NASA/CR—1998-207402

A Semantic Analysis Method for Scientific and Engineering Code

Mark E.M. Stewart
NYMA, Inc., Brook Park, Ohio

Prepared under Contract NAS3–27816

National Aeronautics and
Space Administration

Lewis Research Center

April 1998

Available from

NASA Center for Aerospace Information
7121 Standard Drive
Hanover, MD 21076
Price Code: A03

National Technical Information Service
5287 Port Royal Road
Springfield, VA 22100
Price Code: A03

A SEMANTIC ANALYSIS METHOD FOR SCIENTIFIC AND ENGINEERING CODE

Mark E. M. Stewart
NYMA
2001 Aerospace Parkway
Brook Park, OH 44142

1. Abstract

This paper develops a procedure to statically analyze aspects of the meaning or semantics of scientific and engineering code. The analysis involves adding semantic declarations to a user's code and parsing this semantic knowledge with the original code using multiple expert parsers. These semantic parsers are designed to recognize formulae in different disciplines including physical and mathematical formulae and geometrical position in a numerical scheme. In practise, a user would submit code with semantic declarations of primitive variables to the analysis procedure, and its semantic parsers would automatically recognize and document some static, semantic concepts and locate some program semantic errors. A prototype implementation of this analysis procedure is demonstrated. Further, the relationship between the fundamental algebraic manipulations of equations and the parsing of expressions is explained. This ability to locate some semantic errors and document semantic concepts in scientific and engineering code should reduce the time, risk, and effort of developing and using these codes.

2. Introduction

2.0 Motivation

With revolutionary increases in computer speed, scientific and engineering simulation has moved from the manual calculation of problems to computer simulation codes. Across a wide range of disciplines including aeronautics, astrophysics, combustion, geophysics, molecular dynamics, structural analysis, and weather and climate modeling, computer simulations inexpensively explore important problems that researchers previously studied only with experimental and theoretical methods. However, these scientific and engineering codes involve

a large number of semantic details including the implementation of formulae and manipulation of geometrical concepts. Even with current testing techniques, identifying errors in these semantic details is a problem that significantly hinders scientific code development.

The available software for developing and maintaining this scientific and engineering software includes modern computer languages, syntactic analysis (lint [14], ftnchek [18]), compilation control (make [6]), source debug tools (dbx), and version control (sccs). Further, there are widely accepted techniques for testing scientific programs—solution comparison with available analytic test cases, comparison with available experimental results, examination of the formal and observed order of accuracy of the numerical scheme, as well as verification of convergence. However, these are tests of an ensemble of details that may detect the presence of an error but cannot identify the faulty detail. A program containing either of the errors (2.0)

```
C
      P = RHO * ( E - ( U * U + V * V ) ) * ( GAM - 1. )          (2.0)
C
```

(pressure is incorrectly calculated from density, total energy (intensive), velocity, and the ratio of specific heats) or (2.1)

```
C
      FS(I,J,N) = DW(I+2,J,N) - 2.*DW(I,J,N) + DW(I-1,J,N)        (2.1)
C
```

(the second difference is geometrically incorrect) will fail these traditional testing methods. However, finding these errors can be very difficult. In practise, developers use their knowledge and skill to devise diagnostic tests which narrow a manual search.

In manual checking, developers examine code, interpret its meaning and verify it is correct. However, manual checking is frequently incomplete and incorrect, always time consuming, and limited by the individual's semantic knowledge. To aid in manual interpretation, developers encode some semantic information in variable names, comments, and written documentation which must, however, still be processed manually.

What are the semantic details being checked? They include program execution order and logic which are not considered here. They also include mathematical and geometrical con-

cepts including derivatives, integrals, points, and vectors. Further, simulations use the physical principles and formulae of individual disciplines. For example, in aerodynamics the gasdynamics and Navier-Stokes equations are fundamental formulae. These mathematical, geometrical, and physical equations are an unambiguous domain of knowledge in contrast to the knowledge of other fields. Further, the representation of this mathematical and physical knowledge, in the form of mathematical notation, formulae, and specific terminology has evolved to be simple yet exacting. These properties make equations more suitable for implementation in a rule system than many other domains of knowledge.

This paper exploits these properties to perform automatic testing and documentation of some semantic concepts by using well established compiler techniques. Users must declare in their code some semantic information about primitive code variables. However, the current method parses this embellished code to recognize the patterns of stored mathematical, geometrical, and physical formulae. In this way, the code's semantic implementation is partially checked and documented. In particular, the code lines (2.0) and (2.1) can be recognized as incorrect.

The benefits of automated analysis and verification of code semantics include reduced time, risk, and effort during original code development, subsequent maintenance, second party modification, and reverse engineering of undocumented code. A satisfactory semantic analysis would also provide a useful but not sufficient test for correctness. Further, semantic recognition results are code documentation. These benefits should apply to codes in a wide range of scientific and engineering disciplines.

In the next subsections, related technology is explained, and some basic concepts and notation from compiler parsing are presented. The semantic analysis procedure and a prototype implementation are described in Section 3, some theoretical analyses of the problem are presented in Section 4, and finally, three test problems are shown in Section 5.

2.1 Related Technology

Currently, automatic code testing is largely limited to syntactic testing. Lint [14] and ftnchek [18] test programs for conformance with language syntax and portability rules. However, there are some basic semantic tests performed by these testing codes, such as type checking, which can reveal semantic errors. There are also efforts to organize and manage code

developers [11].

A body of work exists [17,19] in computer science where predicate logic is extended and used for Pascal program verification. Assertions are placed at crucial points in a Pascal code including entry and exit points. These assertions specify the code by defining relationships between program variables which must be true when execution passes the assertion's position in the code. The assertions are parsed with the code into verification conditions which are logical expressions. If all these verification conditions can be proven to be true with a theorem prover, then the program is consistent with its specification. Physical formulae and their geometrical interpretation do not enter this work. However, many practical concepts are similar including the need for parsing, program annotations, and a body of rules.

An influential approach to documentation is WEB [15] which allows natural language documentation to be embedded within a program. Different processors produce either compilable code or a natural language document. In this way, program documentation accompanies a code and is more readily updated when code is modified.

Influential and insightful work exists in fields other than program testing and documentation. Natural language analysis is concerned with dissecting the semantics of written and spoken language. Lexical analysis and parsing are computer algorithms used to automate this analysis [3].

Expert system techniques [12] also provide a means to encode and organize knowledge and to synthesize results from this knowledge. A classic expert system is INTERNIST [20,21], a program which attempts to duplicate the diagnostic reasoning of human medical clinicians. A number of other domain specific expert systems exist [12,23]. Bobrow [4] contains a number of papers on qualitative reasoning about physical systems and in particular about the diagnosis and verification of electronic circuits. The capabilities of expert systems pattern matching differ from programming language parsing. More complex rules can be constructed in an expert system than in a parser as will be explained in section 4. The pattern matching algorithm of choice for expert systems is the Rete algorithm [8].

2.2 Underlying Technology

The semantic analysis procedure developed in the next section uses well established compiler

parsing techniques to recognize the use of mathematical formulae. Since compiler technology is not well known outside computer science, the following subsection briefly explains basic concepts and notation crucial to understanding the subsequent sections. When explained, important terminology is printed in bold type.

In a compiler, a **parser** recognizes from the words and punctuation of a program which **grammar rule** of the programming language is being used. The acceptable programming language order is the syntax of the language, and it is represented by a set of grammar rules called a **syntactic grammar**. These compiler parsing techniques arose from a need for efficient, easily implemented methods for transforming an expressive, high-level programming language, such as FORTRAN or C, into machine code. Following the introduction of the first FORTRAN compiler [22], considerable theoretical and practical efforts were directed at improving parsers. LALR(1) parsing [1,2] emerged as a prominent method of implementing a compiler parser since the allowed grammar rules are expressive enough for most programming languages, and a set of grammar rules, called a **grammar**, can also be automatically converted to a parsing subroutine. In particular, the parser generator YACC [13,16] automatically transforms a set of grammar rules into a very fast parsing subroutine.

The words and punctuation of a programming language are represented by **tokens** and, like a natural language grammar rule, an LALR(1) syntactic grammar rule specifies a way these tokens can be placed in order. Two grammar rules are shown in (2.2). Grammar rules in this paper use a notation similar to YACC's. Unlike the sequential execution of C or FORTRAN code, a rule in a grammar executes when the rule pattern matches the input sequence. With these properties LALR(1) grammar rules are general enough to recognize more than programming language syntax rules, and in Section 3 they will be used to recognize code semantics.

```
var   :  var * var
            { calc_prod(); }
      |  var + var                    (2.2)
            { check_add(); }
      ;
```

Each LALR grammar rule (2.2) always has left- and right-hand sides and is terminated with a semicolon. The pattern to be matched is a sequence of one or more tokens, for example, *var* + *var* or *var* * *var* in (2.2), which is located to the right of the colon or vertical bar. To

the left of the colon is a single token called a non-terminal symbol, for example, *var* in (2.2). Action code may follow the pattern and is enclosed in braces, for example, calc_prod() or check_add(). Action code is supplied by the grammar rule writer and is executed when the rule pattern matches. Multiple grammar rules with the same left-hand side may be combined by replacing the colon with a vertical bar, as in (2.2). Every token in a grammar rule has an associated data structure which stores information about the token and may be used by action code. Among other functions, action code can be used to create and manipulate data structures representing the code. Compiler theory is explained by Aho [1,2].

The tokens and their associated concepts are represented in a hierarchical form resulting from token replacement. When the tokens in the input sequence match the pattern of a grammar rule, they are replaced in the sequence with the left-hand side token, for example, *var* in (2.2). This left-hand token can then be part of yet another pattern match, for example, in the case *var* * *var* + *var*. This hierarchy is not only structural but also semantic with leaves involving details and branches involving substantive concepts. In particular, with semantic grammar rules, one can move from an expression containing program variables to a derivative or integral and to a term in a differential equation.

3. Semantic Analysis Procedure

In this section, the details of an automatic semantic analysis procedure are developed. In particular, it is explained how semantic declarations and program code are represented and translated into semantic statements which can be recognized by semantic parsers. Further, the representation, organization, and storage of mathematical, physical, and geometrical equations in multiple semantic parsers is explained. Finally, prototype software for this semantic analysis procedure is demonstrated.

3.0 Semantic Declarations

A program's primitive semantic quantities must be identified to the semantic analysis procedure, and this is done by including **Semantic declarations** within a user's code. For example, the sample user program line

```
C
      VAR = EI + P / RHO + 0.5 * V * V                    (3.0)
C
```

might be supplemented with semantic declarations to become

```
C?      P == pressure_static, RHO == density_static
C?      V == Speed, EI == internal_energy_intens
C                                                      (3.1)
              VAR = EI + P / RHO + 0.5 * V * V
C
```

The **semantic terms** assigned to program variables, for example, pressure_static or density_static in (3.1), are distinct from tokens and come from a lexicon of semantic terms which are similar to the English technical terms. Since these semantic declarations depend on the user's program they must be provided by the user. However, they are the sole adjustment to a user's code. The declarations are distinguished by "C?" in the first two columns and appear to be comments to a FORTRAN compiler.

The user's program (3.0 or 3.1) is syntactically parsed by the semantic analysis procedure into a **parse tree** (Figure 1) based on the syntactic grammar for a programming language. This syntactic parser could be many established programming languages including C, however, FORTRAN is used here.

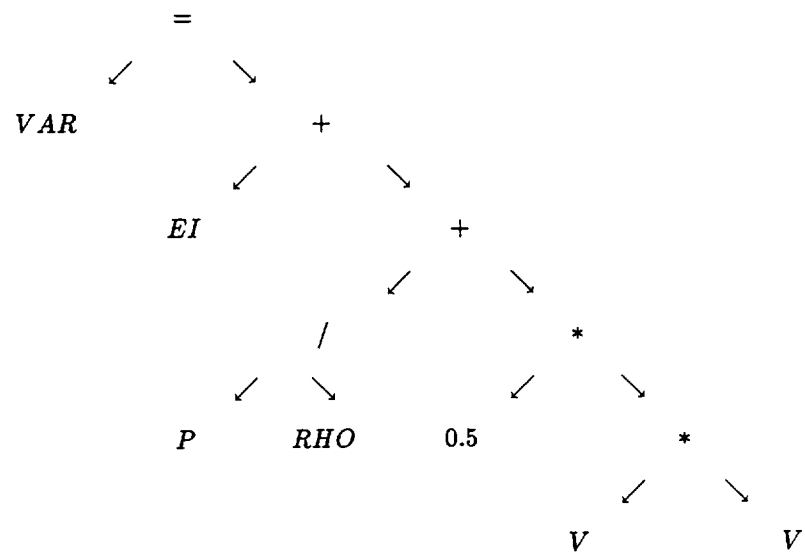

Figure 1 A parse tree representation of the code statement $VAR = EI + P/RHO + 0.5 * V * V$.

3.1 Semantic Initialization—The Annotated Parse Tree

Each leaf and branch node of the parse tree contains a data structure with semantic information. In (3.2) two sample leaf data structures for the variables P and RHO of Figure 1 are shown. A number of specialized slots or members are included in each data structure and

these members correspond to semantic properties of the variable or node. For example, the physical quantity, stencil-wise location, and physical dimensions are some members shown in (3.2). Token values representing semantic concepts are stored in these leaf members, that is, the parse tree is **annotated**.

As a program's semantic declarations are syntactically parsed, the syntactic grammar rule actions transfer the semantic declarations into the corresponding structure members. For example, the declaration of the variable P as pressure_static in (3.1) results in the static pressure semantic token, *static_pressure*, being loaded into the quantity member of the leaf data structure for variable P. The physical dimensions of static pressure, $ML^{-1}T^{-2}$, are known from a semantic program table and are placed in the dimensions member. *RHO* is handled similarly. Unknown or undeclared information is represented by the token, *unknown*.

string:	P	string:	RHO	
quantity:	*static_pressure*	quantity:	*static_density*	
units:	*unknown*	units:	*unknown*	(3.2)
location:	*unknown*	location:	*unknown*	
dimensions:	pointer to $ML^{-1}T^{-2}$	dimensions:	pointer to ML^{-3}	

The **tokens** placed in leaf members of data structures are unique integer values which represent a concept throughout the semantic analysis. Just as "static pressure" is used in the English language as a unique alphabetic representation of the concept of gas pressure, the token *static_pressure* with its integer value is used to uniquely represent static pressure in the semantic analysis. In this way, highly specific properties and concepts may be uniquely represented, recognized, and manipulated. To distinguish tokens in this paper, they are printed in lower case italics.

3.2 Translation of Code to Semantic Statements

Just as the user's program (3.0) may be converted to the parse tree of Figure 1, a branch of the parse tree may be converted back to the original sequence of program elements (3.0). This conversion is a depth-first traversal of the branch while reading the data structure's string member. However, by changing which data structure member (3.2) is read and placed in the token sequence, it is possible to transform the original FORTRAN statements into **semantic statements**. For example, the term *P/RHO*, which is the parse tree branch beyond '/' in Figure 1 (with the member values of (3.2)), is transformed by substitution of

the physical quantity member of the leaf structure into the sequence

$$\textit{static\_pressure},\ /,\ \textit{static\_density},\ \textit{EOS}. \tag{3.3}$$

The sequence is terminated with the punctuation *EOS*. By substituting different members of the leaf structure, a variety of input sequences may be generated. For example, if the dimensions slot is read as the parse tree is traversed, then the token sequence would be

$$M^1 L^{-1} T^{-2},\ /,\ M^1 L^{-3} T^0,\ \textit{EOS}. \tag{3.4}$$

In this way, tokens can be formed into semantic expressions which express different aspects of the code statement, including physical dimensions, grid location, and physical formulae.

3.3 Semantic Grammar Rules

Just as the original program statement (3.0) is parsed by the FORTRAN grammar rules which recognize the acceptable FORTRAN order, the transformed statements, (3.3) and (3.4), may be parsed by grammar rules which recognize semantic patterns. For example, a grammar containing rules (3.5a–f) can recognize the use of a particular formula in the statement fragment (3.3).

speed_squared	:	*speed* * *speed*	(3.5a)
kinetic_energy_intens	:	*half* * *speed_squared*	(3.5b)
work_intens	:	*static_pressure* / *static_density*	(3.5c)
enthalpy_intens	:	*work_intens* + *internal_energy_intens*	(3.5d)
sound_speed_squared	:	*gamma* * *work_intens*	(3.5e)
total_enthalpy_intens	:	*enthalpy_intens* + *kinetic_energy_intens*	(3.5f)

In particular, if the branch of the parse tree in Figure 1 rooted at '/' were transformed to (3.3) and submitted to a parser with the grammar rules (3.5a–f), then rule (3.5c) would recognize this expression. To register this observation, the data structure at the root of the branch being tested—'/' in Figure 1—is annotated by placing the left-hand side token, *work_intens*, in the quantity member.

There is considerable flexibility about which branches can be translated, parsed, and annotated in this manner. In practice, program translations such as (3.3) and (3.4) are performed

during the syntactic parse as the parse tree is constructed. Consequently, subsets of the code statement, corresponding to branches of the parse tree, are compared to the semantic rules. For example, for the parse tree in Figure 1 one would translate and semantically parse the branches corresponding to V^2, $\frac{1}{2}V^2$, $\frac{P}{\rho}$, $\frac{P}{\rho}+\frac{1}{2}V^2$, and $EI+\frac{P}{\rho}+\frac{1}{2}V^2$ at the corresponding points in the syntactic parse. While the syntactic parsing routine is called once for the whole program, a semantic parsing routine is called many times to parse branches involving relatively short token sequences.

3.4 Expert Grammars

Since each rule incorporated into a semantic parser (3.5a-f) will only recognize a specific formula, to increase recognition capabilities, rules must be included which represent fundamental equations, their derivations, and certain variations. The inclusion of rules is perhaps the biggest challenge of developing this analysis procedure.

It is impractical to incorporate the necessary mathematical and physical formulae into a single semantic parsing routine. For simplicity of organization and maintenance, each semantic parsing routine is restricted to the semantic grammar rules for a particular area of expertise, for example, gasdynamics. This set of semantic rules is called a **semantic grammar** or **expert grammar**, and YACC automatically converts it into a subroutine for an **expert parser** or **expert**. Further, any number of areas of expertise can be simultaneously represented with their own expert parsers. Since the semantic grammar rules for an area of expertise can be implemented with some generality, an expert parser can be written once and used by all users.

Annotation of the parse tree with an expert's conclusions stores, organizes, and communicates the semantic information produced by these expert parsers. Each expert parser can study a branch before the branch is enlarged, and any annotation may be used by another expert to recognize a larger portion of an expression. Further, the application of rules to the conclusions of other rules allows a hierarchy of semantic concepts to be constructed which naturally connects details with substantial semantic issues. For example, in the discrete derivative code (3.6) experts would first recognize the differences in U and X, and then the ratio of these differences.

$$DUDX(I) = (U(I+1)-U(I))/(X(I+1)-X(I)) \tag{3.6}$$

Further experts could recognize the use of this discrete derivative in many different equations.

Due to its specialization, an expert grammar must parse unfamiliar tokens from outside its area of expertise. Further, there may be undeclared semantic quantities in the parse tree. The token sequence sent to an expert parsing routine is filtered so that unfamiliar or undeclared tokens are translated to the unknown token, *unknown*. Each grammar contains rules (3.7) to parse these unknown quantities.

```
notknown  :  unknown
          |  notknown * sem_expr
          |  notknown + sem_expr
          |  notknown - sem_expr
          |  notknown / sem_expr
          |  sem_expr * notknown
          |  sem_expr + notknown
          |  sem_expr - notknown
          |  sem_expr / notknown
          |  ( notknown )
          ;
```
(3.7)

3.5 Development of Expert Parsers

In the following subsections, three expert parsers are described for analyzing different semantic aspects of physical formulae. **Aspects** are distinctly different semantic properties of code which must be satisfied. The first expert analyzes the physical dimensions aspect of program statements. Although this test is relatively simple, it provides a very useful test of correctness. The second aspect is recognition of physical formulae, and it is demonstrated with a gasdynamics expert using rules similar to (3.5a–f). Third, geometrical location is determined for array variables defined on a structured grid.

3.5.1 Dimensional Analysis Expert Parser

Physical dimensional analysis provides a necessary semantic test for the correct use of all physical formulae. The code (3.8)

```
C
C?      CC == sound_speed_squared, GAM == ratio_of_specific_heats
C?      RHO == density_static, PROFIL == total_enthalpy_intens
C
        H = RHO * PROFIL - CC / GAM
C
```
(3.8)

would be flagged by this expert as incorrect since the two right-hand side terms have different dimensions. The proper code should have parentheses starting before PROFIL and ending

after GAM.

The dimensional analysis parser's input token sequence contains operators, the token *var* for a known dimension and *unknown* for an unknown one, and this sequence is generated as explained in Section 3.2. *Var's* data structure includes a pointer to its physical dimensions that is available to action code in each grammar rule. The prototype dimensional analysis grammar is the combination of the rules in (3.7) and Figure 2.

```
sem_stmt  :  sem_expr
          |  notknown
          |  sem_stmt sem_expr
          |  sem_stmt notknown
          ;
sem_expr  :  var
          |  var = var
               { check dimensional equality of operands }
          |  notknown = var
               { assign right-hand side dimensions to notknown }
          |  sem_expr EOS
          ;
var       :  var
          |  var * var
               { calculate dimensions of product }
          |  var / var
               { calculate dimensions of quotient }
          |  var + var
               { check dimensional equality of operands }
          |  var - var
               { check dimensional equality of operands }
          |  var ** var
               { check exponent dimensionless;
                 calculate dimensions of result }
          ;
```

Figure 2 Grammar rules for the dimensional analysis expert parser. The rules (3.7) also appear in this expert parser.

In this grammar, action code calculates the dimensions of products and quotients and verifies the equality of operand dimensions on addition, subtraction, and assignment. Unit checking is a potential extension of dimensional analysis.

3.5.2 Formula Recognition Expert Parsers

The formula expert parsers are designed to recognize the use of physical formula in different areas of expertise. In the code

```
C
      P = RHO * ( E - ( U * U + V * V ) ) * ( GAM - 1. )          (3.9)
C
```

if it has been specified or deduced elsewhere that RHO is static density, E is total energy (intensive), U and V are x- and y-velocity, and GAM is the ratio of specific heats, then this code is dimensionally correct. However, (3.9) is not the proper formula for static pressure in two dimensions. If P is declared to be static pressure the expert parser would note an error, and if P is undeclared the code would be noted as misunderstood.

The expert's input token sequence is generated by reading the quantity member during depth-first traversal of a parse tree branch. Each expert grammar has a filter which translates unfamiliar or undeclared tokens to *unknown*. One grammar corresponding to the gasdynamics formulae (3.5a–f) includes the rules in (3.7) and Figure 3. The prototype expert grammar which recognizes gasdynamics equations contains many more rules; however, its structure remains the same. Although Figure 3 excludes the action code, each formula rule has an annotation action which is to annotate the parse tree with the left hand side token.

3.5.3 Location Analysis Expert Parser

The location analysis expert parser recognizes and analyzes the discrete geometrical location of variables and expressions on a structured grid. This information is necessary for recognition and checking of spatial formulae such as spatial derivatives and integrals and the analysis of their accuracy. The expert parser recognizes a subscript notation used by many numerical methods to represent both the physical formulae and, with the subscript, the discrete geometry. For a discrete second difference of a variable, ϕ, the notation would be

$$\phi_{i+1} - 2\phi_i + \phi_{i-1} \tag{3.10a}$$

where i indexes both a discrete coordinate line of points on a structured grid and an array containing a structured grid.

```
sem_stmt        :  sem_expr
                |  notknown
                |  sem_stmt sem_expr
                |  sem_stmt notknown
                ;
sem_expr        :  speed_squared
                |  kinetic_energy_intens
                |  work_intens
                |  enthalpy_intens
                |  ...
                |  total_enthalpy_intens
                |  notknown = notknown
                |  notknown = sem_expr
                      { assign right-hand side quantity to notknown }
                |  sem_expr EOS
                |  ( sem_expr )
                ;
speed_squared   :  speed * speed
                |  velocity_x * velocity_x + velocity_y * velocity_y
                      { verify coordinate system, number of dimensions }
                |  velocity_x * velocity_x + velocity_y * velocity_y
                   + velocity_z * velocity_z
                      { verify coordinate system, number of dimensions }
                ;
kinetic_energy_intens  :  half * speed_squared
                ;
work_intens     :  static_pressure / static_density
                |  internal_energy_intens * ( gamma - one )
                ;
enthalpy_intens :  internal_energy_intens + work_intens
                ;
sound_speed_squared  :  gamma * work_intens
                ;
total_enthalpy_intens  :  enthalpy_intens + kinetic_energy_intens
                ;
internal_energy_intens  :  total_energy_intens - kinetic_energy_intens
                ;
pressure        :  density * work_intens
                ;
```

Figure 3 A subset of grammar rules for the gasdynamics expert parser. The rules (3.7) also appear in this expert parser.

The code (3.10) contains an indexing error,

```
C
      FS(I,J,N) = DW(I+2,J,N) + DW(I-1,J,N) - 2.*DW(I,J,N)          (3.10b)
C
```

and it would not be recognized as a discrete second difference by this expert parser. Depending on the declaration of FS, an error would be declared, or the code would be declared not understood.

Semantic declaration of an array is different than for a scalar variable since the array may have multiple semantic definitions depending on an array index value. The LIST construct is used to create a list or vector of semantic definitions for the index of an array. In (3.11) the LIST construct attaches to the indices of arrays P and X semantic variables representing lines of centroid and vertex coordinates in a structured grid.

```
C
C?      I_C_INDEX == LIST { centroid_line_i }
C?      J_C_INDEX == LIST { centroid_line_j }
C?      P[ I_C_INDEX, J_C_INDEX ]
C                                                    (3.11)
C?      I_V_INDEX == LIST { vertex_line_i }
C?      J_V_INDEX == LIST { vertex_line_j }
C?      X[ I_V_INDEX, J_V_INDEX ]
```

A separate grammar must interpret this semantic declaration for each array reference in a program and determine a stencil-wise location, for example, East_centroid or North_West_vertex. The parse tree is annotated with this result, and the location analysis grammar (Figure 4) uses the value in its input token sequence. In particular, for (3.10b) the indices and expression would be translated to *East_East_centroid* + *West_centroid* - *two* * *Center_centroid*, and this expression cannot be simplified to the intended result, *Center_centroid*.

The grammar shown in Figure 4 is a subset of the prototype location analysis grammar; however, it shows the essence of the location analysis rules. An annotation action is associated with each rule of Figure 4.

3.6 Prototype Semantic Analysis Procedure

A prototype semantic analysis procedure has been constructed by integrating the methods explained in Section 3. In particular, this software uses a syntactic parser to build the parse tree for a user's code. Further, it uses the semantic declarations as initial annotations of this parse tree, and generates semantic statements from these annotations. Extensions of the expert parsers described in Section 3.5 examine the semantic statements and further annotate the parse tree with their conclusions. These elements are accessed and controlled by a graphical user interface (GUI).

```
sem_stmt        : sem_expr
                | notknown
                | sem_stmt sem_expr
                | sem_stmt notknown
                ;
sem_expr        : East_vertex
                | East_centroid
                | Center_centroid
                | West_vertex
                | West_centroid
                | sem_expr EOS
                | ( sem_expr )
                | notknown = sem_expr
                    { assign right-hand side location to notknown }
                ;
East_vertex     : East_centroid + Center_centroid
                    { If quantities differ then return }
                | East_centroid - Center_centroid
                    { If quantities differ then return }
                | East_vertex * East_vertex
                | East_vertex / East_vertex
                | Anywhere * East_vertex
                ;
West_vertex     : Center_centroid + West_centroid
                    { If quantities differ then return }
                | Center_centroid - West_centroid
                    { If quantities differ then return }
                | West_vertex * West_vertex
                | West_vertex / West_vertex
                | Anywhere * West_vertex
                ;
Center_centroid : East_vertex - West_vertex
                    { If quantities differ then return }
                | East_centroid + West_centroid
                    { If quantities differ then return }
                | Center_vertex * Center_vertex
                | Center_vertex / Center_vertex
                | Anywhere * Center_centroid
                ;
```

Figure 4 A subset of the grammar rules for the location analysis expert parser. The rules (3.7) also appear in this expert parser.

In practise, users submit their code and its semantic declarations through the GUI. The result of the analysis is an annotated parse tree, that is, the parse tree with the node data structures (3.2) completed with declared and deduced information. The GUI allows the user to display this information which includes the physical quantity represented by a variable or expression, as well as its physical dimensions, location, and the formula it was recognized

from. A dictionary provides definitions for the semantic tokens. Of course, the available information is limited by the extent of semantic declarations and expert grammar rules as will be explained in Section 4.

This semantic analysis software not only detects some semantic errors but also documents some semantic aspects of the code. The code lines (3.12)

```
C
      PY           = (PX*GXY +QXY)*SX(J)
      P(IWOUT,J) = DIM(P(IWALIN,J),PY)                    (3.12)
C
```

are obscure since they involve derived quantities, and it is not clear which formulae are used. However, the GUI allows the user to display what has been defined and deduced about a variable or expression.

4. Theory

In this section theoretical analyses are used to move beyond specific application and implementation details and understand more general properties of this semantic analysis procedure. Several of the experts have a very simple theoretical analysis. The physical dimensions expert (Figure 2) parses token sequences composed of only seven tokens where five are operators, and it requires no more than a few rules. However, geometrical location analysis (Figure 4) is more complex due to the larger number of potential locations in a stencil and the ways they may be combined. Formula analysis (Figure 3) is the most complex due to the number of physical quantities involved, the algebraic rules satisfied and the equations which can be derived. The following analysis is specialized to formula analysis. In particular, this section demonstrates that the transformations permitted by an LALR(1) grammar are a subset of the fundamental transformations for algebraic equations. The limited capabilities of these grammar rules has implications for what equations must be included in an expert parser, error detection, and computational complexity.

4.1 Rewrite and Reduction Substitution Rules

A grammar rule which specifies the transformation of a sequence of tokens satisfying one pattern into a sequence of tokens satisfying another pattern is a rewrite rule—the token sequence is rewritten. LALR(1) grammar rules are a specialized type of rewrite rules, referred

to here as **reduction substitution** rules since they rewrite and reduce one or more input tokens to a single token, never increasing the number of tokens in the token sequence as it is recognized and simplified with token replacement. The rule (4.1a) is a reduction substitution rule; however, rules (4.1b,c) are not, and could not be implemented in an LALR(1) parser.

$$\begin{aligned} &\textit{speed\_squared} && : \textit{speed} * \textit{speed} && (4.1a)\\ &\textit{speed} * \textit{speed} && : \textit{speed\_squared} && (4.1b)\\ &\textit{enthalpy} + \textit{kinetic\_energy} && : \textit{kinetic\_energy} + \textit{enthalpy} && (4.1c) \end{aligned}$$

Rules (4.1a–c) can be implemented with an expert system.

4.2 Transformation of Equations

The equations represented by LALR(1) grammar rules in expert parsers are generally neither reduction substitution rules nor order dependent. The expressions comprising the left- and right-hand sides of equations have the algebraic properties (4.2) [10].

$$\begin{array}{rcll} a+b & = & b+a & \text{Commutative Law of } + \ (\times) \\ (a+b)+c & = & a+(b+c) & \text{Associative Law of } + \ (\times) \\ a\times(b+c) & = & a\times b+a\times c & \text{Distributive Law} \end{array} \qquad (4.2)$$

Further, equations satisfy the reflexive, symmetric, and transitive properties as well as (4.3a–e) [10],

$$\begin{aligned} &\text{If } E_1 = E'_1,\ E_2 = E'_2 \quad \text{then} \quad E_1 = E_2 \text{ and } E'_1 = E'_2 \\ &\qquad\qquad\qquad\qquad\qquad\quad \text{are equivalent equations} && (4.3a)\\ &\text{If } E_1 = E_2,\ \exists\, E_3 \quad \text{then} \quad E_1 + E_3 = E_2 + E_3 && (4.3b)\\ &\text{If } E_1 = E_2,\ \exists\, E_3 \quad \text{then} \quad E_1 - E_3 = E_2 - E_3 && (4.3c)\\ &\text{If } E_1 = E_2,\ \exists\, E_3 \quad \text{then} \quad E_1 * E_3 = E_2 * E_3 && (4.3d)\\ &\text{If } E_1 = E_2,\ \exists\, E_3 \neq 0 \quad \text{then} \quad E_1/E_3 = E_2/E_3 && (4.3e) \end{aligned}$$

where E_1, E_2, E_3, E'_1 and E'_2 are expressions with identical domains of definition. These properties are fundamental to the manual derivation of expressions and equations. Further, the capabilities of a semantic recognition procedure depend on both the fundamental equations and these transformations.

4.2 Commutative and Distributive Laws

Since the pattern of each grammar rule is order dependent, the pattern matching of formulae must allow for the commutative and distributive laws (4.2). For example, for the code

fragment $P * GAMMA/RHO$ the branch representation may be the left-hand side of Figure 5, but the corresponding token sequence, *static_pressure* * *gamma* / *static_density*, cannot be parsed by the rules (3.5a–f) due to token order. One commutative transformation of the branch is the right-hand side of Figure 5,

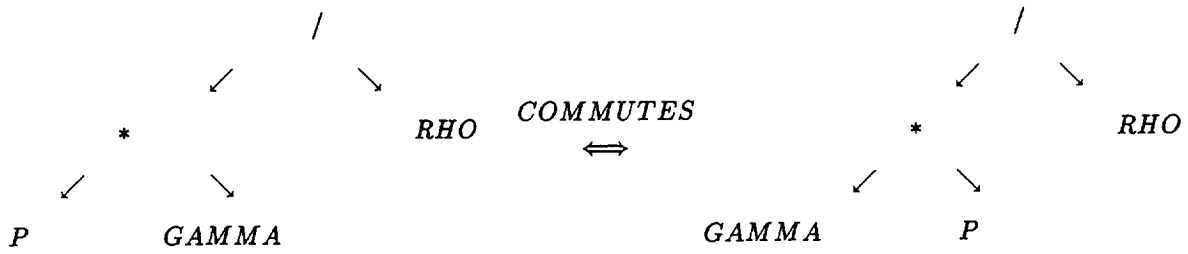

Figure 5 Commutative transformations of the parse tree representation of the code fragment $P * GAMMA/RHO$

and the resulting token sequence, *gamma* * *static_pressure* / *static_density*, can be parsed by (3.5a–f).

A multiplicative or additive expression involving n tokens can be rearranged by the commutative law in up to $n!$ ways. To include all permutations of a formula as grammar rules is impractical for formulae involving a large number of terms. In this paper, an expression's tree representation is transformed into each of the possible permutations, and formula recognition is attempted by each expert. It may be possible to define a normal form [9] for each grammar rule, where a function, $\mathcal{O}(token)$, provides an ordering of tokens which determines the recognizable permutation. However, the token ordering, $\mathcal{O}(token)$, depends on all the grammar rules and their combinations.

The distributive law (4.2) is accounted for by transforming an expression's tree representation into each possible alternate form, and formula recognition is attempted by each expert. For example, for the code $2p_1 - 2p_2$, the binary tree representation may be the LHS of Figure 6, and the distributive transformation would be the right-hand side for which formula recognition is possible.

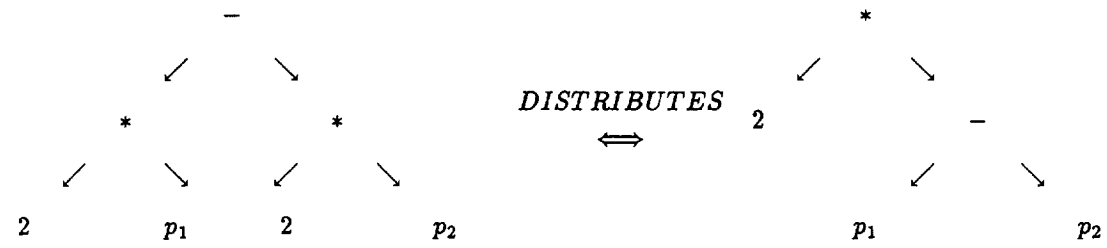

Figure 6 Distributive transformations of the parse tree representation of the code fragment $2p_1 - 2p_2$

4.3 Equation Equivalence Transformations

The equation properties (4.3) are identical to the substitution and solve equation manipulations (4.6a,b)

$$\text{If } E_1 = E_2,\ \exists F + E_1 \quad \text{then } F + E_1 = F + E_2 \quad \text{Substitution} \quad (4.6a)$$

$$\text{If } E_1 = E_2 + E_3,\ \exists E_2 \text{ inverse} \quad \text{then } E_3 = E_1 - E_2 \quad \text{Solve} \quad (4.6b)$$

where E_i, F are expressions containing one or more terms. These rules apply similarly for subtraction, multiplication and division. A special case of the substitution manipulation is the reduction substitution

$$\text{If } T = E,\ \exists F + E \quad \text{then } F + E = F + T \quad \text{Reduction Substitution} \quad (4.6c)$$

where T is a single term, and the rule is also valid for subtraction, multiplication and division.

The transformations (4.2) and (4.6a,b) allow general algebraic derivations. For example, the code (4.7)

```
C?      G == ratio_of_specific_heats, M1 == mach
C
        G = 1.4
        GAM1 = 0.5 * ( G - 1. )
C                                                              (4.7)
        ZED = SQRT( 1. - 2.*(G + 1.) * ( M1**2 * (1. +
       .GAM1 * M1 *M1) / ( 1. + G * M1 * M1 )**2))
C
        M2 = SQRT( (1. - ZED) / ( 1 + G * ZED ))
```

represents the equation

$$z = \left(1 - 2(\gamma+1)\frac{M_1^2(1+\frac{\gamma-1}{2}M_1^2)}{(1+\gamma M_1^2)^2}\right)^{\frac{1}{2}} \quad (4.8)$$

$$M_2 = (\frac{1+z}{1+\gamma z})^{\frac{1}{2}}$$

which is derived from basic aerodynamics and fluid mechanics formulae [7] with the algebraic transformations (4.2) and (4.6a,b).

4.4 Consequences of Using the Reduction Substitution

Choosing to recognize semantic expressions with a LALR(1) parser has a number of consequences for designing rules for expert parsers, error detection, computational complexity, and finite termination of the expert parsers.

The basic physical formulae for the derivation of (4.8) are contained in the prototype expert parser of Section 3.5.2. However, an LALR(1) grammar allows only the reduction substitution (4.6c), and the derivation of (4.8) requires all the transformations (4.6a–c). Consequently, (4.7) cannot be recognized from basic physical formulae by the expert parsers, and the equation (4.8) must be included in the expert parser for recognition to occur. Further, without the theoretical ability to recognize all conceivable derivable physical formulae, it is not guaranteed that the expert parsers can distinguish between an incorrect expression and an unrecognizable yet correct expression. For example, without (4.8) included in an expert parser, the code (4.7) could not be recognized. Consequently, the equations included in an expert parser must be carefully written and more extensive than the fundamental physical equations of a field.

Manual derivation of (4.8) from fundamental aerodynamic and fluid flow equations is a nontrivial search even with the guidance of a derivation. Several substitutions must be manually attempted for each step of the derivation. Consequently, including all the transformations (4.6a–c) and allowing general derivations could lead to expensive searches. Currently, it is preferable to include equations, for example (4.8), in an expert parser once and avoid expensive searches each time the formula is encountered.

Precluding expensive derivations by using only reduction substitutions is not sufficient to yield a linear computational performance. Although the execution time of the expert parsing routines generated by YACC is linear in the number of input tokens, because of the commutative and distributive properties (4.2), the expert parsers may need to see transformations of an expression before recognizing it. Since there could be $n!$ distinct commutative permuta-

tions of n terms in an additive or multiplicative expression, the computational complexity of a semantic analysis can increase beyond linear performance. The worst case performance occurs for expressions which are not recognized, since all possible commutative and distributive transformations of the expression will be considered.

A further consequence of using the reduction substitution is the finite termination of the expert parser. Assuming the reduction substitution rules of the expert parser do not rewrite one token as another single token, then each application of a reduction substitution rule will reduce the number of tokens until termination. At termination either all the tokens have been completely parsed and a single token remains, or some token sequence remains where no patterns are recognized. For rule systems involving general rewrite rules (4.1a–c), it is not as obvious that a sequence will be reduced or that the application of rewrite rules will terminate after a finite number of steps. Considerable theoretical analysis has been applied to this problem [5] for computer algebra systems.

5. Results

In this section the procedure for semantic checking is demonstrated with three test problems. The first problem (Figure 7) demonstrates the gasdynamics expert's ability to recognize gasdynamics equations. The second problem (Figure 8) involves code for the calculation of the discrete, integral convective terms of the Euler equations in a computational fluid dynamics code. The expert for determining geometrical location within a structured grid stencil works with the expert for recognizing integral quantities and the terms of Euler's equations. The third case (Figure 9) is a set of discrete, one-dimensional differential equations taken from fluid dynamics and applied mathematics. Each of these test cases run on engineering workstations in less than a second of CPU time.

The result of each analysis is an annotated parse tree which cannot be completely displayed in a figure. Instead, the right most column of each figure displays the physical quantity deduced for each code line. More detailed information is contained in the annotated parse tree, including the analysis of variables and sub-expressions. The suffixes pum, put, and nd represent per-unit-mass, per-unit-time, and non-dimensional respectively.

```
C?      P == pressure_static, RHO == density_static
C?      GAM == ratio_of_specific_heats
C?      U == speed
C?      M == mach
C?      C == sound_speed
C?      T == temperature_static
C?      R == gas_constant
C?      VOL == volume
C?      CP == specific_heat_cp
C?      CV == specific_heat_cv
C?      A == area
C?      E == energy_internal_pum
C
        vara = (GAM / (GAM - 1.))                          γ/(γ-1)
        varb = M*C                                         Speed
        varc = RHO*U*U + P                                 Momentum_Euler
        vard = (GAM / (GAM - 1.))* P / RHO + 0.5 * U*U     Enthalpy_Total_pum
        vare = 1. + 0.5*(gam-1.)*M*M                       Enthalpy_Total_nd
        varf = 1. + gam*M*M                                Force_Euler_nd
        varg = 1. + M*gam*M                                Force_Euler_nd
        varh = RHO * U * A                                 Mass_put
C
        vari = GAM * P / RHO                               Sound_Speed_Squared
        varj = P / RHO                                     Work_pum
        vark = P / RHO + P / (RHO * (GAM-1.))              Enthalpy_pum
        varl = P / RHO + (1. /(GAM-1.)) * P / RHO          Enthalpy_pum
        varm = P / RHO + (1. /(GAM-1.)) * (P / RHO)        Enthalpy_pum
        varn = E + 0.5 * U*U                               Energy_Total_pum
        varo = RHO * R * T                                 Pressure_Static
        varp = R * T / VOL                                 Pressure_Static
        varq = CP - CV                                     Gas_Constant (R)
        varr = CP / CV                                     Ratio_of_Specific_Heats
C
        vars = c - 0.5 * u*( gam - 1. )                    Riemann_Invariant_A
        vart = c + 0.5 * u*( gam - 1. )                    Riemann_Invariant_B
C
        varu = P /(RHO**GAM)                               Entropy
C
```

Figure 7 The first semantic analysis test case contains gasdynamics equations. The right column displays the physical quantity deduced for the expression.

The first problem (Figure 7) demonstrates one expert's ability to recognize gasdynamics equations. The semantic declarations precede the code, and the equations involve scalar variables without specified locations. Commutative variations of several gasdynamics formulae are included to demonstrate the role of the commutative law in formula recognition. As a result of the analysis, each code expression is recognized by the gasdynamics expert as an instance of a particular gasdynamics formula. Further, a dimensional analysis is per-

formed. These conclusions are stored as annotations of the parse tree. In a Graphical User Interface (GUI) associated with this semantic analysis, these semantic conclusions may be examined.

```
C?   I_C_INDEX == LIST { centroid_line_i }
C?   J_C_INDEX == LIST { centroid_line_j }
C?   I_V_INDEX == LIST { vertex_line_i }
C?   J_V_INDEX == LIST { vertex_line_j }
C
C?   COORDS == LIST { x_coord, y_coord }
C?   VECTOR == LIST { density_static, x_momentum_puv,
     .     y_momentum_puv, energy_total_puv }
C
C?   W[I_C_INDEX,J_C_INDEX,VECTOR]
C?   P[I_C_INDEX,J_C_INDEX]
C?   X[I_V_INDEX,J_V_INDEX,COORDS]
C?   P == pressure
C
C?   I == counter, J == counter
C
     DIMENSION W(20,20,4), P(20,20), X(20,20,2)
C
     ZZ = X(I,J,2) - X(I,J-1,2) + P(I,J)                          Not Understood
     XY = X(I,J,1) - X(I,J-1,1)                                   X_Delta
     YY = X(I,J,2) - X(I,J-1,2)                                   Y_Delta
     PA = 0.5 * ( P(I+1,J) + P(I,J) )                             Pressure_Static
     QSP = (YY*W(I+1,J,2) - XY*W(I+1,J,3))/W(I+1,J,1)             Volume_put
     QSM = (YY*W(I,J,2) - XY*W(I,J,3))/W(I,J,1)                   Volume_put
     FS1 = 0.5*( QSP*W(I+1,J,1) +QSM*W(I,J,1) )                   Mass_put
     FS2 = 0.5*( QSP*W(I+1,J,2) +QSM*W(I,J,2) ) + YY*PA           Force_X_Euler
     FS3 = 0.5*( QSP*W(I+1,J,3) +QSM*W(I,J,3) ) - XY*PA           Force_Y_Euler
     FS4 = 0.5*( QSP*(W(I+1,J,4)+P(I+1,J))+                       Enthalpy_Tot_put
    .          QSM*(W(I,J,4)+P(I,J)))
C
```

Figure 8 The second semantic analysis test case represents finite volume calculations for the Euler equations. The right column displays the physical quantity deduced for the expression.

The second problem (Figure 8) demonstrates the analysis of discrete integral equations and associated grid locations. It includes an error in the first code line, and demonstrates the use of array notation. In particular, the LIST construct introduced in Section 3.5.3 is used to define a semantic variable, COORDS, as the vector (x-coordinate, y-coordinate). When COORDS is associated with an array index as in X[I_INDEX,J_INDEX,COORDS], the third array index of X is defined to have two values: x- and y-coordinate. The semantic variable VECTOR is similar and is defined to represent the vector of dependent variables of fluid dynamics equations, $(\rho, \rho u, \rho v, \rho E)$. The semantic variables I_V_INDEX et al. are

defined as in Section 3.5.3 to represent coordinate lines in a general structured grid. These semantic variables are used to define the indices of arrays W, P, and X, and associate semantic definitions with each index value.

With these array definitions, the semantic analysis process is able to recognize the variables and the use of physical formulae, in particular that XY and YY are differences in x- and y-coordinates, QSP and QSM are the fluid volumes flowing through the face per unit time, and FS1–4 are the components of a discrete approximation to the integral $\oint F \cdot nds$ where $F = (f, g)$, $f = (\rho u, \rho u^2 + p, \rho uv, \rho uH)$, and $g = (\rho v, \rho uv, \rho v^2 + p, \rho vH)$. A dimensional analysis is also performed. All of these conclusions are stored as annotations of the parse tree and are available for examination in the GUI. The analysis also notes that the line involving the variable ZZ is neither dimensionally correct nor a known formula.

The third problem (Figure 9) demonstrates the recognition of several discrete, one-dimensional differential equations from fluid dynamics. Again, array variables are involved and provide stencil locations for use in the analysis, particularly in the recognition of differences and first and second derivatives. In each case the expert recognizes the mathematical formula. As always, a dimensional analysis is performed simultaneously.

6. Conclusions

This paper is motivated by a need for improved tools for verification and documentation of scientific and engineering codes, and it investigates methods for automatically recognizing, checking, and documenting semantic concepts used in these codes. In particular, a scheme and prototype code are presented that recognize and check some mathematical, physical and geometrical formulae. A theoretical analysis of this approach is also presented, and the method is demonstrated with three test cases.

The prototype semantic analysis procedure proves the basic principles of this semantic analysis technique, however additional work is required and a number of issues remain unexplored. First, additional semantic knowledge must be incorporated into the expert parsers to expand recognition capabilities. This semantic knowledge is equations and formulae from additional scientific and engineering fields. Second, there are several semantic aspects which have not

```
C?      I_V_INDEX == LIST { vertex_line_i }
C?      I_C_INDEX == LIST { centroid_line_i }
C?      I == counter
C?      X[I_V_INDEX], P[I_V_INDEX], U[I_V_INDEX]
C?      PHI[I_V_INDEX], DX[I_C_INDEX], DPDX[I_C_INDEX]
C?      DUDX[I_C_INDEX], DPHIDX[I_C_INDEX]
C?      X == x_coord, P == pressure_static, U == speed
C?      RHO == density_static, MU == viscosity
C?      DT == time_step
C
        DIMENSION X(20), P(20), DX(20), U(20)
        DIMENSION DPDX(20), DUDX(20), DUDX1(20)
C
        DX(I) = X(I+1) - X(I-1)                                  X_Delta
        DP = P(I+1) - P(I-1)                                     Press_Delta
        DPDX(I) = DP / DX(I)                                     Press_Deriv_X
        DU = U(I+1) - U(I-1)                                     Speed_Delta
        DUDX(I) = DU / DX(I)                                     Speed_Deriv_X
C
        DU = U(I+1) - U(I)                                       Speed_Delta
        DUDX1(I)= DU / (X(I+1)-X(I))                             Speed_Deriv_X
        DXX = 0.5*((X(I+1)+X(I))-(X(I)+X(I-1)))                  X_Delta
        D2UDX2 = (DUDX1(I) - DUDX1(I-1)) / DXX                   Speed_2Deriv_X
C
C       BURGERS, EULER AND NS EQUATIONS
C
        VAR = U(I) + DT * (U(I) * DUDX(I))                       Speed + Time_Step
                                                                 * Burgers
        VAR = U(I) + DT * (U(I) * DUDX(I)                        Speed + Time_Step
        . - ( 1. / RHO )*DPDX(I))                                * Euler
        VAR = U(I) + DT * (U(I) * DUDX(I) -                      Speed + Time_Step
        . ( 1. / RHO )*DPDX(I) - ( MU / RHO ) * D2UDX2)          * Navier_Stokes
C
```

Figure 9 The third semantic analysis test case includes one-dimensional finite difference approximations for several fluid dynamics equations. The right column displays the physical quantity deduced for the expression.

been explored, including the accuracy of discrete approximations, the consistency of physical assumptions, more complex geometrical concepts, the counting arguments of program loops, conditional statements, and branching. Further work is necessary to bring this technique to users with sufficient utility and convenience.

8. Acknowledgements

This work supported by the Propulsion Systems Base Program at NASA Lewis Research Center through the Computing and Interdisciplinary Systems Office (contract NAS3-27816). Greg Follen and Austin Evans were the monitors. The lexical analysis and FORTRAN syntactic analysis routines are taken from Ftnchek [18]. Wilma Graham has patiently proofread this manuscript. The author thanks Edmane Envia, Rodrick Chima, Jay Horowitz, and Jack Wilson for fruitful discussions. Ambady Suresh, Kevin Lamb and Scott Townsend are due many thanks for considerable direction, advice, and constructive criticism.

9. Bibliography

1. A. V. Aho and S. C. Johnson, LR Parsing, *Computing Surveys* **6**, 2 (1974).

2. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools* (Addison-Wesley, Reading, 1986).

3. J. Allen, *Natural Language Understanding* (Benjamin/Cummings, Menlo Park, 1987)

4. D. Bobrow, *Qualitative Reasoning about Physical Systems* (MIT Press, Cambridge, 1985).

5. B. Buchberger and R. Loos, "Algebraic Simplification," in *Computer Algebra—Symbolic and Algebraic Computation* (Springer-Verlag, New York, 1982), p. 11.

6. S. I. Feldman, "Make–A Program for Maintaining Computer Programs," Comp. Sci. Tech. Rep. No. 57. (AT&T Bell Laboratories, Murray Hill, 1977).

7. J. V. Foa, *Elements of Flight Propulsion* (Wiley, New York, 1960), p. 164.

8. C. L. Forgy, Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, *Artificial Intelligence* **19** (1982).

9. K. O. Geddes, S.R. Czapor, and G. Labahn, *Algorithms for Computer Algebra* (Kluwer Academic, Boston, 1992).

10. W. Gellert, H. Kustner, M. Hellwich, H. Kastner, *The VNR Concise Encyclopedia of Mathematics* (Van Nostrand Reinhold, New York, 1977), p. 84.

11. V. Haase, R. Messnarz, G. Koch, H. Kugler, P. Decrinis, Bootstrap: Fine-Tuning Process Assessment, *IEEE Software* July (1994).

12. P. Jackson, *Introduction to Expert Systems* (Addison-Wesley, Reading, 1986).

13. S. C. Johnson, "Yacc—Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32. (AT&T Bell Laboratories, Murray Hill, 1975).

14. S. C. Johnson, "Lint, A C Program Checker," Comp. Sci. Tech. Rep. No. 65. (AT&T Bell Laboratories, Murray Hill, 1977).

15. D. E. Knuth, "The WEB System of Structured Programming," Computer Science Dept. Rep. STAN-CS-TR-83-980, (Stanford University, Stanford, 1983).

16. J. R. Levine, T. Mason, D. Brown, *Lex and Yacc* (O'Reilly, Sebastopol, 1992).

17. D. C. Luckham, S. German, F. Henke, R. Karp, P. Milne, D. Oppen, W. Polak, S. Scherlis, "Stanford PASCAL Verifier User Manual," Computer Science Dept. Rep. STAN-CS-79-731, (Stanford University, Stanford, 1979).

18. R. K. Moniot, "ftnchek," unpublished except for http://www.dsm.fordham.edu/~ftnchek (Fordham University, New York, 1989).

19. W. Polak, "Program Verification at Stanford: Past, Present, Future," Article in *Workshop on Artificial Intelligence* Informatik Fuchberichte 47, (Springer-Verlag, Berlin, 1981), p. 256.

20. H. E. Pople, "On the Mechanization of Abductive Logic," *Third International Joint Conference on Artificial Intelligence*, 1973.

21. H. E. Pople, J. D. Myers, R. A. Miller, "DIALOG: A Model of Diagnostic Logic for Internal Medicine," *Fourth International Joint Conference on Artificial Intelligence*, 1975.

22. P. B. Sheridan, The arithmetic translator-compiler of the IBM FORTRAN automatic coding system, *Com. ACM* **2:2**, 9-21 (1959).

23. P. H. Winston, *Artificial Intelligence* (Addison-Wesley, Reading, 1984).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1998	Final Contractor Report

4. TITLE AND SUBTITLE

A Semantic Analysis Method for Scientific and Engineering Code

5. FUNDING NUMBERS

WU-509-10-11-00
NAS3-27816

6. AUTHOR(S)

Mark E.M. Stewart

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

NYMA, Inc.
2001 Aerospace Parkway
Brook Park, Ohio 44142

8. PERFORMING ORGANIZATION REPORT NUMBER

E-11149

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135-3191

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA CR—1998-207402

11. SUPPLEMENTARY NOTES

Project Manager, Joseph Veres, Computing and Interdisciplinary Systems Office, NASA Lewis Research Center, organization code 2900, (216) 433-2436.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Categories: 64 and 59 Distribution: Nonstandard

This publication is available from the NASA Center for AeroSpace Information, (301) 621-0390.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

This paper develops a procedure to statically analyze aspects of the meaning or semantics of scientific and engineering code. The analysis involves adding semantic declarations to a user's code and parsing this semantic knowledge with the original code using multiple expert parsers. These semantic parsers are designed to recognize formulae in different disciplines including physical and mathematical formulae and geometrical position in a numerical scheme. In practice, a user would submit code with semantic declarations of primitive variables to the analysis procedure, and its semantic parsers would automatically recognize and document some static, semantic concepts and locate some program semantic errors. A prototype implementation of this analysis procedure is demonstrated. Further, the relationship between the fundamental algebraic manipulations of equations and the parsing of expressions is explained. This ability to locate some semantic errors and document semantic concepts in scientific and engineering code should reduce the time, risk, and effort of developing and using these codes.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software engineering; Computational fluid mechanics	35
	16. PRICE CODE: A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	