The Metasynthesis of Domain-Oriented Languages

University of Regina
Technical Report
Revision 1.1

by Chris Hecker

April 21, 1994

The Metasynthesis of Domain-Oriented Languages
Revision 1.1

Chris Hecker
Computer Science Department
University of Regina
Regina, Canada
heckechr@max.cc.uregina.ca

Abstract

Modern programming languages are difficult to use for many programmers in disciplines other than computer science. These difficulties are not the fault of programmers, but are the result of poor language design: most modern programming languages are designed to solve hardware or technical problems. The design of high-level user-oriented languages, languages which prioritize the user's needs and expectations, can partially remedy this situation. Furthermore, by utilizing each discipline's conceptual model and notation in specialized domain-oriented languages, language designers can fully meet the needs of professionals in other subject areas. A user-oriented language design tool, the Trans-Linguistic Parser, developed by the author, is a first step towards the design of user-oriented languages. As better languages fully utilize human intellectual potential, programmers will experience a rise in both productivity and effectiveness in their programming.

1. Introduction.c.I. Introduction;

"Computer Scientists have so far worked on developing powerful programming languages that make it possible to solve the technical problems of computation. Little effort has gone toward devising the languages of interaction." [NORMAN88, 180.]

Many modern programming languages have a problem: their users have trouble relating to them. These languages, designed as tools to solve technical and machine related problems, are widely used as multi-purpose languages in outside environments. Trained computer professionals, whose problems and knowledge these languages address, have in these languages reasonably effective tools for their own use. For those with different educational backgrounds, or who have other kinds of problems to solve, the same tools are of little use.

Thus, many modern programming languages cannot be effectively be used by a large number of potential programmers. At the same time, most programmers are not well versed in disciplines for which they program. In many cases, neither the user or their hired programmers possess the expertise to write the software. Either the professional must spend long hours learning how to program in a language, or he must spend long hours teaching his programmers about the area he works in.

In the past, computers have not been either powerful or flexible enough to allow a better solution to this problem. However, the recent proliferation of complex, low-cost, high-speed computers presents an alternative: languages oriented towards the human user and her domain of expertise.

Programming language design is at the threshold of a revolution. Technical, computational, and machine-related problems, which for so long dominated the agenda of programming language design, will soon give way to communication and interaction problems. Computer users, who have struggled with arbitrary and unnatural forms of expression in programming, will finally find satisfaction. Faster, more complex compilers and translators are being developed that will allow users to communicate with computers at their own level, the level of ideas, conceptual models, and actions within those models. New user-oriented languages and domain-oriented languages will enable people in all disciplines to program computers effectively. The design of such languages coupled with automated programming software is the next logical step in the history of programming languages [BALZER85].

2. The Development of Computer Science and its Effect on Programming Languages

Language, in its broadest sense, is the method by which humans communicate with the outside world. The internal representation within the human brain must find expression in order to interact with its surroundings. Furthermore, since each human learns at least one language from his or her infancy, the assumptions made by that native language are reflected in the thought patterns of that individual.

Every language has ideas, assumptions, and conceptual models built into it. These reflect the biases of the language's speakers and designers, and the environment in which the language exists. As Sapir and Wharf observed, language defines the way we think. Thus, we create and are created by our languages; we limit them and are limited by them.

Computer science introduces an entirely new perspective to the study of language. For humanity now has at its disposal a problem-solving tool that emulates parts of its own intelligence. Computers share with us the ability to manipulate symbols. However, computers do not yet possess some of the higher attributes associated with intelligence such as volition, complex reasoning, or sentience. Until now, therefore, humans have communicated with computers at a level closer to that of the computer rather than the human capability.

2.1 Historical Reasons for Current Language Design Flaws

Since the earliest days of the electronic computer, programming languages have evolved and developed to facilitate its use. The development of these languages was heavily influenced by the technology of the 1950s and 60s. Computers of that era were much slower, had much less memory, and cost more to produce. The input hardware used for communicating with computers was in the process of invention; this led to slow and cumbersome input devices such as card punches. The architecture of early computers was limited by the Von Neumann bottleneck, which imposed a procedural structure on programming. Such languages are machine-oriented, designed to adapt to the slow, expensive computations, cumbersome input, and bottlenecked architecture. The resulting languages, such as assembly language, FORTRAN, and COBOL, often exhibit simple structure, low-level operations, a linear textual form, a procedural paradigm, and brevity of expression.

The technical issues and research problems of computer science have also contributed to computer languages. Most computer scientists in the 1950s and 60s were concerned with technical or mathematical problems. This produced computer languages concerned with hardware and technical issues, computational efficiency, mathematics, and correct form -- one example is FORTRAN. Programmers who have little experience with technical issues must nevertheless must use a computationally-oriented language derived from this environment.

Later languages such as C stayed true to the machine-oriented model, while languages such as LISP, FP, and Prolog modelled human logical processes or mathematical formulations. However, all were written by computer scientists, and reflected the computer science conceptual models.

"Fourth Generation" languages, developed in the late 1970s and the 1980s, proved a step in the right direction. Database languages like Natural went further than ever before in abstracting implementational details, in the domain of database operations. Simula 67 and CLU introduced object-oriented methods into the software world, which helped users structure and organize their programs. Although these languages represented improvements, they were by no means the user-oriented languages that can bridge the gap between computer science and the outside world.

Despite great improvements in computational power, many of the programming languages used today are still machine-oriented. Indeed, the most popular early languages are widely used. The speed, complexity and interface restraints that early computers imposed on human-computer interaction are still imposed on programmers. The majority of humans must still struggle to learn programming languages. The design and use of programming languages lags behind the power of available technology.

3. User-Oriented Languages

Machine-oriented languages do the human programmer a disservice. The problem inherent in these languages, from a language user's point of view, is that they are constructed to be computationally efficient instead of conceptually efficient. Technical issues, hardware compatibility, and mathematical correctness are currently the prime factors in language design. Ease of interaction and compatibility with human thought is a minor or nonexistent consideration. Even languages designed for non-expert users bring this philosophy to the design process. If programming languages are to be effective for humans, they must be closely matched to human ways of thinking.

The human mind has a different design, needs, and capabilities than a computer. "Possession of human language is associated with a specific type of mental organization, not simply a higher degree of intelligence." [CHOMSKY72, 48] Humans can be expected to be much more productive when they are provided with languages patterned after their own mental organization and affinities. Human thought and expression must be studied to determine how best to develop human-oriented languages.

If human thought needs to be studied in order to create user-oriented languages, then so much more the human use of computers. The human relationship to computers is that of craftsman to tool. Computers are primarily used to solve problems and perform tasks for humans. Since a programming language is a problem-solving tool, humans will expect a language to facilitate problem-solving. Programming languages should be optimized towards efficiency and effectiveness of human tasks, rather than the solution of implementation problems.

Finally, better programming language design requires a knowledge of how humans formulate and express computer programs in particular. Humans tend to formulate computer programs as conceptual models and operations on those models; current programming languages see programs as primitive data structures and procedural operations.

The study of these three areas -- human thought, computer use, and program development -- will allow the construction of user-oriented languages, languages adapted not to the needs of computer scientists and machines, but to human mental proclivities and problem-solving capabilities.

3.1 Human Thought and Expression

The world in which humans live determines to a great extent how they think. Humans spend the first fifteen to eighteen years of their life absorbing essential information about their world. Adult humans thus have a highly developed personal knowledge base, plus an environment which they can read and interpret. They need not formally declare their knowledge, but merely learn to store, retrieve, and interpret it.

Humans can also rely on the world around them to provide much information, both as raw data and as natural and cultural constraints. The "real world" tolerates imprecision in behaviour and incomplete mental knowledge, as information and feedback can be found in the world. Human use of language demonstrates this; even everyday conversations are punctuated by mutually understood implied references, gestures, and ambiguities. Transcripts of everyday conversations may be indecipherable without added information about gestures, voice inflection, and contextual information.

Another aspect of human thinking is strong preference for visual input. The human sense of sight is the most developed and relied-on method of input. Humans interpret the world largely on the basis of what they see; two- or three- dimensional visual communication is often essential. Some of the earliest natural human languages -- Egyptian hieroglyphs and early Chinese characters -- were formed from pictures. In modern times, although textual languages dominate Western languages (primarily due to the relative ease of printing and typing it), maps, diagrams, photographs and videotapes greatly aid the communicative process.

Human thinking and communication utilizes a large bandwidth. This bandwidth includes visual images, but the brain registers other information as well. Multisensory stimuli are registered, remembered, and used in the process of thinking. Gestures, sounds, pictures, text, smells, and tactile inputs all play a part in communication between humans. Communication with humans is more effective when it utilizes a large bandwidth and more than one of the senses.

Emotional security is very important to humans. Humans like to have a sense of control, a feeling that everything is going well. [SHNEIDERMAN80] Placed in a threatening environment, humans will react defensively, and will often think irrationally or fearfully. Non-punative environments where control is established, and events happen in a somewhat predictable order will be more conducive to a human and their ability to work. Humans like to control what they do; simplicity and user-friendliness in design can go a long way to ensuring the effectiveness of a product in human hands.

3.2 Human Problem-Solving Techniques

"Solving problems is a fundamental human activity. In fact, the greater part of our conscious thinking is concerned with solving problems."
- G. Polya [POLYA57]

A computer is ideally a mechanical extension of the human reasoning process. In order to help people solve problems, therefore, computer languages must expedite common procedures used in human problem-solving. G. Polya's analysis of problem-solving catalogues several techniques humans use to solve problems. These range from the strictly mechanistic logic or mathematical techniques, to complex reasoning and visual processing, to intuitive approaches.

Mathematical and logical problem-solving is a widely-used method that utilizes a rigorous set of definitions and techniques. The introduction of a formal mathematical or logical notation clarifies meaning and enables one to completely prove theorems or axioms, or find concrete mathematical solutions to problems. Techniques like mathematical induction and logical proof are straightforward methods useful in solving many problems. These are accompanied by the use of a formal notation -- symbols and operations representing objects and actions in the problem domain -- that can be manipulated according to rules to expedite the solution.

Comparing problems to other, similar problems is a well-known technique. Many new problems have aspects that resemble easier or previously solved problems. Analysis with respect to another problem and solution can lead to partial or complete solutions to the issue in question. If necessary, the current problem may be broken down and solved in parts. Or, it may be a specific instance of a more general problem.

Indirect methods can also work well. These range from logical procedures such as reducing the antithesis of a proposition to an impossible conclusion, to processes used in artificial intelligence, such as working backwards from desired solutions to initial conditions. Hybrid combinations of indirect methods with direct methods are utilized very successfully.

A more nebulous classification of techniques are the intuitive ones, encompassing analogy, visualization, and "bright ideas". Analogy involves an appeal to similar experience outside the problem domain to further understanding of a problem. By linking disparate subject areas, or even appealing to everyday life, humans can often gain greater comprehension of problem issues and solutions. Visualization, the use of visual images to express a problem, is a powerful tool for problem analysis. The human brain seems better able to reason with visual images than abstract models, and visualization of a problem appeals to that human strength. "Bright ideas", or sudden flashes of intuition, are not well understood. A series of dissimilar facts with a hidden link can meld into sudden harmony if the brain thinks about it enough.

Some of these techniques, such as induction, work extremely well with computers. Others, like generalization, are more difficult to realize. Techniques such as analogy appeal directly to human intuition and experience, and are not well suited to a computer. Many of these techniques are uniquely human; at present they are not implemented on a computer. For example, how does one program a computer to have flashes of intuition? However, principles can be derived from human thought, preference, and problem-solving technique that can be applied to languages to make them more intuitively correct and human-friendly.

3.3 The Programming Process.c.The Programming Process;

J. T. Schwartz writes that humans initially conceptualize computer programs as proto-programs -- "intellectual structures ... in which only major rules governing process activity and data transformation appear." [YEH, 188]. Such proto-programs must then be translated into an appropriate form for input into a computer. Thus, the closer the language form is to the user's proto-program, the easier it is for the user to program a computer. Schwartz calls such programs "rubbles", in reference to their loose structure and arrangement of ideas.

The process of computer programming, in its purest form, is translation [KLERER91]. Traditionally, human users have been called on to do the bulk of the translation from human proto-program (or "rubble") to computer program. The amount of translation that needs to be done will depend on the distance between the programmer's conception of a program and its equivalent in the chosen language. Human programmers are forced to structure and optimize their ideas in a series of transformations and translations. [SCHWARTZ78, 190]

First, a procedural structure must be introduced into a program. This organizes processes into groups of like function and organizes their execution. Next, procedures are translated into the subsidiary language forms; modern programming languages force programmers to use a limited set of keywords. Efficient methods of computation are then worked out for use in the low-level program. These methods include special data mappings or combinatorial structures, and reductions in the number of calculations. The latter "can be done by maintaining current values of important expressions." [SCHWARTZ78, p. 190] This allows a programmer to increase processing speed to acceptable levels. By the constructive use of relationships between the data, one can make an algorithm faster by orders of magnitude. Finally, data structures must be constructed to realize the implementation of a more abstract algorithm. These structures are required by lower-level languages in order to handle the data processing of the program.

The job of a user-oriented language is to minimize this translation. Optimization is a process that can and should be automated, ultimately allowing users to program in rubble languages. The more time users spend on determining the translation of their concepts, the less time (proportionally) they will spend on actually working on obtaining the solution to their problem. Thus, the most powerful languages are those in which user specifications are the input [KLERER91].

It is obvious that such very high-level languages will present a host of technical problems to the language designer. These range from optimization of very high-level languages to the interpretation of ambiguous clauses to the utilization of special mappings.

3.4 Factors that Complicate Human-Computer Interaction

"The thing that makes computers so hard to deal with is not their complexity, but their utter simplicity."
- William Kent [KENT78]

The problems experienced by many language users are due to differences between the human programmer and the computer. Humans are very complex organisms; the human brain is typically complex and dwarfs any of today's computers in complexity. The thought processes that accompany even simple actions require complex interactions of neurons. Computers, on the other hand, are a blank slate waiting to be written on. Computers contain no background information about themselves, the world, or their programmers, other than an operating system and the assumptions and provisions of the language designer.

Computers work particularly well with discrete, precise data that can be manipulated and equated. Every piece of information necessary for a computer program to work must be explicitly given to the computer, or be derivable from the explicit data. Case-by-case enumeration and exact comparison are used as primary operations. This forces humans to quantify their knowledge and develop machine representation of it, a difficult and time-consuming process. Indeed, the demands of the computer and the language are often quite foreign to a "real-world" human. Blueprints for a bridge do not explicitly plan for every possible gust of wind, or every possible combination of motor vehicles and cargo loads. Yet this is the level of competence demanded of programmers.

Such precise construction of data structures and a knowledge base is not a part of the usual human experience. Thus, automation of this process -- this is equivalent to using an expert to tell a computer how to optimize instead of letting each individual user do it -- will greatly increase the effectiveness of human programmers. Thus, errors often occur when (relative) precision or completeness of knowledge is required.

As well, computers communicate with a very low bandwidth. Humans, who are very visually oriented, and have four other senses to gather information, are restricted to mere text with many current languages and operating systems. Even with graphical interfaces, humans can only use a small portion of their bandwidth in communication. This communication limitation can affect other facets of programming, for example, computers are difficult to program with vision-based reasoning.

Errors on the part of programmer and user are another concern. Computers will perform faultlessly exactly what they are told to do, and do not make errors as such. Unfortunately, computers have no error resiliency except what is programmed in software. Unexpected user behaviour or bugs in software can cause serious problems for the user..

3.5 Design Principles for User-Oriented Languages

The litmus test of a user-oriented language is whether the language's target user group can use it to solve problems. Language users want an easy-to-use, not overly complex form of expression that lets them describe problems and algorithms. Such a language will not force users to learn a special technical jargon, or a restrictive, bulky, or complex notation. Rather, such an ideal language will be virtually transparent, allowing users to express their information, problems, and algorithms easily and succinctly.

The burden of helping the user must fall on the designer. Too many users will attempt to program in a badly designed language, and blame themselves when their software fails. Many will also construct faulty conceptual models in poor languages, especially if they are trying to implement a simplified or simplistic model. "Users tend to acclimate themselves to the products of bad design simply because they perceive no alternative." [KLERER91, p. 188].

The most important requirement of user-oriented languages is that they be intuitively correct. That is, programming languages must be constructed so that their parts fit logically together, and make sense to a typical user. A user should easily acquire the fundamentals of the language using relatively simple instructions, his own reasoning ability, and the knowledge of systems in general. Programming languages should be logical and internally consistent; for example, the C languages uses single angle brackets (< , >) as less than and greater than operators, but double angle brackets (<<, >>) for bit shift operations. Some users may think double angle brackets refer to "much greater than" or "much less than", or may use them as quotation marks.

User-oriented languages should be transparent -- the implementational and technical details should not distract the user from problem-solving. Explicit "programming" requirements should be minimized; "programmers" shouldn't need a lot of training. Language constructs should map to simple objects or actions that the user understands; computer jargon should be avoided wherever possible. Rigidity in the syntax of the language should be avoided, as rigid syntax leads users to get bogged down with syntax errors. While mathematical algorithms and properties may be desirable in a language, esoteric constructs are not. Intuitively correct languages do not use overly technical constructs, concepts or syntax; neither do they utilize counter-intuitive or contradictory elements.

Concepts need to be related in a meaningful way. Symbols should be clearly and logically defined, and may be used for a single function or multiple, logically related ones. (For example, the period "." used in North American English can indicate the end of a sentence, or a decimal point, or that a word is an abbreviation.) Ambiguities should be avoided. A combination of meaningful keywords and clearly related concepts can lead to programs that are virtually self-documenting. Code properly written in a good user-oriented language should be easily readable, and easily maintainable as a consequence.

If a programming language is to be truly effective, the tools used with it need to be well designed also, and fully integrated. Feedback in debugging should be intelligent and easy to understand. A rich selection of output formats should be supported. The language should be capable of interfacing with other already coded programs. In other words, the new user-oriented language should not force manual translation of old programs.

As well, languages should be designed to minimize the potential for human error and misinterpretation. This is not always the case. The C language, for example, uses '=' for the assignment statement and '==' for the equality test operator. The difference is obvious to a computer, but confusing to most humans.

User-oriented languages must let users conceptualize rather than concentrating on technical or implementation problems. Nonessential technical details can be abstracted, thus reducing the complexity faced by users. The effective use of visual information is a must; humans are extremely visual and can be greatly stimulated and aided by effective visual presentation. This implies that although User-oriented languages can be linear, two- (or even three-) dimensional forms are preferable. Furthermore, User-oriented languages should be easily readable, and well suited to program composition.

User-oriented programming languages should utilize a high bandwidth to communicate with users. With the advent of complex display hardware and software, we now have the ability to transcend mere linear text on a computer. Many operating systems have Graphical User Interfaces (GUIs) which allow users to manipulate text, graphics, file system objects, and applications using graphics rather than text. Computer art flourishes, with applications as diverse as Industrial Light and Magic's special effects to the digital transmission of newspaper photographs. Despite the explosion of computer-based visual technology, the bulk of computer languages still stick to the traditional text-based idiom.

However, some programming languages are now being produced that utilize pictures or non-linear forms. These range from scientific applications to presentation languages like Apple's HyperTalk(tm), an English-like graphics presentation language. [BEEKMAN90] Another user-oriented language which uses a graphics interface is Klerer's Automated Programmer(tm) [KLERER91] is a graphical implementation of mathematical notation. Its basic linguistic structure is a multi-line, multi-clause sentence where each line is a two-dimensional area. The Automated Programmer then translates the structure into C or FORTRAN, which is compiled by a standard compiler.

4. Domain-Oriented Languages

User-oriented languages represent a milestone in the evolution of the information age. With relatively simple, direct, conceptually effective languages, computer programming is made much easier for the majority of people. However, general-purpose programming languages do have some drawbacks.

The biggest objection to general-purpose languages is that they are not specific enough. The very quality that makes a general-purpose language so flexible and thus universally acceptable also makes it difficult to use for some programmers. When a programmer wishes to write an application for his subject area, she must enter in all the background knowledge necessary for the model she works with. For someone with little experience in knowledge representation, this may be an arduous task. Furthermore, by trying to cater to every possible use, a general purpose language will not utilize the strengths of specialized domains such as extant notations and methods. Quite often, a computer science paradigm is reflected in general-purpose languages, which can make them less suitable for specific-domain programming.

Few people have the expertise to write domain-specific software in a general-purpose language. Many domain professionals will implement conventional computer programs incorrectly. Conversely, trained programmers may use the computer language correctly, but will often lack knowledge about the domain they are programming for. In order for a well-written program or system to come into being, the two groups must pool their resources. Even so, few in either group will understand completely what the other side is doing.

Currently, in order for domain experts to acquire enough knowledge to program to their own satisfaction, they must learn the conceptual model and mindset of computer science. This can take a lot of time and effort. One can legitimately as why they should waste their time doing that, when they have a perfectly good conceptual model at their disposal?

Research done by R. E. Meyer shows the learning process can be enhanced by building on one's previously constructed mental models [MEYER81].

Here exists a need for a class of languages that are both user-oriented and able to communicate effectively in terms of the domain conceptual model. Such domain-oriented languages can effectively fill the gap between technical requirements and domain expertise.

4.1 Principles of Domain-Oriented Languages

While each domain will design and use a computer language in a different way, small domain languages have certain principles and properties in common. These principles ensure that each DOL will satisfy the needs of its users. Domain-oriented languages should be completely user-oriented as described in section 3.5. As well, several other properties characterize a good domain-oriented language.

Domain-oriented languages should be an accurate implementation of the conceptual model of the domain as expressed through a notation. Constructs should behave in the language and compiler software as the real objects in the world being model behave. Notations and methods of expression used in the domain should be integrated into the language, yet unconventional, contradictory, or idiosyncratic notation should be avoided. The language should be intuitively correct to users from the target domain.

Conceptual effectiveness is the primary goal of domain-oriented languages. As much as possible, these languages should also be conceptually efficient. Such speed and accuracy of programming can be accomplished by eliminating nonessential syntactic constraints and providing strong error-handling mechanisms. Only syntactic constraints which enforce conceptual or computational principles are useful to a programmer; for example, one may not need to declare variables, or to obey case sensitivity. Similarly, error messages should emphasize conceptual errors. If the language is designed correctly, the only syntactic errors will be reflections of conceptual errors or simple typing mistakes.

Domain-oriented languages should also reflect the dimensionality of their conceptual models. Although not all applications require multi-dimensional input, many (like mathematics, chemistry, and music) "live" in a two- or three-dimensional world. The use of graphical elements in languages will only improve human-computer communication. The ultimate computer languages may find their place in virtual reality rooms, as humans find that they can interact fully with the objects of their programs. A chemist apt at showing a computer how to construct molecules by manipulating the reactants with his bare hands will be reluctant to return to FORTRAN.

4.2 The DOL Environment: a Complete Picture

A complete domain-oriented language is essentially an applications package that enables users to program a computer using their own conceptual model and notation. The DOL allows domain professionals to create programs, manipulate data, and solve problems in their own environment.

In a DOL, several parts combine to make a whole. Users will interact with the DOL through a graphical interface, unless their domain is entirely text-based. This allows the user to enter programs in his own domain notation. When the user is ready to compile the program, the notation may need to be linearized, i.e. converted into a one-dimensional text form. A language translator can then translate the DOL into a lower-level language for compilation. Alternatively, a very high-level compiler may be built to better optimize the program code. A database or knowledge base representing the conceptual model must be built for the programmer/user, as must a procedure library consisting of action primitives. (In other words, the procedural library is the abstraction of technical-level action.)

Essentially, the domain-oriented language designer must build up a domain-specific graphical environment, knowledge base, language form, and procedure library. Several tools will be useful to the designer: language construct templates (production rules, queries, data structures of various types, functional forms, etc), a metalanguage compiler or translator, compiler/debugger toolkits, and the like.

Much time and effort is necessary to build a domain-oriented language for all but the simplest and narrowest domains. The cost of constructing a DOL may prove prohibitive unless one has a large user base to pay for the development. In order to minimize the cost of constructing individual DOLs, a bigger, more general package can be implemented, a DOL construction engine. This engine will allow many domain-oriented languages to be built out of the same package. This should lower the cost of such a language, since we can reduce the duplication required in programming many DOLs individually. A DOL engine will be composed of many of the same parts as the language: a graphical environment, a database/knowledge base, and a low-level language compiler. Design tools must also be included for the DOL engine: conceptual model rendering tools, notation design applications, a metalanguage compiler/translator and a graphical programming environment.

4.3 Conceptual Models.c.Conceptual Models;

Every domain forms its own unique conceptual model of its world and concepts. Certain information about this world is stored in it: entities, relationships between entities, data about entities, and so on. Each domain has its own notation that is used to express concepts and ideas about the domain and its world. Each model also has its own limitations, quirks, and deviations from reality.

Humans learn about real systems by developing mental models of them [NORMAN90, p. 12]. This is typically done by learning the model from a friend, mentor, supervisor, or textbook. Once learned, the model enables one to predict the behaviour of the real world by mentally simulating the model's operation. Using this process, humans are able to manipulate systems to achieve desired results.

In order to facilitate the learning of programming skills in non-computer science trained people, language designers can use this hard-won knowledge. We must take these conceptual models developed over years of domain-specific training, and integrate them into the language. No longer is hardware or even the conceptual model of the computer the primary design consideration. Instead, language design begins with the construction of a computer model of the domain in question. Implementation and efficiency, heretofore the demigods of computer languages, are subordinated for the sake of the human-computer interface.

Conceptual model integration forces the language designer to incorporate into a language the basic model of the systems in question. In other words, much data and relationship information must be entered into the computer for use in the language. The language begins to resemble an application software package, as this may mean inclusion of a database, GUI interfaces, etc., in the language package.

In essence, the job of a DOL designer is that of compiler expert, domain expert, human-computer interaction expert, and an expert in the construction of large-scale data structures. The most difficult part of the job may be translating the domain conceptual model into a computer-resident knowledge base.

4.4 Representing Domain-Specific Notations in Computers.c.Representing Domain-Specific Notations in Computers;

"... the use of signs appears to be indispensable to the use of reason"
 - [POLYA57, 135].

The transformation of a domain's notation is the first step in converting domain knowledge into a DOL. Domain notations are the language of a domain's conceptual model. Every concept or idea present in a conceptual model has a logical representation in a well-formed notation. This notation is usually the currency of a profession; every idea, every thought is expressed in the notation, or, where necessary, a technical vernacular. The goal of any domain notation is, therefore, to provide an unambiguous, conceptually efficient and standard method of communication between members of a certain profession.

4.4.1 Well-Formed Notations

"Intuitively, the intent of designing an automated programmer for scientific programming is to reduce the programming process to one of copying problem-solution methods from a textbook. This is possible in a literal sense only if the textbook is well-formulated." [KLERER91, 185]

The ideal notation will be unambiguous, flexible, and broadly applicable. It will possess a one to one mapping between ideas and symbols, and have a fine enough granularity of concept to encourage exact specification of ideas. Such a notation will be uniform, conforming to accepted domain standards. The notation should be conceptually efficient and effective, and show linkages between entities clearly. It will be intuitively correct to a domain expert, yet easily learned by a student. It should not be too verbose or too cryptic. Finally, it is essential for a notation to be internally consistent.

Unfortunately, few notations possess these characteristics. Typically, notations have grown up over decades (or even centuries) of use and revision. As a consequence, many people in a variety of situations have contributed to the notation. This tends to give a notation flexibility and broad application over the entire field of knowledge; both are necessary for a good notation. However, a notation may also acquire poor characteristics in the course of its history. Different schools of thought may create different notations, or dialects of the same notation. Various dialects may be useful in one area but not in another. Or, advances in the field may render the previous standard notation obsolete.

Mathematical notation provides us with an example. Differentiation, since it was discovered in the seventeenth century, has been given many symbols. These range from a series of prime marks f', to bracketed subscripts f(3), to differential notation df/dx. While all of these symbols are in common usage, many groups have favorite symbols for differentiation. Indeed, they are used interchangeably. This creates a technical problem for a computer-based implementation: three nontrivially different notations can have the exact same meaning.

One must also realize that notations are not always true to their parent conceptual model. One example of this is the Lewis diagram of the chemist's domain. Rather than representing electron orbitals as the 3-D constructs that they are conceptualized as, Lewis diagrams represent them as circular shells, with the electrons fixed in space. Chemists understand that Lewis diagrams are a shorthand form, while beginning students do not. This sort of notation must then be explained or resolved by an implementor.

4.4.2 The Process of Notation Conversion

Thus, converting notations to suitable machine-based representations is not a trivial task. While one tries to eliminate (or work around) redundancy, irregularity, and inconsistency, one is still bound to preserve the flexibility and essential features of a domain notation. Ambiguities, problems, and illogicalities in the notation (particularly in fields which are not rigorous in the development of a notation) may prove that a notation is not suited to extensive automation. "The best that can be accomplished in [not well formulated areas] is to attempt to formulate a language design that minimizes the burden on the user for translation to well-formulated and executable forms." [KLERER91, 186].

When converting a domain notation into a machine representation, it is essential that the language designer fully understand the conceptual model that the notation represents. Relationships between entities, actions performed on entities, and rules governing their behaviour must be known exactly. Without an excellent knowledge of the required information, the designer risks creating an erroneous model in software.

After attaining a full understanding of the domain conceptual model, the domain notation can be studied in detail. Inconsistencies, incompleteness, and ambiguity must be resolved enough that a credible machine representation can be created. (At this point, the designer may discover that a notation is too inconsistent to implement in software without inordinate effort.) A wide range of problems and solutions typical of the domain may be studied, in order to uncover subtle meanings and uncommon usages in the notation. It is essential, for the user's sake, that a DOL be designed to use common domain shorthand forms, and multiple representations of the same idea. At the same time the DOL must be programmed with a knowledge of the difference between the notation and the conceptual model.

Finally, the language designer must determine the required functionality of the language. Knowledge about the problems which require solution is synthesized together with the adapted notation. The functionality of each construct in the notation must be documented, and specifications written as to its purpose, syntax, and effect within sentences of the notation. Once the functionality is determined, it is left to the programmer to implement this functionality by the use of the tools mentioned in section 4.2.

Note that this methodology disagrees with conventional wisdom on programming languages. Technical considerations are left to the compiler writer; the language designer's job is to craft a product for the ease-of-use and conceptual effectiveness of the user.

4.5 Domain-Oriented Languages Now and in the Future

Though most currently popular programming languages leave much to be desired, the evolution of computer languages in the last ten years has clearly shown a move towards domain-oriented languages. This trend should continue and accelerate as the information revolution puts computer technology into the possession of increasingly diverse peoples and subject areas.

The proliferation of database languages, from SEQUEL to QBE types to relational algebras, exemplifies the need for special-purpose languages. The effort of using a general-purpose language to do queries would be nearly equivalent to writing a database language! Of course, many of these languages are typical text-based procedural languages adapted to the requirements of database work. Only QBE uses visual methods, such as setting up a 2-D query chart. As expected, users program with greater speed and confidence when they use QBE as opposed to SEQUEL [REISNER81].

Another area that has found DOLs helpful is that of Mathematics. Math packages, such as Maple(tm) and Mathematica(tm), are popular at universities and higher-end research firms. Maple is a well thought out, well-adapted text-based language that serves its purpose well. It performs all standard math functions, and uses some two-dimensional output forms, albeit consisting only of available text characters. Mathematica is similar in construction, but is implemented in a GUI environment and allows the user to program graphs and other displays.

4.5.1 The Future of Domain-Oriented Languages

The future development of domain-oriented languages will continue. With the advent of virtual reality, and the increasing computing power available, new methods of human-computer communication will be used in many subject areas. Domains which will benefit most from the new technology and design principles are those which are too complex for current languages, or which require detailed graphical representation.

Graphics and virtual reality will prove extremely powerful tools for the language designer. Instead of having to specify in textual detail how to construct data structures or how a procedure is to act on certain objects, future programmers will demonstrate this to the computer by creating and manipulating objects in an on-screen model. By utilizing higher communication bandwidth, and by visualizing the model on which a program is based, such "languages" will appeal directly to the human intellect.

The final form of human-computer communication will be nothing like the programming languages of today. Programmers will utilize virtual reality devices to create and manipulate a conceptual model with their actions, gestures, and voice commands. They will be able to view conceptual models as multidimensional constructs from many angles, and move through the model to explore details of each object and relationship. Operations on objects will be specified by representational actions recorded in the virtual reality environment. Programmers will not just write program code, they will live and work inside the program representation.

4.5.2 Examples of Potential Domain-Oriented Languages.

Three good candidate domains for domain-oriented languages are chemistry, genetics, and music. Chemistry will be the most difficult of the three, because of the huge amount of data needed to implement the chemical conceptual model, and the complexity of that model. A key question for the programmer of a chemistry DOL will be the extent to which the real world can be modeled by equations. For example, unless a suitable equation can be found to describe state transitions (e.g. boiling), the boiling and melting points of all compounds will have to be stored individually. Also, any such language will need a environment that allows for the display, rotation, etc. of molecular structures. Chemistry is one domain ideally suited to advanced techniques like virtual reality. A three-dimensional world where one can illustrate chemical reactions by manipulating molecular models with one's hands is vastly superior to today's primitive keyboard, monitor, and mouse combination.

A somewhat less complex domain is that of genetics. Computer technology is already utilized by geneticists, with the Human Genome Project being the most famous example. Genetic notation is typical of natural science notations; it is very visual and two-dimensional. The key question here is how to represent the notation, since the conceptual model of genetics is relatively easy and straightforward to implement. The simple set of symbols used in genetic notation (see below) is very easy to read and write, and conveys information efficiently. This is a clear-cut case where a two-dimensional notation is much superior to any linear text.

Music is another example of a two-dimensional form which would be utterly meaningless as a linear language. Although some computer languages exist for electronic computer music, strict composition has usually been done with pen and paper. If a two-dimensional language form were created, composition (and allied tasks such as harmonic analysis) might be made easier. Standard musical notation, used to specify the music, could be combined with waveforms and other more modern conventions. Again, this is a possibility that could not be considered outside of a two-dimensional universe.

5. The Trans-Linguistic Parser.c.IV. The Trans-Linguistic Parser;

The Trans-Linguistic Parser was constructed to be the heart of a domain-oriented language engine. It fulfils the role of metalanguage compiler or translator. Although it is a tool with more general uses, in this context its primary function is to translate from a very-high-level DOL into a medium- or low- level language such as C or Icon. The following is a brief summary of its' capabilities and weaknesses.

The parser is designed to work on a BNF-like metalanguage, somewhat like YACC. This metalanguage uses BNF forms: square brackets delineate optional tokens, braces indicate one or more tokens, angle brackets delimit syntax variables, and the meta-equals sign "::=" is used as the assignment statement. Several other conventions apply. The parser uses implied tokenization, which means that tokens need not be specified, but are assumed to be parsed as they are listed in a definition. Variable tokens can be defined as a dollar sign followed by character variables.

The most important part of the parser is its ability to transform a parsed sequence of tokens into a different sequence of tokens, i.e. translating from one syntax into another, perhaps totally different one.

In effect, this enables one to translate from one computer language to another. The TransArrow ("-->") symbol serves as the translating token in a translation equation. The TransArrow indicates to the parser that, given the input sequence from the right hand side (rhs) of the translation equation, the output should be the left hand side (lhs) of the translation equation. Any syntax variables on the rhs will have their corresponding outputs inserted into the lhs of the translation equation. The following example may help to clarify this:

This code is an example of input to be translated into the Icon programming language.

```
d / d T T ^ 3 + T ^ 4 ~
d / d q cos ( sin ( 4 q ^ 6 + 3 x ( q ^ 9 ) ) ) ~
d / d q ln ( cos ( sin ( tan ( ln ( cos ( cos ( 4 y ^ 9 - 3    q ^ 2938 (
2 z ^ 5 ) ) ) ) ) ) ) ) ~
d / d i ln ( 3 x ^ 7 cos ( 4 y ) ) ~
```

The above code is parsed by THE PARSER according to the following metaprogram:

```
START <differentiand> -->;
<differentiand> ::=  d / d <var> {<function>, <polynomial>};
           -->  diff( " {<function>, <polynomial>} " \, " <var> " );
<function> ::=  <fn_type> ( {<function>, <polynomial>} ) -->;
<fn_type> ::= {cos, sin, tan, exp, ln};
<polynomial> ::= <polyterm> [( <polynomial>1 ) , <sign> <polynomial>2] --
>;
<polyterm> ::= [<val>1] <var> [^ <val>2] [<function>] -->;
<val> ::= $(0-9.);
<var> ::= $(A-Za-z);
<sign> ::= ${/+\-};
```

The parsing/translation gives the following result:

```
procedure main()
diff("T^3 +T^4 ","T")
diff("cos(sin(4 q^6 +3 x (q^9 )))","q")
diff("ln(cos(sin(tan(ln(cos(cos(4 y^9 -3 q^2938 (2 z^5 ))))))))","q")
diff("ln(3 x^7 cos(4 y ))","i")
end
```

(Observe that some of the rules in the metaprogram end with the TransArrow. This is merely an indication that the rule should return as output the input it finds in the program, after the program is parsed.)

The Trans-Language Parser is a versatile tool, and I suspect that this example only scratches the surface as to what it can do. For example, the above grammar can be extended to do actual differentiation of exponents and trigonometric functions. It is also possible to extend the parser for use as an inter-language translator. For example, one should be able to write a grammar that translates Pascal to C.

Despite its flexibility, the Trans-Language Parser has some weaknesses that need to be remedied in the next version of the software. The most obvious of these is that both the grammar and the higher-level language source code require all tokens to be separated by spaces. Such a limitation is counter-productive to entering code, and will be eliminated in the next version of the parser. However, since the white space delimitation in the metaprogram is necessary for implied tokenization, it will not be eliminated in the metalanguage. The parser also lacks the capacity to use global variables in its grammars.

6. Conclusion and Further Research.c.V. Conclusion;

In this report I have explored the problem of human-computer communication, particularly that of programming. As people of many diverse backgrounds are required to program computers, it is necessary to find programming languages that can be used effectively by each of them. Many currently popular programming languages fail to meet the needs of these new programmers, because they are based on technical considerations from the 1950s and 60s. Others fail because they are based on the philosophy and conceptual model of computer scientists.

New languages must be developed to meet the needs of a vast array of computer programmers, most of whom have little or no training in computer science. These languages need to be designed, from the start, to address primarily the needs of users. Rather than concentrating on implementation, the new languages should ignore implementational detail and let users express themselves in ways that are conceptually efficient and effective.

Even if new general-purpose languages are designed, language users in many fields will still experience problems. Most users in complex domains such as chemistry, and physics do not have the skill requisite to program a computer with the conceptual models used by their professions. In order to make use of a computer, they must either have a series of computer applications written for them, or a better computer language. Domain-oriented languages, programming languages that include the conceptual model of a domain, will help them fulfill their goals, yet not tie them down to the limitations of applications programs.

The implementation of new languages requires that a series of new tools be developed to facilitate that development. These tools must bridge the gap between conventional programming languages and the new, human-oriented ones. Programming optimizations traditionally done by human programmers need to be automated, to the extent that the users' ideas and requests can be translated by the computer into a formal program for compilation.

Some languages that adhere to these principles already exist; more are sure to follow. As computers become ubiquitous in society, user-oriented and domain-oriented languages will become the norm for all but computer scientists and associated disciplines.

Further research opportunities are plentiful in this field. Automation of the programming process will require the automation of many optimizations now done by humans. These optimizations need to be studied, and algorithms to implement them on computers need to be designed. Practically workable domain-oriented languages will require that these optimizations be done in the most efficient ways possible. One approach worthy of follow-up, at least in the near future, is the multi-stage translation process. The trans-linguistic parser is a first step in this direction.

Design and implementation of conceptual models in a programming language will require careful research in whatever areas are being considered. This may include further research into human psychology and aptitudes, methods of programming and conceptual model specification, and new interaction technologies such as virtual reality devices.

Finally, the construction of domain-oriented languages and DOL engines will be large projects requiring many person-years of labour. This will encompass all of the procedures outlined in section four. The potential gains in productivity promised by this project will make it a worthwhile endeavour.

Acknowledgments.c.Acknowledgments;

This research was supported by an NSERC undergraduate scholarship.

I would like to thank my supervisor, Nalin Wijesinghe, for his help and support in my obtaining the NSERC research grant, in the research itself, and in the preparation of this report. Your generous spirit and wise direction were greatly appreciated.

Thanks to Dr. X. D. Yang for his direction and guidance. Thanks also to Dr. A. G. Law and Dr. Howard Hamilton for their help in applying for funding and co-ordinating the publication of this report.

Appendix A: the Trans-Linguistic Parser
Version 1.0, August 1993.

```
# Trans-Linguistic Parser
# Version 1.0
# August, 1993
# (c) 1993 by Chris Hecker
# All Rights Reserved.
# The TLP translates very high level language text into
#  Icon code by the use of user-supplied BNF-like grammars.

global endchar, tokensep, whitespace, punctuation, digits, listsep

global clist, olist, vchar, mlist #delimiters

global tokentable, transformtable, backslash, syntax_var, errtbl  #
tables

global read_token, var_stack, line_count      # process variables

record delim(ldelim, separator, rdelim, desig)
record syntax_rec(chars, b_type, reps)

# main() is used to parse arguments and handle files.

procedure main(arg)

  local filelist, filecount, parsefileno, outfileno, current, param, file

  initialize_globals()

  # parse arguments to the parser.

  filelist := []
  filecount := parsefileno := outfileno := 0

  every current := !arg do current ? {
      if tab(any('-')) then case param := tab(0) of {
            "p": parsefileno := filecount + 1
            "o": outfileno := filecount + 1
            default: err(10)
      }
      else {
            case (filecount +:= 1) of {
                  parsefileno:  {parsefile := open(current, "r") | err(11)
```

```
                              pfname := current }
                 outfileno  :  {outfile := open(current, "w") | err(12)
                              ofname := current }
                 default    :  put(filelist, current)
                 # what happens if file not there?
            }
        }
  }


# parse files

  write("SDL parser version 0.1 (c) 1993 by Chris Hecker")

  tokenize(parsefile)
  close(parsefile)

  write("Finished Tokenizing.")

  write(outfile, "procedure main()")

  every file := !filelist do {
     write("** Parsing file \"", file, "\".")
     infile := open(file, "r") | err(13)
     parse_code(infile, outfile)
     close(infile)
  }
  write(outfile, "end")
  close(outfile)

  write("Finished Parsing.")
  exit(0)
end


procedure initialize_globals()

  # special csets

  endchar := ';'
  whitespace := ' \l\n\r\t'
  punctuation := ''
  tokensep := punctuation ++ whitespace
  listsep := ','
  digits := '0123456789'

  # delimiters

  clist := delim('[', ',', ']', 0)
  vchar := delim('<', '', '>', 2)
  olist := delim('{', ',', '}', 1)
  mlist := delim('(', ',', ')', 3)

  # the variable-matching stack
  var_stack := []

  # parsing rule storage tables

  tokentable := table([])         # holds patterns to match
  transformtable := table([])           # holds pattern to output

  # substitutes for backslashes in grammar.

  bkslsh_init := [ ["\\[", "["], ["\\]", "]"], ["\\{", "{"], ["\\}",
"}"],
             ["\\\\", "\\"], ["\\,", ","] ]

  backslash := table("")

  every entry := !bkslsh_init do backslash[entry[1]] := entry[2]

  # prepares the syntax variable table

  syntax_var := table(0)

  # reads in error codes, descriptions.

  errtbl := table("Undefined Error")
  errfile := open("SDLp.err", "r")

  while line := read(errfile) do line ?
      errtbl[numeric(tab(many(digits))) ] :=  (tab(many('\t ')), tab(0))

  close(errfile)
end


procedure tokenize(parsefile)
  while rule := read_rule(parsefile) do rule_tokenize(rule)
  return
end


procedure read_rule(parsefile)

  text := ""

  while (char := reads(parsefile, 1)) do
      if char == endchar then {
            suspend text || " "
            text := ""
      }
      else text ||:= char

end
```

```
procedure rule_tokenize(rule)

  token_list := []
  index := -1
  rule ? {
     tab(many(whitespace))
     primetoken := tab(upto(tokensep))
     while &pos < *&subject do  {
           tab(many(tokensep))
           bracket_type := if tab(any(clist.ldelim)) then clist
                       else if tab(any(olist.ldelim)) then olist
                       else &null
           if \bracket_type then {
                 result := parse_list(tab(upto(\bracket_type.rdelim)))
                 put(token_list, push(result, \bracket_type.desig))
                 move(1)
           }
           else {
                 t_l := tab(upto(tokensep))
                 if *t_l = 0 then next
                 case t_l[1] of {
                       "\\"   : put(token_list, backslash[t_l])
                       "$"   : { n := process_syntax_variable(t_l[2:0])
                               put(token_list, "$" || string(n))
                             }
                       default: case t_l of {
                          "::=" : &null
                          "-->" : index := *token_list
                          default: put(token_list, t_l)
                       }
                 }
           }
     }
  }

  tokentable[primetoken] := token_list[1: index + 1]
  transformtable[primetoken] := if index = *token_list
        then list_copy(token_list[1: index + 1])
     else token_list[index + 1 : 0]
  return
end


procedure parse_list(text)

  option_list := []          # no recursive lists.

  text || "," ? repeat {
     tab(many(whitespace))
     if &pos > *&subject then break

     while segment := tab(upto(listsep)) do {
        token_list := []
        segment ? repeat {
           t_l := tab(upto(whitespace) | 0)
           case t_l[1] of {
                 "\\"   : put(token_list, backslash[t_l])
                 "$"   : { n := process_syntax_variable(t_l[2:0])
                         put(token_list, "$" || string(n)) }
                 default: put(token_list, t_l)
           }
           tab(many(whitespace) | 0)
           if &pos > *&subject then break
         }
         if *token_list = 1 then option_list |||:= token_list
         else put(option_list, token_list)
         move(1)
         tab(many(whitespace))
     }
  }

  return option_list
end


procedure process_syntax_variable(syntax_descr)

  static syntax_var_count
  initial syntax_var_count := 0
  syntax_rec_list := []

  syntax_descr ? while any(&cset) do {
     repetitions := numeric(tab(many(digits))) | 1
     every bracket_type := (clist | olist | mlist) do {
           if tab(any(bracket_type.ldelim)) then {
                 result := create_cset(tab(upto(bracket_type.rdelim)))
                 put(syntax_rec_list, syntax_rec(result,
                       bracket_type.desig, repetitions))
                 move(1)
                 break
           }
     }
  }
  if syntax_rec_list ~=== [] then {
     syntax_var[return_val := syntax_var_count +:= 1] := syntax_rec_list
     return return_val
  }
end


procedure create_cset(text)

  expr_cset := ''

  text ? while char1 := move(1) do {
     char1 := cset(if char1 == "\\" then move(1) else char1)
     expr_cset ++:= char1
     if tab(any('-')) then {
           char2 := move(1)
           every char := ascii(char1) to ascii(char2) do
                 expr_cset ++:= cset(chr(char))
     }
  }
  return expr_cset
end


procedure detokenize(token_list, char_count, out)

  alpha := &letters ++ &digits ++ ','

  every item := token_list[index := 1 to *token_list] do {
     next_index := if index < *token_list then index + 1 else &null
     if type(item) == "list" then {
           args := detokenize(item, char_count, out)
           out := args[1]
           char_count := args[2]
     }
     else {
           char_count +:= word_length := *item
           if char_count > 78 then {
                 char_count := word_length
                 out ||:= "\n"
           }
           out ||:= item || if *(cset(item[-1:0]) ** alpha) = 0  |
                    *(cset(token_list[\next_index][1]) ** alpha) = 0
                    then "" else " "
           #vtrace("Cset1: ", ( cset(item[-1:0]) ** alpha))
           #write("Cset2: ", (cset(token_list[\next_index][1]) **
alpha))
           char_count +:= 1
     }
  }
  return [out, char_count]
end



# This section parses the input file(s) and stores the output.

procedure parse_code(infile, outfile)

  start := tokentable["START"][1]
  line_count := 0

  get_scan_line(infile) ?
    while scan_for_token() do {
     line_count +:= 1
     translation := parse(start)
     if *translation = 0 then {
           err(1001, &subject, &pos)
           allow_next_token()
            line_count -:= 1
            next
      }
      out_text := detokenize(translation, 0, "")[1]
      write(outfile, out_text)
    }
  return
end


procedure parse(matchtoken)

  matchlist := list_copy(tokentable[matchtoken])
  if *matchlist = 0 then err(20, matchtoken)
  result_list := list_copy(transptr := transformtable[matchtoken])

  if *transptr ~= 0 then push(var_stack, [])

  every index := 1 to *matchlist do {
      result := &null
      token := matchlist[index]
      vtrace("parse::loop", token, matchlist, index)
      if type(token) == "list" then {
            result := option_parse(token)
            vtrace("parse::list", matchlist, result)
            if /result then {
                  result_list := []
                  break
            }
            if *result = 0 then next
            else if *transptr = 0 then put(result_list, result)
      }
      else if any(vchar.ldelim, token) then {
            var := upto(vchar.rdelim, token) + 1
            result := parse(token[1:var])
            if *result = 0 then {
                  result_list := []
                  break
            }
            put(var_stack[1], [token, peel(result)])
      }
      else if any('$', token) then {
            if t := match_for_svar(syntax_var[numeric(token[2:0])]) then
                  if *transptr = 0 then put(result_list, t)
            else {
                  result_list := []
                  break
            }
      }
      else if scan_for_token() == token then {
            if *transptr = 0 then put(result_list, matchtoken)
            allow_next_token()
      }
      else {
            vtrace("parse::notokenmatch", token, matchlist)
            result_list := []
            break
      }
  }
  if *transptr ~= 0 then {
      replace(result_list, xx:=pop(var_stack))
      reduce(result_list)
  }
  vtrace("parse::2", result_list)
  return result_list
end

procedure option_parse(token_list)

  option_list := list_copy(token_list)
  result_list := []
  if *option_list = 0 then err(21)

  start := if list_type := \numeric(option_list[1]) then 2 else 1
  list_type := \list_type | -1  # change above to if type == integer

  every matchtoken := option_list[start to *option_list] do {
      token := scan_for_token() | break
      if type(matchtoken) == "list" then {
            result := option_parse(matchtoken)
            if *result = 0 then {if list_type = -1 then return [] else
next}
            else put(result_list, result)
      }
      else if any(vchar.ldelim, matchtoken) then {
            var := upto(vchar.rdelim, matchtoken) + 1
            result := parse(matchtoken[1:var])
            if *result = 0 then next
            put(var_stack[1], [matchtoken, peel(result)])
      }
      else if any('$', matchtoken) then {
            if t := match_for_svar(syntax_var[numeric(token[2:0])]) then
{
                  put(result_list, t)
            }
            else case list_type of {
                  clist.desig : next
                  olist.desig : next
                  -1          : return []
            }

      }
      else  if matchtoken == token then {
            put(result_list, matchtoken)
            allow_next_token()
      }
      else case list_type of {
            clist.desig : next
            olist.desig : next
            -1          : return []
      }
      case list_type of {
            clist.desig : return result_list
            olist.desig : return result_list
            -1        : next
      }
  }
  if list_type = olist.desig & *result_list = 0 then fail
  return result_list
end


procedure match_for_svar(matchlist)

  if *matchlist = 0 then fail
  ret_val := ""

  scan_for_token() ? {
      every item := !matchlist do {
            sequence := &null
            vtrace(item.b_type, clist.desig, olist.desig, item.reps)
            sequence := case item.b_type of {
                  (clist.desig | olist.desig) :# imperfect implementation
                        tab(|any(item.chars)\item.reps)
                   mlist.desig : tab(many(item.chars))
                   default : &null
            }
            if /sequence & item.b_type ~= clist.desig then fail
            else ret_val ||:= sequence
      }
      remainder := tab(0)
  }
  if *remainder = 0 then allow_next_token()
  else read_token := remainder

  return ret_val
end


procedure replace(result_list, substitution_list)

  every item := result_list[index := 1 to *result_list] do
      if type(item) == "list" then replace(item, substitution_list)
      else if any(vchar.ldelim, item) then {
            every varb := !substitution_list do
                  if varb[1] == item then {
                        result_list[index] := varb[2]
                        vtrace("replace", varb[1], varb[2])
                  }
      }
  # need to see what happens to substitution_list.
  return
```

```
end

procedure reduce(rlist)

  if type(rlist[1]) == "integer" then {
     if rlist[1] == clist.desig | olist.desig then {
           every item := rlist[index := 2 to *rlist] do {
                 found := &null
                 if type(item) == "list" then {
                       rlist[index] := reduce(item)
                       if *rlist[index] = 0 then next
                       found := 1
                 }
                 else if any(vchar.ldelim, item) then next
                 rlist := item
                 found := 1
                 break
                 }
           if /found then rlist := []
     }
  }
  else every item := rlist[index := 1 to *rlist] do
     if type(item) == "list" then rlist[index] := reduce(item)
     else if any(vchar.ldelim, item) then {
           rlist := []
           break
     }
  return rlist
end

procedure scan_for_token()

  if /read_token then {
     tab(many(whitespace))
     if read_token := \tab(upto(tokensep)) then return read_token
     if &subject := \get_scan_line() then {
           &pos := 1
           return \scan_for_token()
     }
     else fail
  }
  else return read_token
end

procedure allow_next_token()

  read_token := &null
  return
end

procedure get_scan_line(infile)

  static oldfile
  initial oldfile := &input
  repeat {
     vtrace("get", infile, outfile)
     if /infile then infile := oldfile
     if input := \read(infile) then {
           oldfile := infile
           return input || " "
     }
     else break
     vtrace("get2", infile, outfile)
  }
end

procedure peel(plist)

  if type(plist) == "list" then
     if *plist = 1 then
           return peel(plist[1])
     else return plist
  else return plist
end


procedure list_copy(list_to_copy)
  x := list_copy2(list_to_copy)
  return x
end

procedure list_copy2(list_to_copy)
  newlist := copy(list_to_copy)
  every item := newlist[index := 1 to *newlist] do
     if type(item) == "list" then newlist[index] := list_copy2(item)
  return newlist
end

procedure ascii(val)

  return find(string(val), string(&ascii))
end

procedure chr(val)

  return string(&ascii)[val]
end

procedure err(no, arg1, arg2, arg3)

  write("SDL parser: error ", no, " -- ", errtbl[no] | "Undefined Error")
  errclass := no / 10
  case errclass of {
     0 | 1 :    { write("fatal error, terminating.")
             exit(0) }
     2 :   { write("error in token: ", arg1)
             write("fatal error, terminating.")
```

```
               exit(0) }
       100 : { write("line ", line_count, ": ", arg1)
               every 1 to arg2 + 7 + *string(line_count) do writes("-")
               write("^")
             }
       default:   &null
  }
  return
end

# debugging and diagnostic routines.


procedure lipri(lst, tab) #list-print

  every item := !lst do
      if type(item) == "list" then lipri(item, tab || " ")
      else write(tab, item)
  return
end

procedure tapri(tab) #table-print
  tab1 := sort(tab)
  every pair := !tab1 do {
      write(pair[1], ":")
      case type(pair[2]) of {
            "list" : lipri(pair[2], "&")
            "co-expression" : vtrace("tapri", pair[2])
            default:   write(pair[2])
      }
      write("^^^^^")
  }
  return
end

procedure vtrace(a, b, c, d, e, f, g, h); return; end
```

Extended Bibliography.c.Extended Bibliography;

The following books and articles were used in my research and in the preparation of my ideas and this report.

[BACKUS78] Backus, John. "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs" in Communications of the ACM, Vol. 21 no. 8, August 1978, pp. 613-641.

[BALZER85] Balzer, Robert. "A 15 Year Perspective on Automatic Programming" in IEEE Transactions on Software Engineering, Vol. SE-11, no. 11, Nov. 1985, pp. 1257-1267.

[BEEKMAN90] Beekman, George. HyperCard in a hurry. (Belmont, Calif. : Wadsworth Pub. Co., 1990).

[BJORNER78] Bjorner, D., and Jones, C.B., eds., The Vienna Development Method: the Meta-Language. Vol. 61 in Goos, G. and Hartmaanis, J., Lecture Notes in Computer Science (Berlin: Springer-Verlag, 1978).

[CHOMSKY72] Chomsky, Noam. Language and Mind, enlarged edition. (New York: Harcourt Brace Jovanovich, 1972).

[COHEN90] Cohen, Edward. Programming in the 1990s: An Introduction to the Calculation of Programs. (New York: Springer-Verlag, 1990).

[EHRIG92] Ehrig, H., et al. "Introduction of Algebraic Specification. Part 1: Formal Methods for Software Development" and " ... Part 2: From Classical View to Foundations of System Specifications." in The Computer Journal, Vol. 35, no. 5, 1992, pp. 465-477.

[GRISWOLD90] Griswold, Ralph E. and Griswold, Madge T.. The Icon Programming Language 2nd ed. (Englewood Cliffs, N.J. ; Toronto : Prentice Hall, 1990).

[KENT78] Kent, William A. Data and Reality: Basic Assumptions in Data Processing Reconsidered. (Amsterdam: North-Holland, 1978).

[KIPLING84] Kipling-Brown, Ann, and Parker, Monica. Dance Notation for Beginners. (London : Dance Books Ltd., c1984.)

[KLERER91] Klerer, Melvin. Design of Very High-Level Computer Languages : A User-Oriented Approach 2nd ed.. (New York : McGraw-Hill, 1991.)

[LENK81] Lenk, John D. Understanding Electronic Schematics. (Englewood Cliffs, N.J. : Prentice-Hall, 1981.)

[MEALY78] Mealy, George H. "Notions", in [YEH78], pp. 12- 29.

[MEYER81] Meyer, R. E., "How Novices Learn Computer Programming" in Computing Surveys, vol. 13 no. 1, March 1981, pp. 13 - 31.

[NORMAN88] Norman, Donald A. The Psychology of Everyday Things. (New York: Basic Books, 1988).

[POLYA57] Polya G. How To Solve It: A New Aspect of Mathematical Method, 2nd ed. (Garden City, NY: Doubleday Anchor, 1957).

[SALOMON92] Salomon, D. J., "Four Dimensions of Programming-Language Independance", in ACM SIGPLAN Notices, V. 27, no. 3, March 1992, pp. 35-53.

[SCHWARTZ78] Schwartz, J. T., "Program Genesis and the Design of Programming Languages", in [YEH78], pp. 185-215.

[SHNEIDERMAN80] Shneiderman, Ben. Software Psychology : Human Factors In Computer and Information Systems. (Cambridge, Mass. : Winthrop Publishers, c1980.)

[YEH78] Yeh, R. T., ed.   Current Trends in Programming Methodology. (Englewood Cliffs, N.J. : Prentice-Hall, 1978.)