THE MYSTRO SYSTEM:

A COMPREHENSIVE TRANSLATOR TOOLKIT

by

W. Robert Collins & Robert E. Noonan

College of William and Mary
Williamsburg, Virginia 23185

June 1985

(NASA-CR-176350) THE MYSTRO SYSTEM: A COMPREHENSIVE TRANSLATOR TOOLKIT Final Report (College of William and Mary) 37 p HC A03/MF A01 CSCL 09B

N86-16913

Unclas
G3/60 04831

FINAL REPORT

NASA Langley Research Center
Grant NSG-1435

TABLE OF CONTENTS

INTRODUCTION

Mystro is a system that facilities the construction of compilers, assemblers, code generators, query interpretors, and similar programs. It provides features to encourage the use of iterative enhancement[4]. Mystro was developed in response to the needs of NASA Langley Research Center (LaRC) and enjoys a number of advantages over similar systems.

There are other programs available that can be used in building translators[5 12 21 28]. These typically build parser tables, usually supply the source of a parser and parts of a lexical analyzer, but provide little or no aid for code generation. In general, only the front end of the compiler is addressed. Mystro, on the other hand, emphasizes tools for both ends of a compiler.

Even in the construction of the front end of a compiler, our experience with other systems has been that the tedious task of table generation is always automated, but the equally tedious task of managing the text of a translator is largely ignored. A minor change to the grammar may require major changes throughout the translator. In addition to replacing the parser tables, the semantic actions may have to be reorganized, new constants controlling sizes in array declarations may be necessary, a new lexical analyzer may have to be developed, and so on. This is particularly frustrating when using iterative enhancement to implement a language incrementally.

Consequently, an important part of Mystro is the ease and safety with which an evolving translator can be updated. Professional programmers have been able to build translators rather quickly by taking advantage of Mystro's convenience. Students in our compiler construction classes at William and Mary have been able to add new language features to their project compilers as their skills and understanding increase. Using Mystro, they have been able to create compilers, complete with code generation for real machines, in one semester. Our experience has been that this was generally impossible before, even using a parser generator, simply because of the source text management problems.

Another distinguishing aspect of Mystro is its portability. The system can be obtained in versions explicitly designated for several machines, including PRIME's, VAX's, and CDC Cybers. Mystro is designed to be portable across a wide variety of machines without sacrificing features.

In the sections which follow, we discuss the history of the project, we summarize the novel aspects of Mystro, and finally, we give a brief overview of the current system. More complete documentation of Mystro is contained in the appropriate manuals.

HISTORY OF MYSTRO

The Mystro system began as an effort to develop an LR parser generator[1] [2] to facilitate the development of translators and compilers for embedded computers. Since NASA LaRC used CDC Cyber machines and William and Mary had an IBM 370/158, the resulting program had to be reasonably portable. A summary of the project milestones is given in Appendix A; in the remainder of this section we discuss only the highlights of this history.

The first version of the Mystro parser generator (PARGEN) produced SLR(1) parse tables whose table sizes, however, had to be manually edited into the parser program. It was quickly realized that having PARGEN automatically merge constants and other text into the translator under development would be a valuable addition and was added to the next version. At this early stage text management was recognized as an important part of Mystro.

In a separate code generation project (in the summer of 1978), Noonan[22] invented a language (erroneously) named CGGL based on earlier work by Donegan[13]. By late 1978 most of the effort was concentrated on developing the first CGGL translator using the parser generator as a tool. This was to become a standard mode of operation: using a newly developed tool in some other project would lead to further improvements in the tool.

By the spring of 1979 the CGGL translator was sufficiently complete to allow two test cases to be implemented: an Intel 8080 code generator for a small subset of HALMAT and a GE 703 code generator for a larger subset of HALMAT. (HALMAT is the intermediate code produced by the HAL/S front-end.)

Two conjectures were confirmed by the code generation experiments. Both PARGEN and CGGL were viable, useful tools, which necessitated the first set of user manuals, and the experience gained with the CGGL translator confirmed the importance of text management in Mystro. The fact that Mystro parsers needed a syntactic error recovery mechanism also became apparent.

A more powerful CGGL translator (Version 2) was proposed and some enhancements for the HALMAT code generators were implemented. The new code generators were table-driven; previously, the code generators used automatically-produced, voluminous Pascal code. This was a significant improvement for a non-virtual memory machine like the CDC Cyber. These HALMAT code generators became the basis for the CODEGEN skeleton.

Work was also begun on an Ada 79 compiler. To our knowledge the Ada 79 parser produced at William and Mary was the first full parser which did not change the syntax of the language.

In 1979, both PARGEN and CGGL were transported to NASA LaRC and installed on the CDC computers by Computer Sciences Corporation. From this point onward, the CDC (NASA) and IBM (W&M) versions of Mystro began to diverge, with much cross fertilization of ideas but little reuse of code. The divergence arose from the lack of a standard for Pascal and from differences in the CDC and IBM computers and operating systems.

By the summer of 1980, Version 4 of PARGEN had been produced separately both at NASA LaRC and at William and Mary. The implementation of CGGL 2 at William and Mary was nearly complete, although improvements would continue to be added for another two years. On the parser side, the first really good syntactic error recovery scheme based on a one-token repair had been developed and incorporated into the standard parser skeleton. This error recovery scheme was retrofited into the CGGL translator. A panic mode error recovery scheme based on the notion of fiducial symbols was also incorporated into the existing parsers.

However, the Ada effort was set back when both the syntax and semantics of the language were revised at the Ada debut held in September 1980. While the grammar for Ada was easily revised, this necessitated many semantic changes in the ongoing compiler effort.

In the summer of 1981 Mike Donegan left William and Mary for Rice University. Most of the original code in PARGEN and CGGL was written by Mike. With his leaving the Ada compiler effort effectively died, although some work would continue for more than a year. However, the incomplete Ada compiler served as the basis for later compiler skeletons.

In 1982 we undertook the conversion of Mystro from the IBM computer to William and Mary's new Prime supermini computers. As part of this conversion, it was decided to merge the best features of the IBM and CDC versions of Mystro. Furthermore, a single master copy of Mystro was kept at William and Mary and specific versions extracted. This has proved to be an effective scheme.

Some of the portability considerations that were incorporated included identifier length, support for underscores in identifiers, interface to the file system, presence of a value statement, etc. A site-specific Mystro system, Version 6, contained approximately 15K source lines of Pascal, while the all-site system contained almost 22K lines. Several utilities were constructed to aid in producing a specific copy of Mystro.

In 1983, a tree transformer skeleton was generated; this tool was motivated by a compiler done by Collins and Knight for a Pascal-like language for the Intel 8748 chip[8]. As part of this effort, PARGEN and associated parsers were enhanced to allow for syntactically ambiguous grammars, whose parse could be disambiguated using semantic information. This has proved to be one of the most useful and powerful enhancements made.

In 1984, PARGEN was upgraded from NQLALR to full LALR, using an algorithm that uses less space and runs considerably faster. In addition, a new panic mode error recovery scheme was incorporated into the compiler skeletons. The one-token error recovery scheme was augmented with a spelling error corrector and the ability to always back up one token.

NOVEL FEATURES

Mystro includes a number of novel features that are not found in other parser generator systems. These have evolved from the research and development activities undertaken by NASA LaRC. Each of these features will be discussed in turn.

Portability

Portability of the entire system has been a major consideration from the beginning. Most other systems have ignored the portability issue. For example, YACC[21] will run only under the UNIX™ operating system.

Mystro was developed on an IBM 370/158 and then ported to CDC computers at NASA LaRC. These machines differ widely: they have different character sets, different word sizes, one has virtual memory and the other does not, and so on.

In order to miminize portability problems, Pascal was chosen as the implementation language, despite the lack of a standard Pascal. Despite the current Pascal standard, problems still persist. Some compilers (e.g., Berkeley Pascal) are not compatible with the ANSI Pascal Standard[3].

Other problems result from the fact that the new Pascal Standard closely follows the original definition of Pascal promulgated

™UNIX is a trademark of A T & T Bell Laboratories.

in 1975[20]. Many widespread language features are simply omitted from the standard. For example:

- separate compilation,
- the value statement,
- the otherwise construct on a case statement,
- the use of underscores in identifiers.

Additional problems result from compiler-dependent issues:

- the presence and syntax of include directives,
- the presence and syntax of compiler options,
- the number of significant characters in an identifier,
- the maximum size of a set,
- character set dependencies.

Problems are introduced by the computer and/or operating system:

- the presence or absence of virtual memory,
- the interface to the file system,
- the interface (if any) to the command line.

Mystro has consciously addressed all these issues, although our solutions may not please everyone. Further discussion of these issues is contained in the Mystro Installation Guide[24].

Skeletons

Mystro excels as a system in providing standard collections of partial translators or parsers, known as skeletons. Although the skeletons must be enhanced with application-dependent code, they have many useful utility routines: for converting digit strings to numbers, constructing symbol tables, producing a cross reference, etc. These skeletons are equipped with many useful debugging aids, which have proved invaluable to those with a minimal knowledge of LR parsing.

Since the skeletons are (usually) complete programs, developers can execute PARGEN on their grammars, then compile and execute resulting applications. Before adding "semantics," developers can present sample input, trace the parse tree as it is built, and otherwise "test" their grammars.

Skeletons are provided for constructing compilers, assemblers, code generators, query interpretors, tree transformers, and menu-based programs. Although all are LR parsers, each skeleton differs considerably from the others. For example, the compiler skeleton contains routines for maintaining block-structured symbol tables and for full syntactic error recovery; the others do not. The tree transformer's scanner and parser repeatedly read and parse a tree until no further changes are possible.

Complete details of the various skeletons are provided in the Mystro Skeletons Reference Manual[25].

Iterative Enhancement

Many parser generators, such as YACC and Mystro, provide a mechanism for translating references to grammar sysmbols in the semantic code to references to a semantics stack in the executing parser. A few, such as LR[28], provide no such help with semantics.

Again, the support provided by Mystro is superior to other systems with which we are familiar.

Consider YACC: it provides for references to a semantics stack, as does Mystro. However, YACC uses a positional notation rather than a named notation. For example, consider the following production:

<expr> ::= <expr> + <term>

In the associated semantic code, YACC would refer to the **<term>** as "**$3**," while Mystro uses "**<term>** ." If at some later time, this production should be modified so that **<term>** no longer appears in the production, YACC would still have a valid but erroneous reference, while Mystro would detect the error at translation time.

Mystro provides the translation facility not through the parser generator (as does YACC), but through a separate utility called DEREF. This allows a user to modify the parser directly, rather than edit the grammar and then re-execute the parser generator. This is convenient whenever the grammar itself is not changing. The inverse utility RESTORE maps semantic stack references back to grammar references.

Another major difference occurs after the generated translator has undergone many modifications. YACC provides no facility for extracting a grammar and associated semantics from a translator under development. Mystro provides the EXTRACT utility. For example, one application of EXTRACT was in the transition from CGGL 1 to CGGL 2.

In practice we find that EXTRACT gets heavily used because of our reliance on a development philosophy known as iterative enhancement[4]. Using this methodolgy we rapidly develop a prototype and put it in production. Based on experience gained with the prototype, enhancements are designed and added, and the new version put into production. And the cycle repeats itself. Unlike other uses of rapid prototyping, we almost never throw the prototype away, although it may get heavily modified on each iteration.

Ambiguous Grammars

Many parser generator systems allow some form of syntactically ambiguous grammar. For example, most allow shift-reduce conflicts to exist in a parser state, choosing to shift in all such instances. This provides a simple but effective solution to the "dangling else" problem in Pascal and other programming languages.

Unlike most systems, Mystro permits syntactically ambiguous grammars in which the ambiguity is resolved via semantic constraints. This feature has proven to be enormously powerful and useful. All of the skeletons support this form of ambiguity; a few, such as the tree transformer and Glanville[17] [18] skeletons, absolutely require it.

Our exploration of the power of semantically disambiguated grammars has literally just begun. Indeed, Collins and Feyock[10] have noted the similarity of grammars to the use of logic in Prolog. Using ambiguous grammars they have been able to produce a few simple expert systems for aircraft fault diagnosis.

Code Generation

A significant portion of the effort in Mystro has gone into the development of tools to support the code generation process. Much of what has been developed is a result of the early efforts to produce table-driven code generators for HALMAT and for Pascal/48.

One product of this research was the CGGL translator[14]. Besides being used in both the HALMAT and Pascal/48 code generators, it has been used extensively in the compiler construction course at William and Mary. It is also being used in the Modula 2 compiler under construction.

Another product to aid in code generation is the tree transformer skeleton, which has a number of uses. For example, in the Modula compiler it is used not only to "shape" the intermediate code tree just prior to code generation, but also to do all constant expression evaluation, to insert coercion operators where needed, and to simplify relational expressions. Appendix D has a tree transformer example.

Research continues on the development of a language (and translator) for expressing all of the information needed in doing tree transformation, code generation, and peephole optimization.

Syntactic Error Recovery

The early Mystro parsers contained no mechanism to handle syntax errors in the language being parsed. For example, the early CGGL translators aborted upon discovering a syntax error.

The earliest error-handling mechanism incoporated into Mystro parsers was analogous to the methods used in recursive descent translators and in YACC. In the presence of an error the input was scanned until a trustworthy symbol was recognized. This required popping several symbols off the parser stack, replacing them with some nonterminal; there was, however, no simple way to determine the "semantics" associated with the nonterminal. Although this scheme fixed the syntax error, the translator often aborted while processing the semantics of the phantom nonterminal.

Consequently, we established these criteria for a reasonable error recovery scheme.

1 The scheme must be automatically generated from the grammar. In particular, the Mystro user cannot be expected to be knowledgable of either LR parsing or syntactic error recovery.

2 The recovery mechanism must never back up the parser stack and implicitly undo semantic actions. However, input symbols may be freely deleted or modified in any way.

3 The quality of the repair should be commensurate with the error. In particular, one token errors should always be recognized.

What has emerged is a two level scheme. The first level attempts one token repairs in the neighborhood of the detected error. The second level attempts to repair the error by adding and/or deleting multiple input tokens.

In the first level, errors are corrected using a single token repair scheme[19]. The attempted repairs are:

1 Check to see if the current symbol is a misppelled reserved word.

2 Insert legal shift symbols before the current symbol.

3 Replace the current symbol with legal shift symbols.

4 Delete the current symbol

The first of these repairs which succeeds is accepted. A successful repair is one that allows the parser to continue for a fixed number of input tokens without detecting a subsequent error.

If none of these repairs succeed, then the parser backs up a token, if possible, and the entire process is repeated. Currently, the compiler skeleton parses out of phase with the semantics[6], ensuring that the parser can always back up at least one token. This is useful since an error may not be detected until one or more symbols past the point of the actual error.

If the second process fails to find a successful correction, then the best near-correction is taken, provided that such a correction is able to shift the symbol immediately past the original error point. The best near-correction is taken to be the first one that is able to shift the most symbols.

An example of level one recovery is given in Appendix C, using a subset Ada™ grammar. An especially interesting repair occurs on line 13 when the parser successfully corrects the misspelling of the reserved word BEGIN, which is not even detected until line 14.

The level two error recovery is based on the computation of continuation symbols[27] for each parser state. (Using separate error productions[15] is too expensive for an erroneous source program.) These symbols give the shortest possible legal input which can "complete" the current parse. Continuation input is matched against the actual input looking for a symbol in common. If a common symbol is found, then all input tokens up to this symbol are replaced by the required continuation symbols. Otherwise the continuation symbols are used as input, allowing the parser to complete normally.

The second example in Appendix C shows how level two or panic mode error recovery is able to cope with repairs requiring insertion or deletion of more than a single token.

™Ada is a trademark of the U.S. Government (Ada Joint Project Office).

CURRENT STATUS

The current Mystro system (Version 7.3) consists of major tools and utility support programs, Pascal code fragments called skeletons, sample grammars, and supporting documentation. In addition, a number of aids have been created to support the portability of the system.

PROGRAMS	PORTABLE?	LINES
Pargen	yes	4,849
CGGL	no	5,050
Treegen	no	3,823
Meta	no	800
Extract	yes	708
Deref	yes	1,034
Restore	yes	1,034
TOTAL		17,298

Table 1 - Tools and Utilities

PARGEN, CGGL, EXTRACT, DEREF, and RESTORE have already been described. TREEGEN generates abstract syntax trees directly from a grammar. The META utility converts BNF extended with iteration and alternation to standard BNF. These programs and their sizes (in deliverable lines) are enumerated in Table 1. The skeletons are in reality parsers that have been customized for application areas. For example, the two compiler skeletons include code for full

SKELETON	PORTABLE?	LINES
Compiler	yes	3368
Compiler Tree	yes	3289
Assembler	yes	1875
Tree Transformer	yes	871
Query	no	888
Menu	no	1567
Glanville	no	1034
Code Generator	no	326
Glanville Triples	no	1013
TOTAL		14,231

Table 2 - Skeletons

syntactic error recovery. These skeletons are enumerated in Table 2 and documented in the Mystro Skeletons Reference Manual[25].

Also included with Mystro are approximately twenty grammars of various kinds, including programming language grammars (Pascal and Ada), assembler grammars, and query grammars. Manuals and User Guides for the major Mystro components are summarized in Table 3.

To make Mystro portable across a number of compilers, we use portability tools available only at William and Mary. Some are Pascal programs and are potentially useful in other environments. Others are "scripts" of commands, which are used in many cases to

Manual	PAGES
Pargen	26
CGGL	33
Skeletons	26
Installation	6
Meta Program	6
Tape Documentation	2
TOTAL	99

Table 3 - Manuals

"glue" separate filters together into a single "command." These are enumerated in Table 4.

SUPPORT AIDS	TYPE	LINES
Version Extract	Pascal Program	444
Longline	Pascal Program	62
Fix Idents	Pascal Program	540
Crossref	Pascal Program	487
Port All	Command Proc	53
Port	Command Proc	64
Copy to Tape	Command Proc	37
TOTAL		1,687

Table 4 - Portability Aids

CONCLUSIONS

Under a number of distinct measures, the Mystro system is a success. With little publicity other than "word of mouth," Mystro has been distributed to more than thirty distinct sites, including universities, industry, and government. Indeed, NASA LaRC has versions running on the Cyber computers under NOS, on the Primes, on a VAX running VMS, on a VAX running UNIX, and on an Intellimac.

Many of our early sites had Cyber computers. More recently, most of the sites requesting Mystro are VAX's running UNIX, despite the availability of YACC. For example, the *Blaze* language project at ICASE is using Mystro for both the parser for the language and for code generation.

Mystro has been used on a large number of projects sponsored by William and Mary, NASA LaRC, and NASA Lewis Research Center (LeRC). A partial list of these (as reported to the Mystro group) includes:

Project	**Site**
Modula 2 compiler	W&M
Blaze compiler	ICASE
Pascal/48 compiler[8]	NASA LaRC
NSSC II assembler[9]	LaRC
SAGA Software Management System[7]	LaRC
various microcomputer assemblers[9]	W&M
executive, Intel database machine[16]	LaRC

SCMS interface language[26]	LaRC
generating Cyber and VPS JCL[26]	ICASE
hardware design in the FEM project	LaRC
H-Code machine simulator	LaRC
distributed Ada precompiler[11]	NASA LeRC
PL/STAR	LaRC
PL/99	LaRC
real-time expert systems[10]	LaRC

REFERENCES

1 Aho, Alfred V., and Johnson, Steven C. "LR parsing," *Computing Surveys,* 6 (June 1974), pages 99-124.

2 Aho, Alfred V., and Ullman, Jeffrey D. *Principles of Compiler Design.* Addison-Wesley, 1977.

3 *An American National Standard: IEEE Standard Pascal Computer Programming Language.* IEEE, 1983.

4 Basili, Victor R., and Turner, A. Joseph. "Iterative enhancement: A practical technique for software development." *IEEE Transactions on Software Engineering,* SE-1 (December 1975), 390-396.

5 Berger, W. F. "BOBSW 3.0 -- a parser generator." *University of Texas at Austin Technical Report 87,* (November 1978).

6 Burke, M., and Fischer, G. A. "A practical method for syntactic error diagnosis and recovery." *Proceedings of the SIGPLAN 1982 Symposium on Compiler Construction,* (June 1982), 67-78.

7 Campbell, Roy H., and Kirslis, Peter A. "The SAGA project: a system for software development." *SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments,* (April 1984).

[8] Collins, W. Robert, Knight, John C., and Noonan, Robert E. "A translator writing system for micro-computer high-level languages and assemblers." *NASA-AIAA Workshop on Aerospace Applications of Microcomputers,* (November 1980), 179-186.

[9] Collins, W. Robert, Noonan, Robert E., Gregory, Samuel T., Knight, John C., and Hamm, Roy W. "Comprehensive tools for assembler construction." *Software -- Practice and Experience,* 13, (1983), 447-451.

[10] Collins, W. Robert, and Feyock, Stefan. "Syntax programming, expert systems, and real-time fault diagnosis." *Proceedings of the 1985 Eastern Simulation Conference,* Norfolk, Virginia (March, 1985).

[11] Collins, W. Robert, Feyock, Stefan, King, Laurie A., and Morell, Larry. "Moving target, distributed, real-time simulation using Ada," *Proceedings of the 1985 Eastern Simulation Conference,* Norfolk, Virginia (March 1985).

[12] DeRemer, F., and Pennello, T. J. *The MetaWare™ TWS User's Manual.* MetaWare, Santa Cruz, Ca., 1981.

[13] Donegan, Michael K. "An approach to the automatic generation of code generators." Ph.D. Thesis, Rice University, 1973.

[14] Donegan, Michael K., Noonan, Robert E., and Feyock, Stefan. "A code generator generator language." *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction,* (August 1979), 58-64.

15 Fischer, C. N., and Mauney, J. "On the role of error productions in syntactic error correction. *Computer Languages,* 5 (1980), 131-139.

16 Fishwick, Paul A. "HILDA: The Flexible Design and Implementation of a Database Machine Executive." MS thesis, College of William and Mary, 1983.

17 Ganapathi, Mahadevan, and Fischer, Charles N. "Description-driven code generation using attribute grammars." *Ninth Annual ACM Symposium on Principles of Programming Languages,* (January 1982), 108-119.

18 Glanville, R. Steven, and Graham, Susan L. "A new method for compiler code generation." *Fifth Annual ACM Symposium on Principles of Programming Languages,* (January 1978), 231-240.

19 Graham, S. L., Haley, C. B., and Joy, W. N. "Practical LR error recovery." *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction,* (August 1979), 168-175.

20 Jensen, Kathleen, and Wirth, Niklaus. *Pascal User Manual and Report.* Springer-Verlag, 1975.

21 Johnson, S. C. "YACC -- Yet another compiler-compiler." *UNIX Programmer's Manual.* Bell Laboratories, (January 1979).

22 Robert E. Noonan and Patricia Timpanaro. "The application of software engineering techniques to the design of relatively machine-independent code generators." *NASA/AIAA Workshop on*

Tools for Embedded Computing Systems Software, (Nov. 1978), 45-48.

23 Noonan, Robert E., and Collins, W. Robert. "The Mystro System, Version 7.3: Parser Generator User's Guide." *College of William and Mary Technical Report 85-01,* 1985.

24 Noonan, Robert E., and Collins, W. Robert. "The Mystro System, Version 7.3: Installation Guide." *College of William and Mary Technical Report 85-04,* 1985.

25 Noonan, Robert E., and Collins, W. Robert. "The Mystro System, Version 7.3: Skeletons Reference Manual." *College of William and Mary Technical Report 85-01,* 1985.

26 Noonan, Robert E., and Collins, W. Robert. "Construction of a menu-based system," *Software -- Practice & Experience,* (to appear, 1985). Also *ICASE Report No. 85-16,* ICASE, NASA Langley Research Center, Hampton, VA.

27 Rõhrich, Johannes. "Methods for the automatic construction of error correcting parsers." *Acta Informatica,* 13 (1980), 115-139.

28 Wetherell, C., and Shannon, A. "LR -- Automatic parser generator and LR(1) parser." *IEEE Transactions on Software Engineering,* SE-7 (May 1981), 274-278.

APPENDIX A

PROJECT MILESTONES

Spring 1978	PARGEN, v. 1 -- produced SLR(1) parse tables.
Summer 1978	Proposal for a Code Generator Generator Language.
Fall 1978	PARGEN, v. 2 -- included table optimization and text merging phases.
Winter 1979	CGGL, v. 1 -- first production use of PARGEN.
Spring 1979	HALMAT (HAL/S intermediate code) subset code generator produced for Intel 8080 using CGGL.
Summer 1979	HALMAT subset code generator for GE703. Ada 79 parser produced -- first one in nation which did not change syntax of language.
Fall 1979	PARGEN and CGGL ported to the Cybers at LaRC. First PARGEN and CGGL manuals.
Winter 1980	First version of EXTRACT utility.
Summer 1980	PARGEN, v. 4, produced -- improved scheme for text management. Improved EXTRACT in use at W&M and LaRC. First versions of the DEREF and RESTORE utilities. CGGL, v. 2 -- produces table-driven code generator.
Fall 1980	One token syntactic error recovery incorporated into compiler skeleton and into CGGL translator. Four distinct skeletons in use. First program to help maintain Pascal code. System now approximately 15K lines of Pascal plus documentation.

Spring 1981	PARGEN, v. 5 — uses BNF grammars rather than van Wijngaarden notation grammars.
Summer 1981	Mike Doneganx, who wrote much of the original code in PARGEN and CGGL, leaves W&M and Mystro project.
Fall 1981	Bob Collins leaves CSC Mystro group, joins W&M Mystro project. System now consists of 3 manuals, PARGEN 5, CGGL 2, 5 utility programs, and 6 skeletons.
Spring 1982	First Prime version of Mystro. Complete rewrite of PARGEN (v. 6).
Summer 1982	First portable version of Mystro. Compilers supported include IBM, CDC, Prime, and ANSI standard. Total system now 22K lines. First utilities to help with portability.
Summer 1983	Support for semantic disambiguation of parsing added to PARGEN and all skeletons. Tree transformer skeleton produced.
Summer 1984	LALR computation in PARGEN rewritten. Glanville code generator skeleton produced. New panic mode syntactic error recovery added to compiler skeletons; also one token repair enhanced to include spelling error correction and the ability to always back up one token.
Spring 1985	Menu skeleton produced. Subset Modula compiler produced. System now approximately 30K lines of Pascal, excluding Modula compiler

Appendix B

Publications

Robert E. Noonan and W. Robert Collins. "The parser generator as an applications generator." Submitted for publication (1985). Also NASA Contractor Report (1985).

Robert E. Noonan and W. Robert Collins. "Construction of a menu-based system." Submitted for publication (1985). Also NASA Contractor Report 172560 (1985).

Robert E. Noonan. "An algorithm for generating abstract syntax trees." *Computer Languages,* (to appear, 1985). Also NASA Contractor Report 172541 (1985).

Robert E. Noonan and W. Robert Collins. "The Mystro System, Version 7.3: Installation Guide." *College of William and Mary Technical Report 85-04,* 1985.

Robert E. Noonan and W. Robert Collins. "The Mystro System, Version 7.3: Parser Generator User's Guide." *College of William and Mary Technical Report 85-01,* 1984.

Robert E. Noonan and W. Robert Collins. "The Mystro System, Version 7.3: Skeletons Reference Manual." *College of William and Mary Technical Report 85-03,* 1985.

Robert Noonan. "The Mystro System, Version 7.3: CGGL User's Guide 2.11." *College of William and Mary Technical Report 85-02,* 1985.

W. Robert Collins, Robert E. Noonan, Samuel T. Gregory, John C. Knight, and Roy W. Hamm, "Comprehensive tools for assembler construction," *Software - Practice and Experience*, Volume 13, Number 5 (May 1983), 447-451.

Robert E. Noonan. Experiences with a code generation tool. *Proceedings of IEEE COMPCON,* (Fall 1981), 98-105.

W. Robert Collins, Samuel T. Gregory, and Roy W. Hamm, "Supporting the execution of HAL/S programs at NASA LaRC," technical report to NASA Langley Research Center, 1981.

W. Robert Collins, John C. Knight, and Robert E. Noonan, "A translator writing system for micro-computer high-level languages and assemblers," *NASA-AIAA Workshop on Aerospace Applications of Microprocessors,* Bethesda, Maryland, (November 1980), 179-186.

W. Robert Collins, Samuel T. Gregory, and Roy W. Hamm, "An evaluation of two assembler construction tools," report to NASA Langley Research Center, 1980.

Michael K. Donegan, Robert E. Noonan, and Stefan Feyock. "A code generator generator language." *SIGPLAN Symposium on Compiler Construction,* (Aug. 1979), 58-64.

Robert E. Noonan and Patricia Timpanaro. "The application of software engineering techniques to the design of relatively machine-independent code generators." *NASA/AIAA Workshop on Tools for Embedded Computing Systems Software.* (Nov. 1978), 45-48.

Robert E. Noonan. "The design of relatively machine-independent code generators." *NASA Contractor Report 159016, Contract NAS1-14972, Task 14,* (Feb. 1979).

Appendix C

Sample Syntactic Error Recovery

The example below shows the power of the level one error recovery, which effects one token repairs. In particular, the misspelled *begin* on line 13 is not even detected until line 14.

```
  Line#    Source Line

     1   : procedure level_one is
     2   :    a, d : Integer; ;
*** Error                    ^ Unexpected symbol deleted.
     3   :    b    : inetger := 6;
*** Error                 ^ Misspelled "INTEGER" corrected.
     4   :    c      constant integer := 1;
*** Error                  ^ Missing ":" inserted before symbol.
     5   :    g    : booolean;
*** Error                  ^ Misspelled "BOOLEAN" corrected.
     6   :         : Integer;
*** Error          ^ Missing "<ID>" inserted before symbol.
     7   :    f    = integer := 9;
*** Error          ^ Unexpected symbol replaced by ":".
     8   :    z    : contsant integer := 4;
*** Error                  ^ Misspelled "CONSTANT" corrected.
     9   :
    10   :    proc new is
*** Error        ^ Unexpected symbol replaced by "PROCEDURE".
    11   :       i    : : Integer := 9;
*** Error                ^ Unexpected symbol deleted.
    12   :
    13   :    beggin
*** Error          ^ Misspelled "BEGIN" corrected.
    14   :       f := i + i;
    15   :       looop
*** Error           ^ Misspelled "LOOP" corrected.
```

```
   16   :          exit when f = 5;
   17   :          f = f + 1;
*** Error            ^ Unexpected symbol replaced by ":=".
   18   :        en loop;
*** Error           ^ Misspelled "END" corrected.
   19   :     end new;
   20   :
   21   : begin  -- level_one
   22   :     a := 2;
   23   :     iff b > a then
*** Error          ^ Misspelled "IF" corrected.
   24   :        a := b;
   25   :     esle
   26   :        a := a + 1;
*** Error          ^ Unexpected symbol deleted.
   27   :     end if;
   28   :     a = c;
*** Error         ^ Unexpected symbol replaced by ":=".
   29   : end level_one;

   16 errors found.
```

The second example is of level two (or panic mode) error recovery.

```
 Line#    Source Line

    1   : procedure panic_mode is
    2   :     a, b, d : integer;
    3   :
    4   :     procedure new is
    5   :     begin
    6   :        d := a + 1;
    7   :        loop
    8   :           d := d + 1;
    9   :           if d = 5 then
   10   :              a := 1;
   11   :     end;
*** Error          ^ Missing "if" inserted before symbol.
   12   :
   13   :     procedure next is
```

```
*** Error           ^ Missing "end" inserted before symbol.
*** Error           ^ Missing "loop" inserted before symbol.
*** Error           ^ Missing ";" inserted before symbol.
*** Error           ^ Missing "end" inserted before symbol.
*** Error           ^ Missing ";" inserted before symbol.
   14  :    begin  -- next
   15  :       a := 0;
   16  :    end;
   17  :
   18  : begin  -- panic_mode
   19  :    a := 2;
   20  :    if b > a then
   21  :       a := b;
*** Error             ^ Missing "end" inserted before symbol.
*** Error             ^ Missing "if" inserted before symbol.
*** Error             ^ Missing ";" inserted before symbol.
*** Error             ^ Missing "end" inserted before symbol.
*** Error             ^ Missing ";" inserted before symbol.

   14 errors found.
```

Appendix D

Sample Tree Transformer Grammar

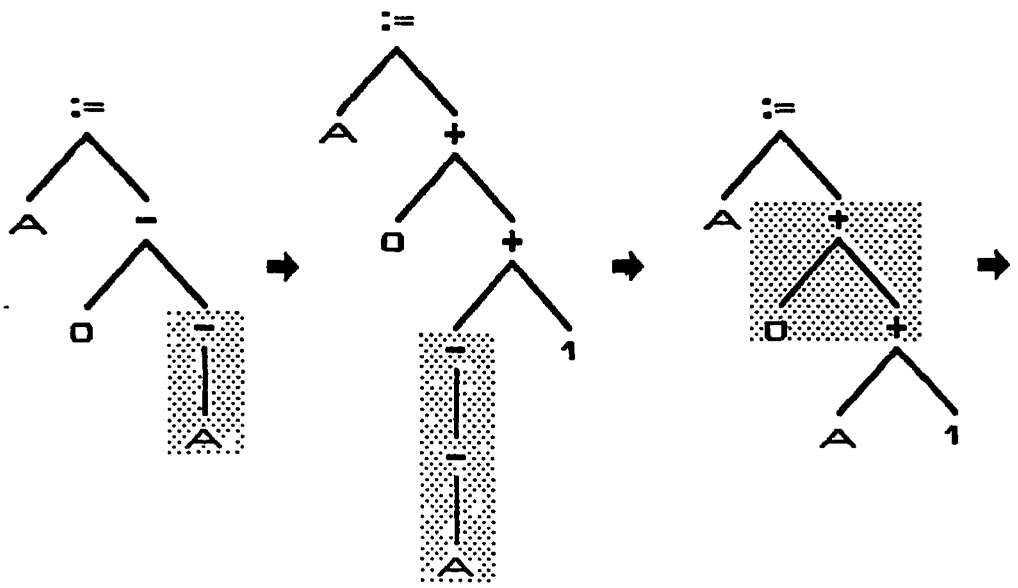
Consider the assignment statement

A := 0 – (–A)

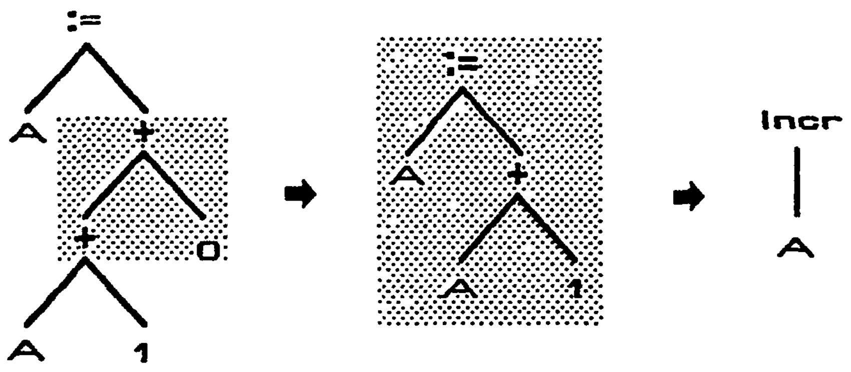
where the first "–" is subtraction and the second complementation. Assume that the computer is without a subtraction operator (e.g., the Intel 8748). Assume also that the computer can increment the values in store. This statement can be optimized to

increment A

The grammar located after the intermediate code trees below is able to effect this optimization as well as others. Each reparsing of an intermediate code tree is given as an arrow between trees. The shaded parts of the trees are about to be transformed.



Rules 3, 4, and 6 effect the first three transformations (arrows).



Rules 5 and 2 effect the last two transformations.

Tree Transformer Grammar

```
? ambiguous
? scan-
*
* This is a sample grammar for the tree transformer.
* Grammar references in the Pascal code, such as <expr> or "lit",
* are pointers to actual Intermediate Code tree nodes.
* A slash "/" in column one indicates that the rule will be
* applied exactly when the boolean variable return is true.
*
<statement>  ::-  <assign>  <end_of_statement>
   translate_tree(<assign>);
<assign>       ::-  assign  var  sum_of  var  lit
*
/  return := equal_tree("var-1", "var-2") and ("lit"^.val = 1);
*
   begin
      make_branch (incr, "var-1", nil, <assign>);
      reparse := true
   end;
<expr>          ::-  subtract  <expr>  <expr>
   begin
```

```
      make_branch (compl, <expr-2>, nil, temp_root_1);
      make_branch (sum_of, temp_root_1, lit_one, temp_root_2);
      make_branch (sum_of, <expr-1>, temp_root_2, <expr>);
      reparse := true
   end;
<expr>          ::- compl compl <expr>
   begin
      <expr>^ := <expr-1>^;
      reparse := true
   end;
<expr>          ::- sum_of <expr> lit
*
/  return := ("lit"^.val = 0);
*
   begin
      <expr>^ := <expr-1>^;
      reparse := true
   end;
<expr>          ::- sum_of lit <expr>
   begin
      exchange ( "sum_of"^.subtree[1], "sum_of"^.subtree[2] );
      reparse := true
   end;
<assign>        ::- assign var <expr>
<assign>        ::- incr var
<expr>          ::- lit
<expr>          ::- var
<expr>          ::- sum_of <expr> <expr>
<expr>          ::- compl <expr>
<assign>        ::- assign var sum_of var lit
```