A Grammar for Spreadsheet Formulas Evaluated on Two Large Datasets

Efthimia Aivaloglou, David Hoepelman, Felienne Hermans
Software Engineering Research Group
Delft University of Technology
Mekelweg 4, 2628 CD Delft, the Netherlands
e.aivaloglou@tudelft.nl, d.j.hoepelman@student.tudelft.nl, f.f.j.hermans@tudelft.nl

***Abstract*—Spreadsheets are ubiquitous in the industrial world and often perform a role similar to other computer programs in many different domains. However, there does not exist a reliable grammar that is concise enough to facilitate research on spreadsheet formula code bases. This paper presents a grammar for spreadsheet formulas that is compatible, is compact enough to feasibly implement with a parser generator, and produces parse trees suited for further manipulation and analysis. We evaluate the grammar against more than one million unique formulas extracted from the well known EUSES and Enron spreadsheet datasets, successfully parsing 99.99%. Additionally, we utilize the grammar to analyze these datasets and measure the frequency of usage of language features in spreadsheet formulas. Finally, we identify smelly constructs and edge cases in the syntax of formulas.**

I. Introduction

Spreadsheets are widely used in industry: Winston [1] estimates that 90% of all analysts in industry perform calculations in spreadsheets. Their use is diverse, ranging from inventory administration to educational applications and from scientific modeling to financial systems. It is estimated that 90% of desktops have Excel installed [2] and that the number of spreadsheet programmers is bigger than that of software programmers [3].

Because of their widespread use, they have been the topic of research since the nineties [4]. Recent spreadsheet research has often focused on analyzing spreadsheets. For example, there have been attempts to visualize spreadsheets [5], [6]. More recently, researchers have attempted to define *spreadsheet smells*: applications of Fowler's code smells to spreadsheets [7], [8], followed by approaches to refactor spreadsheets [9], [10].

These research works analyze the formulas within spreadsheets, and therefore often parse the formulas. This is done either by using simple grammars which have not been evaluated ([10]), or through implied, undefined grammars([5], [7]–[9]). The above analyses are our main motivation towards a defined grammar. Having such a grammar will enable parsing spreadsheet formulas into processable parse trees which can in turn be used to analyze cell references, extract metrics, find code smells and explore the structure of spreadsheets. Essentially, a reliable grammar can facilitate research on the spreadsheet formula code bases. Furthermore, research towards defining and verifying a formula grammar can facilitate exploring and possibly uncovering smelly syntactical constructs that are within the grammar coverage.

To make a grammar suitable for these goals, the requirements that we set for it are (1) to be compatible with the official Excel formula language, (2) to produce parse trees suited for further manipulation and analysis, and (3) to be compact enough to feasibly implement with a parser generator. The approach that we took towards developing the grammar was gradual enrichment through trial-and-error: we started from a simple grammar containing only the well known formula structures, implemented its parser, and provided as input to it formulas extracted from spreadsheet datasets. We used the two major datasets that are available in the spreadsheet research community: The EUSES dataset [11] and the Enron corpus [12], jointly containing over 20,000 spreadsheets. The final grammar resulted from many cycles of parse errors, enrichments and refinements, until all common and rare cases found in the datasets were supported.

The contributions of this paper are (1) a concise grammar for spreadsheet formulas, (2) the evaluation of the compatibility of the grammar using two major datasets, and (3) an analysis of the common formula characteristics and of the rare grammatical cases of the datasets.

The remainder of the paper is organized as follows: In the following section we summarize the basic concepts of spreadsheets and of the formula language. In Section III we present the spreadsheet formula grammar, its lexical and syntactical analysis rules, and details on precedence and ambiguity. Section IV explains how we implemented and evaluated the grammar, presents the obtained results, and analyses the datasets' formula characteristics. In Section V we discuss the grammar and its limitations. Section VI presents related work and Section VII concludes the paper.

II. Background

Spreadsheets are cell-oriented dataflow programs which are Turing complete [13].

A single spreadsheet *file* corresponds to a single *(work)book*. A workbook can contain any number of *(work)sheets*. A sheet consists of a two-dimensional grid of *cells*. The grid consists or verticals *rows* and horizontal *columns*. Rows are numbered sequentially top-to-bottom starting at 1, while columns are numbered left-to-right alphabetically, i.e. base-26 using A to Z as digits, starting at ‘A’, making column 27 ‘AA’.

A cell can be empty or contain a *constant value*, a *formula* or an *array formula*. Formulas consist of expressions which can contain constant values, arithmetic operators and *function calls* such as `SUM(...)` and, most importantly, *references* to other cells. Function arguments are separated by commas.

A. References

References are the core component of spreadsheets. The value of any cell can be used in a formula by concatenating its column and row number, producing a reference like `B5`. If the value of a cell changes this new value will be propagated to all formulas that use it.

When copying a cell to another cell by default references will be adjusted by the offset, for example copying `=A1` from cell B1 to C2 will cause the copied formula to become `=B2`. This can be prevented by making the reference absolute by prepending a `$` to the column index, row index or both. The formula `=$A$1` will remain the same on copy while `=$A1` will still have its row number adjusted.

An alternate style called R1C1 as opposed to the above A1 style exists, but it is very rarely seen or used by users. In R1C1 references one specifies either the offset to a cell between square brackets or its concrete location. In R1C1 style `R[4]C[-2]` means the cell two columns to the left and four rows down, while `R2C2` refers to cell B2. The biggest advantage of R1C1 is that it causes identical formulas to be the same even when they operate on different cells or data because of their position. These properties make R1C1 useful as an internal representation, but the grammar presented in this paper is intended for the common A1 reference style.

References can also be *ranges*, which are collections of cells. Ranges can be constructed by three operators: the range operator `:`, the union operator `,` (a comma) and the intersection operator `␣` (a single whitespace). The range operator creates a rectangular range with the two cells as top-left and bottom-right corners, so `=SUM(A1:B10)` will sum all cells in columns A and B with row number 1 through 10. The range operator is also used to construct ranges of whole rows or columns, for example `3:5` is the range of the complete rows three through five, and `A:D` is the range of columns A through D. The union operator, which is different from the mathematical union as duplicates are allowed, combines two references, so `A1,C5` will be a range of two cells, `A1` and `C5`. Lastly the intersection operator takes only the cells which are in both arguments, `=A:A 5:5` will thus be equivalent to `=A5`.

A user can also give a name to any collection of cells, thus creating a *named range* which can be referenced in formulas by name.

B. Sheet and external references and DDE

By default references are to cells or ranges in the same sheet as the formula, but this can be modified with a prefix. A prefix consists of some identifier, followed by an exclamation mark followed by the actual reference.

The most common use case is to reference another sheet in the same workbook, where the prefix is simply the sheetname: `=Sheetname!A1`. References to external spreadsheet files are also possible, which is done by prepending the file name in between square brackets: `=[Filename]Sheetname!A1` or `=[Filename]!NamedRange`. A peculiar type of prefix are those that indicate multiple sheets: `=Sheet1:Sheet10!A1` means A1 in Sheet1 through Sheet10. Sheet names can also be between single quotes: `='Sheetname with space'!A1`.

In Windows versions of Microsoft Excel formulas can also call external programs through Dynamic Data Exchange (DDE). DDE links are a special case of references, used for receiving data from other applications. They take the form of `=Program|Topic!Arguments`, e.g. `=Database|TableA!Column1`.

C. Array Formulas and Arrays

In spreadsheet programs it is possible to work with one- or two-dimensional matrices.

When constructed from constant values they are called *array constants*, e.g. `{1,2;3,4}`. They are surrounded by curly brackets, columns are separated by commas, and rows by semicolons. Several matrix operations are available, for example `=SUM({1,2,3}*10)` will evaluate to 60.

Array Formulas use the same syntax as normal formulas, except that the user must enter *Ctrl* + *Shift* + *Enter* to signal that it is an Array formula. Excel and LibreOffice surround the formula with curly braces. Google docs works differently and requires the user to surround an array formula with `ARRAYFORMULA(...)`.

Marking a formula as an array formula will enable one- or two-dimensional ranges to be treated as array. For example if `A1`,`A3`,`A3` contain the values `1`,`2`,`3` the array formula `{=SUM(A1:A3*10)}` will evaluate to `60`. Furthermore, an array formula allows the user to return multiple results, which will be presented in multiple cells. The array formula `{={1,2,3}*{4,5,6}}` will show `4`, `10` and `18` in three different cells.

III. Spreadsheet Formula Grammar

For previous and ongoing research the authors needed a grammar for Microsoft Excel spreadsheet formulas with the following requirements:

1) Be compatible with the official language
2) Produce parse trees suited for further manipulation and analysis with minimal post-processing required
3) Be compact enough to feasibly implement with a parser generator

While an official grammar for Excel formulas is published [14], it does not meet the above requirements for two reasons. Firstly, it is over 30 pages long and contains hundreds of production rules and thus fails requirement 3.

Secondly, because of the detail of the grammar and the large number of production rules the resulting parse trees are very complex and fail requirement 2.

For this reason the authors decided to construct a new grammar with the above requirements as *design goals.*

A. Grammar class

While the class of this grammar is not strictly LALR(1) due to the present ambiguity, we implemented this grammar using a parser generator based on LALR(1) grammars. This is because the present ambiguity can be solved by defining operator precedence (section III-D) and manually resolving a conflict (section III-F). These two features are supported by most LALR(1) parser generators.

B. Lexical analysis

Table I contains the lexical tokens of the grammar, along with their identification patterns in the regular expression language. All tokens are case-insensitive.

Lexical analysis requires the scanner to support token priorities. Removing the necessity for token priorities is possible by altering the tokens and production rules, but makes the grammar more complicated and the resulting tree harder to use, thus being detrimental to design goals 2 and 3.

Some simple tokens are directly defined in the production rules in Figure 1 in between quotes for readability and compactness.

1) ***Dates:*** The appearance of date and time values in spreadsheets depends on the presentation settings of cells. Internally, date and time values are stored as positive floating point numbers with the integer portion representing the number of days since a Jan 0 1900 epoch and the fractional portion representing the portion of the day passed.

For this reason, the grammar only parses numeric dates and times and these are not distinguishable from other numbers.

2) ***External References:*** The file names in external references in formulas, both to external files and DDE, are not stored as part of the formula in the Microsoft Excel storage format, but instead are replaced by a numeric index. This index is then stored in a file level dictionary of external references. A formula that is presented to the user as `=[C:\Path\Filename.xlxs]Sheetname!A1` is internally stored as `[X]Sheetname!A1`, where `X` can be any number.

For this reason the presented grammar supports only numeric external references. Adding support for full filenames can be achieved by introducing an additional token or altering the `FILE` token, but that external filenames can be presented to and entered by the user in a myriad of different formats, depending on conditions such as if the file is opened in the spreadsheet program.

Fig. 1: Production rules

```
⟨Start⟩ ::= ⟨Constant⟩
  |  '=' ⟨Formula⟩
  |  ⟨ArrayFormula⟩
⟨ArrayFormula⟩ ::= '{=' ⟨Formula⟩ '}'
⟨Formula⟩ ::= ⟨Constant⟩
  |  ⟨Reference⟩
  |  ⟨FunctionCall⟩
  |  '(' ⟨Formula⟩ ')'
  |  ⟨ConstantArray⟩
  |  RESERVED-NAME
⟨Constant⟩ ::= NUMBER | STRING | BOOL | ERROR
⟨FunctionCall⟩ ::= ⟨Function⟩ ⟨Arguments⟩ ')'
  |  ⟨UnOpPrefix⟩ ⟨Formula⟩
  |  ⟨Formula⟩ '%'
  |  ⟨Formula⟩ ⟨BinOp⟩ ⟨Formula⟩
⟨UnOpPrefix⟩ = '+' | '-'
⟨BinOp⟩ = '+' | '-' | '*' | '/' | '^'
  |  '<' | '>' | '=' | '<=' | '>=' | '<>'
⟨Function⟩ ::= FUNCTION | UDF
⟨Arguments⟩ ::= ⟨Argument⟩
  |  ⟨Argument⟩ ',' ⟨Arguments⟩
⟨Argument⟩ ::= ⟨Formula⟩ | ϵ
⟨Reference⟩ ::= ⟨ReferenceItem⟩
  |  ⟨Reference⟩ ':' ⟨Reference⟩
  |  ⟨Reference⟩ '␣' ⟨Reference⟩
  |  '(' ⟨Union⟩ ')'
  |  '(' ⟨Reference⟩ ')'
  |  ⟨Prefix⟩ ⟨ReferenceItem⟩
  |  ⟨Prefix⟩ UDF ⟨Arguments⟩ ')'
  |  ⟨DynamicDataExchange⟩
⟨ReferenceItem⟩ ::= CELL
  |  ⟨NamedRange⟩
  |  REFERENCE-FUNCTION Arguments ')'
  |  VERTICAL-RANGE
  |  HORIZONTAL-RANGE
  |  ERROR-REF
⟨Prefix⟩ ::= SHEET
  |  FILE SHEET
  |  FILE '!'
  |  QUOTED-FILE-SHEET
  |  MULTIPLE-SHEETS
  |  FILE MULTIPLE-SHEETS
⟨NamedRange⟩ ::= NAMED-RANGE
  |  NAMED-RANGE-PREFIXED
⟨Union⟩ ::= ⟨Reference⟩ | ⟨Reference⟩ ',' ⟨Union⟩
⟨DynamicDataExchange⟩ ::= FILE '!' DDECALL
⟨ConstantArray⟩ ::= '{' ⟨ArrayColumns⟩ '}'
⟨ArrayColumns⟩ ::= ⟨ArrayRows⟩
  |  ⟨ArrayRows⟩ ';' ⟨ArrayColumns⟩
⟨ArrayRows⟩ ::= ⟨ArrayConstant⟩
  |  ⟨ArrayConstant⟩ ',' ⟨ArrayRows⟩
⟨ArrayConstant⟩ ::= ⟨Constant⟩
  |  ⟨UnOpPrefix⟩ NUMBER
  |  ERROR-REF
```

TABLE I: Lexical tokens used in our grammar

Token Name	Description	Regular Expression	Priority
BOOL	Boolean literal	TRUE | FALSE	0
CELL	Cell reference	$? [A-Z]+ $? [0-9]+	2
DDECALL	Dynamic Data Exchange	' ([A-Z0-9_ !@#$%^&*()+={}:;|<>,./?\\] | ")+ '	0
ERROR	Error literal	#NULL! | #DIV/0! | #VALUE! | #NAME? | #NUM! | #N/A	0
ERROR-REF	Reference error literal	#REF!	0
FILE	External file reference	\[[0-9]+ \]	5
FUNCTION	Excel built-in function	(Any entry from the function list[1]) \(	5
HORIZONTAL-RANGE	Range of rows	$? [0-9]+ : $? [0-9]+	0
MULTIPLE-SHEETS	Multiple sheet references	[A-Z0-9]+ : ([A-Z0-9_.]+|'([A-Z0-9_ !^&*()+={}:;|<>,./?\\] | ")+')!	1
NAMED-RANGE	Named range	[A-Z_][A-Z0-9_.]*	-2
NAMED-RANGE-PREFIXED	Named range which starts with a string that could be another token	(TRUE | FALSE | [A-Z]+[0-9]+) [A-Z0-9_.]+	3
NUMBER-LITERAL	An integer, floating point or scientific notation number literal	[0-9]+ ,? [0-9]* (e [0-9]+)?	0
QUOTED-FILE-SHEET	A file reference within single quotes	'\[[0-9]+ \] ([0-9A-Z_ !@#$%^&*()+={}:;|<>,./?\\] | ")+ '!	5
REFERENCE-FUNCTION	Excel built-in reference function	(INDEX | OFFSET | INDIRECT)\(	5
RESERVED-NAME	An Excel reserved name	_xlnm\. [A-Z_]+	-1
SHEET	The name of a worksheet	([0-9A-Z_.]+ | ' ([0-9A-Z_ !@#$%^&*()+=|:;<>,./?\\] | ")+ ') !	5
STRING	String literal	" ([^ "] | "")* "	0
UDF	User Defined Function	(_xll\.)? [A-Z0-9]+ (	4
VERTICAL-RANGE	Range of columns	$? [A-Z]+ : $? [A-Z]+	0

Fig. 3: Syntax diagram of the ⟨*Reference*⟩ production rule with nonterminals expanded

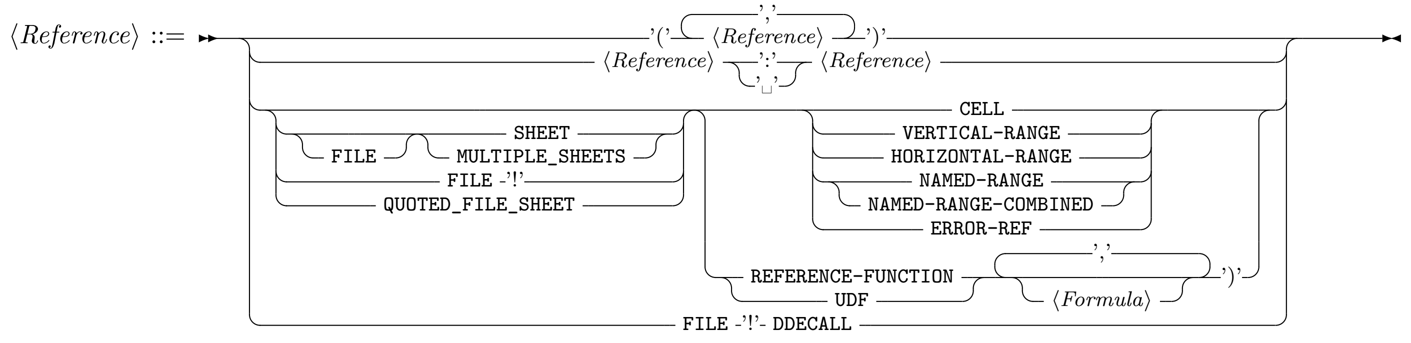


TABLE II: Operator precedence in formulas

Precedence Higher is greater	Operator(s)
1	= < > <= >= <>
2	&
3	+ - (binary)
4	*
5	^
6	%
7	+ - (unary)
8	: , ␣

C. Syntactic analysis

The complete production rules of our grammar in Extended BNF syntax can be found in Figure 1. The start symbol is *Start*.

⟨*Formula*⟩ and ⟨*Reference*⟩ are the two most important production rules in this grammar. These are also illustrated as syntax diagrams, expanded to include lexical tokens, in Figures 2 and 3.

D. Operator Precedence

All operators in Excel are left-associative, including the exponentiation operator which in most other languages is right-associative. In order to resolve ambiguities a LALR parser generator needs the operator precedence to be defined, which can be found in table II.

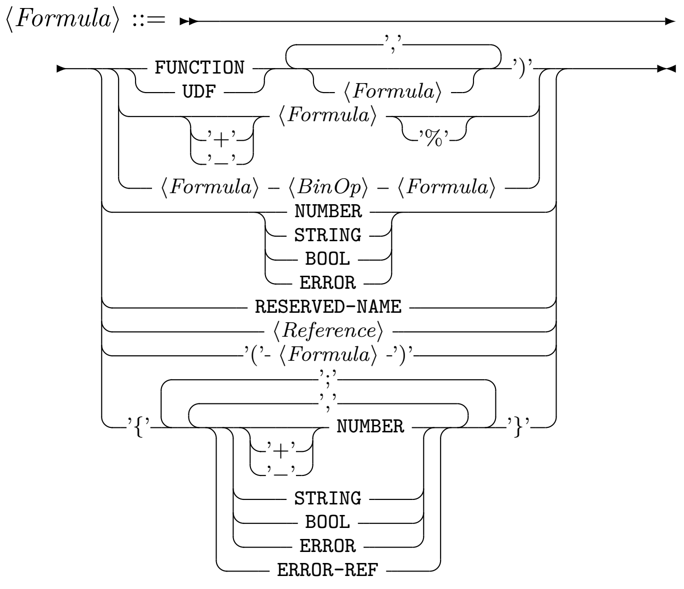
E. Intersection operator

The intersection binary operator in Excel formulas is a single space, while the rest of the language is whitespace independent outside of strings. While this is straightforward to define in EBNF, it can be challenging to implement using a parser generator.

Our parser generator supports a feature called implicit operators which was used to implement this operator. Implicit operators are operators which are left out and only implied, for example in calculus the multiplication operator is often omitted: $5a$ is equivalent to $5 \cdot a$.

[1]A function list is available as part of the reference implementation. Lists provide by Microsoft are also available in [15] and [14].

Fig. 2: Syntax diagram of the ⟨*Formula*⟩ production rule with most production rules expanded



F. Ambiguity

Due to trade-offs on parsing references (see section III-G1) and on parsing unions (see section III-G2) our grammar is not fully unambiguous. Ambiguity exists between the following production rules:

1) ⟨*Reference*⟩ ::= '(' ⟨*Reference*⟩ ')'
2) ⟨*Reference*⟩ ::= '(' ⟨*Union*⟩ ')'
3) ⟨*Formula*⟩ ::= '(' ⟨*Formula*⟩ ')'

A formula like `=(A1)` can be interpreted as either a bracketed reference, a union of one reference, or a reference within a bracketed formula.

In an LALR(1) parser the ambiguity manifests in a state where, on a `')'` token, shifting on rule 1 and reducing on either rule 2 or 3 are possibilities, causing a shift-reduce conflict. This was solved by instructing the parser generator to shift on rule 1 in case of this conflict, because this always results in a correct interpretation and thus in a correct parse tree.

G. Trade-offs

1) ***References:*** References are of great importance in spreadsheet formulas, and thus of interest for analysis. To support easier analysis (design goal 2) references have different production rules than other expressions. This causes references to be easily identified and isolated, but has the downside of increasing ambiguity, as explained in Section III-F.

Another approach would be to parse all formulas similarly and implement a type system, however this would be very detrimental to both ease of analysis (design goal 2) and to ease of implementation (design goal 3).

2) ***Unions:*** The comma serves both as an union operator and a function argument separator. This proves challenging to correctly implement in a LALR(1) grammar.

A straightforward implementation would use production rules similar to this:

⟨*Union*⟩ ::= ⟨*Reference*⟩ ',' ⟨*Reference*⟩
⟨*Arguments*⟩ ::= ⟨*Argument*⟩
| ⟨*Argument*⟩ ',' ⟨*Arguments*⟩

However, this will cause a reduce-reduce conflict because the parser will have a state wherein it can reduce to both a ⟨*Union*⟩ or ⟨*Argument*⟩ on a `,` token. Unfortunately there is no correct choice: in a formula like `=SUM(A1,1)` the parser must reduce on the ⟨*Argument*⟩ nonterminal, while in a formula like `=A1,A1` the parser must reduce to the ⟨*Union*⟩ nonterminal. With the above production rules a LALR(1) parser could not correctly parse the language.

The presented grammar only parsers unions in between parentheses, e.g. `=SMALL((A1,A2),1)`. This is a trade-off between a lower compatibility (design goal 1) and an easier implementation (design goal 3). We deem this decreased compatibility to be acceptable since unions are very rare (see section IV) and all but two were within parentheses (see section V).

Additionally formulas that this grammar does not parse often result in runtime errors after evaluation. For example `=A1,A1` does parse in a spreadsheet program, but produces the error `#VALUE!` on evaluation.

Implementing the straightforward rules above, while desirable, is not possible without using a more powerful grammar class.

IV. Evaluation

In this Section we explain how we implemented and evaluated the grammar using the datasets and we discuss the obtained results and the formula parse failures. In the grammar analysis in Section IV-B we examine how frequently the grammatical features occur in the formulas of the datasets.

The grammar is implemented in the Irony parser generator framework[2] and the resulting parser is available for download[3].

To extract unique formulas and use them as input to the parser we built a tool that opens spreadsheets using the Gembox third party library. The tool reads all cells and identifies which have the same formula in R1C1. It then selects the first cell from each group of cells that share the same formula in the R1C1 style and uses its formula string as input for the parser. It parses only one cell from each R1C1 group—the only differences between the formulas in the same group are the values of the references, so the structure of the produced parse trees is exactly the same.

To evaluate the grammar we apply it for parsing a total of 1,039,751 unique formulas. These originate from

[2]https://irony.codeplex.com/
[3]https://github.com/PerfectXL/XLParser

the two major datasets available in the spreadsheet research community: The EUSES dataset [11], comprising of 4,498 spreadsheets and the Enron email corpus [12], which became available after the Enron company declared bankruptcy, comprising of 16,190 spreadsheets. We were not able to process 1087 (5.25%) of these spreadsheets, either because they are password protected or because of read failures due to the Gembox library. In total, the 19,601 spreadsheets that were processed from the two datasets include 22,632,306 formula cells. These are grouped into 1,039,751 R1C1 groups, which is the number of unique formulas that were used as input to the parser.

To give a rough indication, processing these two datasets and extracting these results takes around 4 hours on a computer with an Intel Core i7 processor, 16GB of RAM and a Solid State Drive.

Out of the 1,039,751 unique formulas from the two datasets that were used as input to the parser, 1,039,709 (99.99%) were parsed successfully. This satisfies our first design goal of compatibility with the official language. Regarding the second and third design goals, the implementation of the parser proved feasible and compact and the resulting parse trees suited for manipulation, having 20 types of non-terminal and 19 types of leaf nodes.

A. Unparsable formulas

The 42 formulas that were not parsed using the grammar defined in Section III are:

- `=-NOX, Regi` and `=-_SO2, Regi` in two different sheets in the Enron dataset. These are cases of an union operations without parentheses that the grammar does not parse as explained in Section III-G2.
- `=+Ë%₀` was included in an Enron file that we assume to be either corrupt or another type of binary file, as the file is indecipherable.
- 39 formulas that are not returned correctly from the Gembox third party library that we use for opening spreadsheets. For example our tool reads and attempts to parse formula `IF(=7,AVERAGE(C4:C11),0)` and fails, but in reality the formula is `IF(B8=7,AVERAGE(C4:C11),0)` which does parse. All these 39 cases are parsed successfully when we manually provide them as input to the parser.

B. Grammar Analysis

The grammar resulted from many cycles of parse errors, enrichments and refinements. There are parts of the grammar that cover a particularly complex set of structures. In this section we analyze the formulas in the datasets and measure the frequency of their characteristics and grammatical structures. We also identify potentially smelly grammatical constructs and edge cases in the syntax of formulas.

1) Formulas and Functions: The ⟨*Formula*⟩ rule covers all types of spreadsheet formula expressions. As illustrated in the syntax diagram in Figure 2, spreadsheet formulas can be constants (`=5`), references (`=A3`), function calls (`=SUM(A1:A3)`), array constants(`={1,2;3,4}`, explained in Section II-C), or reserved names (`=_xlnm.Print_Area`). Function calls invoke actual named (system or user defined) functions or operators applied to one or more formulas.

Table III shows how frequently each of the production rules in Section III-C occurs in the formulas of the two datasets. Jointly, 85.32% of the formulas include a function call. Actual named functions are invoked by 46.7% of the formulas. The vast majority are system defined functions, but there is a significant amount of formula cells (303,789—1.34%) invoking user-defined functions—e.g., `=[1]!erUserEmail(User_Id)`. A special case of user defined functions are the ones created using the Excel xll add-in library. These are found in the dataset invoked as `_xll.functionName`, by 0.61% of the formula cells.

Operators are used in 66.8% of the formula cells, with binary operators being the most common ones, appearing in 59.93% of the formulas. Analyzing the utilization of constants, we find that 39.38% of the formula cells contain at least one; more than one third (35.2%) of the formula cells contain a number and 11.97% are formulas that contain text. Reserved names are uncommon, with 1,281 occurrences of the `_xlnm.Print_Area` and 5 occurrences of `_xlnm.Database`.

Regarding function arguments, spreadsheet systems allow empty ones (e.g. `=SUM(,E35,E37)`) but this is rarely done—in only 0.05% of the formula cells. Unions are also rare. Only 385 formula cells contain a union used as argument, e.g. `=LARGE((F38,C38),1)`. All occurrences were arguments of the `LARGE` and `SMALL` functions—these two functions require an array of cells to be declared as a single argument, necessitating a union if the cells are not in a single range. In the EUSES dataset we also found 19 cases of constant arrays used as arguments, e.g. `=FVSCHEDULE(1,0.09;0.11;0.1)`.

The ⟨*ArrayFormula*⟩ rule, covering formulas surrounded by brackets, is the only part of the grammar that is not evaluated. The Gembox library that we use for reading spreadsheets does not support array formulas—it reads them as regular formulas, without the surrounding brackets. For this reason, we cannot we extract information on their frequency in the two datasets.

2) References: Spreadsheet formulas allow for different types of references, including single cell references, cell ranges, horizontal or vertical ranges, named ranges and reference-returning build-in or user-defined functions. All of these references can be internal (in the same or in different sheets) or external. Syntactically, they can be expressed in a number of ways. The simplest case of a reference to a cell range can be expressed in any of the

TABLE III: Frequency of spreadsheet formulas with specific grammatical structures in the EUSES and Enron datasets

Syntax	Example	Unique formulas		Total formulas	
⟨*Formula*⟩	`=1+2`	**1,039,709**		**22,630,110**	
⟨*Reference*⟩	`=E9/E10`	966,860	92.99%	22,451,956	**99.21%**
CELL	`=A5`	955,518	91.90%	22,342,591	98.73%
⟨*FunctionCall*⟩	`=SUM(A5:A22)`	707,783	68.08%	19,308,203	**85.32%**
⟨*BinOp*⟩	`=H10-H8`	399,046	38.38%	13,562,852	**59.93%**
⟨*Function*⟩	`=SUM(A5:A22)`	293,183	28.20%	10,569,248	**46.70%**
FUNCTION	`=SUM(A5:A22)`	288,929	27.79%	10,459,005	46.22%
⟨*Constant*⟩	`=SUM(A5:A22)`	273,310	26.29%	8,912,021	**39.38%**
NUMBER	`=(B8/48)*15`	251,150	24.16%	7,966,125	**35.20%**
⟨*Prefix*⟩	`=Sheet1!B1`	337,917	32.50%	5,651,635	**24.97%**
SHEET	`=Sheet1!B1`	304,170	29.26%	5,335,009	23.57%
⟨*Reference*⟩ ':' ⟨*Reference*⟩	`=SUM(A5:A22)`	184,877	17.78%	3,852,467	**17.02%**
⟨*UnOpPrefix*⟩	`=+B11+1`	218,527	21.02%	3,283,935	14.51%
STRING	`=COUNTIF(B$4:B$46,">=90")`	57,317	5.51%	2,708,039	**11.97%**
⟨*NamedRange*⟩	`=SUM(freq)`	21,240	2.04%	1,630,263	**7.20%**
BOOL	`=IF(AND(R11=1,R14=TRUE),G19,0)`	7,522	0.72%	1,264,751	5.59%
FILE	`=[11]Sheet1!C5`	104,941	10.09%	1,135,234	**5.02%**
REFERENCE-FUNCTION	`=SUM(J9:INDEX(J9:J41,B43))`	10,515	1.01%	780,050	**3.45%**
QUOTED-FILE-SHEET	`=('[2]Detail I&E'!D62)/1000`	33,782	3.25%	325,499	1.44%
UDF	`=SQRT(_eoq2(C5,C4,C6,C7))`	23,202	2.23%	**303,789**	**1.34%**
'_xll.'	`=_xll.RiskTriang(F9,F7,F8)`	12,426	1.20%	137,886	**0.61%**
ERROR_REF	`=AVERAGE(#REF!)`	3,482	0.33%	123,476	**0.55%**
(' ⟨*Reference*⟩ ')'	`=(2*(B29))/(1+B29)`	5,259	0.51%	85,724	0.38%
VERTICAL-RANGE	`=COUNT(A:A)`	860	0.08%	56,118	**0.25%**
FILE '!'	`=[1]!today`	2,040	0.20%	**28,448**	**0.13%**
ERROR	`=IF(AND(R11=1,R14=TRUE),G19,0)`	380	0.04%	27,245	**0.12%**
'%'	`=IF(E5>I8,3%,0%)`	858	0.08%	16,606	0.07%
Empty argument	`=DCOUNT(Lettergrades,,I80:I81)`	1,343	0.13%	10,512	**0.05%**
Complex range	`=SUM(I8:K8:M8)`	369	0.04%	**8,583**	0.04%
⟨*DynamicDataExchange*⟩	`=TWINDDE|RSFRec!'NGH2 NET.CHNG"`	3,276	0.32%	**3,686**	0.02%
Intersection	`=Ending_Inventory Jan`	298	0.03%	**2,829**	0.01%
MULTIPLE-SHEETS	`=SUM(Sheet1:Sheet20!I29)`	173	0.02%	**1,986**	**0.01%**
Prefixed right reference limit	`=SUM('Tot-1'!$B8:'Tot-1'!B8)`	147	0.01%	**1,501**	0.01%
RESERVED_NAME	`=C23/_xlnm.Print_Area`	76	0.01%	**1,286**	0.01%
UDF reference	`=[1]!wbname()`	332	0.03%	**855**	0.00%
HORIZONTAL-RANGE	`=MATCH(F3,Prices!2:2,0)`	11	0.00%	836	**0.00%**
⟨*Union*⟩	`=LARGE((F38,C38),1)`	10	0.00%	**385**	0.00%
⟨*ConstantArray*⟩	`=FVSCHEDULE(1,0.09;0.11;0.1)`	15	0.00%	**19**	0.00%

following ways:

$$\mathtt{SUM(A1:A2)} \tag{1}$$
$$= \mathtt{SUM(Sheet1!A1:A2)} \tag{2}$$
$$= \mathtt{SUM(Sheet1!A1:(A2))} \tag{3}$$
$$= \mathtt{SUM('Sheet1'!A1:A2)} \tag{4}$$
$$= \mathtt{SUM(Sheet1!A1:Sheet1!A2)} \tag{5}$$
$$= \mathtt{SUM(Sheet1!A1:'Sheet1'!A2)} \tag{6}$$
$$= \mathtt{SUM(namedRangeA1A2)} \tag{7}$$
$$= \mathtt{SUM(A1,A2)} \tag{8}$$
$$= \mathtt{SUM((A1,A2))} \tag{9}$$
$$= \mathtt{SUM(A1:A2:A1)} \tag{10}$$
$$= \mathtt{SUM(A1:A2\ A:A))} \tag{11}$$

The ⟨*Reference*⟩ rule, drawn in diagram 3, covers all types of referencing expressions. It was the rule that was the most complex to devise to cover all cases found in the two datasets.

As shown in Table III, 99.21% of the formula cells in the two datasets contain at least one ⟨*Reference*⟩, and 24.97% of these contain a reference that is not local, since it includes a ⟨*Prefix*⟩, like formulas (2)-(6). External file references exist in almost 5.02% of the formulas. 17.02% of the formulas include a reference to a `':'` separated cell range. Named ranges, like in the case of formula (7), exist in 7.2% of formula cells and, interestingly, horizontal and vertical ranges are rarely used (jointly, in 0.25% of the formulas). 0.55% of formulas include references to errors, e.g. `=#REF!E3`. These referencing errors are more than four times more common than all other types of errors combined—the `ERROR` token exists in 0.12% of the formula cells.

Moving to the edge cases of the grammar, the structures that were less common in the datasets include:

File-only external references
: External references are normally in the form `[File]Sheet!Cell`. In 28,488 formula cells (0.13%), however, the sheet is not specified, e.g. `=[2]!LastTrade`. These are either cases of references to external named ranges or to external UDFs.

Multiple sheet references
: 1,986 cells (0.01%) contain this complex

case of reference, which spans across multiple sheets. An example formula is `=SUM(Sheet1:Sheet10!A5)`, evaluated by summing all cells in position `A5` from `Sheet1` to `Sheet10`.

References to external UDFs
: 855 cells contain references to external user-defined functions, for example `=[1]!SheetName()`.

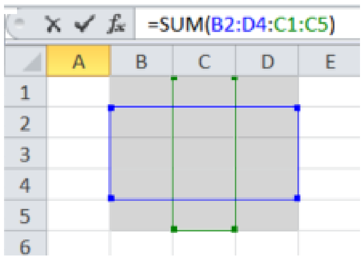
Prefixed right limits
: 1,501 cells include a reference with a prefix in the right limit, like in formulas (5) and (6). In all cases this prefix is identical to the first one, as continuous ranges spanning across multiple sheets are not supported by Excel. Still, this syntax is supported.

A special uncommon case in the grammar are the functions that return references, namely the `INDEX`, the `OFFSET` and the `INDIRECT` functions. For example, `INDEX` returns the reference of the cell at the intersection of a particular row and, optionally, column, so `INDEX(B1:B10,3)` returns a reference to cell `B3` and can be used in a formula as `=SUM(A1:INDEX(B1:B10,3))` being equivalent to `=SUM(A1:B3)`. These functions are relatively common: they are found in 3.45% of the formula cells, with the most common one being the `INDEX` (in 2.47% of formula cells) and the least common one being the `INDIRECT` (in 0.21%). In addition to these three functions, the official formula language specification includes the `IF` and `CHOOSE` functions as able to return references, but there was no formula in the datasets using them as such.

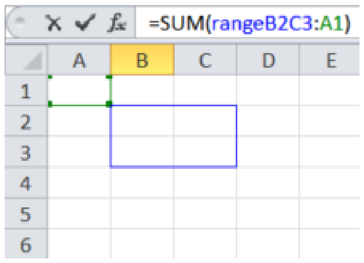
Finally, DDE links, discussed in Section II-B, were found in 3,686 formula cells. Example formulas in the dataset include `=TWINDDE|RSFRecord!'NGH2 NET.CHNG'` and `=GLDDEML|Action!'SGIMP,LA'/100`.

3) Smelly grammar constructs: There are two constructs in the spreadsheet formula grammar that we consider to be smelly, i.e. counterintuitive, inconsistent to the rest of the grammar and error-prone: Complex ranges and the implicit intersect operator.

By *complex ranges* we mean ⟨*Reference*⟩s that include more than two or different types of `':'` separated ⟨*ReferenceItem*⟩s. An example is range `B2:D4:C1:C5`, illustrated in Figure 4a. The smelly aspect of complex ranges is their evaluation. Simple cell ranges are in the form `top-left:bottom-right`, including all cells in between the two limits. However, the limits in complex ranges are not the ones specified in the formula: they are calculated as the upper leftmost and lower rightmost cell in the square that includes all defined cells. For example, range `B2:D4:C1:C5` is equivalent to `B1:D5`. Understanding the limits of the range becomes even less intuitive when vertical or horizontal ranges or named ranges are used, like in Figure 4b where the range is equivalent to `A1:C3`. Moreover, this syntax does not add to expressiveness of the grammar: each range is still calculated as the cells within a single square, but without clearly user-defined limits. The analysis showed that complex ranges are rare: 8,583 formula cells (0.04%) include complex ranges in the Enron dataset, and they are all defined using three cell locations.



(a) A range with four limits, equivalent to `B1:D5` and the area marked gray



(b) A range with a named range, equivalent to `A1:C3`

Fig. 4: Examples of references to complex ranges

The *intersect operator* is included in this discussion because it is not declared as an operator. Intersection between two ranges in the grammar is represented implicitly: by ␣, a single whitespace. For example, `A1:A10␣A1:A5` is equivalent to `A1:A5`. Using this syntax, intersection is an implicit range formation operation, belonging to the ⟨*Reference*⟩ rule, rather than a normal operation declared as a the ⟨*Function*⟩. An advantage of this approach is that it enables more natural language definition of intersections, e.g. `=SUM((Total_Cost Jan):(Total_Cost Apr.))`. An alternative, in our opinion less error-prone definition of intersection, would be a dedicated build-in function used explicitly as `INTERSECT(A1:A10,A1:A5)`. In the two datasets, intersection operations are not common, as they are found in only 2,829 cases of formula cells.

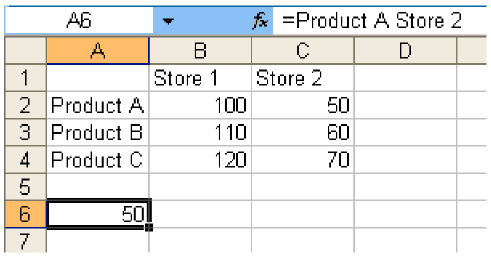
V. Discussion and Limitations

The currently defined Excel grammar is able to parse 99,99% of all formulas from two large and well-known spreadsheet datasets. In this section, we discuss a variety of issues that affect the applicability and suitability of our approach.

A. Version-dependent grammar features

The grammar had been designed as a generic spreadsheet formula grammar, enriched to include all syntactical features found in the two datasets. Both datasets, however, contain spreadsheets created by, or converted to, Excel format. This limits the grammar support for features that are spreadsheet system-dependent or even version-dependent. The built-in functions list, for example, might change across versions, which would make the parser mistakenly recognize build-in as user-defined functions.

Fig. 5: A natural language formula in Excel 2003


A6 =Product A Store 2

	A	B	C	D
1		Store 1	Store 2	
2	Product A	100	50	
3	Product B	110	60	
4	Product C	120	70	
5				
6	50			
7				

Syntactical features have also been deprecated. An example is regular expressions in formulas. Excel allows defining formulas that include regular expressions, for example `=SUM('S*'!A1)` or `=SUM('Sheet?'!A1)`. However, in Excel 2010 and up, regular expressions are instantly resolved—in the example, to the multiple sheet reference `=SUM(Sheet2:Sheet3!A1)`, summing up all `A1` cells between `Sheet2` and `Sheet3`, where the sheets are all matching sheets, except the one that this formula is on. This way, in latest versions of Excel, saved spreadsheets never contain regular expressions.

The use of labels in formulas (referred to as natural language formulas) is another feature that was discontinued in Excel 2007. Labels were the headings that were typed above columns and before rows, and they could be used in formulas instead of defined names or cell ranges. Figure 5 shows an example in Excel 2003, where formula `=Product A Store 2` returns the intersection between the cell range with heading `Product A` and the one with heading `Store 2`. This feature is replaced in newer versions of Excel with the less error-prone named ranges feature. When processing spreadsheets with newer versions of Excel, the references that include labels are automatically converted to cell-only references—in the example, the formula is converted to `=C2`. The grammar does not support the use of labels, and it would mistakenly parse them as named ranges.

B. Internationalization

Excel formulas differ depending on the language of the software. For example function arguments are separated by a semicolon instead of a comma in locales that use the comma as a decimal separator: the formula `=SUM(1.5,A1)` in the English version would be shown as `=SOM(1,5;A1)` in the Dutch version. Our grammar is only for the English locale. Grammars for other locales can be gotten by replacing delimiters, error values and function names by their localized versions.

It is worth noting that Excel will always save formulas in either a locale-independent binary format (Excel 2003 and earlier format) or in its English version (Excel 2007 and later format). When interacting with Excel through its API two versions of the formula can be read or written: the English version and the version in the current locale. This makes a grammar for the English version useful, since the parser can process all spreadsheets as long as their formulas are read using the English locale.

C. Rejection of invalid formulas

As stated in the design goals in Section III, the goal of this grammar is to facilitate analysis of formulas, which means correctly parsing valid spreadsheet formulas. Rejecting invalid formulas is not among the primary goals of this grammar, as the parser will normally not encounter invalid formulas in Excel files. Furthermore, while there exist two extensive datasets of valid formulas, to the best of our knowledge there are no such datasets of invalid formulas. As such we expect that the presented grammar is too broad and will parse a large number of formulas which are not valid. Using this grammar to parse possibly-invalid formulas like user-input might thus require additional safeguards.

The following is a non-exhaustive list of restrictions on the validity of formulas not encoded into the presented grammar[4]:

- Functions cannot have more than 255 arguments.
- Function calls cannot be nested more than 64 levels deep.
- The row number cannot exceed 1,048,576 (2^{20}) and the column number cannot exceed XFD (2^{14}).
- A formula cannot be longer than 8,192 (2^{13}) characters.

VI. Related work

Most related to our research is the work of Badame and Dig [10] who, as part of their proposed spreadsheet refactoring approach, presented a grammar for spreadsheet formulas. However, they do not evaluate their grammar, and upon inspection one can see that key ingredients are missing: e.g. external references, intersections and unions and named ranges. An extension of the same grammar was used to refactor formulas by Hermans and Dig [9].

Efforts to reverse-engineer a formal language specification based on existing artifacts have been used successfully for other languages, including COBOL [16] and C# [17].

As explained in the introduction, there is a large body of related work that relies on parsing spreadsheet formulas in order to analyze spreadsheets. This includes our own work in which we have created an algorithm to visualize spreadsheets as dataflow diagrams [5], and subsequently on detecting smells in spreadsheets [7], [8]. Related approaches exist, for example the work of Cunha that have worked on code smells [18] and smell-based fault localization [19]. These papers too analyze spreadsheet formulas but do not detail whether they use a grammar or what alternative they use.

VII. Conclusion

In this paper we present a grammar for spreadsheet formulas that we evaluated against over one million unique formulas and that successfully parses 99.99%, covering a

[4]These restrictions are in place in the Excel 2007 and later format. The Excel 2003 and earlier format often has lower limits.

particularly complex set of structures. The grammar was used to analyze the formulas in the datasets and to measure the frequency of their characteristics and grammatical structures. We found uncommon cases in the syntax of formulas, and we identified complex ranges and the implicit intersect operator as potentially smelly grammatical constructs.

The grammar is compact, consisting of 20 production rules and producing processable parse trees, suited for further manipulation and analysis. We believe that the grammar is reliable and concise enough to facilitate further research on spreadsheet formula code bases. It has already been applied in other works for analyzing formula characteristics, calculation chains and code smells and for applying formula transformations. The grammar parser is available as open-source software.

A weak point of the presented grammar is that the full extend of compatibility with the official Microsoft Excel grammar is unknown. An improvement could be made to this grammar by comparing it to the official specification, either by improving compatibility or by extending the number of known limitations. In general the problem of determining whether two context-free grammars are equivalent is undecidable, but in practice several techniques have been successfully used for this purpose [20], [21].

Another improvement could be made by testing the grammar against datasets containing spreadsheets made in other spreadsheet programs, and by testing the grammar against datasets of spreadsheets containing spreadsheets made in other companies.

References

[1] W. Winston, "Executive education opportunities," *OR/MS Today*, vol. 28, no. 4, pp. 8–10, 2001.

[2] L. Bradley and K. McDaid, "Using bayesian statistical methods to determine the level of error in large spreadsheets," in *Proc. of ICSE '09, Companion Volume*, 2009, pp. 351–354.

[3] C. Scaffidi, M. Shaw, and B. A. Myers, "Estimating the numbers of end users and end user programmers," in *Proc. of VL/HCC '05*, 2005, pp. 207–214.

[4] D. Bell and M. Parr, "Spreadsheets: A research agenda," *SIGPLAN Notices*, vol. 28, no. 9, pp. 26–28, 1993.

[5] F. Hermans, M. Pinzger, and A. van Deursen, "Supporting professional spreadsheet users by generating leveled dataflow diagrams," in *Proc. of ICSE '11*, 2011, pp. 451–460.

[6] K. Shiozawa, K. Okada, and Y. Matsushita, "3d interactive visualization for inter-cell dependencies of spreadsheets," in *Proceedings of The IEEE Information Visualization Conference (INFOVIS)*. IEEE, 1999, pp. 79–83.

[7] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and visualizing inter-worksheet smells in spreadsheets," in *Proc. of ICSE '12*, 2012, pp. 441–451.

[8] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting code smells in spreadsheet formulas," in *Proc. of ICSM '12*, 2012.

[9] F. Hermans and D. Dig, "Bumblebee: A refactoring environment for spreadsheet formulas," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE*, 2014, pp. 747–750.

[10] S. Badame and D. Dig, "Refactoring meets spreadsheet formulas," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on.* IEEE, 2012, pp. 399–409.

[11] M. Fisher and G. Rothermel, "The euses spreadsheet corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083242

[12] B. Klimt and Y. Yang, "The enron corpus: A new dataset for email classification research," in *Machine Learning: ECML 2004*, ser. Lecture Notes in Computer Science, J.-F. Boulicaut, F. Esposito, F. Giannotti, and D. Pedreschi, Eds. Springer Berlin Heidelberg, 2004, vol. 3201, pp. 217–226. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30115-8_22

[13] F. Hermans, "Excel turing machine." [Online]. Available: http://www.felienne.com/archives/2974

[14] Microsoft, "Excel (.xlsx) extensions to the office open xml spreadsheetml file format." [Online]. Available: https://msdn.microsoft.com/en-us/library/dd922181(v=office.12).aspx

[15] Microsoft, "Excel functions (alphabetical)." [Online]. Available: https://support.office.com/en-in/article/Excel-functions-alphabetical-b3944572-255d-4efb-bb96-c6d90033e188

[16] M. Van Den Brand, M. Sellink, and C. Verhoef, "Obtaining a cobol grammar from legacy code for reengineering purposes," in *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, electronic Workshops in Computing. Springer verlag.* Citeseer, 1997.

[17] V. V. Zaytsev, "Recovery, convergence and documentation of languages," Ph.D. dissertation, Vrije Universiteit, 2010.

[18] J. Cunha, J. P. Fernandes, J. Mendes, and J. S. Hugo Pacheco, "Towards a Catalog of Spreadsheet Smells," in *The 12th International Conference on Computational Science and Its Applications*, ser. ICCSA'12, vol. 7336. LNCS, 2012, pp. 202–216.

[19] R. Abreu, J. Cunha, J. a. P. Fernandes, P. Martins, A. Perez, and J. a. Saraiva, "Smelling faults in spreadsheets," in *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '14. Washington, DC, USA: IEEE Computer Society, 2014, to appear.

[20] R. Lämmel and V. Zaytsev, "An introduction to grammar convergence," in *Integrated formal methods.* Springer, 2009, pp. 246–260.

[21] B. Fischer, R. Lämmel, and V. Zaytsev, "Comparison of context-free grammars based on parsing generated test data," in *Software Language Engineering.* Springer, 2012, pp. 324–343.