

2013 International Conference on Computational Science

Self-Checking Spreadsheets: Recognition of Semantics

M. E. M. Stewart*

Westlake, Ohio 44145, USA

Abstract

This paper demonstrates a self-checking (self-validating) spreadsheet. This checking analyzes the meaning or semantics of the spreadsheet's variables and equations using a parsing scheme. These semantics go beyond dimension, unit, and type checking to include the physical and mathematical formulae that dominate science, engineering, and mathematics. The spreadsheet is a client, JavaScript web application working with a server application. Entries in the spreadsheet are analyzed by semantic parsers on the server, and the representation and recognition of spreadsheet semantics are detailed. The intent of this prototype is to reduce the errors—errors in meaning—that commonly occur when spreadsheets are used. A prototype has been available on the Internet since early 2010 at <http://semantics.grc.nasa.gov/cgi-bin/spread.cgi>.

© 2013 The Authors. Published by Elsevier B.V.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science

Keywords: Semantic Analysis; Validation; Parsers; Spreadsheets; Engineering Analysis; Mathematical Methods

1. Introduction

Spreadsheets are notorious for errors! In particular, the meaning or semantics of the contained values and formulae are prone to mistakes that are hard to find. Although it is difficult to display a spreadsheet's contents, the greater difficulty is representing and checking the semantics of variables and formulae.

What are these semantics? Engineering and science *is* predictions about our world: the trajectory of a rocket, the strength of a building, the performance of an electronic component, the cause and rate of global warming, the performance of a financial instrument. For centuries, the basis for this quantitative reasoning—the ontology—has been mathematical, physical, and engineering quantities and formulae; Newton's second law

* Corresponding author. Tel.: 1-440-801-1739.

E-mail address: Mark.E.Stewart@nasa.gov.

(1687) is a classic example that relates mass, acceleration, and force in specific physical circumstances. Physical quantities have dimensions and units[†] that must be consistent, and, hence, can be easily checked. Further, to make valid predictions, the equations, physical quantities, and their units and dimensions must be used correctly.

Yet, representing and analyzing the semantics of variables and formulae is a complex problem. Beyond some well-known ones, equations are numerous, if not unlimited. Manually checking a spreadsheet or program is difficult, and people think of this task as a job requiring ‘intelligence’. A human must have knowledge of the domain be it finance or mathematics, fluid mechanics, or quantum mechanics. Each field of expertise has concepts, formulae, and methods of reasoning.

The prototype addresses this problem by checking the spreadsheet’s contents against semantic analysis parsers containing this stored semantic information. Once the user identifies the semantics of primitive cells (quantity, units, vector component), then the formulae generated from these primitive cells are recognized by the semantics parsers.

The current work extends the state of the art. Type checking is a feature of modern compilers, and the early programming language FORTRAN [1] tracked integer and real types of expressions. Characterizing memory storage for variables and expressions has been extended, in subsequent programming languages, to coercion, overloading, and strong typing, and then further abstracted to classes and kinds [2]. Type systems have been further extended to spreadsheets. Erwig and Burnett [3] describe an error checking procedure for spreadsheets. They test for ‘units’ that are broader than measurement standards; their ‘units’ describe what objects are: apples or oranges. Propositional logic expressions are evaluated in a σ -calculus (first-order functional language); traditional mathematical, physical, and engineering equations are not apparent. Ahmad et. al. [4] describe a type checking process for spreadsheets with is-a and has-as relationships. Kennedy [5] discusses classical unit and dimensional analysis in terms of propositional logic and functional languages, and proves several theoretical results. The Buckingham π theory [6] exploited dimensional and unit consistency within an equation to predict equations, and it was a formalization of earlier work by Raleigh [7] and Bertrand [8]. Hence dimensional and unit consistency have been understood for more than a century.

Spreadsheets offer an opportunity for the automated analysis of mathematical, scientific, and engineering semantics. They are a valuable tool in wide use, yet simple enough to provide a testing ground for the representation and analysis of meaning. In contrast, errors in computer programs are also common and hard to find, but computer programs allow more complex features (arrays, pointers, subroutines), control structures (loops, conditional statements), and are often much bigger. Consequently, spreadsheets are an ideal application to test the automated analysis of the meaning of equations.

In the following section, the prototype Internet application’s features and operation are described. In Section 3, the representation and recognition of knowledge by semantic parsers is explained.

2. Features and Operation of the Self-Checking Spreadsheet

This prototype self-checking spreadsheet combines four key components: a JavaScript spreadsheet, AJAX messaging between the client web application and the server, a server based semantic parsing program, and descriptive pop-ups in the client.

[†] Here, dimensions are combinations of length, time, mass, temperature, amount of a substance, electric charge, luminous intensity, plus angle, and money. Further, units are specific measurement standards of these dimensions; the corresponding SI units are meter, second, kilogram, degree Kelvin, mole, Coulomb, candela, and degree angle; money has no SI unit, but US\$ or € are examples.

First, an existing, open source, JavaScript spreadsheet (<http://www.simple-groupware.de>) was supplemented with self-checking features. Figure 1 shows this web application written in JavaScript, HTML, Tcl/Tk and run on current web browsers. The basic functionality of a spreadsheet is provided, but the sophisticated features of a commercial product are not present.

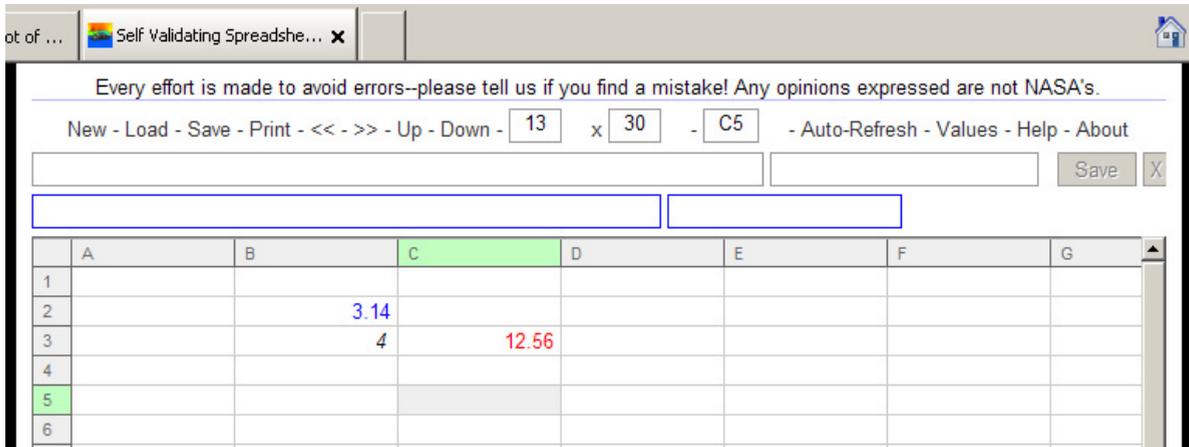


Figure 1: Simple spreadsheet, written in JavaScript, provides a spreadsheet as a web application. Cell B3 was identified as a DIAMETER, and units “km” were selected; Cell B2 was not identified since its’ unique value, 3.14, identifies it when used in the proper context.

Second, using AJAX, brief messages are relayed to a central server running the equation recognition program. Messages, or requests, concern identifying cell values (i.e. stress, interest rate, density) and also cell equations. In Figure 2, a cell is given the value 5.2, but the identity of this quantity is also needed; the user input of “p” results in AJAX requests to the server that return potential identities. “p” may refer to the symbol for momentum, or the first letter of pressure. Each additional letter invokes an AJAX message to the server that returns different possible identities for the cell.

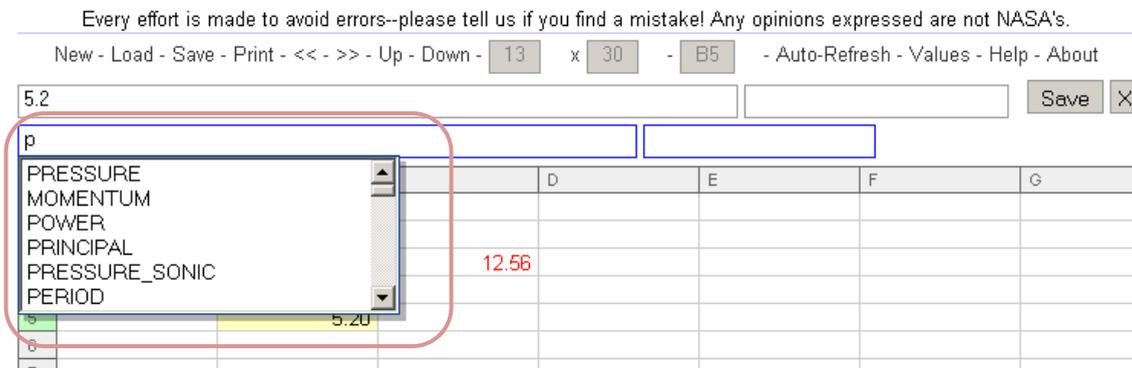


Figure 2: AJAX messages allow the central server to interact with user inputs. Starting with “p” as an input, identity suggestions are returned by the server based on matching against symbols and names.

Further, AJAX requests also communicate the equations contained in cells to the central server that runs the third part of this process—equation recognition parsers. This optimized program for recognizing equations is a descendant of the author’s semantic analysis program [9]. It recognizes equations and their constituent physical and mathematical quantities. Only known, stored equations and quantities can be recognized. This

equation recognition program is explained in greater detail in Section 3. Equation recognition results are returned to the spreadsheet by AJAX messages.

Fourth, Figure 3 shows how descriptive information is available in pop-ups in the spreadsheet. The server also generates supplementary information (definitions, symbols, units, warnings) that is passed back to the client and used in pop-ups. This information is available when the user moves the cursor over a cell.

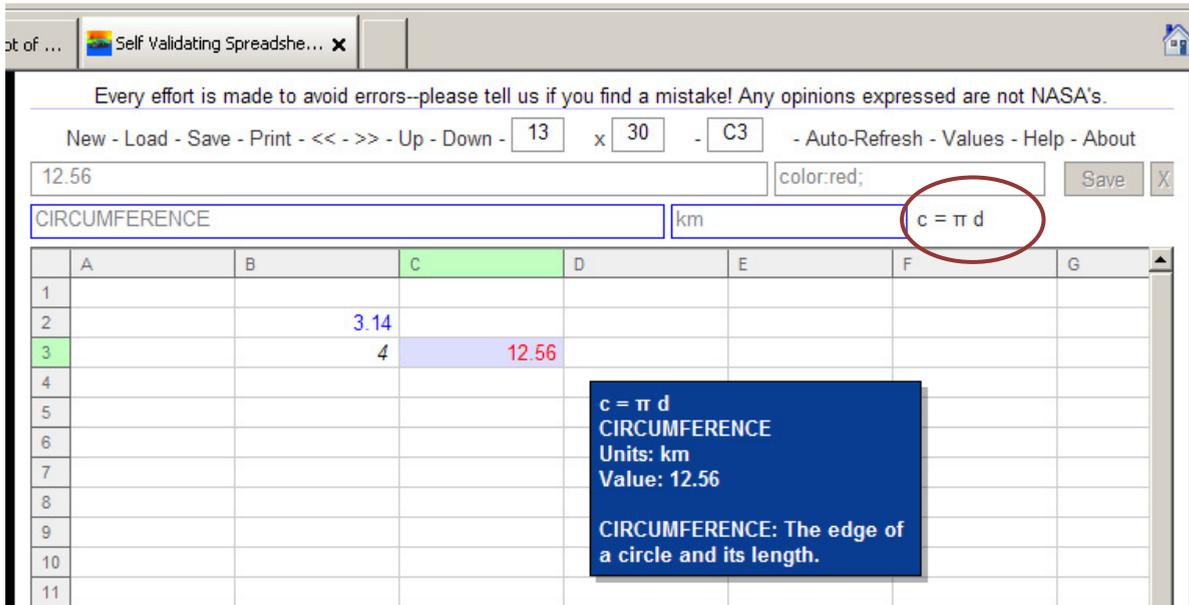


Figure 3: Relevant information is returned by the equation recognition program to the client spreadsheet for display. When the cursor is over a cell, a pop-up (blue rectangle) displays symbols, identity, units, definition, and any warnings. The deriving equation (red circle) is also shown.

When a cell is updated by the user, all dependent cells are identified by the client, and an AJAX message is sent, client to server, with all the equations from dependent cells. The server's equation recognition software determines updated identities and returns—in an AJAX message—conclusions to the client application. These conclusions are displayed.

Figure 4 provides an extended example of calculating the calibration of a sonic nozzle.

3. The Semantic Analysis Parsers

In this section, the semantic analysis of spreadsheet equations is detailed. Key concepts include tokenizing physical and mathematical concepts and translating semantic rules into parsers that recognize token sequences, namely, a spreadsheet's equations.

3.1. Tokenizing Concepts

Importantly, the validation of equations involves representing a concept with a symbol or token. Tokens are indivisible, atomic units that are uniformly understood, and can be used in expressions. The physical quantity velocity is an example. Fundamentally, language tokenizes concepts. We use the word "velocity" to represent the physical concept of the distance travelled in a unit of time in a direction; everyone implicitly agrees on the

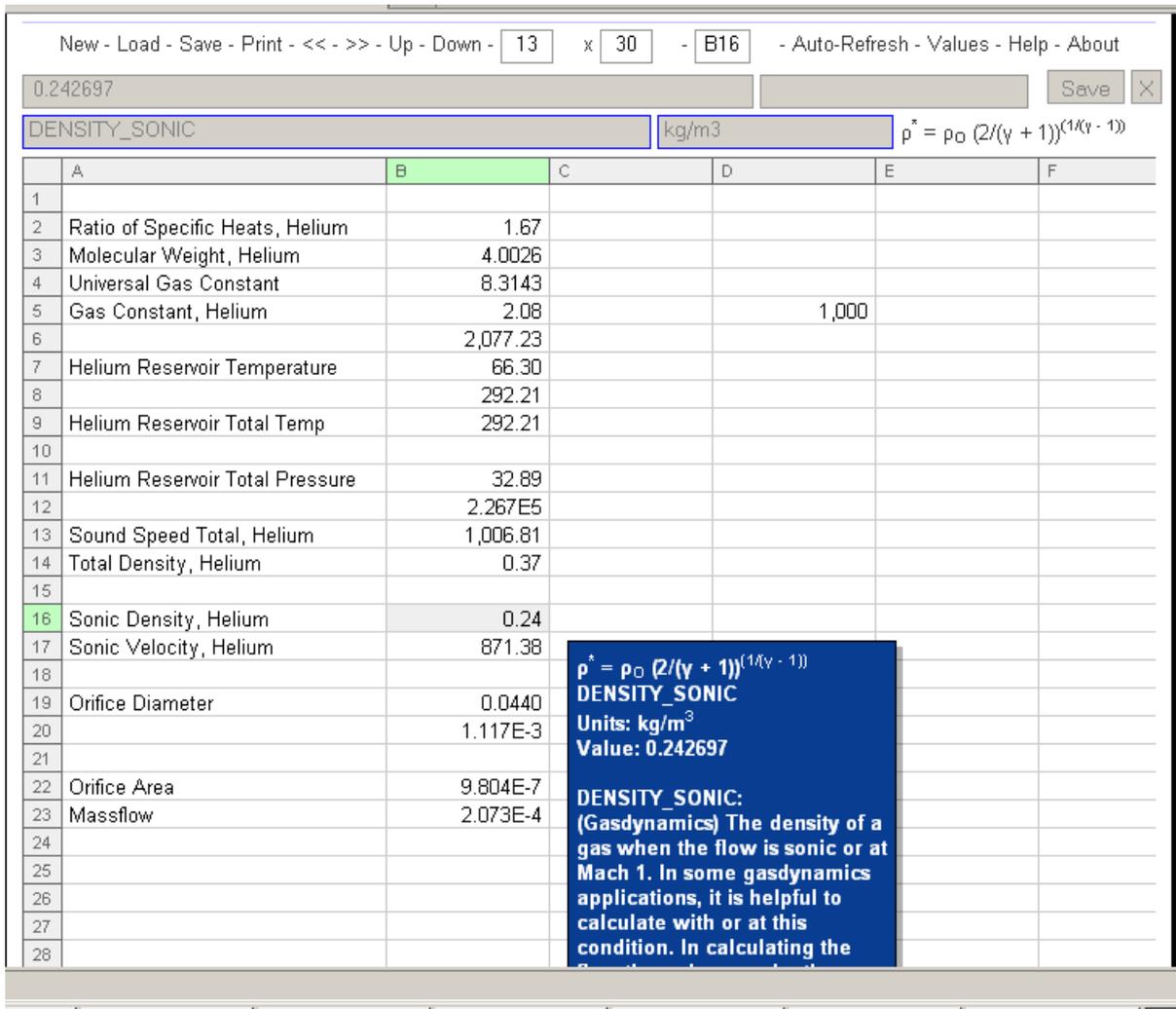


Figure 4: Example of a spreadsheet that checks a calibrating nozzle calculation. The pop-up refers to the highlighted cell, B16.

meaning (ambiguity can exist, malapropism and colloquialism occur), and we are free to use the token in expressions. When parsing is taught as part of compiler design, tokenization is implicitly understood.

The current semantic analysis program contains 970 tokens. Hundreds of rules (Table 1) represent physical, mathematical, and engineering equations.

3.2. Recognizing Sequences of Tokens

Substituting tokens for concepts is not enough to recognize semantics. Sequences of these tokens must be recognized. Fortunately, compiler development in Computer Science forced intense research (Section 3.6), and robust tools—plus understanding—emerged for token sequence recognition. These tools, parsers, are applicable here.

Table 1: Sample rules within the semantic analysis parsers that allow simplification of expressions of concept tokens.

1	Force	←	Mass * Acceleration
2	Momentum	←	Mass * Velocity
3	Speed_Squared	←	Speed * Speed
4	Energy_Kinetic	←	Half * Speed_Squared
5	Work	←	Pressure_Static / Density_Static
6	Enthalpy	←	Energy_Internal + Work
7	Sound_Speed_Squared	←	Gamma * Work
8	Enthalpy_Total	←	Enthalpy + Energy_Kinetic
9	Volume	←	Length * Area
10	Area	←	Length * Height
11		←	Length * Width
12	Voltage	←	Current * Resistance
13	Circumference	←	Pi * Diameter
14	Diameter	←	2 * Radius

The formulae of science, engineering, and mathematics can be thought of as sequences of concepts that are re-used in myriad different combinations. Generally, each sequence of concepts reduces to a single concept[‡]. Table 1 shows some examples. The current semantic analysis program contains hundreds of rules. These rules comprise not only physical and mathematical equations, but also expressions for units, dimensions, vector quantities, and location. Parsers were developed for compilers to transform between the characters of the program text and high-level expressions. In a similar manner, they can be applied here to transform equations into simplified expressions.

A parser generator transforms rules (Table 1) into a set of states and a transition table between these states. These transition rules strictly navigate from the current state to the next state based on the next token in the sequence. Yacc [10] is a widely used parser generator; Bison [11] is the modern implementation, and Aho et. al. [2] explain the theory of parser generators. Although parsers are simple, they have remarkable computational economy. Examination of the algorithm [2] makes this clear.

The equation recognition program and its parsers can only recognize equations that it “knows”, that is, has a rule for. If the recognition program were asked to recognize an obscure equation, it would fail without the required stored rule. This feature is similar to human expert behavior.

3.3. How Does a Parser Recognize an Expression? An Extended Example

The most important step in the 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 (1), the parser sequentially examines the RHS of (1). When the parser reads *Energy_Internal* (EI), it anticipates Rule 6 from Table 1 and expects the tokens ‘+’ and *Work*.

[‡] Exceptions exist, particularly in complex derivations. Analysis is the current focus.

$$\text{VAR} = \text{EI} + \text{P} / \text{RHO} + 0.5 * \text{V} * \text{V} \quad \text{where} \quad \begin{array}{ll} \text{EI} & == \text{Energy_Internal} \\ \text{P} & == \text{Pressure_Static} \\ \text{RHO} & == \text{Density_Static} \\ \text{V} & == \text{Speed} \end{array} \quad (1)$$

When the parser reads *Pressure_Static* (P), it expects Rule 5 to produce *Work* and the tokens ‘/’ and *Density_Static* to appear next. When the parser sees all the tokens of Rule 5, it reduces them to the token *Work*, and when all the tokens of Rule 6 are present, they are reduced to the token *Enthalpy*. The next token is ‘+’, and the parser anticipates Rule 8. Similarly, the parser recognizes Rules 4, 3, and 8 to infer that VAR represents *Enthalpy_Total*.

The parser rules (Table 1) 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 are automatically converted to a subroutine by the parser generator Yacc [10]. The parser generator considers all the input rules, tabulates what tokens and rules can appear next in any legal, conceivable input token sequence. An analogy is calculating all the possible moves in a chess game, except that grammar rules simplify the myriad input possibilities into a small number of exemplar rules.

Note also that each step in the recognition process depends on correctly performing the previous step; an error can dramatically reduce recognition.

3.4. Addition of Equation Rules to Parsers

Rules can be added to the parsers. Currently, the process is manual: a specification of rules must be edited to include the additional equations; updated code is automatically generated. Table 1 is a simplified version of this specification of rules, and the specification is, in fact, the input to the parser generator Yacc which generates parser code.

Typically, equations are decomposed into binary forms. For example, ρUA from fluid mechanics would be broken into mass flux, ρU , and mass flux times area. Based on experience from hundreds of equations and tokens, equations are unambiguous and not prone to generating severe conflicts within the grammar[§]. This observation is consistent with the centuries-long, rigorous development of scientific, engineering, and mathematical concepts and formula: ambiguous quantities and equations are refined, clarified or superseded with better models.

Perhaps the greatest limitation of this approach is the vast number of physical, mathematical, and engineering formulae and concepts that exist. The formulae learned through the end of an undergraduate education are numerous and widely applicable, but, beyond that point, they are even more extensive. The parsing paradigm should remain valid for large numbers of equations, but incorporating this volume of information is a daunting task.

3.5. Transformation of Token Sequences

In general, equations can be transformed by commutative, associative, and distributive laws. The rules (Table 1) do not account for these variations, but the semantic analysis program does. Commutative, associative, distributive, and anti-distributive variations of an equation are generated and analyzed by the parsers.

[§] Mathematical rules can be difficult due to their generality. For example, a derivative is variable dependent.

For example, in Figure 3, if the formula was “Radius * π * 2”, then it would not be recognized until a commutative transformation was applied, say “ π * 2 * Radius”. In particular, the diameter expression can be recognized first by parser Rule 14, and “ π * Diameter” can be recognized by Rule 13.

The spreadsheet’s equations are stored as a parse tree with annotations for declared and deduced properties. The transformations are accommodated by modifying this parse tree. The binary tree is adjusted to commute an expression, associate a recognized sub-expression (i.e. reform “2 * Radius” as a proper sub-tree), and distribute an expression.

3.6. History of Parsing

Although parsing a sequence of tokens is used here to recognize physical, engineering, and mathematical semantics, parsing is a common problem in computer science. Parsers were introduced to computer science during the early, formative years of programming languages. Their development was driven by the efficiency of writing computer programs in a high-level language instead of simple machine instructions.

Parsers transform between the characters of the text of a program and high-level expressions, including formulae. It discovers enough of the program’s meaning to do machine operations correctly. The first high level compiler was FORTRAN [1]; team members subsequently contributed to Backus-Naur Form (BNF) and the influential ALGOL 60 report [12]. Many approaches to parsing were explored, but one approach—involving considerable research and engineering—developed into the modern parsing standard, Yacc [10]. Knuth’s [13] introduction of left to right (LR) parsing algorithms and DeRemer’s [14] LALR methods are among the notable milestones.

BNF is a precise method of describing the syntax of a programming language as a set of string rewriting rules, but it was quickly realized that linguists had been using an identical notation to describe languages. In about the 4th century B.C., Panini [15] used a similar formalism to describe Sanskrit with precision and perfectionism. Panini’s grammar included phonemes, derived and compounded words, grammatical rules, accents (sound variations), and meaning. Early 20th century linguists extended this approach [16]. In 1956, Chomsky [17] pursued formal rules in language and introduced the Chomsky Hierarchy. Importantly, BNF is identical to a context-free grammar and is among the simpler grammars of the hierarchy. Human languages may not be exactly context-free, but they are almost so [18].

Summary and Conclusions

This paper describes a prototype, self-checking spreadsheet. This tool goes beyond unit and type checking to validate scientific, engineering, and mathematical semantics. Spreadsheet features and its internet-based client/server implementation are presented. The representation and recognition of spreadsheet semantics are explained in detail. The intent of this work is to quickly and automatically find errors in spreadsheets and reduce the cost of these errors.

There are three principal conclusions from this effort. First, using parsers to represent and recognize semantics is a robust, efficient, and versatile way to encode a large number of equations. Further, this approach differs from propositional logic approach of type and unit checking procedures. Second, remarkably, this successful paradigm extends even further to include the grammar of spoken languages as a context-free grammar.

Lastly, the greatest limitation of this approach is the vast number of physical, mathematical, and engineering formulae and concepts that exist. The paradigm remains valid, but incorporating this volume of information is a daunting task.

Acknowledgements

Three anonymous reviewers have provided valuable comments that substantially improved the paper.

References

- [1] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes and R. Nutt, "The FORTRAN Automatic Coding System," in *IRE-AIEE-ACM '57 Western Joint Computer Conference: Techniques for Reliability*, Los Angeles, CA, 1957.
- [2] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Reading: Addison-Wesley, 1986.
- [3] M. Erwig and M. Burnett, "Adding Apples and Oranges," in *Practical Aspects of Declarative Languages*, 2002, pp. 173-191.
- [4] Y. Ahmad, T. Antoniu, S. Goldwater and S. Krishnamurthi, "A type system for statically detecting spreadsheet errors," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, 2003.
- [5] A. J. Kennedy, "Relational Parametricity and Units of Measure," in *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages*, Paris, 1997.
- [6] E. Buckingham, "On Physically Similar Systems; Illustrations of the Use of Dimensional Equations," *Physical Review*, vol. 4, no. 4, p. 345-376, 1914.
- [7] Rayleigh, "On the Question of the Stability of the Flow of Liquids," *Philosophical magazine*, vol. 34, pp. 59-70, 1892.
- [8] J. Bertrand, " Sur l'homogénéité dans les formules de physique," *Comptes Rendus*, vol. 86, no. 15, p. 916-920, 1878.
- [9] M. E. M. Stewart and S. Townsend, "An Experiment in Automated, Scientific-Code Semantic Analysis," in *AIAA-99-3276*, June 1999.
- [10] S. C. Johnson, "Yacc--Yet Another Compiler Compiler," *Comp. Sci. Tech. Rep. No. 32*, AT&T Bell Laboratories, Murray Hill, NJ, 1977.
- [11] J. R. Levine, *Flex & Bison*, Sebastopol, CA: O'Reilly Media, 2009.
- [12] P. Naur (Editor), "Revised report on the algorithmic language Algol 60," *Comm. ACM*, vol. 6, no. 1, pp. 1-17, 1963.
- [13] D. E. Knuth, "On the Translation of Languages from Left to Right," *Information and Control*, vol. 8, pp. 607-639, 1965.
- [14] F. DeRemer, "Simple LR(k) grammars," *Comm. ACM*, vol. 14, no. 7, pp. 453-460, 1971.
- [15] S. C. Vasu, *The ashtadhyayi of panini* (edited and translated into English), Motilal Banarsidass, Delhi, India, 1891.
- [16] L. Bloomfield, *Language*, New York: Henry Holt, 1933.
- [17] N. Chomsky, "Three Models for the Description of Language," *IRE Trans. Info. Theory*, vol. 2, no. 3, pp. 113-124, 1956.
- [18] G. K. Pullum and G. Gerald, "Natural Languages and Context-Free Languages," *Linguistics and Philosophy*, vol. 4, pp. 471-504, 1982.