JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 0000; **00**:1–24
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

Parsing Excel formulas: A grammar and its application on four large datasets

Efthimia Aivaloglou[1]*, David Hoepelman[1], Felienne Hermans[1]

Software Engineering Research Group, Delft University of Technology, Mekelweg 4, 2628 CD, Delft, The Netherlands

SUMMARY

Spreadsheets are popular end-user programming tools, especially in the industrial world. This makes them interesting research targets. However, there does not exist a reliable grammar that is concise enough to facilitate formula parsing and analysis and to support research on spreadsheet codebases. This paper presents a grammar for spreadsheet formulas that can successfully parse 99.99% of more than eight million unique formulas extracted from four spreadsheet datasets. Our grammar is compatible with the spreadsheet formula language, recognizes the spreadsheet formula elements that are required for supporting spreadsheets research, and produces parse trees aimed at further manipulation and analysis. Additionally, we utilize the grammar to analyze the characteristics of the formulas of the four datasets in three different dimensions: complexity, functionality and data utilization. Our results show that 1) most Excel formulas are simple, however formulas with more than 50 functions or operations exist, 2) almost all formulas use data from other cells, which is often not local, and 3) a surprising number of referring mechanisms are used by less than 1% of the formulas. Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: spreadsheets; syntax; formula grammar

1. 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. In a survey held in 2003 by the US Bureau of Labor Statistics [2], over 60% of 77 million surveyed workers in the US reported using spreadsheets, making this the third most common use of computers, after email and word processing. A more recent survey among 95 companies world-wide placed spreadsheets fourth, after email, browsing and word processing, accounting for 7.4% of computer time [3]. It is estimated that the number of spreadsheet programmers is bigger than that of software programmers [4, 5].

Because of their widespread use, spreadsheets have been the topic of research since the nineties [6]. Recent research has often focused on analyzing and visualizing spreadsheets [7, 8]. More recently, researchers have attempted to detect data and table clones in spreadsheets [9, 10] and to define *spreadsheet smells*: applications of Fowler's code smells to spreadsheets [11, 12, 13], followed by approaches to refactor spreadsheets [14, 15] and to apply testing practices on spreadsheets [16]. These research works analyze the formulas within spreadsheets, and therefore often involve formula parsing. This is done by using simple grammars which have not been

*Correspondence to: Software Engineering Research Group, Delft University of Technology, Mekelweg 4, 2628 CD, Delft, The Netherlands. E-mail: e.aivaloglou@tudelft.nl evaluated (like in [15] or the formula pattern recovery technique in [13]), through implied, undefined grammars ([7, 11, 12, 14, 16]), or through entirely bypassing formula parsing and using string comparison operations ([10]).

The above analyses are our main motivation towards defining a formula 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, evaluate formula similarity, and explore the structure of spreadsheets. Essentially, a reliable and consistent grammar and its parser implementation, available to the spreadsheet research community, can support research on spreadsheet formula codebases and can enhance the understanding and usability of research results.

A grammar suitable for this goal should be compatible with the official Excel formula language, produce parse trees suited for further manipulation and analysis, and recognize the spreadsheet formula elements that are required for supporting spreadsheets research. We identified and examined two existing grammars: the official, published grammar for Excel formulas [21] and the grammar implemented by the formula parser of the Apache POI Java API for Microsoft Documents[†], and found that neither of them fulfills those requirements.

In this paper we present a grammar that can support research on spreadsheet formulas. We further utilize the grammar to analyze more than eight million unique formulas originating from the three major datasets available in the spreadsheet research community, namely EUSES [17], Enron [18] and FUSE [19], along with a fourth dataset of that we accumulated through crawling the WikiLeaks website. The goal of the analysis is to obtain an understanding of how people program in the spreadsheets formula language by quantitatively evaluating the characteristics of spreadsheet formulas in terms of complexity, functionality and data utilization. Specifically, we explore the following research questions:

RQ1 What are the size and complexity characteristics of Excel formulas?
RQ2 Which types of functions and operations are commonly invoked in formulas?
RQ3 How is input data used in formulas?

The contributions of this paper are (1) a concise grammar for spreadsheet formulas, (2) the evaluation of the compatibility of the grammar using four major datasets, and (3) an exploratory study of the formulas in the datasets in terms of complexity, utilized data and functionality.

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 3 we discuss our design process and present the spreadsheet formula grammar, its lexical and syntactical analysis rules, and details on precedence and ambiguity. Section 4 explains how we implemented and evaluated the grammar. The analysis of the datasets with respect to the research questions is presented in Section 5. In Section 6 we compare the grammar to alternative grammars and parsers and we discuss the grammar and its limitations. Section 7 presents related work and Section 8 concludes the paper.

2. BACKGROUND

Spreadsheets are cell-oriented dataflow programs which are Turing complete [20]. 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 of horizontal *rows* and vertical *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. Functions can be built-in or user-defined (UDFs), which are created using the Visual Basic for Applications programming language.

[†] `https://poi.apache.org/spreadsheet/`

2.1. 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 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.

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. 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 returns only the cells which are occurring in both ranges, `=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. Cells can also be grouped into *tables*, which can be used in *structure references*. This is a special case of references, introduced in Excel 2007, that allow the use of keywords and table headers as row and column specifiers.

2.2. Sheet and External References

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

A reference to another sheet in the same workbook is indicated by using the sheetname as prefix: `=Sheetname!A1`. References to external spreadsheet files are defined by prepending the file name in between square brackets: `=[Filename]Sheetname!A1`. A peculiar type of prefix are those that indicate multiple sheets: `=Sheet1:Sheet10!A1` means cell A1 in Sheet1 through Sheet10. Sheet names are enclosed in single quotes if they contain special characters or spaces, e.g. `='Sheetname with space'!A1`.

2.3. Array Formulas and Arrays

In most 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 signal that it is an array formula, usually by pressing *Ctrl* + *Shift* + *Enter*. Marking a formula as an array formula will enable one- or two-dimensional ranges to be treated as arrays. For example, if `A1`,`A3`,`A3` contain the values `1`,2,3, the array formula `{=SUM(A1:A3*10)}` will evaluate to `60`.

3. SPREADSHEET FORMULA GRAMMAR

To support research on the spreadsheet formula codebases, a grammar for Microsoft Excel spreadsheet formulas should satisfy the following requirements:

1. Compatibility with the official language
2. Produce parse trees suited for further manipulation and analysis with minimal post-processing required

3. Recognize the spreadsheet formula elements required for supporting spreadsheets research

Examining previous research works (including [11, 12, 13, 14, 15, 16]), we find that the spreadsheet formula elements that a grammar for this purpose needs to recognize include functions calls (of build-in and user-defined functions), function arguments, data (of different types) and references (to internal and external cells and ranges of different types).

We identified and examined two existing grammars: the official, published grammar for Excel formulas [21] and the grammar implemented by the formula parser within the Apache POI Java API for Microsoft Documents, and found that none of them fulfills the above requirements, which is further discussed in Section 6.1. For these reasons the authors decided to construct a new grammar with the above requirements as design goals.

3.1. Design Process

The approach that we followed towards developing the grammar was gradual enrichment through trial-and-error: we started from a simple grammar containing only the most common and well known formula structures and implemented it using a parser generator. Subsequently, we repeatedly tested it against formulas extracted from spreadsheet datasets, leading to further enrichments and refinements, until all common and rare cases found in the datasets were supported.

For the first version of the grammar [22] we used the two major datasets that were available in the spreadsheet research community at the time, namely the EUSES dataset [17] and the Enron [18] corpus, from which we extracted 1,035,586 unique formulas. From the simple grammar that we started from until the published version that parses 99.99% of those formulas, we fixed a total of 85 parse issues. We used GitHub for tracking the parse errors that were produced and pull requests for integrating fixes and refinements to the parser.

Since that version, two factors facilitated further refinement of the grammar: first, making its parser open source and increasing its visibility following [22] enabled users to test it with their datasets and discover new parse errors. This was the case with structured references, which the grammar did not support because they are a relatively new feature, not used in the formulas of the EUSES or the Enron datasets. Second, two new large datasets became available for testing, the FUSE corpus [19] and a dataset that we accumulated by crawling the WikiLeaks website, jointly containing over 7,541,840 unique formulas.

Once the grammar presented in [22], enriched to support structured references, was tested against the new datasets, we found that 1,387 of those formulas were not parsed. The process that we followed was similar as with the first two datasets: initially, for each formula we did a preliminary search for the reason causing the parse failure and we grouped the failing formulas into 7 categories of parse errors. Then we modified or enriched the grammar with the missing lexical or syntactical constructs, updated the parser implementation, added unit tests, and re-tested against the datasets, repeating the process until all but 457 formulas were successfully parsed —as explained later in Section 4.1, all but 2 of those formulas do not cause actual parse errors.

3.2. Grammar Class

While the class of this grammar is not strictly LALR(1) due to the ambiguity that will be discussed in Section 3.7, we implemented this grammar using a LALR(1) parser generator. The present ambiguity can be solved by defining operator precedence (section 3.5) and manually resolving conflicts (Section 3.7). These two features are supported by most LALR(1) parser generators.

3.3. Lexical Analysis

Table I contains the lexical tokens of the grammar, along with their identification patterns in the regular expression language and their priorities. All tokens are case-insensitive. Characters are defined as unicode characters x9,xA,xD and x20 and upwards.

Some simple tokens (e.g. '%', '!') are directly defined in the production rules in Section 3.4 in between quotes for readability and compactness.

Table I. Lexical tokens used in the grammar

Token Name	Description	Contents	Priority
BOOL	Boolean literal	TRUE | FALSE	0
CELL	Cell reference	$? [A-Z]+ $? [1-9][0-9]*	2
DDECALL	Dynamic Data Exchange link	’ ([^ ’] | ”)+ ’	0
ERROR	Error literal	#NULL! | #DIV/0! | #VALUE! | #NAME? | #NUM! | #N/A	0
ERROR-REF	Reference error literal	#REF!	0
EXCEL-FUNCTION	Excel built-in function name followed by ‘(’	(Any entry from the function list[3]) \(	5
FILE	External file reference	\[[0-9]+ \]	5
HORIZONTAL-RANGE	Range of rows	$? [0-9]+ : $? [0-9]+	0
NR	Named range	[A-Z_\\$\square_1$][$\square_4$]*	-2
NR-COMBINATION	Named range which starts with a string that could be another token	(TRUE | FALSE | [A-Z]+ [1-9][0-9]*[A-Z\\_.?$\square_1$]) [$\square_4$]+	3
SR-COLUMN	Column definition in structured references	[\w\.]+	-3
NUMBER	An integer, floating point or scientific notation number literal	[0-9]+ .? [0-9]* (e [0-9]+)?	0
REF-FUNCTION	Excel built-in reference-returning function ‘(’	(INDEX | OFFSET | INDIRECT)\(	5
REF-FUNCTION-COND	Excel built-in conditional reference function ‘(’	(IF | CHOOSE)\(	5
RESERVED-NAME	An Excel reserved name	_xlnm\. [A-Z_]+	-1
SHEET	The name of a worksheet	$\square_2$+ !	5
SHEET-QUOTED	A sheet reference in single quotes	($\square_3$ | ”)* ’ !	5
MULTIPLE-SHEETS	A reference to multiple sheets	$\square_2$+ : $\square_2$+ !	1
MULTIPLE-SHEETS-QUOTED	A multiple sheets reference in single quotes	($\square_3$ | ”)+ : ($\square_3$ | ”)+ ’ !	1
STRING	String literal	" ([^ "] | "")* "	0
UDF	User Defined Function followed by ‘(’	(_xll\.)? [A-Z_\][A-Z0-9_\\.$\square_1$]* \(	4
VERTICAL-RANGE	Range of columns	$? [A-Z]+ : $? [A-Z]+	0
Placeholder character	Placeholder for	Specification	
$\square_1$	Extended characters	Non-control Unicode characters x80 and up	
$\square_2$	Sheet characters	Any character except ’ * [ ] \ : / ? ( ) ; { } # " = < > & + - * / ^ % , ␣	
$\square_3$	Enclosed sheet characters	Any character except ’ * [ ] \ : / ?	
$\square_4$	Valid named range characters	A-Z0-9\\_.?$\square_1$	

[3] A function list is available as part of the reference implementation. Lists provided by Microsoft are also available in [23] and [21].

3.3.1. ***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.

3.3.2. ***External References*** The file names of external references in formulas 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.xlsx]Sheet1!A1` is internally stored as `[X]Sheet1!A1`, where `X` can be any number.

For this reason the presented grammar supports only numeric file names in external references. Adding support for full filenames can be achieved by introducing an additional token or altering the `FILE` token. Note that external filenames can be presented to, and entered by, the user in a number of different formats, depending on conditions such as whether or not the file is open in the spreadsheet program.

3.4. Syntactical Analysis

The complete production rules of our grammar in Extended BNF syntax are listed below. Patterns inside { and } can be repeated zero or more times. The start symbol is $Start$. An example parse tree produced using this grammar is drawn in Figure 7(b).

⟨*Start*⟩ ::= ⟨*Formula*⟩
| '=' ⟨*Formula*⟩
| '`{=`' ⟨*Formula*⟩ '`}`'

⟨*Formula*⟩ ::= ⟨*Constant*⟩
| ⟨*Reference*⟩
| ⟨*FunctionCall*⟩
| '`(`' ⟨*Formula*⟩ '`)`'
| ⟨*ConstantArray*⟩
| `RESERVED-NAME`

⟨*Constant*⟩ ::= `NUMBER` | `STRING` | `BOOL` | `ERROR`

⟨*FunctionCall*⟩ ::= `EXCEL-FUNCTION` ⟨*Arguments*⟩ '`)`'
| ⟨*UnOpPrefix*⟩ ⟨*Formula*⟩
| ⟨*Formula*⟩ '`%`'
| ⟨*Formula*⟩ ⟨*BinOp*⟩ ⟨*Formula*⟩

⟨*UnOpPrefix*⟩ ::= '+' | '−'

⟨*BinOp*⟩ ::= '+' | '−' | '`*`' | '/' | '`^`' | '`&`'
| '<' | '>' | '=' | '<=' | '>=' | '<>'

⟨*Arguments*⟩ ::= ⟨*Argument*⟩ { '`,`' ⟨*Argument*⟩ } | ϵ

⟨*Argument*⟩ ::= ⟨*Formula*⟩ | ϵ

⟨*Reference*⟩ ::= ⟨*ReferenceItem*⟩
| ⟨*RefFunctionCall*⟩
| '`(`' ⟨*Reference*⟩ '`)`'
| ⟨*Prefix*⟩ ⟨*ReferenceItem*⟩
| `FILE` '`!`' `DDECALL`

```
⟨RefFunctionCall⟩ ::= ⟨Union⟩
  | ⟨RefFunctionName⟩ ⟨Arguments⟩ ‘)’
  | ⟨Reference⟩ ‘:’ ⟨Reference⟩
  | ⟨Reference⟩ ‘␣’ ⟨Reference⟩

⟨ReferenceItem⟩ ::= CELL
  | ⟨NamedRange⟩
  | VERTICAL-RANGE
  | HORIZONTAL-RANGE
  | UDF ⟨Arguments⟩ ‘)’
  | ERROR-REF
  | ⟨StructuredReference⟩

⟨Prefix⟩ ::= SHEET
  | FILE SHEET
  | FILE ‘!’
  | MULTIPLE-SHEETS
  | FILE MULTIPLE-SHEETS
  | ‘'’ SHEET-QUOTED
  | ‘'’ FILE SHEET-QUOTED
  | ‘'’ MULTIPLE-SHEETS-QUOTED
  | ‘'’ FILE MULTIPLE-SHEETS-QUOTED

⟨RefFunctionName⟩ ::= REF-FUNCTION
  | REF-FUNCTION-COND

⟨NamedRange⟩ ::= NR | NR-COMBINATION

⟨Union⟩ ::= ‘(’ ⟨Reference⟩ { ‘,’ ⟨Reference⟩ } ‘)’

⟨StructuredReference⟩ ::= ⟨SRElement⟩
  | ‘[’ ⟨SRExpression⟩ ‘]’
  | NR ⟨SRElement⟩
  | NR ‘[’ ‘]’
  | NR ‘[’ ⟨SRExpression⟩ ‘]’

⟨SRExpression⟩ ::= ⟨SRElement⟩
  | ⟨SRElement⟩ (‘:’ |‘,’) ⟨SRElement⟩
  | ⟨SRElement⟩ ‘,’ ⟨SRElement⟩ (‘:’ | ‘,’) ⟨SRElement⟩
  | ⟨SRElement⟩ ‘,’ ⟨SRElement⟩ ‘,’ ⟨SRElement⟩ ‘:’ ⟨SRElement⟩

⟨SRElement⟩ ::= ‘[’ (NR | SR-COLUMN) ‘]’
  | FILE

⟨ConstantArray⟩ ::= ‘{’ ⟨ArrayColumns⟩ ‘}’

⟨ArrayColumns⟩ ::= ⟨ArrayRows⟩ { ‘;’ ⟨ArrayRows⟩ }

⟨ArrayRows⟩ ::= ⟨ArrayConst⟩ { ‘,’ ⟨ArrayConst⟩ }

⟨ArrayConst⟩ ::= ⟨Constant⟩
  | ⟨UnOpPrefix⟩ NUMBER
  | ERROR-REF
```

⟨*Formula*⟩ and ⟨*Reference*⟩ are the two most important nonterminals of the grammar. These are also illustrated as syntax diagrams, with most production rules expanded, in Figures 1 and 2.

The ⟨*Formula*⟩ rule covers all types of spreadsheet formula expressions: they can be constants (`=5`), references (`=A3`), function calls (`=SUM(A1:A3)`), array constants (`={1,2;3,4}`, explained in Section 2.3), or reserved names (`=_xlnm.Print_Area`). Function calls invoke actual named

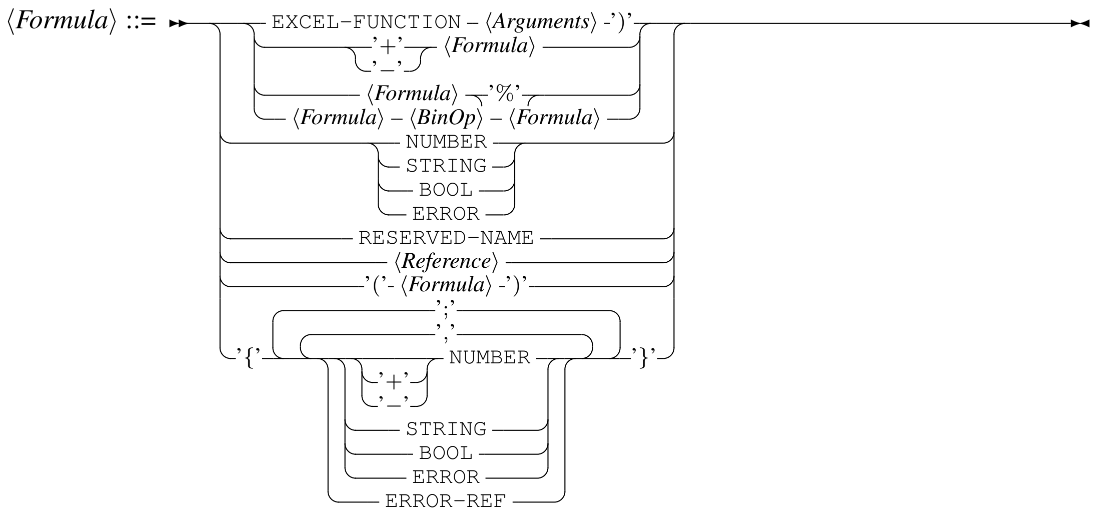


Figure 1. Syntax diagram of the ⟨*Formula*⟩ production rule with most production rules expanded

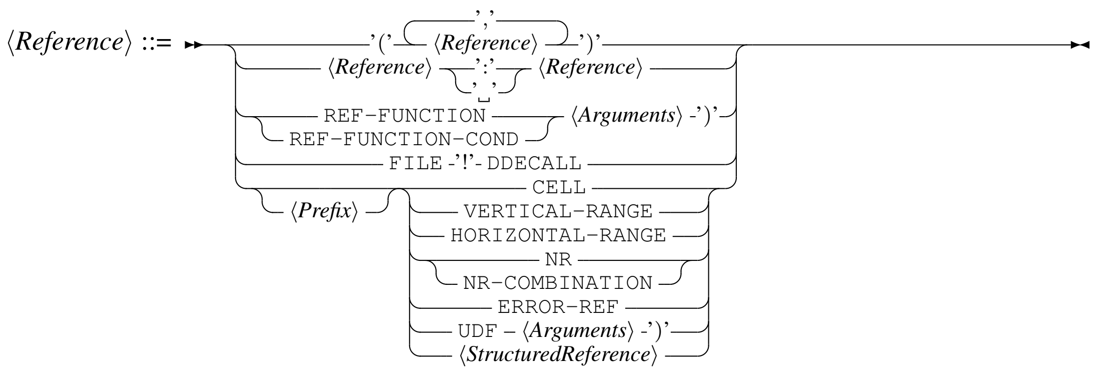


Figure 2. Syntax diagram of the ⟨*Reference*⟩ production rule with most production rules expanded

(built-in or user defined) functions or operators applied to one or more formulas. A special case of built-in functions are those that always return references, namely the `INDEX`, `OFFSET` and `INDIRECT` and the conditional functions that sometimes return references, namely `IF` and `CHOOSE`. 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)`.

The ⟨*Reference*⟩ rule covers all types of referencing expressions, which are diverse. The simple case of a reference to a cell range can be expressed in any of the following ways:

Table II. Operator precedence in formulas

Precedence (higher is greater)	Operator(s)
1	= < > <= >= <>
2	&
3	+ - (binary)
4	∗ /
5	^
6	%
7	+ - (unary)
8	,
9	␣
10	:

```
  SUM(A1:A2)                (1)
= SUM(Sheet!A1:A2)          (2)
= SUM(Sheet!A1:(A2))        (3)
= SUM('Sheet'!A1:A2)        (4)
= SUM(Sheet!A1:Sheet!A2)    (5)
= SUM(namedRangeA1A2)       (6)
= SUM(A1,A2)                (7)
= SUM((A1,A2))              (8)
= SUM(A1:A2:A1)             (9)
= SUM(A1:A2 A:A)            (10)
```

The ⟨*Reference*⟩ rule, as shown in Figure 2, supports internal (in the same or in different sheets), or external single cell references, cell ranges, horizontal and vertical ranges, named ranges and reference-returning, built-in or user-defined, functions.

3.5. 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 as listed in Table II.

3.6. Intersection Operator

The intersection binary operator in Excel formulas is a single space. While this is straightforward to define in EBNF, it can be challenging to implement using a parser generator.

The parser generator we used for implementing the grammar 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 $5 \cdot a$.

3.7. Ambiguity

Due to trade-offs on parsing references (see Section 3.8.1) and on parsing unions (see Section 3.8.2) our grammar is not fully unambiguous. Ambiguity exists between the following production rules:

1. ⟨*Reference*⟩ ::= ‘`(`’ ⟨*Reference*⟩ ‘`)`’

2. ⟨*Union*⟩ ::= ‘`(`’ ⟨*Reference*⟩ { ‘`,`’ ⟨*Reference*⟩ } ‘`)`’
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 a 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 (bracketed ⟨*Reference*⟩) in case of this conflict, because this always is a correct interpretation and thus results in correct parse trees.

3.8. Trade-offs

3.8.1. ***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 3.7.

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

3.8.2. ***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*⟩ } | ϵ

However, this will cause a reduce-reduce conflict because the parser will have a state wherein it can reduce to both a ⟨*Union*⟩ or an ⟨*Argument*⟩ on a `,` token. Unfortunately there is no correct default 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 parses 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 (discussed in Section 5) and, in the datasets we used, all but two were with parentheses (Section 4.1).

4. EVALUATION AND DATASET

In this section we explain how we implemented and evaluated the grammar using four large datasets and we discuss the obtained results and formula parse failures.

The grammar is implemented using the Irony parser generator framework[§]. Irony fulfills the requirements set in Section 3.2: it is a LALR(1) parser generator, is enables defining operator precedence as listed in Table II, and it includes a method, `PreferShiftHere()`, for resolving shift-reduce conflicts as specified in Section 3.7. In addition, Irony enables defining terminals that produce tokens with empty text through a special terminal type, `ImpliedSymbolTerminal`, which we used for defining the intersection operator explained in Section 3.6. At the same time, with Irony the entirety of the grammar was coded directly in C#, which enabled streamlining the development of the parser and of its evaluation tools in the .NET platform.

The resulting parser, named XLParser, is available for download[¶]. An online demo is also available.[‖]

[§]`https://irony.codeplex.com/`
[¶]`https://github.com/PerfectXL/XLParser`
[‖]`http://xlparser.perfectxl.nl/demo`

Table III. The datasets used for evaluation and analysis

	Spreadsheets			Cells		
Dataset	Total	Processed	With formulas	Non-empty	Formulas	Unique
Euses	4,498	4,106	2,599	9,315,032	1,260,877	88,675
Enron	16,190	15,900	9,297	106,131,592	21,160,856	951,366
Fuse	249,376	238,991	14,789	400,776,397	11,056,536	1,110,880
WikiLeaks	109,475	107,061	73,639	2,080,034,916	155,862,445	6,426,505
Total	379,539	366,058	100,324	2,596,257,937	189,340,714	8,577,426

To extract unique formulas from spreadsheets and use them as input to the parser we built a tool that opens spreadsheets using a third-party library called Gembox[**] and converts the ones in Excel version prior to 2007 (with `xls` file extension) to Excel 2007 format. This tool reads all cells and identifies the formulas that are unique when adjusted for cell location (R1C1 representation), thus rejecting the formulas with adjusted references due to cell copying (e.g. formulas `=C1` and `=C2` are considered the same if contained in cells A1 and A2 respectively). The tool then uses each unique formula string as input to the parser.

To evaluate the grammar we attempt to parse a total of 8,577,426 unique formulas. These originate from the three major datasets available in the spreadsheet research community, the EUSES dataset [17], published in 2005 and consisting of 4,498 spreadsheets, the Enron email corpus [18], which became available after the Enron company declared bankruptcy in 2001, consisting of 16,190 spreadsheets, and the recently published FUSE corpus [19], consisting of 249,376 spreadsheets, along with a fourth dataset of 109,475 spreadsheets that we accumulated through crawling the WikiLeaks website. The original spreadsheets in the datasets are of various Excel versions. The Euses and the Enron datasets contain spreadsheets in the Excel binary file format (`.xls` files), while the last two datasets include more recent spreadsheets, saved in Excel 2007 and later workbook format (`.xlsx`, `.xlsm` and `.xlsb` files). We were not able to process 13,481 (3.55%) of all spreadsheets, either because they are password protected, or because of read failures in the Gembox library. Table III summarizes the data obtained from each dataset. In total, the 366,058 spreadsheets that were processed from the four datasets include 189,340,714 formula cells with 8,577,426 unique formulas.

Out of the 8,577,426 unique formulas that were used as input to the parser, 8,576,969 (99.99%) were parsed successfully, with 2 of the failing ones being actual parse errors, as will be explained in the following section.

4.1. Unparsable Formulas

The 457 formulas that were not parsed using the grammar defined in Section 3 are:

- `=-NOX, Regi` and `=-_SO2, Regi`, found in two different workbooks in the Enron dataset. These are cases of an union operations without parentheses that the grammar does not parse as explained in Section 3.8.2.
- 28 formulas originating from 13 spreadsheets that Excel does not evaluate either. Those formulas include indecipherable characters, for example `=+Ë%` in a file that appears to be corrupt, and they result in `#NAME?` errors.
- 371 formulas originating from 67 spreadsheets that are not returned correctly from the Gembox library. For example, our tool reads and attempts to parse the formula `=1 SUMM BS!A3:K3` and fails, but in reality the formula is `='1 SUMM BS'!A3:K3` which can be parsed. All these 371 cases were parsed successfully when we exported them into a flat file,

[**]`http://www.gemboxsoftware.com/`

manually corrected them to match their original form, and provided them as input to the parser.

- 56 formulas originating from 2 spreadsheets that include filenames with brackets and are not returned correctly as numeric. For example, formula `='C:\[FY2014WSSBBasic[1].xls].xls]Basic Grant'!A6` is read as `='[1].xls].xls]Basic Grant'!A6`, which is not parsed, as the grammar is designed to support numeric-only filenames for the reasons explained in Section 3.3.2. All those 56 cases are parsed successfully when the filename is replaced with a numeric one.

5. RESULTS

In the following sections, for each of the research questions, we describe the results obtained through the analysis of the 8,576,969 parsed unique formulas in the dataset.

5.1. Formula Complexity

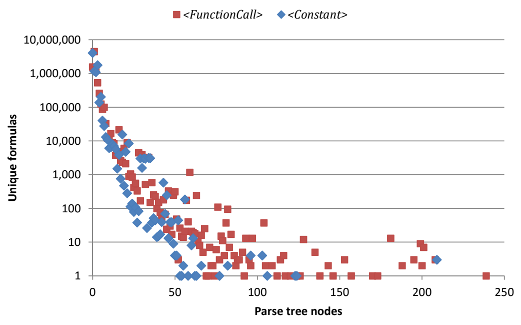
The complexity of the formulas is reflected in the structure of the produced parse trees. To demonstrate how the parse trees can be used for analyzing formula characteristics, we calculated four formula complexity indicators: the number of functions and constants, the conditional depth, the formula depth, and the operator depth. The results, in terms of number of unique formulas, are plotted in Figure 3.

The majority (51.94%) of the unique formulas have exactly one $\langle$*FunctionCall*$\rangle$ node —they invoke one Excel function or they include one binary or unary operation. 18.18% of the formulas have no such nodes, while 85.16% have two or less. A small number of formulas (2,356) have more than 50 $\langle$*FunctionCall*$\rangle$ nodes, and up to 239. Examining the utilization of constants, 47.1% of the unique formulas do not include any, while 26.2% include one or two constants. More than 10 constants are used in 0.95% of the formulas, while 278 formulas were found to contain more than 50 and up to 209 constants.

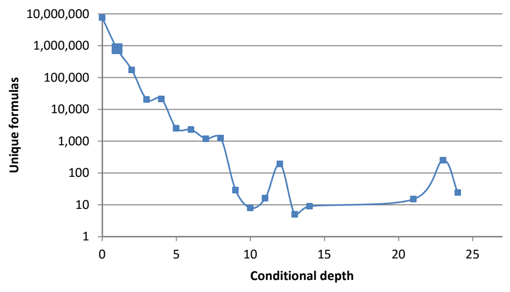
The conditional depth of formulas is the number of nested conditional function calls that they include. We measured it as the depth of function-invoking parse tree nodes (`EXCEL-FUNCTION` and `REF-FUNCTION-COND`) created for conditional functions[††]. Formula `ROUND(IF(D6>1600,D9,(1+(1600-IF(D6>D7,D6,D7))*D8)*D9),0)` (1), for example, has a conditional depth of 2. We found that 7,549,671 (88.02%) of the formulas in the datasets do not include any conditional function call and 805,359 (9.39%) include only one. On the opposite end, 7,791 formulas include 5 or more nested conditional function calls and 288 parse trees have conditional depth over 20.

The formula depth and the operator depth are indicators of the formula complexity in terms of calculation operations. The first is the depth of $\langle$*Formula*$\rangle$ nodes in the tree, while the latter of function-invoking $\langle$*FunctionCall*$\rangle$, $\langle$*RefFunctionCall*$\rangle$ and `UDF` nodes. In the example in Figure 7a, those are 2 and 1 respectively. The majority of the formulas in the dataset are as simple as this example or simpler: 68.72% of the parse trees have formula depth up to 2, and 55,2% have operator depth 0 or 1. However, as shown in Figures 3c and 3d, significantly complex formulas have also been found. 52,539 trees have formula depth greater than 11 and 79,885 trees have operator depth greater than 8, which are the formula and operator depths of formula (1). All formulas with operator depths over 65 involve simple but repetitive operations (additions, string concatenations or multiplications) of a large group of numbers or cells. For example, the formula with the largest operator depth of 208 is `320+204+...+616`, adding a total of 209 numbers.

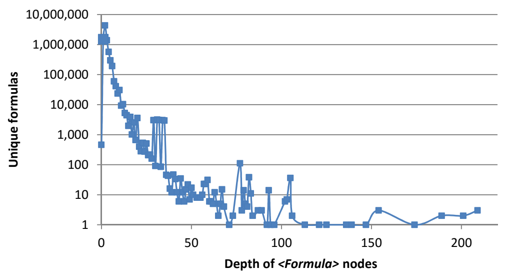
[††]Conditional functions are the `IF`, `COUNTIF`, `COUNTIFS`, `SUMIF`, `SUMIFS`, `AVERAGEIF`, `AVERAGEIFS` and `IFERROR`.



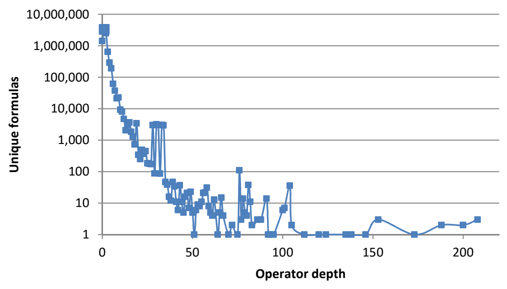
(a) Number of functions and constants



(b) Depth of conditional function nodes



(c) Depth of ⟨*Formula*⟩ nodes



(d) Depth of function-invoking nodes

Figure 3. Complexity indicators for the unique formulas in the four datasets combined (vertical axes are in logarithmic scale)

RQ1: The majority of Excel formulas are small and simple: 70% of the unique formulas in the datasets include up to one function or operation and up to two constants. 88% of the unique formulas in the datasets do not invoke conditional functions. There exist surprisingly large and complex formulas. These include formulas that invoke simple but repetitive operations, but also formulas with many nested conditional functions.

5.2. *Functions and Operations*

Table IV shows the number of parsed formulas in the four datasets that contain each type of parse tree node. All parsed formulas contain at least one ⟨*Formula*⟩ node. 86.61% of the formulas include a ⟨*FunctionCall*⟩. Built-in value-returning functions are invoked by 42.70% of the formulas, the majority of which contain exactly one function call. This is derived from the data plotted in Figure 4a, showing the number of `EXCEL-FUNCTION` nodes in the parse trees of the unique formulas in datasets. We found that 6.13% of the unique formulas contain two or more of such nodes.The largest number of `EXCEL-FUNCTION` nodes was found in a 1.231 character-long formula that invoked repetitively 7 distinct functions for a total of 91 times.

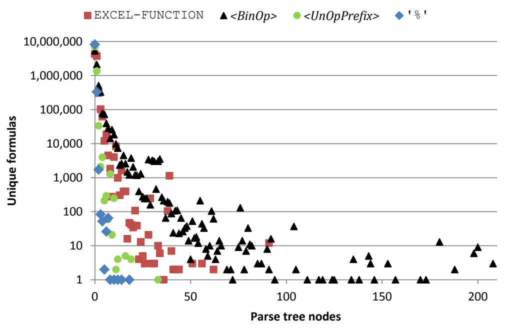
A significant amount of formulas (1,963,284 or 1.04%) invoke user-defined functions—e.g., `=erUserEmail(User_Id)`. A special case of user defined functions are the ones created using an Excel add-in. These are invoked as `_xll.functionName` in 0.65% of the formulas.

Operators are common, with binary operators appearing in 48.58% of the formulas and 18.47% of the formulas containing prefix operators. Figure 4a plots the number of operators of each type. Exactly one binary operator is found in 24.69% of the unique formulas, 9.77% have two or three and 3.95% have more than three and up to 239.

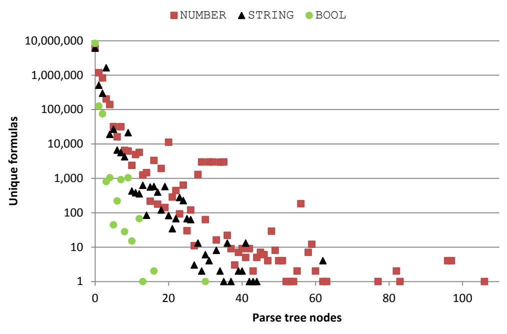
Table V lists the most frequently used functions and the operators, along with their total occurrences in the unique formulas of the four datasets combined. The frequency of operators is similar across datasets, with the plus being the most popular one in all four. The frequency of

Table IV. Frequency of spreadsheet formulas with specific grammatical structures in the combined EUSES, Enron, Fuse and WikiLeaks datasets

Syntax	Example	Unique formulas		Total formulas	
⟨*Formula*⟩	`=1+2`	**8,576,969**	100%	**189,335,068**	100%
⟨*Reference*⟩	`=E9/E10`	8,048,520	93.84%	187,297,690	**98.92%**
CELL	`=A5`	7,961,614	92.83%	186,125,582	98.30%
⟨*FunctionCall*⟩	`=SUM(A5:A22)`	7,017,678	81.82%	163,982,378	**86.61%**
⟨*BinOp*⟩	`=H10-H8`	3,294,376	38.41%	91,984,979	**48.58%**
EXCEL-FUNCTION	`=SUM(A5:A22)`	4,271,892	49.81%	80,850,975	**42.70%**
⟨*Constant*⟩	`=A5+134`	4,537,173	52.90%	80,185,313	**42.35%**
NUMBER	`=(B8/48)*15`	2,473,935	28.84%	67,421,408	**35.61%**
⟨*Prefix*⟩	`=Sheet1!B1`	2,918,585	34.03%	54,019,709	**28.53%**
⟨*Reference*⟩ ':' ⟨*Reference*⟩	`=SUM(A5:A22)`	1,509,887	17.60%	46,037,124	**24.32%**
⟨*UnOpPrefix*⟩	`=+B11+1`	1,399,182	16.31%	34,963,857	**18.47%**
⟨*RefFunctionName*⟩	`=SUM(J9:INDEX(J9:J41,B43))`	722,407	8.42%	28,592,583	**15.10%**
REF-FUNCTION-COND	`=IF(A1<0,0,1)`	682,713	7.96%	27,747,635	**14.66%**
STRING	`=IF(AD3<0,"buy","sell")`	2,529,561	29.49%	25,350,954	**13.39%**
SHEET	`=Sheet1!B1`	1,271,751	14.83%	20,255,725	10.70%
BOOL	`=IF(AND(R11=1,R14=TRUE),G19,0)`	204,060	2.38%	17,442,457	9.21%
FILE	`=[11]Sheet1!C5`	755,875	8.81%	15,759,824	**8.32%**
QUOTED-FILE-SHEET	`=('[2]Detail I&E'!D62)/1000`	318,600	3.71%	10,466,380	5.53%
VERTICAL-RANGE	`=COUNT(A:A)`	244,153	2.85%	7,927,994	**4.19%**
⟨*NamedRange*⟩	`=SUM(freq)`	206,336	2.41%	3,389,407	**1.79%**
REF-FUNCTION	`=SUM(J9:INDEX(J9:J41,B43))`	49,753	0.58%	1,967,840	**1.04%**
UDF	`=SQRT(_eoq2(C5,C4,C6,C7))`	72,552	0.85%	**1,963,284**	**1.04%**
ERROR-REF	`=AVERAGE(#REF!)`	57,375	0.67%	1,435,260	**0.76%**
_xll.	`=_xll.RiskTriang(F9,F7,F8)`	50,575	0.59%	1,221,357	**0.65%**
'(' ⟨*Reference*⟩ ')'	`=(2*(B29))/(1+B29)`	29,191	0.34%	1,201,914	0.63%
'%'	`=IF(E5>I8,3%,0%)`	334,627	3.90%	792,299	0.42%
Empty argument	`=DCOUNT(Lettergrades,,I80:I81)`	13,180	0.15%	324,621	**0.17%**
Intersection	`=Ending_Inventory Jan`	17,817	0.21%	**225,621**	0.12%
FILE '!'	`=[1]!today`	9,152	0.11%	**171,048**	**0.09%**
External UDF reference	`=[1]!wbname()`	7,074	0.08%	**139,851**	**0.07%**
HORIZONTAL-RANGE	`=MATCH(F3,Prices!2:2,0)`	5,140	0.06%	137,418	**0.07%**
⟨*StructureReference*⟩	`=MAX(Vertices[In-Degree])`	1,016	0.01%	54,153	0.03%
ERROR	`=IF(R14=TRUE,G19,#N/A)`	549	0.01%	28,706	**0.02%**
MULTIPLE-SHEETS	`=SUM(Sheet1:Sheet20!I29)`	289	0.00%	**25,218**	**0.01%**
Complex ranges	`=SUM(I8:K8:M8)`	433	0.01%	**9,166**	**0.00%**
Prefixed right ref. limit	`=SUM('Tot-1'!$B8:'Tot-1'!B8)`	345	0.00%	**3,941**	**0.00%**
DDECALL	`=TWINDDE|RSFRec!'NGH2 NET.CHNG'`	3,279	0.04%	**3,689**	0.00%
FILE MULTIPLE-SHEETS	`=SUM([2]Section3A:formulas!B11)`	32	0.00%	**1,054**	0.00%
⟨*ConstantArray*⟩	`=FVSCHEDULE(1,0.09;0.11;0.1)`	75	0.00%	**743**	0.00%
RESERVED_NAME	`=C23/_xlnm.Print_Area`	32	0.00%	**672**	0.00%
⟨*Union*⟩	`=LARGE((F38,C38),1)`	24	0.00%	**578**	0.00%



(a) ⟨*FunctionCall*⟩-related nodes



(b) ⟨*Constant*⟩-related nodes

Figure 4. Frequency of different types of functions, operations and constants in the parse trees of the unique formulas in the four datasets combined (vertical axes are in logarithmic scale)

Table V. Frequency of the most common functions and operators found in the unique formulas of the four datasets

Operators		Functions	
Operator	Occurrences	Function	Occurrences
+	3,427,080	CONCATENATE	1,780,969
-	1,312,314	IF	916,201
/	1,145,218	SUM	832,769
*	1,079,082	ROUND	624,448
=	617,288	SUMIF	368,163
&	502,307	VLOOKUP	301,178
%	337,259	DAYS360	253,058
>	272,438	SUBTOTAL	165,182
<	83,287	COUNTIF	126,791
<>	37,494	AND	83,588
>=	17,577	SUMIFS	74,019
<=	16,028	ISNA	47,918
^	13,308	ISERROR	46,652

functions varies, with `IF` and `SUM` being the ones in the top-ten most frequently appearing in all four datasets. From the top-five functions in each dataset, the ones that are missing from the table are the `AVERAGE` (fifth most frequent function in the Enron dataset) and the `INDIRECT` (third most frequent one in the Euses dataset). Examining the most rarely used Excel build-in functions, we find that `STEYX`, `VARA`, `CONVERT`, `EXPONDIST`, `WEIBULL`, `BETADIST` and `COSH` are used only one time each, the first two in the Euses and the remainder in the Fuse dataset.

Regarding function arguments, spreadsheet systems allow empty arguments (e.g. `=SUM(,E35,E37)`) but this is rarely done—in only 0.17% of the formulas. In the EUSES and Fuse datasets we also found 743 cases of constant arrays used as arguments, e.g. `=FVSCHEDULE(1,{0.09;0.11;0.1})`. Reserved names are uncommon, with 667 occurrences of the `_xlnm.Print_Area` and 5 occurrences of `_xlnm.Database`.

Analyzing the utilization of constants in formulas, we find that 42.35% of the formulas contain at least one; more than one third (35.61%) of the formulas contain a number and 13.39% are formulas that contain text. Figure 4b plots the number of constants of different types in the parse trees. One to four numbers is used in 27.18% of the unique formulas and 0.56% contain more than 10 numbers and up to 209 (was left out of the plot for readability). For textual constants, an interesting finding is that there are most commonly three of them in formulas —19.09% of the unique formulas contain three `STRING` tokens in their parse trees, while 9.38% contain one or two.

Table VI. Frequency of the most common constants found in the unique formulas of the four datasets combined

Numeric		Non-Numeric	
Constant	Occurrences	Constant	Occurrences
0	1,021,557	""	1,951,521
1	491,056	")"	1,565,605
1000	468,986	"("	1,561,982
9	185,929	FALSE	284,898
30	179,070	"*"	157,521
2	170,822	"."	118,833
-1	133,278	"-"	89,572
100	95,286	"Y"	59,001
12	75,176	TRUE	15,106
3	74,671	"YES"	13,609

Table VI lists the most frequently used constants, along with their total occurrences in the unique formulas of the four datasets combined. The frequency of numerical constants is similar across datasets, but this is not the case with the non-numerical ones, with `FALSE` and `"-"` being the only ones in the top-twenty of all datasets.

The array formulas production rule, covering ⟨*Formula*⟩s 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 four datasets.

RQ2: 87% of the total formulas in the datasets include at least one function or operation: 43% of the formulas include Excel value-returning functions, with `IF` and `SUM` being common in all datasets, 49% include binary operations, with addition being by far the most common one, and 18% include prefix operators. User-defined functions are found in 1% of the formulas. 42% of the total formulas contain at least one constant, usually a number, with 0 being the most common one.

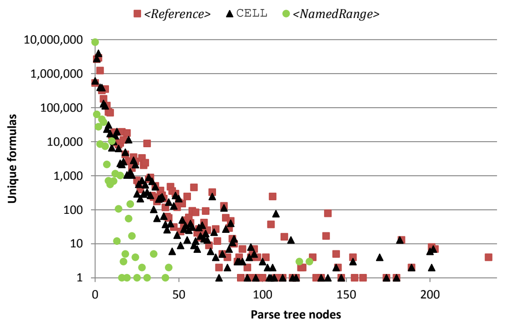
5.3. *Input Data*

Spreadsheet formula calculations can be performed using data from other cells, internal or external, local or in different sheets. This data is used in formulas by specifying references to these cells. 98.92% of the formulas in the four datasets contain at least one ⟨*Reference*⟩. Figure 5a plots the number of references in the parse trees of the unique formulas. One or, more commonly, two reference nodes are found in the parse trees of the majority of the formulas (in 64.61%), while 1.82% have more than 10 reference nodes and up to 235.

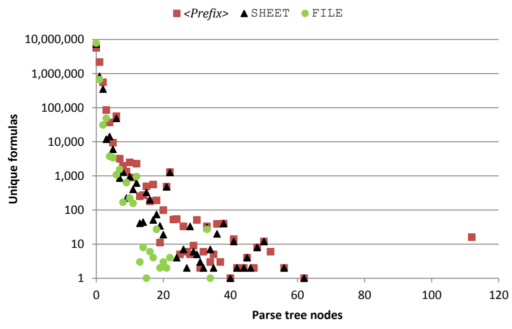
As explained in Section 2.1, references can include cell ranges, which can be constructed by three operators: the range operator `:`, the union operator `,`, and the intersection operator ␣. The range operator is the most common one, with 24.32% of the formulas including at least one instance of it. In the four datasets, intersection operations are found in 225,621 formulas (0.12%), the majority of which (80%) being in the Fuse dataset alone. Unions are found in only 578 formulas, e.g. `=LARGE((F38,C38),1)`. The majority of the occurrences are arguments of the `LARGE`, `SMALL` and `RANK` functions—these functions require a range of cells to be declared as a single argument, necessitating a union if the cells are not in a single range.

References to named ranges exist in 1.79% of formulas. The Enron dataset differs significantly from the other ones in this aspect, having named ranges in 7.24% of its formulas. While most formulas with named ranges refer to only one, there exist complex cases: 3.215 unique formulas were found with more than 10 ⟨*NamedRange*⟩ nodes, and even 6 with more than 100.

28.53% of the total formulas contain a reference that is not local, since it includes a ⟨*Prefix*⟩. External file references exist in 8.32% of the formulas. Figure 5b plots the number of ⟨*Prefix*⟩ nodes in the parse trees of the unique formulas. Most formulas that utilize prefixed references have exactly



(a) ⟨*Reference*⟩-related nodes



(b) ⟨*Prefix*⟩-related nodes

Figure 5. Frequency of references and prefixes in the parse trees of the unique formulas in the four datasets combined (vertical axes are in logarithmic scale)

one `SHEET` or `FILE` node in their parse trees. Only 1.07% of the unique formulas include more than one external file references and up to 34, although all references in this case were to the same external file.

Interestingly, horizontal and vertical ranges are rarely used (in 0.07% and 4.19% of the formulas respectively), especially in the Enron and the Euses datasets, where they appear jointly in 0.25% of the formulas. 0.76% of formulas include references to errors, e.g. `=#REF!E3`. These reference errors are by far the most common type of error —the `ERROR` token exists in only 0.02% of the formulas.

Moving to the edge cases of the grammar, the structures that are least common in the datasets include:

File-only external references

External references are normally in the form `[File]Sheet!Cell`. In 171,048 formulas (0.09%), however, the sheet is not specified, e.g. `=[2]!LastTrade`. These are cases of references to either external named ranges or external UDFs.

References to external UDFs

139,851 formulas (0.07%) contain references to external UDFs, for example `=[1]!SheetName()`.

Structured references

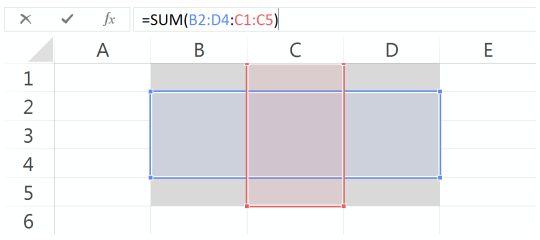
54,153 formulas (0.03%) from the Fuse and the WikiLeaks datasets contain this relatively new type of reference, introduced in Excel 2007. An example is `tblData[[#This Row],[Year]]`, returning the value of the cell in the current row of the defined table `tblData` in the table column with header `Year`.

Multiple sheet references

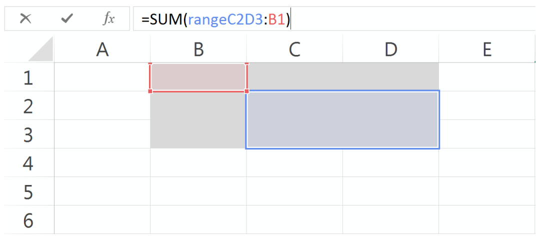
25,218 formulas (0.01%) contain this complex case of reference, which spans across multiple sheets. An example formula is `=SUM(Sheet1:Sheet10!A5)`, evaluated by summing all cells in position A5 from Sheet1 to Sheet10. In 1,054 formulas, the reference is to external files.

Complex ranges

9,166 formulas (0.005%) contain ⟨*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 6a, which is equivalent to `B1:D5`. The range 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. Understanding these limits is even less intuitive when vertical or horizontal ranges or named ranges are used, like in Figure 6b. The majority (95%) of the complex ranges in the dataset are defined using three cell locations.



(a) A range with four limits `B2:D4:C1:C5`, equivalent to the area marked gray `B1:D5`



(b) A range with a named range `rangeC2D3:B1`, equivalent to the area marked gray `A1:C3`

Figure 6. Examples of references to complex ranges

Prefixed right limits

3,941 formulas (0.002%) include a reference with a prefix in the right limit, e.g. `=SUM('Deals'!F9:'Deals'!F16)`, which is equivalent to `=SUM('Deals'!F9:F16)`. 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.

Examining the special case of Excel functions that return references (the `INDEX`, `OFFSET` and `INDIRECT`), they are found in 1.04% of the formulas, with the most common one being the `INDEX` (in 0.69% of formulas) and the least common one being the `OFFSET` (in 0.10%). While the `IF` and `CHOOSE` functions can be part of reference expressions, there were no formulas in the datasets using them as such. An example of using those functions in this way would be `=SUM(IF(A1=1,A2,A5):A10)`, which is equivalent to `=SUM(A2:A10)` if `A1` is `1` and to `=SUM(A5:A10)` otherwise.

Another rare case of references are the dynamic data exchange links, recognized using token `DDECALL`, which were found in 3,686 formulas of the Enron and the Euses datasets and in only 3 formulas of the other two. These take the form of `=Program|Topic!Arguments`, e.g. `=Database|TableA!Column1`, and are used in Windows versions of Microsoft Excel to receive data from other applications.

RQ3: Almost all formulas (98%) use data from other cells, through various types of references. References in the form of cell ranges are used in one fourth of the formulas, but unions and intersections are rare. 29% of the formulas refer to cells outside their worksheet, and even 8% of the formulas include references to cells in external files. References to named ranges, to horizontal and vertical ranges and to reference-returning Excel functions are rare. In less than 0.1% of the formulas we found multiple sheet references, complex ranges, structured references or dynamic data exchange links.

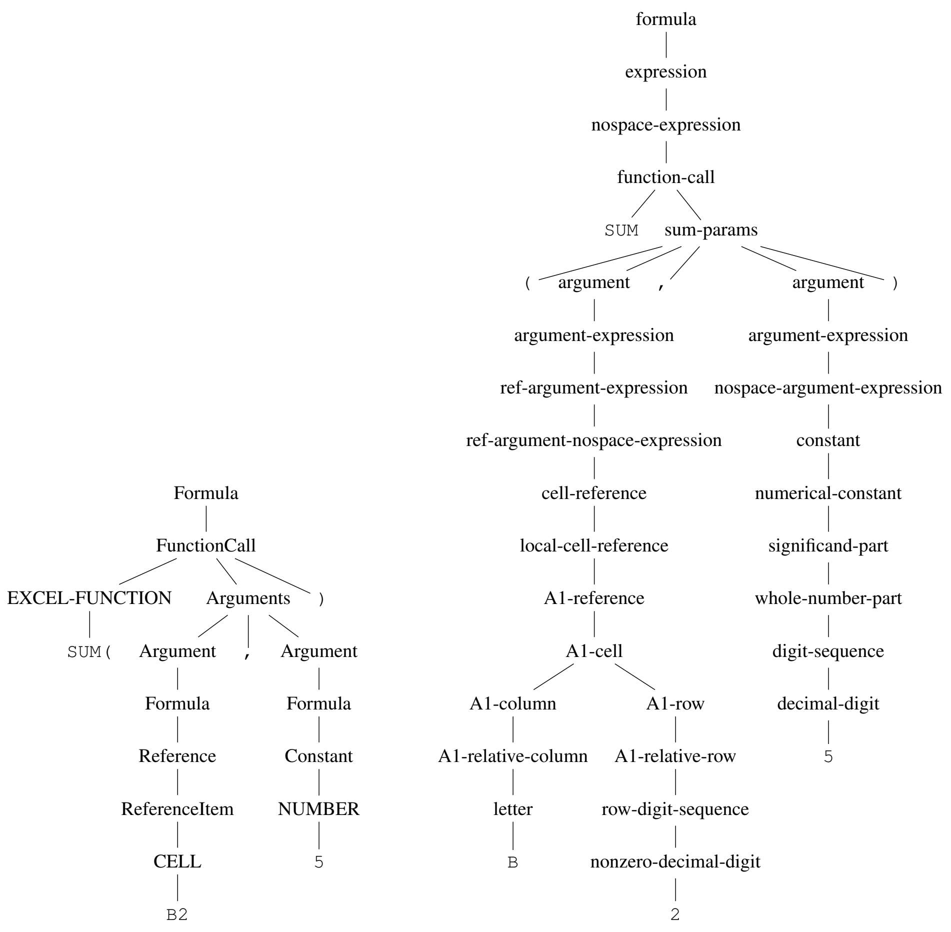
6. DISCUSSION AND LIMITATIONS

The currently defined formula grammar is able to parse 99,99% of the 8,577,426 unique formulas in the four datasets. In this section, we compare it to other grammars and parsers and we discuss a variety of issues that affect its applicability and suitability.

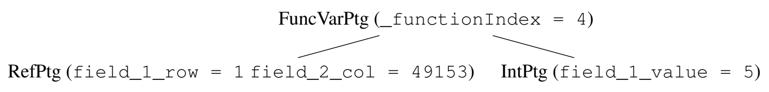
6.1. *Alternative grammars and parsers*

We have identified two alternatives to the proposed grammar and its parser implementation: The official, published grammar for Excel formulas [21] and the formula parser implementation within the Apache POI Java API for Microsoft Documents[‡‡]. Of the related works discussed in this paper,

[‡‡] `https://poi.apache.org/apidocs/org/apache/poi/ss/formula/FormulaParser.html`



(a) Parse tree produced using the grammar defined in this paper (b) Microsoft Excel parse tree, constructed based on reference [21]


FuncVarPtg (_functionIndex = 4)
RefPtg (field_1_row = 1 field_2_col = 49153)
IntPtg (field_1_value = 5)

(c) Parse tree produced by the Apache POI (v3.15) FormulaParser
‡‡

Figure 7. Parse trees for formula `SUM(B2,5)`

[13] and [15] utilize Apache POI for processing spreadsheets, but we are not aware of any related works utilizing either the official Excel formula grammar or Apache POI for formula parsing.

From the requirements set in Section 3, the official Excel formula grammar naturally fulfills the first one, on compatibility. However, it is too granular for our purpose —it is over 30 pages long and contains hundreds of production rules. Because of its detail and the large number of production rules, the resulting parse trees are very complex and thus fail requirement 2. An example is given in Figure 7(b): the relatively simple formula `SUM(B2,5)` results in a 37-node tree with a depth of 18 nodes. For our purpose of facilitating research on spreadsheet formulas, we need a grammar that provides a different level of detail, just-enough to satisfy requirement 3.

Examining the Apache POI formula parser (the current version of which is v3.15), the first issue that we encountered is the lack of grammar specification: We found no published or defined grammar, apart from a high-level grammar composed of 4 BNF syntax rules in the comments of the FormulaParser class. The grammar specification can therefore only be retrieved by reverse-engineering the implementation. Moreover, the FormulaParser class is marked to be 'for POI internal use only', both in its source code and in its documentation. The produced parse tree is not offered through an interface while the root node, which is required for traversing it, is declared as a private property, not exposed outside the FormulaParser class.

The Apache POI formula parser fails our second requirement because of the parse tree structure. As demonstrated in the example in Figure 7(c), the parse trees it produces are condensed. However, this is at the expense of defining many different types of edge nodes to represent different syntactical cases, each type with its own properties and methods (in the current version, there exist 66 types of edge nodes, counted as the members of the org.apache.poi.ss.formula.ptg package). For example, reference `A1` would be represented as an edge node of type `RefPtg`, reference `A1:A3` as an `AreaPtg` and reference `Sheet3!B6` is an `Ref3DPxg`. For a simple task like finding the cell references of a formula, we would therefore need to explore and handle all different types of edge nodes that might relate to references through various properties. Similar to the official Excel formula grammar specification, it is built for a different purpose than the proposed grammar, i.e., to facilitate the evaluation of formulas, and this makes it less suitable for the intended use.

Finally, comparing it to the proposed grammar, Apache POI has not been tested against and improved based on the datasets discussed in this paper. To reach this conclusion, we compiled a list of the latest grammar cases that were found in the datasets and our parser had to be enriched to support according to the process described in Section 3.1. We tested the Apache POI formula parser against those cases and we found 6 cases that caused it to generate parse errors and incorrect parse trees. Example grammatical cases that we found that Apache POI in its current version does not support are intersections between named ranges (e.g. `SUM(January Sales)`), ranges with error references (e.g. `SUM(REF!:REF!)`), quoted multiple sheet references (e.g. `SUM('sales 1:sales 10'!F9)`), which it incorrectly recognizes as single-sheet references (to non-existent worksheet `'sales 1:sales 10'`), and references to quoted sheets in external files (e.g. `'[file.xlsx]final sales'!A20`), which it incorrectly recognizes as references to local worksheets.

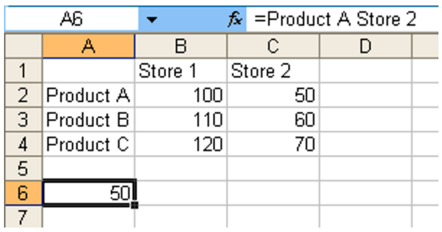
6.2. Dialects

While other spreadsheet programs (e.g. Numbers, LibreOffice, Google Sheets) have generally adopted the Excel formula syntax, there are slight differences between these programs and even between Excel versions. Our grammar has been designed as generically as possible and has been enriched to include all syntactical features found in the four datasets. All datasets, however, contain spreadsheets created in, or converted to, the Excel 2007 format, as explained in Section 4. The conversion does not affect the function names of previous Excel versions that have been deprecated after Excel 2007, but it does affect deprecated syntactical features like labels or regular expressions, which are described below. This limits the grammar support for language elements that are spreadsheet system-dependent or even version-dependent.

Certain differences between spreadsheet systems or versions are irrelevant to the grammar; for example, differences in the number or the type of built-in function arguments would not affect the grammar. However, other differences will require grammar adjustment. Examples are new syntactical features, like the structured references that were introduced in Excel 2010 and the grammar was adjusted to support, and changes in the built-in functions list, which would make the parser mistakenly recognize built-in functions as user-defined functions. Another example is found in LibreOffice, which uses ~ as the union operator instead of `,`. The presented grammar will need to be modified to account for these differences before it can be used on other dialects.

Syntactical features have also been deprecated between Excel versions. 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

More information on those cases can be found in the issues 60979 to 60984 that we opened in the Apache POI project, accessible through `https://bz.apache.org/bugzilla/show_bug.cgi?id=<issue-number>`



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				

Figure 8. A natural language formula in Excel 2003

expressions are instantly resolved—in the example, to `=SUM(Sheet2:Sheet3!A1)`, summing up all A1 cells between Sheet2 and Sheet3, where the sheets are all sheets matching the regular expression, except the one that the formula is on. This way, in Excel versions 2010 and up, saved spreadsheets never contain regular expressions.

The use of labels in formulas (referred to as natural language formulas) is another feature that was discontinued from Excel 2007 onwards. 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 8 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`. Our grammar does not support labels, and it would mistakenly parse them as named ranges.

6.3. *Internationalization*

The presentation of Excel formulas to the user differs depending on the language settings 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 supports only the English locale. Grammars for other locales can be derived by replacing delimiters, error values and function names with their localized versions.

It is worth noting that Excel will always save formulas in either a locale-independent form (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 always available English locale.

6.4. *Rejection of Invalid Formulas*

As stated in the design goals in Section 3, 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 four big datasets of valid formulas, no such datasets of invalid formulas exist. As such, we expect that the presented grammar will parse formulas which are not valid. Using this grammar to parse possibly-invalid formulas like user-input might thus require additional safeguards.

On one point we know the grammar to be too broad: Excel places several limitation on formulas like the number of arguments of a function (255), nested function calls (64), row number (2^{20}), column number (2^{14}) and total formula length (2^{13}), with lower numbers in older file formats. Our grammar does not enforce any of these limits.

6.5. Parse Tree Correctness

While we have empirically shown a high compatibility in terms of successful parse rate, we do not have as much evidence that the produced parse trees are correct as this is only tested by usage and unit tests in the reference implementation. We have manually sampled numerous parse trees and we have found them to be correct. We believe it is unlikely that a formula parsed with the presented grammar would be interpreted differently by Excel, but we do seek additional feedback on possible erroneous parse trees from the research community .

7. RELATED WORK

Efforts to reverse-engineer language characteristics based on existing artifacts have been successful for other languages, including COBOL [24] and C, C++, C# and Java [25].

Most related to our research on the spreadsheet formula language is the work of Badame and Dig [15] 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, unions, named ranges and operator precedence. An extension of the same grammar was used to refactor formulas by Hermans and Dig [14].

There exist other works that analyze spreadsheets and their formulas, but the analysis is limited to a single dataset. [17] presents the EUSES dataset, along with summary statistics on its formula functions, input cells and data types, which are obtained using the Excel VBA API. A similar analysis for the Enron dataset is presented in [18], using data that was obtained with a previous version of the parser proposed in this paper.

Furthermore, there is a large body of related work that relies on parsing spreadsheet formulas to analyze spreadsheets. This includes our own work, in which we have created an algorithm to visualize spreadsheets as dataflow diagrams [7], and subsequently on detecting smells in spreadsheets [11] and on applying testing practices on spreadsheets [16]. Related approaches exist, for example the work of Cunha et al. that have worked on code smells [26] and smell-based fault localization [27] and the work of Cheung et al. on cell clustering and smell detection [12]. These papers also analyze spreadsheet formulas but do not detail which analysis method or grammar they use for formula parsing. The table clone detection mechanism recently presented by Dou et al. in [10] relies on detecting the similarity between spreadsheet formulas, but utilizes string comparison of their R1C1 representation, along with undefined techniques for recognizing constants and external references.

8. CONCLUSION

In this paper we (1) present a grammar for spreadsheet formulas, (2) evaluate it against over eight million unique formulas, successfully parsing 99.99%, and (3) use it to analyze the formulas in the dataset in terms of complexity, functionality and utilization of data.

The grammar is compact and produces 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 codebases. It has already been applied in other works for analyzing formula characteristics, calculation chains and code smells and for applying formula transformations. The XLParser is published as open-source software.

A point of improvement for the grammar is that its exact compatibility with the official Excel grammar is unknown. A comparison to the official specification could lead to either improving compatibility, or extending the number of known limitations. In general, the problem of determining

`https://github.com/spreadsheetlab/XLParser/issues` whether two context-free grammars are equivalent is undecidable, but in practice several techniques have been successfully used for this purpose [28, 29].

The analysis of the formulas in the dataset revealed that the majority are small and simple, but also that surprisingly large and complex formulas, with more than 50 functions or operations, exist. In terms of functionality, the majority of the formulas include at least one function or operation and almost half of them contain at least one constant. Almost all formulas were found to use data from other cells, which are often to cells of different worksheets and of external files. Various types of references were found: References in the form of cell ranges are used often, in one fourth of the formulas, but unions, intersections, and references to named ranges, to horizontal and vertical ranges and to reference-returning Excel functions are rare. In less than 0.1% of the formulas we found multiple sheet references, complex ranges, structured references or dynamic data exchange links.

REFERENCES

1. Winston W. Executive education opportunities. *OR/MS Today* 2001; **28**(4):8–10.
2. Computer and internet use at work in 2003. *Technical Report*, USA Bureau of Labor Statistics 2005. URL `http://www.bls.gov/news.release/pdf/ciuaw.pdf`.
3. An analysis of computer use across 95 organisations in europe, north america and australasia. *Technical Report*, Wellnomics 2007. URL `http://www.wellnomics.nl/assets/Uploads/WorkPace/News/Wellnomics-white-paper-Comparison-of-Computer-Use-across-different-Countries.pdf`.
4. Scaffidi C, Shaw M, Myers BA. Estimating the numbers of end users and end user programmers. *Proc. of VL/HCC '05*, 2005; 207–214.
5. Hermans F, Jansen B, Roy S, Aivaloglou E, Swidan A, Hoepelman D. Spreadsheets are code: An overview of software engineering approaches applied to spreadsheets. *23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2016.
6. Bell D, Parr M. Spreadsheets: A research agenda. *SIGPLAN Notices* 1993; **28**(9):26–28.
7. Hermans F, Pinzger M, van Deursen A. Supporting professional spreadsheet users by generating leveled dataflow diagrams. *Proc. of ICSE '11*, 2011; 451–460.
8. Shiozawa K, Okada K, Matsushita Y. 3d interactive visualization for inter-cell dependencies of spreadsheets. *Proc. of INFOVIS*, IEEE, 1999; 79–83.
9. Hermans F, Sedee B, Pinzger M, Deursen Av. Data clone detection and visualization in spreadsheets. *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, IEEE Press: Piscataway, NJ, USA, 2013; 292–301. URL `http://dl.acm.org/citation.cfm?id=2486788.2486827`.
10. Dou W, Cheung SC, Gao C, Xu C, Xu L, Wei J. Detecting table clones and smells in spreadsheets. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, ACM: New York, NY, USA, 2016; 787–798, doi:10.1145/2950290.2950359. URL `http://doi.acm.org/10.1145/2950290.2950359`.
11. Hermans F, Pinzger M, van Deursen A. Detecting and visualizing inter-worksheet smells in spreadsheets. *Proc. of ICSE '12*, 2012; 441–451.
12. Cheung SC, Chen W, Liu Y, Xu C. Custodes: Automatic spreadsheet cell clustering and smell detection using strong and weak features. *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, ACM: New York, NY, USA, 2016; 464–475, doi:10.1145/2884781.2884796. URL `http://doi.acm.org/10.1145/2884781.2884796`.
13. Dou W, Xu C, Cheung SC, Wei J. Cacheck: Detecting and repairing cell arrays in spreadsheets. *IEEE Transactions on Software Engineering* March 2017; **43**(3):226–251, doi:10.1109/TSE.2016.2584059.
14. Hermans F, Dig D. Bumblebee: A refactoring environment for spreadsheet formulas. *FSE 2014*, 2014; 747–750.
15. Badame S, Dig D. Refactoring meets spreadsheet formulas. *Proc. of ICSM 2012*, IEEE, 2012; 399–409.
16. Hermans F. Improving spreadsheet test practices. *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, IBM Corp.: Riverton, NJ, USA, 2013; 56–69.
17. Fisher M, Rothermel G. The euses spreadsheet corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. *SIGSOFT Softw. Eng. Notes* May 2005; **30**(4):1–5, doi:10.1145/1082983.1083242.
18. Klimt B, Yang Y. The enron corpus: A new dataset for email classification research. *Machine Learning: ECML 2004*, vol. 3201. Springer Berlin Heidelberg, 2004; 217–226, doi:10.1007/978-3-540-30115-8_22.
19. Barik T, Lubick K, Smith J, Slankas J, Murphy-Hill E. Fuse: A reproducible, extendable, internet-scale corpus of spreadsheets. *Proc. of the 12th Working Conference on Mining Software Repositories (Data Showcase)*, 2015.
20. Hermans F. Excel turing machine. URL `http://www.felienne.com/archives/2974`.
21. Microsoft. Excel (.xlsx) extensions to the office open xml spreadsheetml file format. URL `https://msdn.microsoft.com/en-us/library/dd922181(v=office.12).aspx`.
22. Aivaloglou E, Hoepelman D, Hermans F. A grammar for spreadsheet formulas evaluated on two large datasets. *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, 2015; 121–130.
23. Microsoft. Excel functions (alphabetical). URL `https://support.office.com/en-in/article/Excel-functions-alphabetical-b3944572-255d-4efb-bb96-c6d90033e188`.

24. Van Den Brand M, Sellink M, Verhoef C. Obtaining a cobol grammar from legacy code for reengineering purposes. *Proc. of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications. Springer verlag*, 1997.
25. Zaytsev VV. Recovery, convergence and documentation of languages. PhD Thesis, Vrije Universiteit 2010.
26. Cunha J, Fernandes JP, Mendes J, Hugo Pacheco JS. Towards a Catalog of Spreadsheet Smells. *Proc. of ICCSA'12*, vol. 7336, LNCS, 2012; 202–216.
27. Abreu R, Cunha J, Fernandes JaP, Martins P, Perez A, Saraiva Ja. Smelling faults in spreadsheets. *Proc. of ICSME'14*, IEEE Computer Society, 2014; 111–120.
28. Lämmel R, Zaytsev V. An introduction to grammar convergence. *Integrated formal methods*, Springer, 2009; 246–260.
29. Fischer B, Lämmel R, Zaytsev V. Comparison of context-free grammars based on parsing generated test data. *Software Language Engineering*. Springer, 2012; 324–343.