

# AN EXPERIMENT IN AUTOMATED, SCIENTIFIC-CODE SEMANTIC ANALYSIS

Mark E. M. Stewart  
Scott Townsend  
Dynacs Engineering, Inc.  
2001 Aerospace Parkway  
Brook Park, OH 44142

## Abstract

This paper concerns a procedure that analyzes aspects of the meaning or semantics of scientific and engineering code. This procedure involves taking a user's existing code, adding semantic declarations for some primitive variables, and parsing this annotated code using multiple, independent expert parsers. These semantic parsers are designed to recognize formulae in different disciplines including physics, numerical methods, mathematics, and geometry. The parsers will automatically recognize and document some static, semantic concepts and help locate some program semantic errors. Results are shown for two intensively studied codes and two blind test cases. These techniques may apply to a wider range of scientific codes. If so, the techniques could reduce the time, risk, and effort required to develop and modify scientific codes.

## Introduction

From a syntactic or programming language perspective, scientific programs are uses of a programming language that specify how numbers are to be manipulated. However, from the perspective of semantics or meaning, scientific programs involve an organization of physical and mathematical equations and concepts. The programs from a wide range of scientific and engineering fields use and reuse these fundamental concepts in different combinations. This paper explains an experiment in representing, recognizing, and checking these fundamental scientific semantics.

What motivates this experiment is that semantics is a central issue in the development and modification of scientific code. Reducing the errors in a scientific or engineering program until its results are trusted involves ensuring the program's semantics are correct. Further, this development process is expensive and time consuming because it is primarily a manual task. The

existing software development tools (lint, ftnchek, make, dbx, SCCS, call tree graphs, memory leak testing) do not fully alleviate this problem and deal only superficially with semantics. Further, verification techniques (comparison with available analytic and experimental results; verification of convergence and order of accuracy) can only detect the presence of an error; *finding* this error often leads to a time-consuming manual search. For example, the second difference code (1) contains a geometrical error in the grid index I, which is exceedingly hard to find manually.

$$FS(I,J) = DW(I+2,J) - 2.*DW(I,J) + DW(I-1,J) \quad (1)$$

However, it can be found automatically with this semantic analysis procedure.

Semantics' role in program modification is similar to its role in code development. Understanding another programmer's code is usually frustrating and time consuming, and to understand code well enough to modify it confidently requires a large time investment. Suggestive variable names, program comments, program manuals, and communications with the developer are means to convey an understanding of a code, but these methods are often neither adequate nor efficient. This semantic analysis procedure can represent and recognize important code details.

Modern programming practice attempts to reduce the number of code development errors and to ease code modification. Software reuse (through subroutine libraries) and object-oriented programming<sup>1</sup> also target these problems, but these techniques cannot help when modifications and custom software are required. Recently there has been work in high-level specification languages<sup>2</sup> where a symbolic manipulation program (Maple, Mathematica) is used to write subroutines or even programs. Yet another attempt to solve these software problems is the field of formal methods<sup>3</sup> that uses logic, set theory, functions, and algebra to develop

mathematical models for systems and to rigorously prove code properties.

Part of the problem with these tools is that it is difficult to represent knowledge. Using the classical notation and methods of mathematics and physics simplifies knowledge representation in this work, yet representations (or ontologies) have been developed in other modern fields. Using an ontology for engineering knowledge representation and tool integration has been studied<sup>4</sup>. In natural language understanding, parsing has been combined with an ontology to recognize and represent the semantics of written text<sup>5</sup>. The use of both parsing and an ontology makes the natural language work similar to the current experiment with scientific programming.

The limitations of these existing tools and approaches and the cost of manual semantic analysis are the motivations for the current experiment. As a complementary tool, automated semantic analysis<sup>6</sup> could reduce the time, risk, and effort during original code development, subsequent maintenance, second party modification, and reverse engineering of undocumented code.

This paper follows experimental report form with a thesis, procedure, results, discussion, and conclusion.

### Thesis

The thesis of this semantic analysis experiment is that fundamental physical and mathematical formulae and concepts are reused and reorganized in scientific and engineering codes. Further, a procedure, which combines a parser<sup>7,8,9</sup> with other methods, can recognize each reuse.

If this experiment in automated analysis succeeds, the resulting tool would help locate errors during code development and document code for modification.

### Procedure

In outline, the current procedure for testing this thesis consists of four key stages. First, the user adds **semantic declarations** to his/her existing program (2).

```
C? MA == mass
C? ACC == acceleration
FF = MA*ACC (2)
```

Distinguished by “C?” these declarations provide the mathematical or physical identity of primitive variables

in the user’s program. Second, the procedure syntactically parses the user’s program into a data structure representation. Third, a translation scheme converts statements in the user’s FORTRAN program into statements in different context languages. For example, the FORTRAN expression in (2) is converted to the physical dimensions expression (3) and the physical quantity expression (4).

$$(M) * (L*T**-2) \quad (3)$$

$$\text{mass} * \text{acceleration} \quad (4)$$

These context languages reflect the different **aspects** of program statements that scientists and engineers analyze. Aspects include mathematical or physical quantity, geometrical (grid) location, geometrical entity, vector entity, dimensions, and units. Fourth, independent expert parsers examine the translated phrases and attempt to recognize formulae from their area of expertise. For example, a dynamics expert parser would include the rule (5), be able to recognize the phrase (4) as “force” due to Newton’s law, and assign this result to FF in (2).

$$\text{force} : \text{mass} * \text{acceleration} \quad (5)$$

Further, the units expert parser can reduce (3) and verify units. The other expert parsers act similarly (see Table 1).

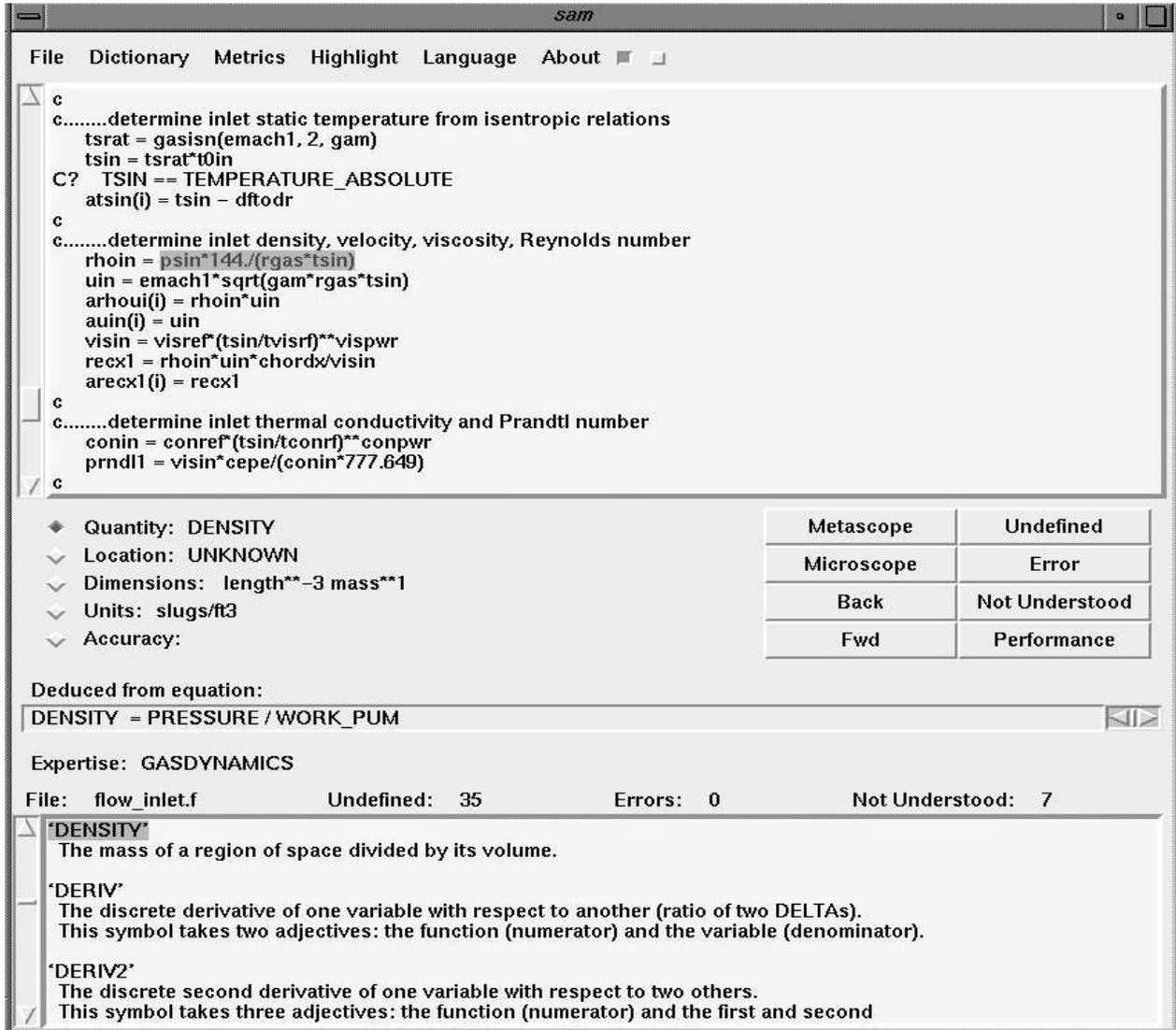
Analyzed Aspect	Parsers	Parser Rules	Fundamental Equations
Quantity-Math	3	410	49
Quantity-Physical	3	615	94
Value / Interval	2	201	27
Grid Location	4	1648	232
Geometrical Entity	1	441	20
Vector Entity	1	305	11
Non-Dimensional	1	72	5
Dimensions	1	55	10
Units	1	81	19

**Table 1:** Aspect analyses performed by the semantic analysis procedure including number of parsers for each aspect, number of Yacc<sup>8</sup> parser rules, and fundamental equations. Rule (5) is a fundamental equation; some equations require several parser rules.

When an expert parser recognizes an expression, it annotates the data structure representation of the user’s

program with the observation. Other expert parsers can use this observation to recognize more of the expression. Further, the annotated data structure representation contains all the results of the semantic analysis, and a graphical user interface (GUI) displays these results as shown in Figure 1. The user may point to variables and

expressions in his/her code, and the GUI displays any semantic interpretation and its derivation. The GUI highlights recognized errors, undefined quantities, and unrecognizable expressions. Further, the GUI provides detailed scientific and technical definitions and explanations.



**Figure 1:** GUI display for the semantic analysis program. The top window displays a user’s code; variables and expressions may be selected for explanation. The middle region explains this selected text. In this case, the physical quantity is density, it does not have a grid location, and it has the displayed dimensions, units, and derivation. The bottom region displays the semantic dictionary/lexicon.

Extended Example of Parsing for Code Recognition

The most important step in the above procedure is the expert parsers' analysis. An example of how parser rules operate to recognize an expression is instructive. To determine the meaning of the variable VAR in (6), the parser sequentially examines the RHS of (6). When the parser reads *energy<internal>* (EI), it anticipates rule (7d) and the tokens '+' and *work*.

C? P == pressure<static>, RHO == density<static>

C? V == speed, EI == energy<internal>

C

$$\text{VAR} = \text{EI} + \text{P} / \text{RHO} + 0.5 * \text{V} * \text{V} \quad (6)$$

When the parser reads *pressure<static>* (P), it expects rule (7c) to produce *work* and the tokens '/' and *density<static>* to appear. When the parser sees all the tokens of rule (7c), it reduces them to the token *work*, and when all the tokens of rule (7d) are present, they are reduced to the token *enthalpy*. The parser anticipates rule (7f) next. Similarly, the parser recognizes rules

(7b), (7a), and (7f) to infer that VAR represents *enthalpy<total>*.

speed\_squared : speed \* speed (7a)

energy<kinetic> : half \* speed\_squared (b)

work : pressure<static>/density<static> (c)

enthalpy : energy<internal> + work (d)

sound\_speed\_squared : gamma \* work (e)

enthalpy<total> : enthalpy + energy<kinetic> (f)

(These rules have been simplified to intensive quantities.)

The parser rules (7) do not execute sequentially as with statements in a conventional programming language. Instead the parser determines if and how the rules appear in the input. Parser rules (7) are automatically converted to a subroutine by the program Yacc<sup>8</sup>. Table 2 gives a flavor of the expert parser rules. Note also that each step in the recognition process depends on correctly performing the previous step; an error can dramatically reduce recognition.

Mathematical, Numerical Quantity	Physical Quantity	Grid Location, Geometrical Entity
$q \leftarrow q + 0$	$p \leftarrow F / \text{area}$	$l \leftarrow l_1 \pm l_2$
$q \leftarrow q * 1$	$F \leftarrow m * A$	$l \leftarrow l_1 */ l_2$
$0 \leftarrow q_1 - q_2$	$W \leftarrow F * \text{length}$	$g \leftarrow g_1 \pm g_2$
$\Delta q \leftarrow q_1 - q_2$	$R \leftarrow R_u / \text{Mol. wt.}$	$g \leftarrow g_1 */ g_2$
$2q \leftarrow q_1 + q_2$	$R \leftarrow C_p - C_v$	<b>Vector Entity</b>
$\Delta^2 q \leftarrow q - 2q + q$	$R_u \leftarrow k * N_A$	
Polynomials	$C_p \leftarrow \Sigma (\text{Mass Fract.} * C_p)$	$v \leftarrow v_1 \pm v_2$
$q^2 \leftarrow q * q$	$\gamma \leftarrow C_p / C_v$	$v \leftarrow v_1 */ \text{scalar}$
$\Sigma q \leftarrow q + q + \dots$	$w \leftarrow p / \rho$	surface $\leftarrow v_1 * v_2$
$\partial q / \partial x \leftarrow \Delta q / \Delta x$	$c^2 \leftarrow \gamma * p / \rho$	scalar $\leftarrow \text{scalar} \pm */ \text{scalar}$
$\partial^2 q / \partial x^2 \leftarrow \Delta^2 q / \Delta^2 x$	$^{\circ}\text{C} \leftarrow ^{\circ}\text{K} - 273.15$	scalar $\leftarrow \text{Dot Product}$
$\partial q / \partial y \leftarrow \partial q / \partial x * \partial x / \partial y$	$^{\circ}\text{F} \leftarrow 1.8 * ^{\circ}\text{C} + 32$	<b>Non-Dimensionalization, Dimensions, Units</b>
vol $\leftarrow \text{length} * \text{area}$	$p / \rho \leftarrow R * T$	
area $\leftarrow \text{length} * \text{length}$	$\partial m / \partial t \leftarrow \rho * U * A$	$D \leftarrow D_1 \pm D_2$
<b>Number Value, Number Interval</b>	$E_k \leftarrow 1/2 * m * U^2$	$D \leftarrow D_1 */ D_2$
	$e_k \leftarrow 1/2 * U^2$	$D \leftarrow \text{ftn}(D_1)$
$n \leftarrow n_1 \pm n_2$	$e_i \leftarrow 1/(\gamma-1) * p / \rho$	$d \leftarrow d_1 \pm d_2$
$n \leftarrow n_1 */ n_2$	$h \leftarrow e_i + w$	$d \leftarrow d_1 */ d_2$
$n \leftarrow n_1 ** n_2$	$h_o \leftarrow h + e_k$	$d \leftarrow \text{ftn}(d_1)$
$n \leftarrow \text{ftn}(n_1)$	$v \leftarrow \mu / \rho$	$u \leftarrow u_1 \pm u_2$
$r \leftarrow r_1 \pm r_2$	Reynolds $\leftarrow \rho * U * \text{length} / \mu$	$u \leftarrow u_1 */ u_2$
$r \leftarrow r_1 */ r_2$	$u * \partial u / \partial x - (1/\rho) * \partial p / \partial x$	$u \leftarrow \text{ftn}(u_1)$
q = Math/Numerical Quantity; l = Grid Location; g = Geometrical Entity; v = Vector Entity; n = Number Value; r = Number Interval; D = Non-Dimensionalization; d = Dimensions; u = Units		

**Table 2:** A sampling of expert parser rules used in the semantic analysis method. Many rules are condensed. Due to decomposition a single operation may involve multiple independent aspects (units, grid location and quantity for *x\_coordinate - x\_coordinate*), and several rules from this table can apply to it.

### Properties of the Procedure

Several additional features and properties of this automated semantic analysis procedure deserve mention: semantic declaration terms, mathematical rules, the generality of recognition, the nature of error detection, and the presence of ambiguities.

### Terms in Semantic Declarations

The code fragment (6) includes four semantic declarations. The six defining terms *pressure*, *static*, *density*, *speed*, *energy*, and *internal* are from a lexicon of terms (currently 380), and they closely resemble English technical terms. Further, the knowledge representation uses adjective terms, such as *static* in *pressure<static>*, to modify terms and reduce their number. Multiple adjectives are possible. For example, the term *derivative* takes two adjectives, *derivative<pressure time>*, to represent  $\partial p/\partial t$ .

### Mathematical Rules

The physical rules (7) differ from mathematical rules, since mathematical rules apply to any mathematical or physical quantity. For example, to detect a discrete difference,  $\Delta q$ , the pattern is *variable-variable* where *variable* is any quantity. This pattern matches excessively, and when it does match, additional code compares the aspects of the two variables. If the variables are identical, then the expression is zero; if the variables differ in location only, then the expression is a discrete difference,  $\Delta q$ ; if the variables differ in geometrical entity in a specific way, then the expression is a second order Jacobian.

### Non-General Derivation

The general derivation of equations is dependent upon the fundamental physical equations, the algebraic properties of mathematical expressions (Commutative, Distributive and Associative laws), and the transformation laws of equations (equation substitution and algebraic solution). However, this general derivation of equations is a non-trivial and potentially expensive search.

In comparison, the rules within a Yacc<sup>8</sup> parser (or more formally a LALR(1)<sup>7,8</sup> grammar) are a specialized type of rewrite rule, referred to here as reduction substitution rules since they rewrite and reduce one or more input tokens to a single token. The rule (8a) substitutes one token (LHS) for two tokens (RHS).

enthalpy<total> : enthalpy + energy<kinetic>(8a)  
enthalpy + energy<kinetic>: enthalpy<total> (b)  
enthalpy<kinetic> + enthalpy: enthalpy + energy<kinetic> (c)

However, the substitution (8b) and the commutative transformation (8c) cannot be directly implemented in a Yacc grammar since they rewrite to more than one token.

Although parsers allow fast and efficient rule recognition, the parser rules will allow only a small subset of the algebraic and equation laws to be implemented. Consequently, by using only Yacc parsers, it is not possible to perform general derivations and compare them with expressions in scientific and engineering codes.

### Enhanced Recognition

To resolve this recognition problem, the parsers are supplemented with five methods. The first three of these methods give a moderate algebraic search. This strategy tries to avoid expensive general searches each time the formula is encountered. The existing evidence indicates this is a reasonable choice.

In the first method, the experts are applied incrementally to expression components. In (9) the sub-expressions  $p/\rho$ ,  $u^2$ , and  $\frac{1}{2}u^2$  are separately referred to each of the expert parsers for analysis.

$$p/\rho + \frac{1}{2} u^2 \quad (9)$$

Second, between calls to the expert parsers, methods are applied that commute, associate, and distribute (and inverse distribute) the expression.

Third, some limited substitutions can be performed between referrals to the expert parsers. Examples include,  $2a_i \Leftarrow a_{i+1} + a_{i-1}$  or the normalizing transformation  $u^2 \Leftarrow u*u$ .

A fourth method of enhancing recognition is **decomposition** of semantic knowledge. For example by analyzing a quantity's axis, the vector entity expert parser can recognize a dot product (10) almost independently of the physical quantity,  $u$ . (Note: verifying that  $u_x^2$ ,  $u_y^2$ , and  $u_z^2$  are otherwise identical is the further necessary test.)

$$u_x^2 + u_y^2 + u_z^2 \quad (10)$$

This near independence of aspects extends to the analysis of mathematical/physical quantity, number value, number interval, grid location, geometrical entity, vector entity, non-dimensionalization, dimensions, and units.

In the fifth method, constants are identified when the expert parser's input is prepared. For example, in (11) the constant 32 is distinguished from other constants for the parser analyzing temperature equations.

$$^{\circ}\text{F} = 1.8 * ^{\circ}\text{C} + 32 \quad (11)$$

The use of 32 in (11) is an example of how this enhanced recognition process is context sensitive<sup>7</sup>, since in other contexts this constant can have other meanings.

### Error Detection

This semantic analysis procedure detects errors with direct tests of some code aspects including dimensions, units, and non-dimensionalization. For other code aspects, including mathematical/physical quantity, the semantic analysis procedure attempts to recognize formulae. Unrecognized code may be incorrect or a correct formula beyond the scope of the stored rules. For example, the procedure would declare the aerodynamics equation (12) unrecognized (pressure is incorrectly calculated from density, total energy, velocities and the ratio of specific heats); (12) cannot be declared in error since it may be an unknown rule.

$$P = \text{RHO} * (\text{E} - (\text{U} * \text{U} + \text{V} * \text{V})) * (\text{GAM} - 1) \quad (12)$$

However, in cases where multiple conditions must be satisfied before recognition, the failure of one condition is evidence of an error. For example, to recognize a second-difference, formula and geometrical conditions must be satisfied; however in (1) the geometrical condition is not satisfied and an error is suspected. Although this semantic analysis procedure is not a direct error tester, by reviewing the analysis, users can identify errors relatively easily.

### Ambiguities

A further theoretical issue is that ambiguities exist in the static analysis of scientific code. The final identity of the variable P is ambiguous in (13).

$$\begin{aligned} \text{C? } P &== \text{pressure}\langle\text{static}\rangle \\ \text{C? } \text{RHO} &== \text{density}\langle\text{static}\rangle \\ \text{CC} &= \text{GAM} * P / \text{RHO} \\ P &= 1 \end{aligned} \quad (13)$$

After P is assigned the indistinct value 1, it may still represent pressure, or it may represent something else with the constant value 1. This ambiguity would not exist if the assignment were  $P = 3.14$  or  $P = \text{RHO}$ . The procedure resolves this ambiguity by assuming no memory re-use; the new number value is set, and a

warning is generated. The ambiguity can be resolved if the user rewrites the code so that P is not re-used.

Another ambiguity can exist when deducing an array's layout. In a static analysis, the indices in array assignment can be under-specified and suggest different array layouts. In (14), it is not clear if  $N < 3$ ,  $N > 3$ , or  $N = 3$  since the value of N is not known.

$$\begin{aligned} \text{C? } N &== \text{number}\langle\text{species}\rangle \\ \text{C? } R &== \text{radius} \\ \text{DIMENSION } A &(10) \\ A(3) &= 0. \\ A(N) &= R \end{aligned} \quad (14)$$

This ambiguity is resolved with semantic declarations for array structure or by assuming (and noting) a case.

### Results

The results take two forms: the recognition of code semantics and the generality of this recognition capability.

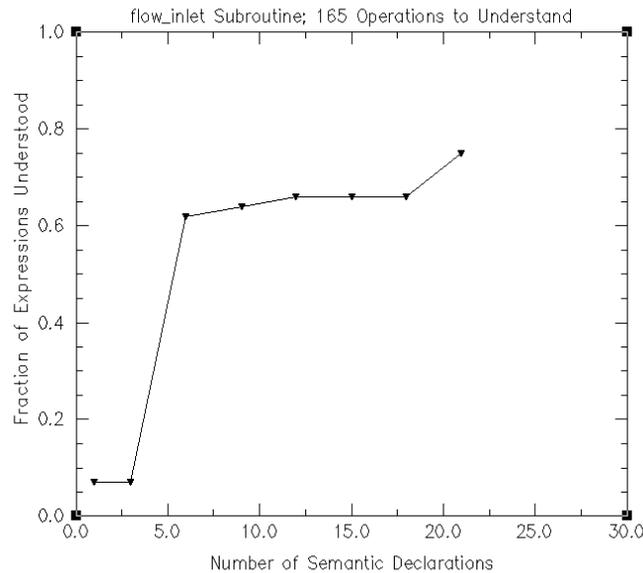
### Recognition of Code Semantics

The results of analyzing two development codes demonstrate recognition of code semantics. One program performs data reduction for experimental data analysis, and the other is twenty subroutines from the ALLSPD<sup>10</sup> code. ALLSPD is a 3D, implicit, generalized curvilinear coordinate code for chemically reacting flows. In development test cases, the procedure developers devise and test expert parser rules, and correct any errors. Hence these are not blind test cases.

Highlighted in the GUI of Figure 1 is a recognized expression from the first development code. Other recognized formulae include temperature formulae, viscosity and thermal conductivity calculated from the power law, Reynolds number, and Prandtl number. Most of the not-understood code corresponds to program variables that are defined by function calls and logical expressions. The semantic analysis coding needed for these cases has not yet been developed.

To measure this recognition of code semantics, the procedure examines each operation,  $a \otimes b$  where  $\otimes \in \{ +, -, *, /, ** \}$ , intrinsic function reference,  $\text{ftn}(a)$ , and array reference,  $a(i,j,k)$ , in the code. The recognition fraction is the fraction of these operations/references where the mathematical/physical quantity is understood. In Figure 2, the recognition fraction for the first development case is plotted against

## Expression Understanding .vs. Semantic Declarations



**Figure 2:** Graph showing the increase in expression understanding as semantic declarations are added to a data reduction subroutine. The subroutine contains 160 operations to understand and approximately 100 lines of code.

an increasing number of semantic declarations. The curve is offset from the origin since some trivial expressions are recognizable without semantic declarations.

Twenty ALLSPD subroutines (5500 FORTRAN statements) form an additional development case where the analysis tool achieved an understanding fraction of 0.51 (see Figure 3). Input and primitive variables were semantically declared. The user effort to prepare these declarations is modest compared to the effort of syntactically declaring all variables as required by modern programming practice and some programming languages.

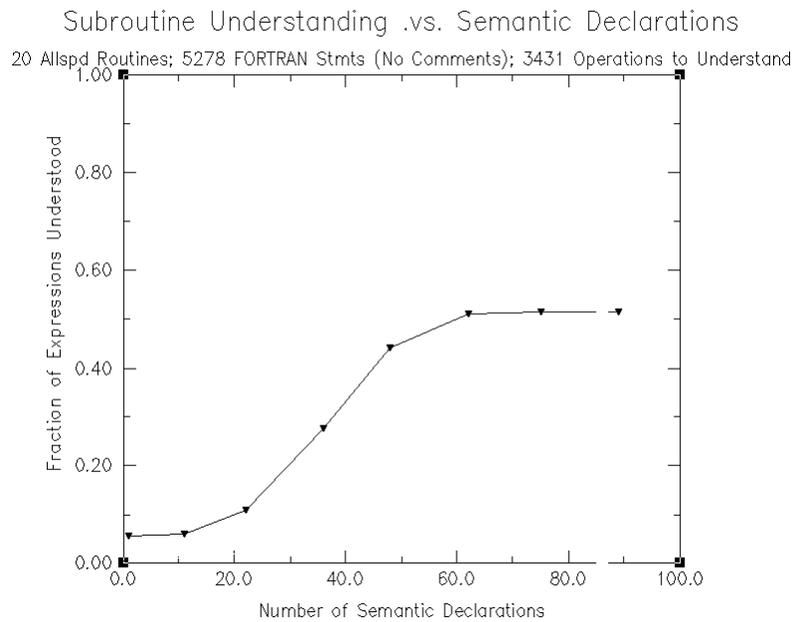
### Generality of Recognition Capability

Blind tests of two codes demonstrate the generality of this semantic analysis procedure. These test cases are the ENG10<sup>11</sup> code with 20k lines of FORTRAN code (loc) and the ADPAC<sup>12</sup> code (86k loc). Both are explicit, finite-volume fluid dynamics codes. The procedure developers examined these test codes to determine semantic declarations for coordinates and solution variables. The semantic analysis program was not modified to correct errors.

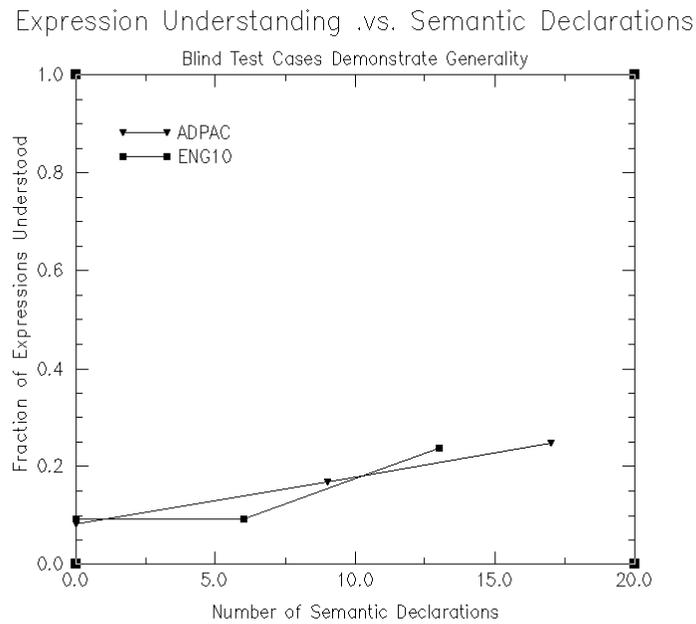
Recognition results for these blind test cases are shown in Figure 4, and they demonstrate a general semantic recognition capability. Of course, these results reflect the analysis code's level of development and not the quality of these blind test cases. One would expect the development codes to have a higher recognition fraction since the procedure developers correct expert parser errors found during development. Additional work on the procedure will improve these preliminary results.

### Discussion

This automated semantic analysis procedure has properties advantageous for analysis of scientific programming. First, parsers can recognize mathematical, physical, and geometrical knowledge in code. Second, parsers can encapsulate formulae into independent modules. Third, these rules are largely fundamental, which increases generality, and they are largely aspect-independent, which reduces complexity. The data structures used to represent the code are effective and allow manipulations. Last, the economics of this procedure appear to be favorable. In particular, the number of semantic declarations is reasonable, and the execution time on a modern workstation is under one minute per thousand lines of code.



**Figure 3:** Graph showing the increase in expression understanding as semantic declarations are added to twenty subroutines from the ALLSPD code. The subroutines contain 5278 non-comment FORTRAN statements and 3431 operations to understand. Further work will increase the understanding fraction. The analysis results reflect the analysis code’s quality and not the quality or ability of the ALLSPD code.



**Figure 4:** Graph showing the increase in expression understanding as semantic declarations are added to two blind test cases. The ADPAC codes contain 86k loc, and the ENG10 code contains 20k loc. The understanding fraction for development codes is higher than blind test case codes because the procedure developers correct expert parser errors found during testing. Further work will increase the understanding fraction. The analysis results reflect the analysis code’s quality and not the quality or abilities of the ADPAC or ENG10 codes.

### Potential Limitations

Despite these advantages, several potential limitations have been identified and must be monitored as the procedure develops. First, the procedure lacks the code to analyze the following constructions correctly: array structure, boundary conditions, logical expressions, subroutine calls and the call tree. As the procedure develops this infrastructure problem will disappear. Second, as noted earlier, when attempting to recognize an expression, this procedure does not perform a general derivation/search but only a moderate search. These searches will limit recognition but not excessively. Third, a stability problem could exist because errors and omissions in the expert parser rules can reduce the recognition fraction. This loss of recognition occurs because recognition involves long inference chains that can be broken by an error. Fourth, although adding rules requires moderate expertise, the large number of physical formulae may mean it takes significant effort to incorporate these rules into the procedure. Fifth, the current implementation uses sizeable amounts of memory, and for big analyses, execution is memory bound.

Further, to achieve the best results with this procedure requires a particular style of structured programming. For example, the real constants in (15) are evaluations of  $\gamma/(\gamma-1)$ ,  $2/\gamma$ , and  $(\gamma-1)/\gamma$  for a particular value of the ratio of specific heats,  $\gamma$ .

$$\begin{aligned} \text{numer} &= 3.8249 * (r^{**1.4771}) * (1 - r^{**0.26145}) * (1 - \text{beta}^{**4}) \\ \text{denom} &= (1 - r) * (1 - (\text{beta}^{**4}) * r^{**1.4771}) \end{aligned} \quad (15)$$

As numbers they are hard to recognize, and to be recognized, this code would have to be rewritten.

### Conclusions

We spend too much time slaving over our codes, analyzing details; and this experiment strives to automate these menial chores. Further, its use of fundamental representation and expert parsers provides an example for automating other scientific and engineering tasks, such as design. As detailed in our discussion section, if this procedure is to achieve its full potential, we must tackle problems that fall into three categories:

- Develop the procedure's infrastructure: array structure, logical expressions, subroutine calls
- Extend discipline detail by adding physical, mathematical, and geometrical rules
- Improve generality, utility, and economy

### Acknowledgments

The lexical analysis routines and FORTRAN grammar are from `ftnchek`<sup>13</sup>. The GUI routines use Tcl/Tk<sup>14</sup>. This work was supported by the NASA High Performance Computing and Communications program through the Computing and Interdisciplinary Systems Office (contract NAS3-98008) at NASA Glenn Research Center. Greg Follen, Joe Veres, and Karl Owen were the monitors. The authors thank Ambady Suresh for helpful discussions about this work.

### Bibliography

- <sup>1</sup> W. Y. Crutchfield and M. L. Welcome, "Object Oriented Implementation of Adaptive Mesh Refinement Algorithms," *Scientific Programming* 2 (1993) 2, 145-156.
- <sup>2</sup> E. Kant, "Synthesis of Mathematical Modeling Software," *IEEE Software*, May 1993.
- <sup>3</sup> J. Woodcock and M. Loomes, *Software Engineering Mathematics* (Pitman, London, 1988).
- <sup>4</sup> M. R. Cutkosky, R. S. Engelmire, R. E. Fikes, et. al., "PACT: An Experiment in Integrating Concurrent Engineering Systems," *IEEE Computer*, Jan. 1993.
- <sup>5</sup> J. Allen, *Natural Language Understanding* (Benjamin/Cummings, Menlo Park, 1987).
- <sup>6</sup> M. E. M. Stewart, "A Semantic Analysis Method for Scientific and Engineering Code," *NASA CR 207402*, April 1998.
- <sup>7</sup> A.V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools* (Addison-Wesley, Reading, 1986).
- <sup>8</sup> S. C. Johnson, "Yacc--Yet Another Compiler-Compiler," *Comp. Sci. Tech. Rep. No. 32*. (AT&T Bell Laboratories, Murray Hill, 1977).
- <sup>9</sup> J. R. Levin, T. Mason, and D. Brown, *Lex and Yacc* (O'Reilly, Sebastopol, 1992).
- <sup>10</sup> J.-S. Shuen, K.-H. Chen, and Y. Choi, "A Coupled Implicit Method for Chemical Non-equilibrium Flows at All Speeds," *J. of Comp. Phys.*, 106, No. 2, 306, 1993.
- <sup>11</sup> M. E. M. Stewart, "Axisymmetric Aerodynamic Numerical Analysis of a Turbofan Engine," *ASME Paper 95-GT-338*, 1995.
- <sup>12</sup> E. Hall, N. J. Heidegger, and R. A. Delaney, "ADPAC v 1.0 – User's Manual," *NASA CR 1999-206600*, Feb. 1999.
- <sup>13</sup> R. K. Moniot, "ftnchek" <http://www.dsm.fordham.edu/~ftnchek> (Fordham University, New York, 1989).
- <sup>14</sup> J. K. Ousterhout, *Tcl and the Tk Toolkit* (Addison-Wesley, Reading, 1994).