

Within the module may be declarations of constants, types, variables and events just as in a standard main program. Among the types, of course, may be capsule specifications. Following these declarations come declarations of the headings of routines that are to be part of the environment. These headings are identical to the headings in normal declarations.

Any identifier defined in a declaration module may be referenced in any other module compiled using the environment created from the declarations. This mechanism allows routines compiled separately to call each other and to use the same global constants, types, variables and events. Compilation of a declaration module creates a description of an environment, which is written into the file specified in the declaration heading.

9.2.2 Routine modules

Routine modules are used to provide the bodies of routines and capsules declared in an environment.

<routine module> ::=
    <routine module head> <routine module block> .
<routine module head> ::= <environment head> Routines
    (<identifier> {, <identifier>});
<environment head> ::= Environment (<file identifier>) ;
<routine module block> ::= <constant definition part>
    <type definition part> <variable declaration part>
    <event declaration part> <routine declaration part>

The file specified in the environment head provides the environment in which the module is to be compiled. The list of identifiers in the heading tells the compiler which of the routines defined in the module are to match declarations in the environment and thus are to be callable from outside of the module. This list may also include capsule names, indicating that the bodies for the named capsules will be found in the routine declaration part, corresponding to capsule specifications in the environment. Routine modules may also include declarations of constants, types, variables, events and local routines. The variables so declared are statically allocated and thus retain their values between calls to routines in the module. A routine module de-

fines a new name scope, so identifiers used in the global environment may be redefined within the module. When a routine declared in the global environment is defined in a routine module, the declaration of its parameters is repeated and the types must match those specified in the environment declaration. (The parameter names need not match those in the declaration.)

9.2.3 Environment extension modules

Environment extension modules are used to make additions to an environment.

<environment extension module> ::=
    <extend heading> <declaration block> .
<extend heading> ::=
    Extend (<old file identifier>, <new file identifier>) ;

Environment extension modules may add any kind of declaration to the environment, but cannot change any existing ones. The environment description from the old environment file is expanded to describe the extended environment and is written as the new environment file.

9.2.4 Main program modules

Main program modules are used to compile the main program body. They look exactly like regular PASCAL programs except

that the heading is prefixed by an environment heading to supply an environment file specification.

<main module> ::=

<environment head> <program heading> <block> .

If any routine modules have been compiled in environments produced by extending earlier environment declarations, the main program module must be compiled in the last of the extended environments. Only a linear succession of environments may be used to compile the modules that make up a program.

10. System events

10.1 Escape events

RunError

This escape event occurs when a run-time error is detected that should terminate execution. After generating an appropriate diagnostic message, the system will cause this event rather than terminating the program. This will allow only a routine or process to be terminated by the error rather than the whole program. If no handler exists for this event, termination of the whole program will occur. The errors that result in

this event include such things as arithmetic faults, out-of-range subscripts, use of invalid pointers or references, use of fields of inactive variants, file errors, etc. No provision is made to communicate the nature of the error to a handler.

10.2 Signal events

There are some system signal events that will be available to handlers only if they are explicitly declared in the event declaration part of the program.

10.2.1 Entry and exit events

Entry to and exit from any routine (including system routines) can be signal events, if they have been explicitly declared as such. The events are declared using the following:

```
Signal Event <routine identifier>$Entry ;
Signal Event  <routine identifier>$Exit ;
```

These events have as parameters those that exist for the routine, which are named in the routine declaration. They are referenced in a handler just as they would be within the routine. In handlers for entry and exit events, the CurrentContext value for the routine can be referenced as EventContext. (Handlers execute in the context that is cur-

rent for the routine in which they are attached.) In a handler for the exit event for a function, the return value may be referenced as ExitResult.

If the routine is a coroutine, the entry event occurs each time a process for which the coroutine is the archetype is invoked and the exit event occurs each time it suspends. The identity of the particular process involved may be referenced in such a handler using the process reference variable EventProcess.

10.2.2 Data base events

Entry and exit events for type-generic system-provided routines and, in particular, for the data base routines need a further specification of the type of object being operated on. Thus a declaration of the exit event for storing an object of some type would appear as:

```
Signal Event  <type>$Store$Exit ;
```

Handlers for such events provide a data base demon capability in TELOS. The type separation of data base events minimizes the number of handler invocations that occur, since handlers need not decide if they are applicable to the particular type of object involved in the data base operation.

10.3 Data base conflict events

This system signal event need not explicitly be declared. If a change is made to a data base object in a context that has descendants, a possible ambiguity in the meaning of that change exists. If the object has previously been changed in one or more of these descendants, there is a question as to how far the new change should propagate. The event mechanism is used to allow the programmer to explicitly decide the question. For each such conflicting change, the following signal event occurs:

```
Signal Event DBOConflict
  (Obj : DB->Any; Change, Conflict : ContextRef) ;
```

A pointer to the DBO and contexts in question are available as parameters. The handler may make a new change to the object in the context referenced by Conflict, it may use Delete to remove the old change altogether, or it may allow the old change to stand. Use of Change, Replace or Delete within the handler may result in a DBOConflict event in some descendant contexts of Conflict. Execution of the handler must end with a call to Resume or a RunError occurs. If no explicit new change to the DBO is made, the old change stands. If no handler exists for this event, the system handler generates a diagnostic message and a RunError escape event occurs.

CHAPTER 4

Language Design Goals and the Design of TELOS

The programming language design process is far from an exact science. It involves trade-offs among design goals and resolution of conflicts between an overall design philosophy and requirements for the intended use of a language. Since TELOS is an extension of PASCAL, its design has certainly been influenced by the "PASCAL philosophy" of programming language design. However, the complexity of TELOS represents a large step away from the sparseness that characterizes PASCAL. The following presentation of design goals is based fundamentally on ideas presented by Wirth [Wir74], Hoare [Hoa73], and Spencer, Tremblay and Sorensen [STS77]. These goals represent an emerging approach to programming language design that has gained a fairly wide, but by no means universal, acceptance.

PASCAL is the most prominent example of a design based on the goals discussed below. TELOS does not satisfy them to the extent that PASCAL does, principally because of a different relative emphasis among the goals. Following the discussion of the design goals is an analysis of the design of TELOS, particularly concerning its divergence from the PASCAL philosophy.

Design Goals

The design goals that constitute this emerging approach include:

(1) simplicity - Language concepts should be as clear and easy to understand as possible.

(2) minimality - A language should include the minimum useful set of opeations for effective, efficient programming in the intended application area.

(3) programming effectiveness - The language should contribute to programming effectiveness by aiding and encouraging good program design.

(4) ease of debugging - The necessity for debugging should be minimized by encouraging good design and the implementation should guarantee security to reduce the scope of bugs.

(5) readability - Programs should be comprehensible to a human reader.

(6) compilability and efficiency - The language constructs should enable fast translation to efficient object code.

(7) security - All restrictions in a language should be enforceable, preferably at compile-time.

The goals enumerated above are interdependent to some considerable extent. As they are developed below, this interdependence will be discussed as it influences the requirements which derive from particular goals.

Simplicity

Simplicity should not be confused with generality or lack of structure in a language. Limitless generality or complete orthogonality (any possible combination of features is allowed, e.g., complex integers) leads only to a language that is difficult to master or impossible to implement completely or efficiently. Rather, simplicity is achieved through the inclusion in a language of only those features which are transparently understandable and which are based on concepts developed from the intended use of the language.

Modularity (not needing to know all of a language to use it) has been suggested as an alternative to simplicity. The design of PL/1 is based on the modularity concept [RR65]. The principal shortcoming of this approach is probably in the area of debugging. It is necessary for a programmer to understand what incorrectly written programs do in order to debug them, a task complicated by the accidental invocation of unknown features. Such problems make modularity an inadequate substitute for simplicity.

ALGOL 68 [VW75] is a language that has been designed to have maximum generality rather than simplicity. Its lack of popular support despite its powerful features is at least partly due to its overwhelming complexity. It also presents implementation difficulties, particularly requiring considerable run-time checking to produce a secure implementation [Tho77]. TELOS includes features of greater complexity than

those in PASCAL, but has not moved toward allowing the generality of ALGOL 68. Its intended use for AI programming has led particularly to the inclusion of features oriented toward handling complex data structures and control schemes.

Minimality

Minimality is a goal closely related to simplicity. Certainly the inclusion of only the minimum useful set of operations necessary in a language will enhance its simplicity. The fewer features there are in a language, the more easily it can be mastered. Minimality can also make another important contribution to a language. It enhances the development of an efficient, reliable compiler and good documentation.

The idea of minimality has not been ignored during the design of TELOS, but it has not had a dominant effect on the resulting language. The usefulness of features for programming effectiveness has been considered to be of greater value than the contribution their absence would make to minimality.

Programming Effectiveness

A programming language exerts a strong influence on how a programmer formulates and solves a problem. It is thus essential that a language provide assistance in program development. One aspect of this assistance is program docu-

mentation. The readability of a language is an important part of making a program largely self-documenting. An even more important aspect is that of program design. The language should allow the program to record design decisions in such a way that they are easily understood and modified as required. This goal is best met by the inclusion of abstraction capabilities, with procedural abstraction being the most common example. The importance of abstraction is that the intent of a programmer need not be obscured by implementation details. Abstraction also serves the purpose of localizing the effect of implementation changes and helps to manage program complexity.

Most languages support procedural abstraction; other kinds, such as data abstraction, are far less common. No matter what features are available, it is important that the language assist in the use of conventions that ensure cooperation among the various parts of a program. Such programming effectiveness is one of the primary concerns in the design of TELOS, motivated by the necessity of being able to handle the complex structures used in AI programs and by the need to manage the complexity of AI programs. Abstraction mechanisms have been a major result of this emphasis.

Ease of Debugging

Many of the other goals contribute to the next goal - that programs written in the language can easily be de-

bugged. An understandable and readable language is obviously important. A language that provides abstraction capabilities and otherwise encourages good design will minimize the necessity of debugging. A secure implementation will restrict the scope of bugs, catching errors where they occur rather than by their frequently obscure side effects.

Other aspects of the design of a language also contribute to ease of debugging, principally by outright elimination of the possibility of some errors. High-level languages in general remove the possibility of branching to arbitrary places in a program or even into data. Strong typing and mandatory declarations give the compiler the information to detect many cases of misuse of variables. PASCAL's restrictions on pointers eliminate many runaway-pointer errors and make dangling pointers inexpensive to detect. Attempts to provide this kind of aid to debugging have influenced a number of features of TELOS. Aspects of overseer control of processes and of the event mechanism are motivated by this goal.

Readability

There has lately been an increasing recognition of the importance of the readability of programs. It is necessary that a program communicate well with a computer, but communication with humans is even more important. Programs must be read to be debugged and extended, frequently by program-

mers other than the original author. A program that is not understandable is almost worthless.

Since external documentation and comments are often incomplete and out of date, the program text becomes the main source of information. This makes the programming language a crucial factor in producing a readable program. While it is important to be able to write programs easily, any version of a program is only written once. Since they are read many times, it is more important that programs be readable than writeable [Hoa73]. APL "one-liners" are an extreme example of the sacrifice of readability to convenience in writing programs using powerful operations.

A programming language contributes to readability if its syntax is reflective of the operations it denotes. Abbreviations and defaults produce obscure programs, not simple ones. Complex operations should not occur without indication in the syntax. Syntactic constructs that look nearly alike should not denote vastly different operations. Finally, it is necessary to be aware that humans and computers do not process complicated syntactic structures by the same mechanisms. The fact that some syntactic construct is unambiguous to a parser is not a guarantee that it is readable by a human. Readability has been a high-priority consideration in the design of TELOS. The dependence of AI program development on experimentation and modification requires that programs be very readable.

Compilability and Efficiency

For a language to be of any practical use at all, it must be possible to implement all of its features. This requires that at least some thought be given to implementation during the design phase. It is further desirable that the compilation task be as simple as possible, in order to minimize the complexity of the compiler. Simplicity of compilation is achieved through minimizing the amount of context necessary to parse constructs successfully. The use of the same operators and punctuation for different purposes should thus be minimized. Finally, fast compilation makes debugging more economical by limiting the cost of recompilation of a program when it is being tested and amended.

It is also important that a language be designed to allow the generation of efficient object code. Wirth goes so far as to recommend [Wir74] that the mere fact that a feature is costly to implement might be a reason to exclude it from a language. It is often stated that as machines become faster and cheaper, efficiency becomes less important. Hoare argues [Hoa73], however, that it will always be better to use a machine more efficiently, no matter how fast it is. He further argues against reliance on optimizing compilers, principally because so few reliable ones exist for languages other than FORTRAN.

The number of features included in TELOS will certainly increase the complexity of a compiler but, in general, the

features are syntactically independent of one another. (This reduces one kind of compiler complexity but introduces more syntactic constructs.) Efficiency has not been ignored, but it has not been an overriding concern. Wirth's suggestion for exclusion is considered to be overly restrictive. One of the most important efficiency considerations in TELOS is that the cost associated with any particular feature not be borne by programs that do not use that feature, i.e., features should be designed (and implemented) so that they do not involve a general overhead cost.

Security

Security is an often-overlooked aspect of language design. It is of little use for a language to include restrictions that are not enforceable. Restrictions that are ignored by compilers because of their costliness are equally useless. Features should be designed so that a compiler can check as much as possible during compilation. Required run-time checks should be kept to a minimum and elaborate run-time checking should be avoided so that checking will not be turned off in production runs because of its cost.

ALGOL 68 is an example of a language that requires costly and elaborate run-time checks to guarantee security [Tho77]. Even PASCAL, which can mostly be checked at compile-time, includes some features that require rather complex run-time checks [FL77]. Since security makes an im-

portant contribution to ease of debugging, the features of TELOS have been designed with considerable attention paid to security consideratons. As much checking as possible is designed to be done at compile-time, e.g., the use of multi-type variables only within a typecase statement.

While these goals generally support one another, there are also some conflicts. The most obvious conflict is between simplicity and minimality on one hand and some aspects of programming effectiveness on the other. For example, the inclusion of extensive features to facilitate abstraction certainly complicates a language. Such trade-offs are the essential challenge of language design.

The Design of TELOS

Since TELOS is intended for use as an AI programming language, it is necessary to consider how the requirements of AI programming interact with the goals discussed above. As outlined in the introductory chapter, AI programming requires capabilities for specifying and manipulating complex environment structures, for specifying and manipulating structurally complex and variable data objects, for experimenting with control strategies, and for building models incrementally. The data base and context mechanisms of TELOS are meant to fulfill the first of these requirements. A number of features, particularly capsules and the standard

routines that operate on pointer closures, are motivated toward the second. Overseers are intended to facilitate control-regime definition capabilities, aided by the event mechanism. Incremental model building is served primarily by the modular compilation capability, but also by such things as multi-type pointers and routine references, as will be seen below.

Another major concern is the problem of managing the complexity of AI programs. This concern becomes even more crucial as AI research progresses to challenge increasingly difficult problems.

In view of these requirements, the design of TELOS emphasizes some of the design goals from the previous chapter at the expense of others. Programming effectiveness receives primary emphasis. Readability is considered important, due to the highly experimental, evolutionary nature of AI programming based on iterative modification and extension of programs. Ease of debugging is important for the same reasons, along with the security of implementation necessary to achieve it. Reasonably fast compilation is important for incremental model building, and run-time efficiency must certainly be considered since it was among the main shortcomings of earlier AI languages that have resulted in their limited use.

Simplicity and minimality, while not ignored, are deliberately de-emphasized, particularly where they would conflict with the goal of programming effectiveness.

Resolution of Design Goal Conflicts

Considering the trade-offs among design goals more explicitly, the foremost is between programming effectiveness and minimality. The abstraction capabilities provided by capsules and overseers are costly in terms of added complexity of the language. An extreme emphasis on minimality would probably have resulted in their exclusion from TELOS. However, the usefulness of abstraction for the management of program complexity is such that the contribution of these features to programming effectiveness makes them well worth their price.

Constructors are an example of emphasis on readability at the expense of minimality. The capability to describe in a pictorial manner (and thus create) structured objects or patterns to match structured objects increases readability (and writeability as well). Again, the feature makes the language no more powerful, since use of New and a series of assignment statements can accomplish the same task. In this case, the contributions to readability and writeability are seen to outweigh the considerations of minimality.

Ease of debugging tends to suffer with the inclusion of such complex control structures as conditional interrupts

(demons) and suspendable processes. The event mechanism, used to implement conditional interrupts (via signal events) and control transfers from processes (via suspend events), allows the inclusion of these control structures but provides a programmer with a means of keeping them under control. Using the event mechanism for these purposes is not as simple as such alternatives as global declaration of demons and more primitive process control transfers. It has the advantages of keeping the scope of event handlers limited (for efficiency) and very well-defined by the program text (for ease of debugging and readability). Once again the goal of simplicity suffers.

The inclusion of a built-in data base mechanism in TELOS is motivated by the goals of efficiency and programming effectiveness. Since a data base mechanism is required by many AI applications, especially given the shift in AI paradigms in the last ten years [Fei77], inclusion of the mechanism which relieves programmers of the necessity of building their own has been deemed important for TELOS. (The tools to do so are available, principally the capsule mechanism and the Match function, though implementing the required associative references with these tools would be no simple task for each programmer who had to do it). The indexing mechanism is made more efficient by allowing compile-time specification of indexing information (in type declarations). Thus, while the data base routines provide

capabilities at a higher level than other operators and routines in TELOS, their inclusion can be seen to enhance programming effectiveness and to allow a more efficient implementation of an important tool for AI programming.

The modular compilation features of TELOS, again a step away from minimality, provide support for incremental model building (programming effectiveness) and help minimize compilation costs. Separation of the implementation of routines and capsules from the specification of their external characteristics provides a way to significantly reduce recompilation costs during debugging or extension of a program. Further, program modularity is improved by allowing a collection of routines to have access to statically allocated data structures not declared as "absolutely" global variables.

CHAPTER 5

Analysis of Individual TELOS Features

Capsules

Since abstraction has been a major focus of the preceding discussion, those features of TELOS that provide such capabilities will be the first considered. The capsule definition mechanism is provided to enable the creation of data abstractions. That is, a data type and the operators that may manipulate it are defined as a capsule type. (Type parameterization in effect allows the definition of a set of such types in a single specification.) Capsules are related to CLU clusters [Lis76] and ALPHARD forms [WLS76], but are different in a number of details. The separation of a capsule definition into two parts emphasizes the nature of the abstraction. Such separation is similarly pronounced in ALPHARD and LIS [IF76]. The capsule specification part defines the external properties of a capsule type. This definition strictly delineates the interaction between objects of the capsule type and other parts of the program. A routine using such objects need be concerned with no other details of the capsule, nor need a programmer writing such a routine - or reading it.

Capsule operations are named by

<capsule type identifier> $ <capsule routine identifier>

For any capsule operation that requires an instance of the capsule as a parameter (as all probably will, since there is no automatic association of routines with particular instances, as in SIMULA 67 [DMN70]), the explicit presence of the capsule type name is redundant. However, it does improve readability by making clear the type of operand (or operands) required. It also simplifies compilation of the routine call, since the particular routine being called can always be recognized without reference to the actual parameters.

Capsule routine operands are required to be explicit, among other reasons, in order to eliminate any syntactic bias toward unary operators on capsule objects. SIMULA 67 class attributes which happen to be procedures or functions have access to an implicit parameter, an instance of the class of which they are attributes. This instance is provided by the routine name syntax. If such a routine is to operate also on a second instance (for example, to perform a comparison), it must be provided as a normal parameter. A similar TELOS capsule routine will have both objects provided as explicit parameters. This also results in a clear distinction in TELOS between capsule routines, which are associated with a capsule type, and exported capsule header variables, which are referenced as part of a particular instance of a capsule type. There is thus no "uniform refer-

ence" problem [GM74] in TELOS, since capsule routines and header variables are distinct entities.

Among the items that may be exported from capsules are types. When the type is structured so as to have nameable components (or operations, if the type is a capsule), these names are not automatically exported. Only those which appear within a parenthesized "exports" list following the type or subordinate capsule name may be used outside of the capsule body. The list capsule exemplified in Appendix A illustrates the utility of exporting some but not all details of a type defined within a capsule. This exporting mechanism borrows from ideas in EUCLID [LHLMP77]. By exporting only the information field of the record type cell, routines outside of the capsule are allowed direct reference to items of the list. This enables efficient and convenient referencing and changing of list item content while reserving for the appropriate capsule routines the altering of list structure. These capsule routines need (and have) access to the structure-defining pointer fields contained in cell, which are not exported.

The power of the capsule mechanism is enhanced further by the ability to define capsules within capsules. Advantages of partial exportation like those just discussed are available for capsule type definitions as well as for simpler type definitions, e.g., record definitions. It is also possible for any number of instances of a subcapsule to ref-

erence a single instance of an outer capsule. This allows sharing of data between the subcapsule instances, providing a capability similar to common data structures of ALPHARD forms [WLS76], but in a more powerful manner since any number of distinct instances of the outer capsule, and thus of the shared data, may exist. Subcapsule definitions also provide a means of defining capsule operators that may operate on more than one capsule type. However, this capability as provided in TELOS does not have the symmetry that might be desired, since the definition of one of the capsule types must be subordinate to the other.

A minor syntactic extension to PASCAL provides for convenient referencing of capsule components, such as a cell from the list capsule example. The consideration of a <function designator> as a <variable> rather than as a <factor> allows a function call to start a variable qualification as an identifier can. Thus, if a function returns a pointer, it is syntactically legal to immediately dereference the pointer; that is, <function designator>-> is a legal <referenced variable>. Since TELOS allows any assignable type to be returned as a function value, other qualifications of the result of a function are also possible. If a function returns a record, for example, a component reference to a field of that record is allowed using <function designator>.<field identifier> .

Use of assignment statement syntax in connection with capsule variables and capsule components enhances readability by making data flow easier to follow. For capsule variables this is accomplished by compiling the assignment as a call to a procedure in the capsule named Assign. Partial exporting of capsule component types, as discussed above for the list example, provides direct assignment to capsule components.

The construction of an actual parameter array from an indefinite number of parameter values (separated from fixed parameters by a semicolon) is particularly intended for use with capsules. It can provide a variable number of parameters to a capsule constructor function, such as the one in the list capsule example. Together with the transformation of values available to construct arrays of pointers or arrays of patterns, parameter array construction enables the specification of powerful capsule pattern operations. Since capsules may include matchroutines that interpret such parameter arrays in any way a programmer wishes, it is easy to specify such possibilities as SNOBOL-type pattern matching [GPP71] that includes backtracking over candidate structures.

The capsule body provides an implementation for the routines defined in the specification part. This separates implementation details from the definition of the external properties of the capsule and also enables effective use of

the modular compilation features of TELOS in association with capsules. A capsule implementation may be altered and recompiled without requiring recompilation of any routines that use the capsule.

Thus the capsule mechanism contributes to programming effectiveness by providing an abstraction capability, by separating interface specifications from implementation details, and by localizing the effect of program changes. The type parameterization of capsules contributes to this localization by allowing a single definition for type generators, like lists and stacks, whose properties are independent of their component object types (which must be specified before a specific type is fully defined).

Coroutines, Processes and Overseers

Many of the problem-solving strategies used in AI programming, particularly those which involve the creation of alternative states of the data base through use of contexts, are most easily implemented when the language provides the ability to suspend a computation and possibly resume it later. To provide this capability, the mechanisms to create processes from coroutines and suspend them using suspend events are included in TELOS. (SAIL [Van73] and QLISP [Wil76] are examples of other languages designed for AI programming which provide a somewhat different but similarly motivated process capability.) Coroutines are distinguished

from procedures mainly to promote ease of debugging. Coroutine activations are referred to as processes to denote that they may exist in a suspended state, awaiting later reactivation. There are no capabilities for parallel execution of processes, but pseudo-parallel execution may be programmed as an overseer. By restricting the ability to cause suspend events only to coroutines (and routines defined within them), the effect of a suspend event becomes obvious from the program text - a process for which the coroutine is the archetype is suspended by the event. If any procedure could be executed as a process and thus any procedure could cause a suspension, it would not be possible to know from the text if a procedure causing a suspend event had itself been used to create a process or if it had been called by the process archetype procedure (perhaps through a number of intermediate routines). Since TELOS does not allow procedures to be executed as processes nor coroutines to directly call other coroutines, this source of confusion is eliminated.

The overseer is introduced to encapsulate the details of process creation and manipulation used to implement a particular problem-solving strategy. Overseers are intended to provide control abstractions for process control regimes in much the same way that capsules provide data abstractions. The creation of processes is a capability reserved for overseers (except for the case of processes at-

tached to capsules, which have a different use, as described below). Using processes and the data context mechanism, an overseer may implement a backtracking mechanism like that of PLANNER [Hew72], a best-first search strategy (see Appendix B) or any control regime tailored to a particular problem. The overseer truly does encapsulate the details of such a regime. It exerts complete control over the processes it creates, since they may only pass control to one another by going through the overseer. Overseers are allowed to cause suspend events, along with coroutines, for the purpose of constructing multi-level control structures through recursive use of overseers. That is, a process being controlled by a given overseer may invoke an overseer (perhaps the given one) in its problem-solving attempts, or an overseer may call itself recursively.

The second use of processes is to obtain a generator capability. Use of the Step suspend event by a coroutine allows its use in the special iteration statement provided for this purpose. The point of this capability is again abstraction. By putting in a coroutine the mechanism for determining the next value on which to iterate, it is separated from the body of the iteration. Similar capabilities are provided in ALPHARD and CLU [see SWL77 and LSAS77]. Readability is enhanced by separating these different aspects of an iterative algorithm, and programming effectiveness is served by localizing the details of constructing or finding

the "next" operand object apart from the details of the operation performed on each object produced. Allowing any routine to use the step statement does not violate the restrictions necessary to keep the use of processes as easy to debug as possible, since suspend events other than Step and EOP in a coroutine cause an error event if the coroutine is used in a step statement.

The final use of processes is their attachment to instances of capsule types for purposes of generating successive states of a data structure. For this purpose, any routine within a capsule may create a process, but it may only be attached to an instance of the capsule type and only routines within the capsule may manipulate it. (A similar capability is available in SIMULA 67, through use of DETACH in a class body [see DMN70].) Since the idea of successive states of a data structure is analogous to the idea of successive states of a process, allowing a process to be attached to a capsule enables an intuitively natural and efficient implementation of the generation of states of the capsule data structure, particularly when the next state may depend in a complex way on how previous states have been generated. This will provide a more readable and efficient mechanism for determining how the next state is to be generated than requiring that an explicit determination be made from values in the structure.

Events

The event mechanism is another feature that plays multiple important roles. The use of suspend events to effect process suspension can improve the readability of process-overseer interaction through the use of event parameters for communication. In conjunction with the attachment of a process to a particular overseer or capsule instance, the use of suspend events and their parameters for process control and communication will help to make programs using processes more understandable and easier to debug. If the alternate approach had been taken, providing only a simple suspend primitive (as in SIMULA 67 or 2.PAK [Mel74], for example), communication would have to be by less distinct means. Requiring a list of event handlers as part of the statement that invokes a process makes explicit the expected suspension states.

Signal events are included in TELOS to implement conditional interrupt (demon) mechanisms. This is a capability which, like process specification and control, is hard to use efficiently and understandably if sufficient controls are not provided by the programming language. Allowing event handlers to be activated only for an explicit scope contributes to such controls. Even allowing some dynamic activation and deactivation of handlers (with Enable and Disable) can be accomplished without causing unmanageable complications (such as can result from PL/1 ON Conditions

[Mac77]), since this can be done only within handlers, restricting the scope of dynamically activated handlers to that of the handler in which they are activated, and thus making this scope just as visible as that of non-dynamic handlers. Allowing a group of related handlers to be bound together as a parameterizable team can be used to make handler associations explicit and to eliminate tedious repetition of similar handler definitions.

Entry and exit signal events are an important part of the features of TELOS that provide the demon capability. This capability is oriented toward triggering event handlers (demons) on actions defined in terms of routine calls and exits. The TELOS design represents a compromise between the full generality of watching for actions like assignment to particular variables (which is prohibitively expensive) and the minimality which would require a programmer to code explicit routine calls at every point where a monitor invocation might be needed (which brings back the involvement with details that demons relieve). The design of the event mechanism has been much influenced by the desire to avoid any significant expense for programs that do not use events. Checking for the presence of an enabled event handler can be added to routine entry and exit code without a significant increase in overhead. More specialized events can be defined by a programmer, but an explicit use of an event statement is necessary to make them occur.

Given the existence of the event mechanisms for suspend and signal events, adding escape events for program-detected error conditions (or other non-standard exit conditions) provides a considerable improvement in error handling over PASCAL (among other things, providing user control of this error handling) while adding little complexity to the language. Escape events are distinctly different from signal events because there can be no resumption of the computation that causes an escape event. As a result, no allowance need be made for such a possibility. The informational nature of signal events presents a very different case, where the possibility of resumption is assumed and there even need not be any handler active for an event. If no handler is active, resumption is automatic (i.e., the event occurrence has no effect).

Since handlers can often use more information than simply that some event has occurred, events may have parameters in a manner completely equivalent to routine parameters. There is particular use for Var parameters in signal event handlers that return control to the interrupted computation. The parameters of a suspend event provide for the explicit communication "channels" between an overseer and a process it is controlling as described above. Finally, the inclusion of the routine parameters in exit events allows for a convenient implementation of exit functions [Tei75]. This

can be done by attaching a handler for a routine exit event directly to a call of the routine.

Type specifications are required on data base routine entry and exit events to take advantage of the natural type segmentation of the data base. If such specification were not required on data base routine events, it would be necessary for handlers meant for maintaining data base consistency and integrity to check their applicability, perhaps by pattern match, every time a data base routine was called. With type segmentation, it is necessary to invoke only those handlers that operate on the type of object stored or changed by the data base routine.

The new form of <compound statement> included in TELOS (BeginX <statement list> Except <handler list> End) for the attachment of a set of event handlers has utility in implementing control structures such as structured exits or a situation case statement [Zah74]. The particular syntax used, including use of BeginX in place of Begin, is for the purpose of readability. It is necessary to make it clear that there are some handlers active for the statement, but putting them before the statement list for which they are active would misleadingly place the main emphasis on the handlers. The resolution is to provide BeginX to indicate that the compound statement is of this special type and to separate the handlers from the statements with Except, indi-

cating that occurrence of one of the events is a non-standard flow of control.

In their various uses, events provide improved readability, ease of debugging and programming effectiveness. The work of Goodenough [Goo75] was a major influence on this part of TELOS.

The Data Base

Among the principal reasons for the inclusion of the data base mechanism in TELOS are its general utility for AI programming and the efficiency gains that can come from its implementation as part of the language. This latter point is particularly important given the high degree of interaction among the data base constructs, contexts and matchroutines. The context-dependent values of data base objects can be implemented efficiently [HLTZ77], but the implementation must be done in conjunction with the implementation of the data base. It would not be possible to efficiently add the context mechanism of TELOS on top of a previously existing, non-context-structured data base representation. The indexing specifications that are part of type definitions allow compile-time processing of information that would otherwise have to be handled at run-time. The indexing part of a matchroutine is, of course, completely directed toward providing more efficient associative re-

trieval by limiting the number of objects that must be fully examined by the matchroutine.

The fact that a data base object may take on different values for different contexts is a departure from previous AI languages, in which an object was either present or absent (or perhaps unspecified) in any particular context (see particularly QLISP [Wil76] and CONNIVER [MS72]). One point of the TELOS version of data base objects is to minimize the cost of updating indexing information when only a limited part of an object is changed (using the Change procedure rather than a simple assignment). It is less expensive to specify that only some part of the object is to be changed than to first delete the object and then to replace it with a slightly different one.

Another innovation of TELOS is the capability to reference objects in the data base using special pointers. (It is necessary that such a capability exist to make multi-valued objects usable.) Having available a non-associative reference into the data base allows explicit cross-referencing among data base objects, removes the necessity of reaching data base objects only via expensive associative retrieval, and removes the requirement of always copying objects when they are found in the data base by an associative search function.

Data base component pointers are available in order to provide efficient reference into data base objects. Unlike

data base object pointers, which are context relative, a component pointer references a component of a particular DBO value (that is, the value the DBO has for some specific context). Thus component pointers are much like working space pointers, differing in that they may not be used to alter the objects they reference by direct assignment. (The procedure Change must be used to make changes in data base objects, since such changes might require updating the indexing information used for associative referencing.)

The ReadOnly parameter mode is necessary for passing objects referenced by data base component pointers as parameters. That is, components referenced by DBCP's cannot be passed as Var parameters, since this might result in an attempt to change a data base object illegally. In fact, ReadOnly parameters have a general utility. This parameter mode combines the efficiency of Var parameters (only an address need be provided to a called routine) with the security of value parameters (the routine may make no changes to the object provided as an actual parameter).

The patterns used for associative referencing are based on matchroutines for two principal reasons. The first is that such a mechanism makes it unnecessary for the language to include a wide variety of pattern matching primitives, keeping the language simpler. Only the single function Match is provided, which applies matchroutines to candidate objects. Three type-generic matchroutines (MatchValue,

MatchAny and MatchObject) are available for use by Match, along with programmer-defined matchroutines. The second reason is that dynamically created structures, such as can be defined by capsules, are best matched procedurally rather than by picturing (although facilities are provided to enable a programmer to specify in capsule definitions quite useful kinds of pattern "picturing"). The interactions of the index part of a matchroutine with the associative retrieval routines also gives the capability of procedural specification of search restrictions (in contrast to only statically supplied information being available with a completely non-procedural pattern picturing mechanism).

Contexts

An important point about the context mechanism in TELOS is that no new contexts are created implicitly and thus the current context of a routine never changes without an explicit programmer request. The automatic context creation and switching that appears in PLANNER and FUZZY can result in unnecessary overhead and may require effort on the part of a programmer to avoid the results of the built-in mechanisms when some other result is desired. Elimination of automatic context creation enables a programmer to minimize the total number of contexts created during a program execution. This is a basic assumption of the efficient context

implementation described in [HLTZ77], designed for use in the implementation of TELOS.

Elimination of automatic context switching is a result of the elimination of such things as automatic context creation upon routine entry. If no contexts are explicitly created in a program, the entire execution of the program will take place in the original context that exists when execution begins. (The initial value of CurrentContext references this context.) The incontext statement is used to switch execution of a routine to a new context. The only other way the execution context can change is when a process is invoked, since the execution context is part of the information saved when a process suspends. A directly called routine (a procedure, function or overseer) always begins execution in the context in which the calling routine was executing.

Protection is another important aspect of the context mechanism. A routine may reference only contexts which it can name and no standard routines are provided to find names of any ancestors of a context. Thus if a routine has access to no context reference variables or parameters, it may name only its initial CurrentContext and any descendants of that context. This protection allows an overseer complete control over the interaction between processes it controls, as far as the data base and relative variables are concerned. If each process is created in a separate context and is giv-

en no information about other contexts, relative isolation is possible. Conversely, if a set of processes execute in a shared context or can all name some commonly known context, interaction via the data base may freely occur. In general, a routine may protect its CurrentContext from a routine it calls by creating a descendant context and calling the routine in this descendant (using InContext).

The incontext statement makes context changes visible from the program text. Execution is switched to a different context for a clearly delimited scope. This makes a program significantly more readable than is the case if context switching is unrelated to program structure (as can be the case in CONNIVER and QLISP).

The execution context provides an implicit context in which all relative variables and data base object pointers are evaluated and it is also the context in which associative retrieval takes place. It is possible to cause relative variables or DBOP's to be evaluated in some other context without using an incontext statement. Such evaluation can be specified using

<context reference>.<variable>

where <variable> is a relative variable or a DBOP. This form of reference is useful when only a few references to a context are needed, thus avoiding the (small) overhead involved in a context change. It also allows references to more than one context within a single statement. Having the

context reference prefix available to direct evaluation to a specific context guarantees that the incontext mechanism results in no loss of generality.

Relative variables are available for several purposes. They provide mainly for use of the context mechanism for applications where associative referencing may not be needed or desired (for example, because of its cost). They are also useful in conjunction with the use of the data base as a world model. In such a case, contexts may represent tentative modifications of the model and the relative variables are useful for recording descriptions of the modifications. It is for this reason that relative variable values for different contexts are independent of each other.

One important context routine is Finalize. It is used to cause some tentative modifications of the data base, represented by some particular node in the context tree, to be made "final" in some ancestor of that node, which then has all of its descendant contexts deleted (to avoid possible race conditions). This is, of course, a relatively complex and specialized operation, but one which is difficult for a programmer to implement and which is required by the "tentative modification" interpretation of contexts. The inclusion of a context-relative data base and the provision of such operations as Finalize in TELOS are crucial to use of the language for AI applications that make use of large, dynamic knowledge bases.

Arrays

The extensions to PASCAL array type definitions in TELOS are aimed mainly toward programming effectiveness considerations. They are influenced by a number of proposals for such extensions, but mainly by [Pok76]. The mechanisms that allow differently sized arrays with the same component type to be used as actual parameters corresponding to a single formal parameter of a routine are meant to remedy one of the main shortcomings of PASCAL [see Hab74]. Allowing the creation of dynamically sized arrays with New provides a useful capability without affecting the static-size property of variables in PASCAL (a property highly conducive to program run-time efficiency). This is an example of a clean extension of a PASCAL feature, not adding to the cost of programs that do not use dynamically sized arrays. The heap is an obvious place from which to allocate such dynamic objects. The ability to create such dynamic arrays is important in AI programming, with its heavy use of dynamically varying structures. Recompilation whenever a differently-sized array is needed is not an acceptable alternative.

Multi-type Pointers

Multi-type pointers have a purpose similar to that of the array extensions. Multi-type pointers and constructors of arrays of multi-type pointers provide the flexibility to

easily specify and build heterogeneous structures, avoiding the clumsiness (and syntactical messiness) of variants for this simple case where each variant would have only a single pointer field. Allowing pointers to reference objects of any type further provides a capability not available using variants. The restrictions placed on the use of these pointers, requiring that the objects they reference only be accessed within a typecase statement, maintain the advantages of PASCAL's strongly typed pointers. The desired flexibility is attained at a small cost in additional complexity of the language. Pointers that are unrestricted in what types they may reference can be an aid in evolutionary program development. Use of multi-type pointers and typecase statements may result in more readable programs than use of variants for the same purpose.

Parameter Forms

Parameter forms and the references based on them play an important part in TELOS. Routine references are, of course, crucial to parameter records. Use of routine references as the means for passing routines as parameters is one way to remedy a significant security problem in PASCAL, i.e., the lack of specification for formal procedures and functions. Having process references based on parameter forms allows overseers to reference the parameters of the processes they control, while maintaining a type security

that can be checked at compile time. Parameter forms are also available to make capsule parameter specifications more readable. Thus parameter forms, by the classifications they provide, contribute to several TELOS design goals.

Parameter Records

Parameter records are a feature with several important uses in TELOS. Matchroutine parameter records are the basis of the "pattern" mechanism, essential for associative retrieval from the data base, but generally available for any use through the Match function. Allowing patterns to be represented as programmer-accessible data structures allows patterns to be incrementally constructed as an alternative to the picturing mechanism provided by pattern constructors, as well as to be examined and modified dynamically.

A second possible use of parameter records is for the dynamic construction of "programs." For instance, a program may "learn" a task as a structure of routine calls or a planning program may create plans that are similarly constituted. This is an important capability if TELOS is to provide a viable alternative to LISP-based languages for AI work. Since programs and data are not interchangeable in PASCAL, as they are in LISP, parameter records provide a well-defined mechanism for program construction by programs, where basic operations are routine calls.

The final principal anticipated use of parameter records is to enable pattern-directed routine invocation (or other kinds of generalized routine invocations). The PatternGet data base routine can be used to find objects containing references to routines and to set up a parameter record as part of the match. The parameter record can then be used to invoke the routine. The separation of finding the routine reference from invoking the routine is a manifestation of the general attempt in TELOS design to provide more control over such powerful features than has been provided in previous AI languages (yet no duplicated or otherwise extra work has to be done because of this separation).

The requirement that Var and ReadOnly parameter record fields be set through use of the procedure Bind serves to emphasize the distinction between such fields and value parameter fields. Value parameter fields may have values asigned to them like standard fields, while Var and ReadOnly parameter fields are assigned direct references to objects as their values by Bind. This distinction is quite necessary for purposes of efficiency and communication of results.

Parameter records are thus an aid to programming effectiveness on several counts. Their use in this last case to provide a generalization of a feature widely used in AI programming is an example of providing the needed effectiveness

while at the same time giving the programmer more control. This can lead to greater program efficiency and ease of debugging.

Constructors

Constructors for structured objects and pattern objects (matchroutine parameter records) are a feature that enhances the readability and writeability of programs that deal with such structures as their primitive objects. AI programs typically do so; thus this pictorial representation is important in TELOS, particularly for pattern objects. The natural usefulness of constructors can be seen in the use of a similar feature by Wirth in [Wir76]. TELOS constructors go somewhat further, providing some conversions in the construction of pattern objects and arrays of pointers. These conversions are included because their absence would make construction of these objects so cumbersome syntactically as to negate the benefits of having the constructors in the first place.

Modular Compilation

The modular compilation features of TELOS are essential in making the language routine oriented rather than having it deal only with whole programs. Influences on this part of TELOS include LIS [IF76] and SUE [CH73]. Particularly in applications where routines are used as a means of knowledge

representation, it is important to be able to add (or modify) one or a few routines without requiring recompilation of an entire program. In conjunction with the capsule definition feature, modular compilation allows separation of the external characteristics of data types and routines from their underlying implementations. This makes experimentation with various implementations convenient. Modular compilation thus can provide benefits in terms of both compilation efficiency and programming effectiveness.

Pointer Closures

A number of standard routines in TELOS operate on the entire pointer closure from a designated object. (Recall that the pointer closure from an object includes all objects which can be reached by following working space pointers, starting from the object.) These routines include Store, Fetch, CopyStructure and DisposeStructure. They operate in this manner in order to support the use of complex, linked structures as basic operand objects. Their orientation is particularly useful for working with structures originating from capsule header instances.

There are a number of means available for refining the effect of routines that operate on pointer closures. The Unstored attribute, which may be associated with a working space pointer field in a record type declaration, is available to limit the extent of a structure entered into the data

base by Store. (This attribute indicates that neither the pointer nor the object it points to is to be included in the data base structure.)

The division of a working space object into several different data base objects is also possible. This can be accomplished if pointers that connect the parts of an object that are to be separate DBO's are Unstored, and "paired" data base object pointers are included (in the definition of the record being used in the structure) corresponding to each of the connecting pointers. A programmer-supplied routine can traverse the structure, using Store to create a DBO for each component object in the working space that is to be stored as an independent DBO and assigning the resulting DBOP to the DBOP field paired with the pointer to the working space component object. Such a routine is particularly appropriate for inclusion within a capsule. A single Fetch will copy only a single DBO into the working space, so dividing a structure like this when it is entered into the data base limits the extent of a later retrieval as well.

Syntax

Several new kinds of brackets ([! !], [? ?] and [* *]) are included in the syntax of TELOS. While adding some syntactic complexity to the language, they are important contributions to readability of the constructs they delimit. To specify constructors and event handler lists using only

the parentheses or square brackets also used for other purposes would require a reader to distinguish among the several different bracket meanings on the basis of semantic context. The special brackets also simplify compilation of the new constructs.

CHAPTER 6

TELOS Subsets

Although the original motivation of the design of TELOS was to create a language for use in artificial intelligence programming, the resulting language has features that are useful for a wider range of programming tasks. For some of these tasks, not all TELOS features are needed, as indeed is the case for certain specialized kinds of AI programming, so it is of interest to examine the utility of certain isolatable subsets of the features of TELOS. Each of the subsets identified below is designed to a be a coherent extension to PASCAL and, except in the case of the first, to some or all of the previous subsets.

The first extension level is a group of minor extensions to PASCAL. It includes modular compilation, addition of an Otherwise clause to the case statement, subpool extensions to the dynamic allocation and de-allocation procedures, array extensions for generalized array parameters and some run-time determination of array sizes, data object constructors, and generalized function returns (any assignable type may be returned and <function designator> considered a <variable>).

These minor extensions, especially the case statement and array extensions, improve several features of PASCAL that have been subjected to frequent criticism [for example,

see Hab73]. The modular compilation features alleviate the inconvenience of having to compile all parts of a program in a single compiler operation, but retain the advantages of compile-time checking. Constructors and generalized function returns improve the convenience of using the language and the readability of programs written in it. The subpool extensions to the dynamic allocation mechanism simplify storage reclamation. (Automatic garbage collection is a possible alternative to subpools. There are, however, a number of problems with implementing automatic garbage collection given the features included in PASCAL [see FL77].)

The second extension level, including and significantly extending the first, centers around several major extensions to PASCAL. This level includes capsules, escape and signal events (and teams), multi-type pointers and typecase statements, parameter forms and routine references, and parameter records. The main thrust of the first extension level was extension and improvement of existing PASCAL features; the thrust of the second level is to provide a number of new kinds of constructs.

The inclusion of the event mechanism provides for a well-structured exception handling capability, a serious lack in PASCAL and, for that matter, in other languages [Goo75]. The by now well-established utility of a data abstraction mechanism [see LZ74 and WLS76 and previous discussion] justifies the inclusion of capsules in this subset.

The rest of the extensions in this group are included for the general flexibility they provide and because together they constitute the larger part of the type creation mechanisms in full TELOS and thus are needed as a base for the later extension levels. (Given the desire to keep each extension level distinct, all of these type definition capabilities must be included in this level.) Routine references are principally useful at this level to correct the shortcomings of PASCAL with respect to the passing of routines as parameters.

The third extension level includes two alternative sets of features. Both include PASCAL and all of the first two extension levels. The first set at the third level is comprised of control abstraction extensons, i.e., overseers and coroutines, process reference variables, suspend events, process creation (NewProcess and Clone) and control (Continue and Terminate), Step events and ForStep, and process attachment to capsules. These process features provide capabilities which enable easy and convenient implementation of discrete simulation models, multiprocess coroutine regimes, and pseudo-parallel processing.

The second set at the third level is comprised of data base extensions, i.e., pattern constructors, matchroutines, the associatively referencable data base without contexts, all data base routines except FindContext, data base object and component pointers, and ReadOnly parameters. The basic

data base mechanism and the featues that support it (in particular, the pattern mechanisms needed for associative referencing) will support the numerous applications that depend on a conveniently definable and associatively referencable data base, but one which does not include the complications of context-structuring and interaction with processes. Contexts are among the more AI-specific features of TELOS and do add to the cost of the data base implementation. Since the control abstractions and data base extensions are independent of one another and thus are presented as parallel alternatives at this extension level, it would be possible to combine them without the inclusion of any additonal features.

The fourth and final level of extension, providing the full set of TELOS features, does bring together the control abstraction and data base extensions and adds to them the context features. These include the routines that create and operate on different data base versions related to each other in a tree structure, context-relative variables, context reference variables and CurrentContext, the incontext statement, FindContext, and DBOConflict events. The data base is more expensive to use with contexts, but provides the support necessary for applications which need to proceed by making tentative, reversible changes to the data base with alternative changes possibly existing simultaneously

(e.g., AI applications where the data base represents some kind of cognitive model being used for planning).

This segmentation of TELOS into subsets serves several purposes. It partitions the features, suggesting stepwise implementation and suggesting subset implementations including only those features needed for a particular application area. The segmentation also aids in understanding where the numerous features fit into the total design and how they relate to each other. The result is an improved intuition of the point and power of the whole language.

Summary of Extension Levels

Level 1: Minor extensions to PASCAL

Modular compilation, Otherwise, subpool allocation and de-allocation, array extensions, object constructors, generalized function returns.

Level 2: Major extensions to PASCAL

Escape and signal events, teams and BeginX compound statement; capsules; multi-type pointers, typecase statement and extended With statement; parameter forms and routine references; parameter records.

Level 3A: Control abstraction extensions

Overseers and coroutines, process reference variables, suspend events, process creation (NewProcess and Clone) and control (Continue and Terminate), Step events and ForStep, processes attached to capsules.

Level 3B: Data base extensions

Pattern constructors, matchroutines, data base without contexts, all data base routines except FindContext, data base object and component pointers, ReadOnly parameters.

Level 4: Context extensions (full TELOS)

Contexts (i.e., tree-structuring of different data base versions), context routines, FindContext, context-relative variables, context reference variables and CurrentContext, incontext statement, DBOConflict events.

Appendix A

A Capsule Example

The list capsule below illustrates some of the data abstraction facilities of TELOS. It defines doubly linked lists, lists identified by pointers to their headers. The control abstraction example in Appendix B illustrates use of lists so defined.

```
list = Capsule (comp_type [ := ])
  Exports
    cell (item), head, list_ptr, cell_ptr,
    comp_array, header_delete;

  Type
    list_ptr = ->list;  cell_ptr = ->cell;
    comp_array = array [1..Maxint] of comp_type;
    cell_kinds = (header, component);
    cell = Record
      pred, succ : cell_ptr;
      Case kind : cell_kinds of
        {header : empty}
        component : (item : comp_type)
      End {cell};

  Header
    head : cell_ptr
  End {Structure};

  Escape Event header_delete;

  Function create : list_ptr;
  Function construct (a : comp_array*) : list_ptr;
  Procedure insert_after (c : cell_ptr;
                          val : comp_type);
  Procedure delete (Var c : cell_ptr);
  Function succ_element (c : cell_ptr) : cell_ptr;
  Function pred_element (c : cell_ptr) : cell_ptr;
  Coroutine elements (l : list_ptr; Var c : cell_ptr);
End {list specification};
```

```
Capsule list  {body of list capsule}

  Function create : list_ptr;
    Var lp : list_ptr;
  Begin
    New (lp);
    With lp-> Do Begin
      New (head, header);
      With head-> Do Begin
        pred := head;
        succ := head
      End {With head->}
    End {With lp->};
    create := lp
  End {create};

  Procedure insert_after (c : cell_ptr;
                          val : comp_type);
  Var
    new_cell : cell_ptr;
  Begin
    New (new_cell, component);
    new_cell->.item := val;
    With c-> Do Begin
      new_cell->.succ := succ;
      new_cell->.pred := c;
      succ->.pred := new_cell;
      succ := new_cell
    End {With c->}
  End {insert_after};

  Function construct (a : comp_array*) : list_ptr;
  Var
    i : 1..Maxint;
    lp : list_ptr;
  Begin
    lp := create;
    For i := LoBound(a,1) To HiBound(a,1) Do
      insert_after (lp->.head->.pred, a[i]);
  End {construct};

  Procedure delete (Var c : cell_ptr);
  Begin
    If c->.kind = header Then
      Escape header_delete
    Else
      With c-> Do Begin
        pred->.succ := succ;
        succ->.pred := pred
      End {With c->};
    c := Nil
  End {delete};
```

```
  Function succ_element (c : cell_ptr) : cell_ptr;
  Begin
    succ_element := c->.succ
  End {succ_element};

  Function pred_element (c : cell_ptr) : cell_ptr;
  Begin
    pred_element := c->.pred
  End {pred_element};

  Coroutine elements (l : list_ptr; Var c : cell_ptr);
  Begin
    c := l->.head->.succ;
    While c->.kind <> header Do Begin
      Suspend Step;
      c := c->.succ
    End {While}
  End {elements}
End; {list body}
```

APPENDIX B

A Control Abstraction Example

The overseer attempt specified below is given a task description and a set of routines (for example, a set determined by some associative reference) that might be able to perform the task. These routines will use the event step_done to inform the overseer of their estimated progress (unless they report win or lose). The overseer will win (report success to its caller) as soon as any of the routines have done so. It will lose (report failure to its caller) if all of the routines lose. If neither of these cases have occurred, it will invoke the routine that last returned the highest progress estimate.

```
Type
  task_desc = Record
      { description of the job to be performed by
        the coroutines provided to the overseer }
    End;
  task_routine = Coroutine (t : task_desc);
  tref = RoutineRef task_routine;
  routine_list = list (tref);
  routine_list_ptr = routine_list$list_ptr;

Suspend Event step_done (eval : Integer);
Escape Event win;
Escape Event lose;

Overseer attempt (task : task_desc;
                  try_list : routine_list_ptr);
  Type
    process_rec = Record
        last_eval : Integer;
        ctxt : ContextRef;
```

```
        p : ProcessRef task_routine
      End;
    process_list = list (process_rec);

  Var
    try : routine_list$cell_ptr;
    process_cell, best_process
      : process_list$cell_ptr;
    plist : process_list$list_ptr;
    proc_rec : process_rec;
    max_eval : Integer;

  Procedure find_max_eval;
    Begin
    max_eval := 0;
    ForStep process_list$elements (plist, process_cell) Do
      With process_cell->.item Do
        If last_eval > max_eval Then Begin
          max_eval := last_eval;
          best_process := process_cell
          End
    End {find_max_eval};

Begin
max_eval := 0;   best_process := Nil;
plist := process_list$create;
process_cell := plist.head;

{ First, create a process corresponding to each
  coroutine on the try list and allow it to run to
  its first evaluation point}

ForStep routine_list$elements (try_list, try) Do
 With proc_rec Do Begin
  last_eval := 0;
  ctxt := CreateContext (CurrentContext);
  Incontext ctxt Do  p := NewProcess try (task);
  Continue p
   Until
    step_done : Begin
      last_eval := step_done.eval;
      process_list$insert_after
          (process_cell,proc_rec);
      process_cell :=
        process_list$succ_element (process_cell);
      If last_eval > max_eval Then Begin
        best_process := process_cell;
        max_eval := last_eval   End {If};
      End {step_done handler};
```

```
      lose : Begin
        Terminate(p);  delete(ctxt);
        End {lose handler};

      win : Begin
        Finalize (CurrentContext,ctxt);
        Escape win  {exit indicating win}
        End {win handler}
      End { Continue p }
    End {With and For};

{ If execution reaches this point, all routines
  on the try list have been executed to their first
  suspension and none have  indicated a win. }

While max_eval > 0 Do Begin
  { Indicate current state to a higher level overseer.}
   Suspend step_done (max_eval);

  { Execution continues here when restarted. }
  Continue best_process
   Until
    win : Begin
      Finalize (CurrentContext,best_process->.item.ctxt);
      Escape win  {exit indicating win}
      End {win handler};

    lose : Begin
      process_list$delete (best_process);
      find_max_eval
      End {lose handler};

    step_done : Begin
      best_process->.item.last_eval :=
        step_done.eval;
      If step_done.eval >= max_eval
        Then max_eval := step_done.eval
        Else find_max_eval
      End {step_done handler}
    End { Continue best_process }
  End { While };

{ If all of the processes indicate "lose",
  find_max_eval will find the list empty
  and set max_eval to 0.  Then the While
  loop will terminate and the following
  statement will be executed. }

Escape lose

End {attempt};
```

REFERENCES

Bob72 Bobrow, D.: "Requirements for Advanced Programming Languages for List Processing Applications", Communications of the ACM 15, 7 (July 1972).

BR64 Bobrow, D. and B. Raphael: "A Comparison of List Processing Computer Languages", Communications of the ACM 7, 4 (April 1964).

BR74 Bobrow, D. and B. Raphael: "New Programming Languages for AI Research", Computing Surveys 6, 3 (September 1974).

BW73 Bobrow, D. and B. Wegbreit: "A Model and Stack Implementation of Multiple Environments", Communications of the ACM 16, 10 (October 1973).

CH73 Clark, B. and J. Horning: "The System Language for Project SUE", SIGPLAN Notices, 8, 2 (February 1973).

DMN70 Dahl, O., B. Myhrhaug and K. Nygaard: "SIMULA Common Base Language", Norwegian Computing Center 1970.

Dav73 Davies, D. J. M.: POPLER 1.5 Reference Manual. University of Edinburgh, TPU Report No. 1, May 1973.

Fei77 Feigenbaum, E. A.: "The Art of Artificial Intelligence", Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massechusetts (1977).

Fik75 Fikes, R. E.: "Deductive Retrieval Mechanisms for State Description Models", Proceedings of the Fourth International Joint Conference on Artificial Intelligence, Tiblisi, Georgia, USSR (1975).

FL77 Fischer, C., and R. LeBlanc: "Efficient Implementation and Optimization of Run-time Checking in PASCAL", Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices 12, 3 (March 1977).

Goo75 Goodenough, J. B.: "Exception Handling: Issues and a Proposed Notation", Communication of the ACM 18, 12 (December 1975).

GM75 Geschke, G. and J. Mitchell: "On the Problem of Uniform References to Data Structures", Proceedings of the 1975 International Conference on Reliable Software, SIGPLAN Notices 10, 6 (June 1975).

GPP71 Griswold, R., J. Poage and I. Polonsky: The SNOBOL4 Programming Language, 2nd. Ed., Prentice-Hall, 1971.

Hab73 Haberman, N.: "Critical Commeents on the Programming Language PASCAL", Acta Informatica 3, 1 (1973).

Hew69 Hewitt, C.: "PLANNER: A Language for Proving Theorems in Robots", Proceedings of the First International Joint Conference on Artificial Intelligence, Washington, D. C. (1969).

Hew71 Hewitt, C.: "Procedural Embedding of Knowledge in PLANNER", Proceedings of the Second International Joint Conference on Artificial Intelligence, London (1971).

Hew72 Hewitt, C.: "Description and Theoretical Analysis (Using Schemata) of PLANNER, A Language For Proving Theorems and Manipulating Models in a Robot", MIT AI Memo 251, 1972.

HLTZ77 Honda, M., R. LeBlanc, L. Travis and S. Zeigler: "An Improved Data Context Mechanism", MACC Technical Report No. 47, University of Wisconsin - Madison, October, 1977.

Hoa73 Hoare, C. A. R.: "Hints on Programming Language Design", Stanford Technical Report CS403, 1973.

Hor76 Horning, J. J.: "Some Desirable Properties of Data Abstraction Facilities", SIGPLAN Notices 11, 2 (February 1976).

IF76 Ichbiah, J. and G. Ferran: "Separate Definition and Compilation in LIS and its Implementation", CII-Honeywell Bull, Louveciennes, France, 1976.

JW75 Jensen, K. and N. Wirth: PASCAL User Manual and Report, 2nd Edition, Springer-Verlag, New York, 1975.

LeF74 LeFaivre, R. A.: Fuzzy Problem Solving, MACC Technical Report No. 37, University of Wisconsin - Madison, September 1974.

LHLMP77 Lampson, B., J. Horning, R. London, J. Michell and G. Popek: "Report on the Programming Language EUCLID", SIGPLAN Notices 12, 2 (February 1977).

Lis76 Liskov, B.: "An Introduction to CLU", Computation Structures Group Memo 136, MIT Laboratory for Computer Science, February 1976.

LSAS77 Liskov, B., A. Snyder, R. Atkinson and C. Schoffert: "Abstraction Mechanisms in CLU", Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices 12, 3 (March 1977).

LSW76 London, R., M. Shaw and W. Wulf: "Abstraction and Verfication in ALPHARD: A Symbol Table Example", Technical Report, Carnegie-Mellon University, December 1976.

LZ74 Liskov, B. and S. Zilles: "Programming with Abstract Data Types", Proceedings of ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices 9, 4 (April 1974).

Mac77 MacLaren, M. D.: "Exception Handling in PL/1", Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices 12, 3 (March 1977).

MBMR73 Myopoulos, J., N. Badler, L. Melli and N. Roussopoulos: "1.PAK: A SNOBOL-based Programming Language for Artificial Intelligence", Proceedings of the Third International Joint Conference on Artificial Intelligence, Stanford, California (1973).

McD76 McDermott, D.: "AI Meets NS", SIGART Newsletter, No. 57, April, 1976.

Mel74 Melli, L. F.: The 2.PAK Language: Primitives for AI Applications. TR 73, University of Toronto, Dept. of Computer Science, December 1974.

Mel75 Melli, L. F.: "The 2.PAK Language: Goals and Descriptions", Proceedings of the Fourth International Joint Conference on Artificial Intelligence, Tiblisi, Georgia, USSR (1975).

MH69 McCarthy, J. and P. Hayes: "Some Philosophical Problems from the Standpoint of Artificial Intelligence", in B. Meltzer and D.Mitchie (ed.) Machine Intelligence 4, American Elsevier, 1969.

MS72 McDermott, D. and G. Sussman: "The CONNIVER Reference Manual", AI Memo NO. 259, MIT Project MAC, May 1972.

Pok76 Pokrovsky, S: "Formal Types and their Application to Dynamic Arrays in PASCAL", SIGPLAN Notices 11, 10 (October 1976).

RDW73 Rulifson, J., J. Derkson and R. Waldinger: "QA4, A Procedural Calculus for Intuitive Reasoning", Technical Note 73, SRI AI Center, November 1973.

REFN73 Reddy, D., L. Erman, R. Fennell and R. Neely: "The Hearsay Speech Understanding System: An Example of the Speech Recognition Process", Proceedings of the Third International Joint Conference on Artificial Intelligence, Stanford, California (1973).

RR65 Radin, R. and P. Rogoway: "NPL: Highlights of a New Programming Language", Communications of the ACM 7, 4 (April 1964).

RS73 Reboh, R. and E. Sacerdoti: "A Preliminary QLISP Manual", Technical Note 81, SRI AI Center, August 1973.

Sho76 Shortliffe, E.: Computer-based Medical Consultations: MYCIN, Elsevier, 1976.

SIGART/SIGPLAN77 Unpublished panel discussion at the SIGART/SIGPLAN Symposium on Artificial Intelligence and Programming Languages, University of Rochester, August 1977. (Participants: G. Sussman (CONNIVER/PLANNER), J. Feldman (SAIL), R. Waldinger (QA4/QLISP), J. Davies (POPLER))

SM72 Sussman, G. and D. McDermott, "Why Conniving Is Better than Planning", AI Memo No. 255A, MIT Project Mac, April 1972.

STS77 Spencer, H., J. Tremblay and P. Sorenson: "Programming Language Design", Technical Report 77-5, Department of Computational Science, University of Saskatchewan, June, 1977.

SWC70 Sussman, G. J., T. Winograd and E. Charniak: "Micro-Planner Reference Manual", AI Memo No. 203, MIT Project MAC, July 1970.

SWl77 Shaw, M., W. Wulf and R. London: "Abstraction and Verification in ALPHARD, Designing and Specifying Iteration and Generators", Proceedings of an ACM

Conference on Language Design for Reliable Software, SIGPLAN Notices 12, 3 (March 1977).

Tei69 Teitelman, W.: "Toward a Programming Laboratory", Proceedings of the First International Joint Conference on Artificial Intelligence, Washington, D. C. (1969).

Tei75 Teitleman, W.: INTERLISP Reference Manual, Xerox Palo Alto Research Center, 1975.

Tei77 Teitleman, W.: "A Display Oriented Programmer's Assistant", Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massechusetts (1977).

THLZ77 Travis, L., M. Honda, R. LeBlanc and S. Zeigler: "Design Rationale for TELOS, a PASCAL-based AI Programming Language", Proceddings of a Symposium on Artificial Intelligence and Programming languages, SIGPLAN Notices 12, 8 (August 1977).

Tho77 Thompson, C. M.: "Error Checking, Tracing and Dumping in an ALGOL 68 Checkout Compiler", SIGPLAN Notices 12, 7 (July 1977).

Van73 VanLehn, K. A.(ed.): "SAIL User Manual", Stanford AI Laboratory Memo AIM-204, July 1973.

Wil76 Wilber, B.: "A QLISP Reference Manual", SRI AI Note 118, 1976.

Wir71 Wirth, N.: "The Programming Language PASCAL", Acta Informatica 1, 1 (1971).

Wir74 Wirth, N.: "On the Design of Programming Languages", in Information Processing 74, North-Holland, 1974.

Wir76 Wirth, N., Algorithms + Data Structures = Programs, Prentice-Hall, 1976.

WLS76 Wulf, W., R. London and M. Shaw: "Abstraction and Verfication in ALPHARD: Introduction to Language and Methodology", Technical Report, Carnegie-Mellon University, June 1976.

Zah74 Zahn, C.: "A Control Statement for Natural Top-down Structured Programming", In Programming Symposium, Lecture Notes in Computer Science, Vol. 19, Springer-Verlag 1974.