Tool-assisted Spreadsheet Refactoring and Parsing Spreadsheet Formulas

by

David Hoepelman

MASTER THESIS

22-10-2015

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science
in Computer Science - Software Technology

Faculty Electrical Engineering, Mathematics and Computer Science (EEMCS)
Delft University of Technology

Thesis supervisor:
Felienne Hermans

Preface

Author:	David Jonathan Hoepelman
Student ID:	1521969
Master:	Computer Science
Specialization:	Software technology
Version:	22-10-2015
Defense date:	17-11-2015 16:00
Defense location:	Snijderszaal, Faculty EWI room LB01.010, Mekelweg 4, Delft

Abstract

Spreadsheets have a life-cycle similar to that of other software: they are inherited throughout an organization, are maintained by different users, and evolve over time to meet changing requirements. This leads to increased complexity and technical debt. In software engineering, refactoring is used to combat these problems by improving software structure without altering behavior. This technique can also be applied to spreadsheets.

In this thesis we present an improved version of the spreadsheet refactoring tool BumbleBee, extended with six refactorings: EXTRACT FORMULA, INLINE FORMULA, INTRODUCE CELL NAME, GROUP REFERENCES, INTRODUCE AGGREGATE and INTRODUCE CONDITIONAL AGGREGATE. The INLINE FORMULA, GROUP REFERENCES and INTRODUCE CONDITIONAL AGGREGATE refactorings were not implemented before and EXTRACT FORMULA and INTRODUCE CELL NAME improve upon previous implementations. To support these refactorings and facilitate future spreadsheet research the formula parser used needed improvements. We implemented these improvements and released the result as the open-source software package XLParser, a stand-alone C# parser for spreadsheet formulas. XLParser was evaluated on more than a million unique formulas from industrial datasets, and successfully parsed 99.999‰.

Thesis committee:

Chair:	Prof. Dr. A. van Deursen	Faculty EEMCS, Delft University of Technology
Committee Member:	Dr. Ir. F.F.J. Hermans	Faculty EEMCS, Delft University of Technology
Committee Member:	Dr. C. Hauff	Faculty EEMCS, Delft University of Technology

Contents

Chapter 1

Introduction

Like all people, I sometimes get asked what I do for a living. When I tell someone I am writing my master thesis in Computer Science, their eyes start to glaze over as they anticipate some explanation peppered with terms they will not understand about. I then tell them my thesis is about spreadsheets and ask if they have ever worked with Excel, and nearly everyone who has ever worked in business or research has. Nearly everyone has a horror story about that one unmaintainable spreadsheet that they had to work on, or that day their reporting system broke down because 2009 turned into 2010 and the spreadsheet only looked at the last digit.

This anecdotal evidence is mirrored in research. Panko [1] estimates that 80% to 95% of businesses use spreadsheets in one of their processes. Furthermore, almost all spreadsheets contain at least one error, and 1 to 5% of spreadsheet cells contains an error according to Panko [2]. Spreadsheets perform roles very similar to software in that they perform business-critical roles, are inherited throughout the organization and maintained by different users and accrue technical dept during and after the initial development period [2]. In short, spreadsheets can be classified as programs, and spreadsheet creators as end-user programmers.

This view, "spreadsheets are code", could be the one-sentence summary of the ideology of the Spreadsheet Lab, which is a part of the TU Delft Software Engineering Research Group (SERG). Using this view as a baseline, the group works on translating tried and proven software engineering methods to the spreadsheet domain so that they can be used to improve spreadsheets, spreadsheet development practices and help spreadsheet programmers. As part of this effort, a spreadsheet formula refactoring tool called BumbleBee was developed by Hermans and Dig [3]. This tool allows a formula to be transformed into another by defining a transformation rule, which works very similar to a pattern or regular expression replacement in a text editor.

However this approach has the downside that it can only consider one formula, and not the spreadsheet as a whole. This leads to a lack of power to implement all spreadsheet refactorings, such as those implemented by Badame and Dig [4] in earlier work. I joined the group to extend the capabilities of BumbleBee so that it could take context into account when performing refactorings, and implement more refactorings in BumbleBee.

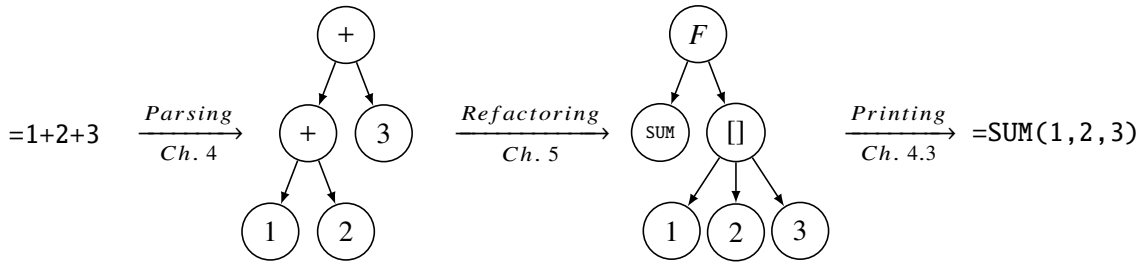


Figure 1.1: Overview of the refactoring process

After the initial literature research, I started implementing refactorings, but encountered a fundamental problem in doing so. The standard way of implementing refactorings, illustrated in Figure 1.1, is by parsing the source code to an Abstract Syntax Tree (AST), which represents the structure of the program. This AST can then be manipulated into the desired form, after which it can be converted back to source code (this is called printing or pretty-printing). While BumbleBee contained a home-grown parser, I found a range of formulas that were either not parsable, or parsed into an incorrect AST. This made me refocus the purpose of my thesis into making a better parser for Excel formulas, as this would not only be very useful for implementing refactorings but would be beneficial to all future spreadsheet research projects. Using this new parser, I implemented several refactorings, which are described in Chapter 5.

1.1 About this thesis

1.1.1 Contribution

The contributions of this thesis are twofold. Firstly I improved and open-sourced a stand-alone formula parser called XLParser, which is available online[1]. The parser was tested on over a million formulas and failed to parse merely two formulas. Details of this parser are published by Aivaloglou, Hoepelman and Hermans [5]. This paper is partially re-used in this thesis report.

The second contribution is an improved version of BumbleBee, available online [2], which implements the refactorings described in Chapter 5: EXTRACT FORMULA, INLINE FORMULA, INTRODUCE CELL NAME, GROUP REFERENCES, INTRODUCE AGGREGATE and INTRODUCE CONDITIONAL AGGREGATE.

Of these refactorings, INLINE FORMULA, GROUP REFERENCES and INTRODUCE CONDITIONAL AGGREGATE where not implemented in any previous work known to us, and EXTRACT FORMULA and INTRODUCE AGGREGATE offer improvements over previous implementations [4].

[1] https://github.com/spreadsheetlab/XLParser
[2] http://spreadsheetlab.org/2015/10/12/bumblebee-an-excel-refactoring-add-in/

1.1.2 Attribution

This thesis was performed at the TU Delft Spreadsheet Lab, and is partially based on a collaborative effort in that group. During this thesis the Excel Formula parser XLParser was developed, and a paper describing it was accepted into the 15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2015): *A Grammar for Spreadsheet Formulas Evaluated on Two Large Datasets* by Efthimia Aivaloglou, David Hoepelman and Felienne Hermans [5]. This paper is attached verbatim as published in appendix B. Chapter 3 is based on Section II, and was primarily written by the thesis author. Chapter 4 is an updated and extended version of Section III, and was primarily written by the thesis author with contributions from Efthimia Aivaloglou, except for Chapter 6, which incorporates parts of Section IV of the paper which was primarily written by Efthimia Aivaloglou, who also performed the evaluation.

The grammar implementation (XLParser) was primarily done by the thesis author, but the implementation is based on the previous (un-named) parser which was primarily developed by Efthimia Aivaloglou and Felienne Hermans. The spreadsheet scantool used to evaluate XLParser, which extracts formulas from spreadsheet files, was developed as part of Infotron B.V.[3], with many authors. The thesis author did not contribute to this tool.

The refactorings described in Chapter 5 were added to the existing BumbleBee Excel add-in developed by Felienne Hermans, but these refactorings were solely implemented by the author with very little of the existing infrastructure used.

1.1.3 Outline

Chapter 2 details previous and related work on spreadsheet refactoring. A passing knowledge of Excel and more in-depth knowledge of Excel formulas is needed to read the rest of this thesis, which is bundled in Chapter 3.

Chapter 4 covers how XLParser parses spreadsheet formulas and why it was designed as it was. Chapter 5 describes which spreadsheet refactorings were implemented and how this was done. Chapter 6 provides an overview of the evaluation done and Chapter 7 contains concluding remarks.

[3] `http://www.infotron.nl/`

1.2 Timeline and decisions taken

December 2014	Literature study on refactoring, refactoring spreadsheets, converting spreadsheets to programs.
	Thesis topic selection.
January 2015	Studied practicality of generic spreadsheet refactoring language based on BumbleBee transformation language, deemed inviable.
	Gathered existing refactorings from spreadsheet literature and translated Fowler refactorings.
	Decided which refactorings to initially implement
	Familiarization with existing BumbleBee code
February 2015	Implementing INLINE FORMULA
	Extended BumbleBee and parser to account for sheet and file names
March 2015	Implementing EXTRACT FORMULA
	Writing of paper "End user programming" [4]
April 2015	Writing of paper "End user programming"
	Various improvements to parser
May 2015	Decision to rewrite parser to solve several fundamental problems
	Start work on XLParser
	Implementing INTRODUCE (CONDITIONAL) AGGREGATE
	Implementing GROUP REFERENCES
June 2015	Continued work on XLParser: Initial release
	Writing of XLParser paper [5]
July 2015	Changing of refactoring UI to context-aware right-click menu, similar to IDE's
August 2015	Continued work on XLParser: Several fixes to XLParser parse trees
	Camera-ready adjustment of XLParser paper
	Constructed demo application[5] for XLParser which shows the parse trees.
September 2015	Continued work on XLParser: Adding structured references, file paths
	Porting BumbleBee and refactorings to XLParser-based implementations
	Writing of this thesis
October 2015	Implementing INTRODUCE CELL NAME
	Porting BumbleBee and refactorings to XLParser-based implementations.
	Writing of this thesis

[4] First version was rejected from IEEE special issue on Refactoring: Accelerating Software Change. A significantly extended and improved version was submitted to ICSE later, but I did not contribute to these changes.

[5] `http://xlparser.perfectxl.nl/demo/`

Chapter 2

Previous and related work

2.1 Refactoring

Refactoring is "the process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure" [6]. The process is probably as old as programming itself, and was known since at least 1986 as "Restructuring" [7]. The term "Refactoring" was coined by Opdyke in 1992 [6] and originally specifically meant the restructuring of Object Oriented Programs, but nowadays the term is also used in other paradigms. Over time the popularity of both the practice and the term "Refactoring" increased, greatly helped by Fowler's 1999 book "Refactoring: improving the design of existing code" [8], which has become the canonical reference. Currently, all major programming Integrated Development Environments like Visual Studio, Eclipse, Netbeans and IntelliJ include support for automated code refactorings.

2.2 Spreadsheet smells

Fowler [8] also introduced the concept of "code smells": allegorical bad smells in code which are signs that the code has design problems and could benefit from refactoring. The concept of code smells has been translated to the spreadsheet domain by both Cunha et. al [9] and Hermans et. al [10–12]. Of particular relevance are the MULTIPLE REFERENCES, MULTIPLE OPERATIONS, LONG CALCULATION CHAIN and DUPLICATED FORMULAS "formula smells" defined by Hermans et. al [11, 12] because these can be solved with the refactorings presented in this thesis.

2.3 Spreadsheet models

The tool presented in this thesis, BumbleBee, aims to empower spreadsheet users from within their familiar environment, Microsoft Excel, to improve their spreadsheets by giving them additional capabilities.

An alternative approach is to restrict the options spreadsheet users have, resulting in safer spreadsheets similar to how schemas and foreign key constraints in databases and static typing in general purpose programming languages prevent certain categories of errors. This can be done by defining a high level model representing the data and generating a type-safe spreadsheet from this model, effectively using spreadsheets as the target environment for a compiler. The way this model is constructed differs, and has been done in general purpose programming languages [13, 14], templates [15–20] and relational database schemas [21]. Models can also be inferred from existing spreadsheets [22, 23].

2.4 Refactorings provided by excel

Several useful refactorings are already provided by Excel, although Excel does not call them refactorings.

The best example of this is the cut, copy and paste functionality of Excel. If a user cuts a selection of cells and pastes it elsewhere, all references to those cells made in other cells will be moved as well. For example if a user cut-pastes cell `A1` to location `C3`, the formula `=A1` will be changed to `=C3` in other cells, even though they were not selected by the user. This is very similar to the MOVE METHOD [8] refactoring, because not only are the cells (method contents) themselves moved, references to them (call sites) are adjusted for the new location as well. Another option Excel provides when copy-pasting is "Paste Values", with which a formula is replaced by its evaluated value and remains constant from that point on.

The fact that Excel has built-in support for these refactorings shows that there is a need among spreadsheet users to change their spreadsheets without altering functionality, to refactor them, and that spreadsheet users are likely already comfortable with the concept of refactoring, albeit it not by name.

2.5 Spreadsheet refactoring

In addition to Excel, two commercial Add-Ins are known to the author that implement refactorings. The Power Utility Pak [24] contains a UNAPPLY NAMES refactoring, which changes a named range to its location, e.g. `=TAX_RATE*100` to `=$A$1*100`. This is the exact inverse of the INTRODUCE CELL NAME refactoring defined in Section 5.3. It also contains a "Error Condition Wizard", which is very similar in purpose to the GUARD CALL refactoring defined by Badame and Dig [4]. The ASAP Utilities for Excel [25] Add-In contains a "Change formula reference style" function, which implements the MAKE CELL CONSTANT Refactoring described by Badame and Dig [4], and also contains its own version of UNAPPLY NAMES. The algorithms used and popularity of these Add-Ins is unknown, but their existence is additional indication that advanced spreadsheets users recognize the usefulness of refactoring spreadsheets.

The first Excel Add-In specifically developed for refactoring, called RefBook, was presented by Badame and Dig [4]. In this tool, seven refactorings are presented, four of which have been re-implemented and improved in this thesis in Chapter 5.

Hermans et. al [12] present four refactorings designed to each solve a specific spreadsheet smell, which is an indicator that something might be wrong with the spreadsheet. EXTRACT (COMMON) SUBFORMULA is very similar to Refbook's EXTRACT ROW OR COLUMN, and is implemented in this thesis as EXTRACT FORMULA. The proposed GROUP REFERENCES refactoring is implemented in this thesis, as is the proposed MERGE FORMULAS refactoring under the name INLINE FORMULA.

Hermans and Dig [3] developed a refactoring tool called "BumbleBee" for Excel which operates using a different principle. Instead of implementing a fixed set of refactorings, it uses "transformation rules" which consist of two formulas, with extended syntax introducing placeholders for certain constructs such as formulas, cells or ranges. For example the transformation rule =IF(F_1>F_2, F_1, F_2) ↔ MIN(F_1,F_2) indicates that these two formulas are equivalent to each other and can be transformed into each other. However this approach is contained within one formula and can thus only be used to implement intra-formula refactorings: refactorings which both only affect and require information from a single formula. This thesis was started as an attempt to introduce BumbleBee with inter-formula refactorings, refactorings which affect or require information from more than a single formula.

2.6 Parsing Spreadsheet Formulas

Most spreadsheet tools process formulas in one way or another. However, all of them have one or more reasons which make them unsuitable for our purposes, which was the motivation to improve the previously proprietary parser used by BumbleBee and release this as open-source, see Chapter 4.

Parser is proprietary

The most obvious case is the Microsoft Excel formula parser itself, which is not available for usage by external programs or Add-Ins. Several research projects process formulas, but do not make available their parser or grammar used, which is the case with Baryowy's et. al CheckCell [26] and all of the work by Cunha et. al [9, 23, 27].

Parser is not advanced enough for refactoring purposes

Several grammars are available online [28, 29], and a grammar is published as part of RefBook [4], but all were found to contain errors, especially in the areas of operator precedence, reference expressions or references to sheets or files. These errors were deemed crucial to solve by the thesis author, as making errors in parsing has a high chance of resulting in errors, and thus violating the user expectation that a refactoring will not introduce errors.

Parser is not stand-alone

Several open-source programs [30–32] can process Excel formulas in one way or another. However, these parsers are deeply tied into the product, which makes using them from different program difficult. One could re-implement the grammar used by these parsers, but the grammars are not available separatly, thus requiring the grammar to first be extracted before it can be re-implemented. This has not been attempted, but would likely result in grammar suffering from the same deficiencies as the official grammar provided by Microsoft, which is unsuitable for reasons described in Section .

While it would have been possible to decouple the parser from an existing product, this likely would have been more difficult and produced worse results than improving the existing parser, which was done for this thesis.

Chapter 3

Anatomy of spreadsheets and spreadsheet formulas

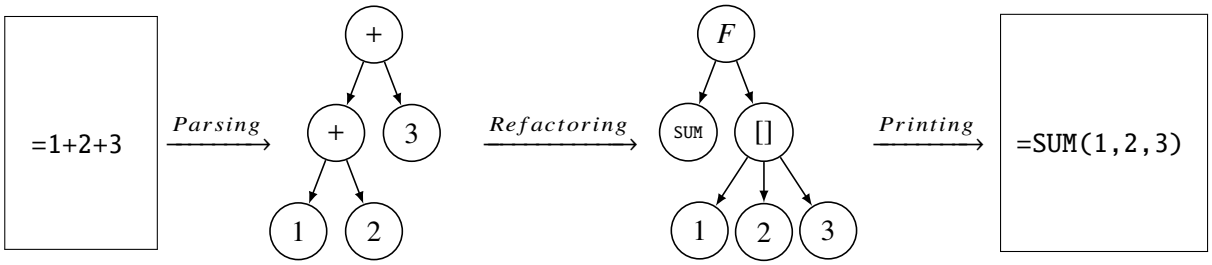
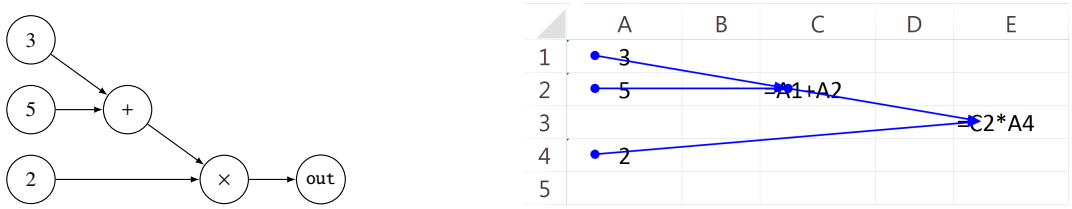


Figure 3.1: Overview of the refactoring process

In order to be able to create a parser or refactor-tool for spreadsheets, a solid understanding of their inner workings is necessary, especially of the formula language. This chapter details the properties and syntax of spreadsheet and formula strings, shown boxed in Figure 3.1.

By a large margin, the most widely used spreadsheet system is Microsoft Office Excel, which has a self-reported install base of 1.2 billion users [33]. Two less-popular but still common implementations are Apache OpenOffice Calc and LibreOffice Calc, which evolved from the same product and thus have largely identical semantics. Apache OpenOffice has a self-reported *total* download count of 150 million [34], while LibreOffice has a self-reported download count of 120 million *from unqiue IP adresses* [35], both excluding external download locations like Linux distribution repositories. Google Sheets is another common implementation, and is special because it was the first widely used web-based collaborative spreadsheet program. Google does not report usage numbers, but in 2010 has said it is used by "tens of millions" of users [36]. There are many other spreadsheet implementations, but none of them come close to the user base of the above programs.

When referencing "all" spreadsheet programs, this indicates the three implementations previously mentioned: Microsoft Excel, LibreOffice and Google Sheets. These three implementations are the systems studied because of their market share.



(a) Dataflow program (b) Spreadsheet implementation

Figure 3.2: An example dataflow program and its spreadsheet implementation

All mainstream implementations use the following model:

- 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** (table) of **cells**.
- A vertical unit in the grid is called a **column** and a horizontal unit a **row**. 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. A column or row can also mean all cells contained in that column or row.
- A **cell** can contain a **constant value** of any type, a calculation called a **formula** or a matrix calculation called an **array formula**.
- An (array) formula can **reference** other cells to use their values. When the value of a referenced cell changes, this new value is propagated and the dependent formula values are recalculated.

This model is a variation of the dataflow programming model. A dataflow program is a directed graph, where data flows between operation in nodes along the graphs edges. In spreadsheets, cells represent the nodes of a dataflow program and edges are represented by references. An example dataflow program and its spreadsheet implementation can be seen in Figure 3.2.

The spreadsheet model is Turing complete, as proven by an Excel 2010 implementation of a Turing machine [37].

3.1 Formulas

All spreadsheet programs currently use dialects of the same formula language. Two of these dialects have been standardized: Office Open XML spreadsheet language is a standardization of the Excel language and OASIS OpenFormula, which is part of the OpenDocument standard and aims to provide a specification for all dialects. Both dialects are very similar and this section covers both unless otherwise noted.

Formulas consist of an expression which can contain constant values, function calls and operators and, most importantly, references to other cells. A cell is identified as a formula cell because all formulas must start with the equals sign =.

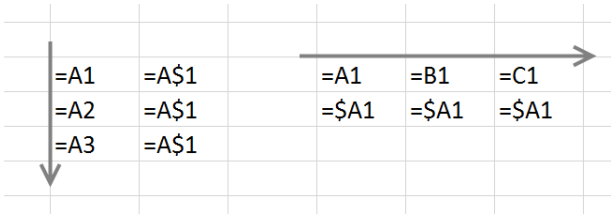


Figure 3.3: Different copy-paste behavior depending on $ modifier, copy-direction is given by the arrow

3.1.1 Function calls and operators

Function calls are performed, similar to other programming languages, by starting with the function name, followed by the arguments in parentheses, separated by a comma. All spreadsheet implementations provide a range of built-in functions, and in most spreadsheet implementations it is possible to define new functions yourself. However in current implementations this is not done directly inside the spreadsheet, instead using an alternate programming language. In Microsoft Excel and LibreOffice this is done with a variant of the BASIC programming language, while in Google Sheets this is done with Javascript. A way for the user to define functions in the spreadsheet itself has been proposed by Peyton Jones et. al [38], but as of now has not been implemented in mainstream spreadsheet programs yet.

The binary operators + - * / = >= <= < > and <> (inequality) can be used according to their usual semantics. + and - are available both unary (=-1) and binary (=1-1). Additionally the % postfix unary operator is defined to transform a number into a percentage (divining it by 100), ^ is the exponentiation operator and & is the text concatenation operator.

Spreadsheet programs contain three fairly unique binary operators, the semantics of which are detailed in Subsection 3.1.2. Firstly there is the range operator : then the union operator (, in Excel and ~ in OpenFormula) and lastly the intersection operator (␣ in Excel and ! in OpenFormula). Note that OpenFormula diverges from the Excel syntax, possibly because the comma is already used in other places in the language and a single space as an operator is highly unusual. The Excel characters for the operators will be used in the remainder of this thesis.

3.1.2 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`. This is called A1-style referencing and is by far the most common in modern spreadsheet implementations. 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 when copied, as ilustrated in Figure 3.3.

	A	B	C	D
1	Year	Revenue	Expenses	Profit
2	2010	1067000	1015000	=[@Revenue]-[@Expenses]
3	2011	1079000	985000	=[@Revenue]-[@Expenses]
4	2012	1071000	1008000	=[@Revenue]-[@Expenses]
5	2013	1050000	1015000	=[@Revenue]-[@Expenses]
6	2014	1140000	970000	=[@Revenue]-[@Expenses]
7				
8				
9			Total:	=SUM(Budget[Revenue])-SUM(Budget[Expenses])
10				

Figure 3.4: Example usage of structured references

Ranges

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 space). 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. For example one can give the name `TAX_RATE` to cell `A2` and then use this in a formula: `=C3+C3*TAX_RATE` instead of `=C3+C3*$A$2`.

Structured References

A recent addition to the Excel formula language introduced in Excel 2007 are structured (table) references. To use this feature, a table must be given a name and column headers. One can then reference a column in the table by entering `TableName[ColumnName]`. Inside the square brackets reference operators can be used to construct more complex references, `TableName[Column1,Column4]` references two columns.

There is no way to reference a specific row, except the current row, for example if a formula is placed in `A3` it can only reference row number 3. The `#This Row` keyword and the `@` operator are used for this: `TableName[#This Row]` and `TableName[@]` both reference the current row number in the provided table, and `TableName[@ColumnName]` references the cell in the provided column of the current row number.

This feature is meant to make formulas easier to read, by replacing references with human readable names as can be seen in Figure 3.4: the formula `=SUM(B2:B6) - SUM(C2:C6)` can instead be written as `=SUM(Budget[Revenue]) - SUM(Budget[Expenses])`.

	A	B
1	1	=A1+1
2	2	=A2+1
3	3	=A3+1

(a) A1-style formulas

	1	2
1	1	=RC[-1]+1
2	2	=RC[-1]+1
3	3	=RC[-1]+1

(b) R1C1-style formulas

Figure 3.5: A1 vs R1C1 style on identical formulas with respect to cell position

R1C1 reference style

An alternate style called R1C1 as opposed to the above A1 style exists, but it is only rarely used by users in modern spreadsheet implementations. In R1C1 reference style 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, illustrated in Figure 3.5. These properties make R1C1 useful as an internal representation in spreadsheet implementations and in a spreadsheet refactoring tool.

Non-local references

References refer to cells or ranges in the same sheet as the formula by default, but this can be modified with a prefix. A non-local reference consists of a prefix indicating the location, followed by an exclamation mark, followed by the actual reference.

The simplest case is a reference to another sheet in the same workbook, where the prefix is simply the sheet name: `=Sheetname!A1`. Sheet names can also be between single quotes if they contain special characters: `='Sheetname with space'!A1`. References to external spreadsheet files are also possible, which is done by providing the file name in between square brackets and optionally the file path: `=[Filename]Sheetname!A1` or `='C:\Path\[Filename]Sheet'!A1`. A peculiar type of prefix are those that indicate multiple sheets: `=Sheet1:Sheet10!A1` means A1 in Sheet1 through Sheet10.

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`.

3.1.3 Case sensitivity

Formulas are case-insensitive outside of the trivial case of string literals. Identifiers have a canonical capitalization, and while a user can type the identifier with any casing only the canonical form will be displayed. While the canonical capitalization of built-in identifiers, functions and reserved names is mostly uppercase, the canonical capitalization of user defined identifiers and named ranges, is as the user defined them originally.

3.1.4 Whitespace sensitivity

The Excel formula language is whitespace sensitive in several places:

- Whitespace is not allowed between function names and the argument list: `=SUM (1)` is invalid.
- Whitespace is not allowed inside internal or external references: `=Sheet1 !A1` is invalid.
- The intersection operator is a single space: `=A:A 3:3` is the intersection of column `A` and row `3`, equivalent to `=A3` (Excel formula language only).

3.2 Array Formulas and Arrays

In spreadsheet formulas it is possible to transform one- or two-dimensional matrices.

When constructed from constant values they are called *array constants*, e.g. `{1,2;3,4}` constructs a two-by-two matrix. 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 Sheets 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 reference ranges to be treated as matrices, and several matrix operators and functions will be available. 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.

3.3 Type system

The formula language uses a weak type system, because most types can be coerced into others. T he following types exist:

Boolean values are either `TRUE` or `FALSE`. Booleans can be coerced to string and numbers, where `TRUE` will become `"True"` and `1` and `FALSE` will become `"False"` and `0`.

Numeric values are in the range of 8-byte IEEE doubles. Numbers can be provided as integers, decimals or in scientific notation. When coerced to booleans 0 will become `FALSE`, all other values will be `TRUE`. Numbers can also be coerced into strings, or type-casted with the `TEXT` function.

String values are any Unicode character enclosed in quotation marks ". Two quotation marks serve as the escape character, thus `""""` represent the string ". If the contents of a cell start with a ' the rest of that cell content is interpreted as a string.

When coerced to booleans all strings except the empty string are `TRUE`, the empty string is `FALSE`. When coerced to a numeric value the spreadsheet program will accept any string representing valid numeric user input and otherwise give the error `#VALUE!`. Explicit conversion to a numeric value is done with the `VALUE` function.

Error values are `#DIV/0!`, `#NAME?`, `#NULL!`, `#NUM!`, `#N/A!`, `#VALUE!` and `#REF!`. Errors behave similar to exceptions in that they will propagate throughout a calculation. Errors cannot be coerced.

Ranges and arrays are one- or two-dimensional matrices of any non-array values. Arrays are rarely used outside of array formulas, but ranges are very common in formulas. However, these types usually only serve as inputs for functions and are thus fairly transparent to the user outside of array formulas. Both types usually cannot be coerced, doing so will result in the `#VALUE!` error.

Some other "display types" exists, these can change the way the data is presented to or validated from the user and can have implications when inter-operating with other programs. Usually the user can mark a cell as containing one of these types, or Excel can automatically mark a cell to be of this type based on heuristics. In formulas and internally these types are all represented by one of the above types. A few of these are commonly used:

Dates and times are internally stored as a floating point with the integer portion being the number of days since the epoch January 1st 1900, incorrectly considering 1900 a leap year, and the remainder being the portion of the day that passed. Excel displays dates and times as is customary in the locale of the user. When interoperating a date or time value will be exported as a datetime type value of that system.

Currency is stored as other numbers and displayed in the format customary for the specific currency. When interoperating with some other systems currency values will be exported using arbitrary-precision arithmetic formats.

Percentages are stored as other numbers, but displayed as if multiplied by 100%.

Chapter 4

Parsing spreadsheet formulas

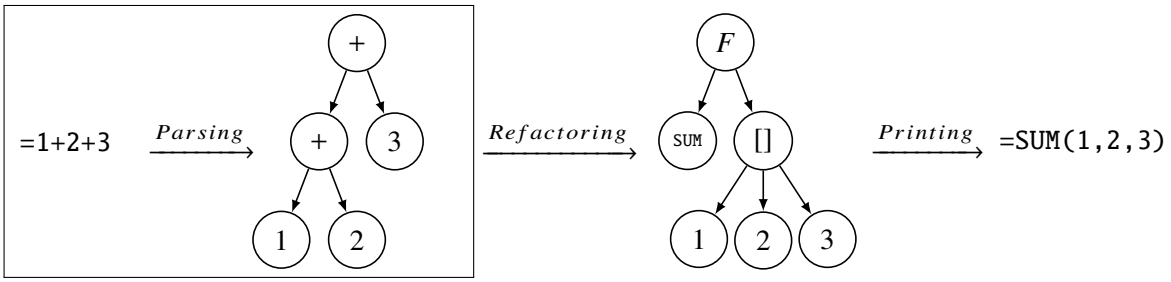


Figure 4.1: Overview of the refactoring process

This chapter details parsing, converting a string formula to an AST, shown boxed in Figure 4.3. It assumes the reader is familiar with basic parser theory, a good overview of which can be found in [39].

4.1 Motivation

In order to implement refactorings of spreadsheets, a refactoring tool must be able to manipulate spreadsheets. The usual way to implement refactorings is by manipulating the original program's AST until it represents the desired program, and then print that back to a string. Excel exposes an API to retrieve and update the contents of a spreadsheet file, but this API only provides access to formula strings, and does not expose its parser or formula ASTs representations. Because of this, a refactoring tool needs to contain a parser for Excel formulas.

BumbleBee relied on a parser developed over the years for previous research, however this parser had accrued technical debt due to new rules being added over time, sometimes in an inconsistent manner or not supporting the whole language. Furthermore the parser interpreted some language constructs wrong and missed several features, which made implementing refactorings hard and error-prone. For example, operator precedence was not taken into account, causing the formula `=A1 + A2 * A3` to be parsed as `=(A1 + A2) * A3`. The existing parser thus was insufficient to correctly implement the refactorings presented in this thesis. Thus an additional goal for this thesis became to create a better parser.

For BumbleBee and other research on spreadsheet formulas the following design goals were formulated for the parser and grammar within the Spreadsheet lab group:

1. The parser must be compatible with the official language
2. Produced parse trees must be suited for further manipulation and analysis with minimal post-processing required
3. The grammar must be compact enough to feasibly implement with a parser generator

While an official grammar for Excel formulas is published [40], 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.

Because there is no suitable parser and grammar available that satisfy the above requirements, we decided to clean up and partially rewrite the parser. The end result of this effort is an independent, open-source parser for Excel formulas called XLParser[1], about which a paper was published in IEEE conference SCAM 2015 [5].

4.2 Parser implementation

The existing parsing was built using the Irony parser framework[2], which is a C# parser generator that produces parsers based on the LALR(1) algorithm using a grammar defined in C#.

Strictly speaking Irony produces a parse tree, however this tree is fairly high-level for a parse tree, leaving out elements such as punctuation and whitespace, and no separate AST is (currently) constructed in XLParser or BumbleBee, instead this tree is directly manipulated. To avoid confusion and use usual nomenclature we will from now on refer to this tree as the AST.

4.2.1 Lexical Analysis

Table 4.1 contains the lexical tokens of the grammar, along with their identification patterns in a simple regular expression language. All tokens are case-insensitive. Characters are defined as Unicode code points x9 (tab), xA (newline), xD (carriage return) and x20 (space) and upwards.

This grammar requires the parser to support token priorities, which Irony does. 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 (e.g. '%', '!') are directly defined in the production rules in Figure 4.2 in between quotes for readability and compactness.

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

Token Name	Description	Contents	Priority
BOOL	Boolean literal	TRUE | FALSE	0
CELL	Cell reference	$? [A-Z]+ $? [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	(Any entry from the function list[3]) \(	5
FILE	External file reference using number	\[[0-9]+ \]	5
FILENAME	External file reference using name	\[$\square_4$+ \]	-1
FILEPATH	Windows file path	[A-Z] : \\ ($\square_4$+ \\)*	0
HORIZONTAL-RANGE	Range of rows	$? [0-9]+ : $? [0-9]+	0
MULTIPLE-SHEETS	Multiple sheet references	(($\square_2$+ : $\square_2$+)(' ($\square_3$ | '')+ : ($\square_3$ | '')+ ')) !	1
NAME	User Defined Name	[A-Z_\\][A-Z0-9_.$\square_1$]*	-2
NAME-PREFIXED	User defined name which starts with a string that could be another token	(TRUE | FALSE | [A-Z]+[0-9]+) [A-Z0-9_.$\square_1$]+	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 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	Quoted worksheet name	$\square_3$+ ' !	5
STRING	String literal	" ([^ "] | "")* "	0
SR-COLUMN	Structured reference column	\[[A-Z0-9_.$\square_1$]+ \]	-3
UDF	User Defined Function	(_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 ' * [ ] \ : / ?	
$\square_4$	Filename characters	Any character except " * [ ] \ : / ? < > |	

[3] A function list is available as part of the reference implementation.
Lists provided by Microsoft are also available in [41] and [40].

Table 4.1: Lexical tokens used in the XLParser grammar

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[5] and the fractional portion representing the portion of the day passed.

When extracting formulas from spreadsheets, only the floating point value is available. The parser will thus never encounter the formatted notation of the date. For this reason, the grammar only parses numeric dates and times and these are not distinguishable from other numbers.

[5] Note that 1900 is incorrectly considered a leap year, due to a bug in Lotus 1-2-3 (first released in 1983) which was deliberately copied into the first Excel release and has since then been preserved for backwards compatibility reasons.

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

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

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

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

⟨*UnOpPrefix*⟩ ::= '+' | '-'

⟨*BinOp*⟩ ::= '+' | '-' | '*' | '/' | '^'
| '<' | '>' | '=' | '<=' | '>=' | '<>'

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

⟨*Argument*⟩ ::= ⟨*Formula*⟩ | ϵ

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

⟨*RefFunctionCall*⟩ ::= '(' ⟨*Union*⟩ ')'
| ⟨*RefFunctionName*⟩ ⟨*Arguments*⟩ ')'
| ⟨*Reference*⟩ ':' ⟨*Reference*⟩
| ⟨*Reference*⟩ '␣' ⟨*Reference*⟩

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

⟨*NamedRange*⟩ ::= ⟨*Name*⟩

⟨*Name*⟩ ::= NAME | NAME-PREFIXED

⟨*File*⟩ ::= FILE
| FILENAME
| FILEPATH FILENAME

⟨*Prefix*⟩ ::= SHEET
| ''' SHEET-QUOTED
| ⟨*File*⟩ SHEET
| ''' ⟨*File*⟩ SHEET-QUOTED
| FILE '!'
| MULTIPLE-SHEETS
| ⟨*File*⟩ MULTIPLE-SHEETS

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

⟨*Union*⟩ ::= ⟨*Reference*⟩ { ',' ⟨*Reference*⟩ }

⟨*ConstantArray*⟩ ::= '{' ⟨*ArrColumns*⟩ '}'

⟨*ArrColumns*⟩ ::= ⟨*ArrRows*⟩ { ';' ⟨*ArrRows*⟩ }

⟨*ArrRows*⟩ ::= ⟨*ArrConst*⟩ { ',' ⟨*ArrConst*⟩ }

⟨*ArrConst*⟩ ::= ⟨*Constant*⟩
| ⟨*UnOpPrefix*⟩ NUMBER
| ERROR-REF

⟨*StructuredReference*⟩ ::= ⟨*SRCol*⟩
| '[' ⟨*SRExpr*⟩ ']'
| ⟨*Name*⟩ ⟨*SRCol*⟩
| ⟨*Name*⟩ '[' ⟨*SRExpr*⟩ ']'

⟨*SRExpr*⟩ ::= ⟨*SRCol*⟩
| ⟨*SRCol*⟩ ':' ⟨*SRCol*⟩
| ⟨*SRCol*⟩ ',' ⟨*SRCol*⟩
| ⟨*SRCol*⟩ ',' ⟨*SRCol*⟩ ':' ⟨*SRCol*⟩
| ⟨*SRCol*⟩ ',' ⟨*SRCol*⟩ ',' ⟨*SRCol*⟩
| ⟨*SRCol*⟩ ',' ⟨*SRCol*⟩ ',' ⟨*SRCol*⟩ ':' ⟨*SRCol*⟩

⟨*SRCol*⟩ ::= FILENAME
| '[' ⟨*Name*⟩ ']'
| '[' SR-COLUMN ']'

Figure 4.2: Production rules used in the XLParser grammar

4.2.2 Syntactical Analysis

The complete production rules of the grammar are listed in Extended BNF syntax in Figure 4.2. Patterns between { and } can be repeated zero or more times. The start symbol is *Start*.

The ⟨*Formula*⟩ rule covers all the expressions which can be used in spreadsheet formulas: constants (`=5`), references (`=A3`), function calls and operators (`=SUM(A1,A2)`), array constants (`={1,2;3,4}` and reserved names (`=_xlnm.Print_Area`). The ⟨*Reference*⟩ rule covers a subset of expressions known as references expressions, expressions which can return a reference. These can be internal or external cell and range references, functions and operators which can return references, named ranges, structured references and dynamic data exchanges.

Precedence	Operator(s)	Description
1	= < > <= >= <>	Logical comparison
2	&	Text concatenation
3	+ - (binary)	Addition and subtraction
4	∗ /	Multiplication and division
5	^	Exponentiation
6	%	Division by 100
7	+ - (unary)	No effect and inverting number sign
8	,	Range union
9	␣	Range intersection
10	:	Range construction

Table 4.2: Operator precedence in formulas, with larger numbers indicating higher precedence

4.2.3 Precedence and ambiguity

The production rules are ambiguous, which means they cannot be directly used in a parser generator based on the LALR(1) algorithm like Irony.

To resolve ambiguity with operators, e.g. whether to parse `=1+2*3` as `=(1+2)*3` or `=1+(2*3)`, operator precedence and associativity rules are defined. These can be found in Table 4.2.

However, even with precedence and associativity rules the grammar is still not fully un-ambigious. This is due to trade-offs on parsing references (see Section 4.4.1) and parsing unions (see Section 4.4.2). 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, which Irony produces, this 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 is a correct interpretation and thus results in correct ASTs.

[5] This is contrary to most other languages, where the exponentiation operator is right-associative.
In Excel `2^1^2` will be $(2^1)^2 = 4$, while in most other languages it will be $2^{1^2} = 2$

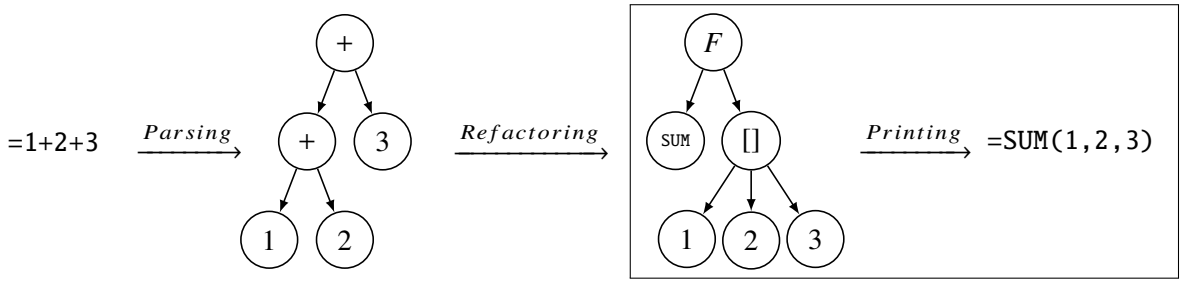


Figure 4.3: Overview of the refactoring process

4.3 Pretty-printing the formula AST

Pretty-printing a formula AST is the reverse operation of parsing, and must be done to convert a refactored AST back to a string as seen in Figure 4.3.

Pretty-printing is quite straightforward: it can be done by describing for each tree node type how it can be translated back into a string, most of the time this is the exact reverse of the parser production rule. Printing is done by starting at the root of the tree and calling the print function recursively for each child, because nodes with children need to know the printed form of their children. A slightly simplified and compacted version of the XLParser code responsible for printing can be found in Appendix A.1.

4.4 Trade-offs

The grammar presented in this chapter contains some trade-offs, partly due to the Excel language itself, partly due to design decisions. These are detailed in this section.

4.4.1 References

References play an important role in the spreadsheet paradigm and therefore in the formula language. Particularly reference expressions, expressions which evaluate to a reference, are a subset of expressions and several operators and functions only accept reference expressions. For example the formula `=SUM(IF(...):A1)` is valid, while `=SUM((1+1):A1)` is not, because `IF` can return a reference while `+` cannot and the `:` operator only operates on references.

This is not unique to reference expressions, for example the operator `+` only operates on numeric values, making the expression `="a"+1` invalid. What does make reference expressions special is how Excel treats them. A formula which uses a non-reference expression where a reference expression is required, like the previously mentioned `=SUM((1+1):A1)`, will result in a parse error which means excel will not accept this formula from the user. By contrast, `="a"+1` will only result in the runtime error `#VALUE!`, but will still be parsed and evaluated.

For XLParser we had three options: do not concern ourselves with invalidly type expressions, incorporate the reference expression rules into the grammar, or implement a type system similar to how this would be done in a full compiler and reject invalidly typed expressions. The first option is by far the simplest, but would result in a lot of invalid formulas being accepted, the second option would result in a more complicated grammar and might not even be possible, while the third option would result in an additional layer on top of the parser generator.

Because references are of great interest when analyzing formulas and already had additional grammar rules, the second option seemed to be achievable and acceptable and this is the route XLParser took and successfully implemented. A downside of this approach turned out to be some additional ambiguity, as explained in Section 4.2.3.

4.4.2 Unions

The comma serves both as the union operator and the 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*⟩ { ‘`,`’ ⟨*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). This lower compatibility is deemed acceptable, because unions are only extremely rarely used. In the evaluation as described in Chapter 6 unions were only encountered in 0.002% of formulas.

Additionally formulas that this grammar does not parse often result in an error value after evaluation in Excel. For example `=A1,A1` does parse in Excel, but produces the error `#VALUE!` on evaluation.

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

4.5 Improvements over existing parser

Improvements made to XLParser compared to the existing parser fall into the following categories:

More frequent rejecting of invalid formulas

XLParser is often less forgiving than the previous parser, and rejects more types of invalid formulas. This is most prominently noticable in reference expressions, becaues the old parser does not differentiate between reference and non-reference expressions. Therefore formulas like `=1 1` and `=LARGE((1,2,3),4)` are considered valid, while they are not and would be rejected by Excel.

Broader parsing of valid formulas

As detailed in Chapter 6, XLParser has a very high parse success rate.

Several language features were absent in the previous parser. Examples are ranges with multiple limits (`=SUM(A1:B2:C3:D4)`), structured table references (`=TableName[ColumnName]`), array constants `=SUM({1,2,3})` and functions in reference expression (`=SUM(IF(TRUE,A1,B2):C5)`).

Furthermore the previous parser relied on a tool which extracted the formulas as stored in spreadsheets, while BumbleBee is used as an Excel add-in and therefore receives its formulas from Excel. These formulas sometimes slightly differ in at least one aspect: when external files are referenced a numeric reference is stored while Excel provides either the filename or the file path and name. Thus a formula could be received as `=[1]Sheet!A1` from the tool, and `=[File]Sheet!A1` or `='C:\Path\[File.xlxs]Sheet'!A1` by an Excel Add-In. XLParser supports al three formats, while the previous parser only supported the first.

AST improvements

Correctness

While the AST correctness is unverified for both XLParser and the previous parser, several improvements have been made. Operator precedence has been mentioned before, this was not taken into consideration in the previous parser version, providing very problematic in BumbleBee's use-case. Several smaller corrections have also been made. For example in the previous version `=F(1,,1)` and `=F(1,1)` produced an identical AST, while they have a different meaning, especially in the case of user defined functions.

Homogenization

The previous parser was constructed with a clean base grammar, but over time additions were made to deal with constructs which could not be parsed. This caused the rules and therefore the AST to become inconsistent. XLParser solves this by reducing the grammar to a clean grammar again and removing inconsistencies. However, this advantage is subjective and hard to quantify. An example of this are user defined functions which produced a different AST depending on whether they were internal `=UDF()` or external `=[1]UDF()`. Another example are prefixes, all of the following used different tokens and productions rules: `=Sheet!A1`, `='Sheet'!A1`, `=[1]Sheet!A1` and `='[1]Sheet'!A1` while in XLParser the tokens are unformalized, and the production rules are cleaner in the authors opinion.

Chapter 5

Refactoring spreadsheets

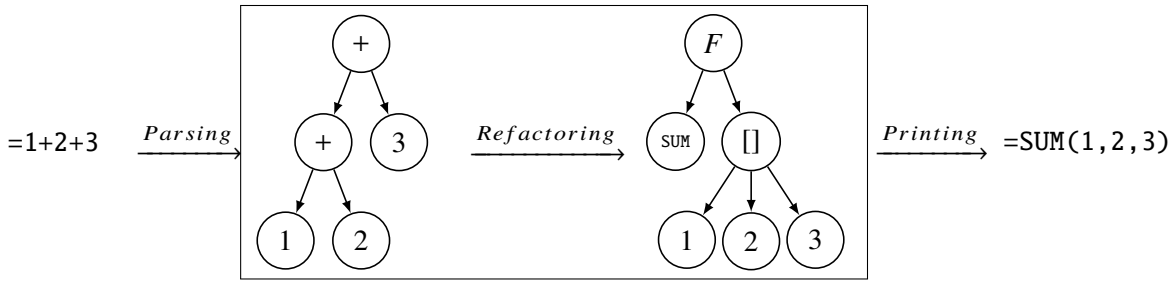


Figure 5.1: Overview of the refactoring process

Refactoring a spreadsheet involves changing the worksheets, cells and formulas in a workbook. Excel provides an API to change the worksheet and cells, and most other elements of a workbook. When it is desired to refactor formulas this means the original formula string must be changed into a new formula string. This is usually implemented by parsing the formula, performing the desired transformations on the AST and then printing the AST back to a string form [8]. The inner workings of the parser are described in Chapter 4. This chapter covers how the AST is transformed for each refactoring, as seen in Figure 5.1.

The refactorings are implemented in the BumbleBee Excel Add-In[1], and presented to the user through a context menu as seen in Figure 5.2. This context-menu automatically determines if a refactoring can be performed on the specific selected cell(s) and disables inapplicable refactorings.

[1] http://spreadsheetlab.org/2015/10/12/bumblebee-an-excel-refactoring-add-in/

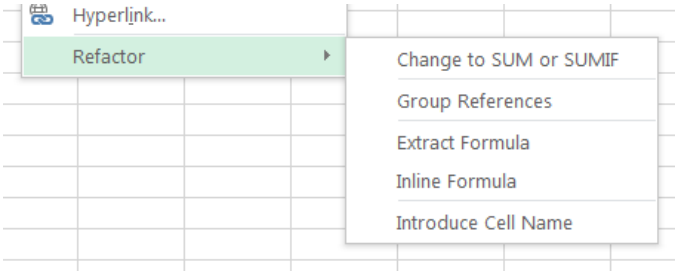
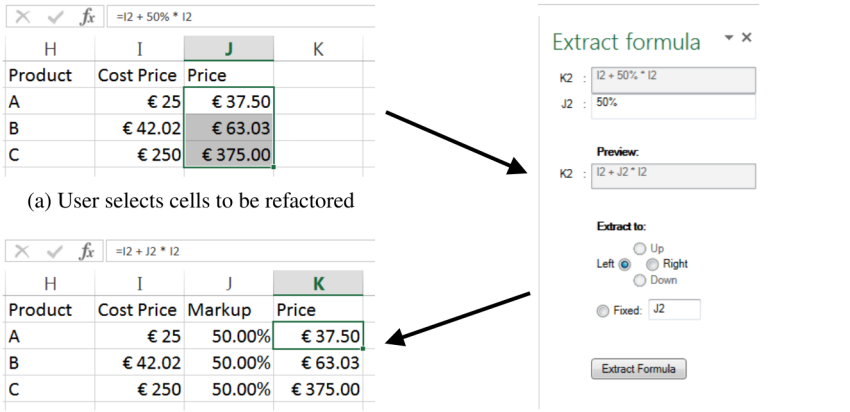


Figure 5.2: BumbleBee context menu



(a) User selects cells to be refactored

(b) User selects subformula to be extracted

(c) Refactoring has been performed

Figure 5.3: An example application of EXTRACT FORMULA

5.1 EXTRACT FORMULA

The goal of the EXTRACT FORMULA refactoring is to move part of a formula expression, a sub-formula, to another cell, which has a number of potential use cases:

- Magic numbers or other constants can be extracted to a separate cell, thus making them easy to adjust.
- A large or complicated formula can be made easier to understand by splitting it into more smaller components. This solves the MULTIPLE OPERATIONS smell [12].
- Reduce duplication in a formula by extracting common sub-formulas into another cell. This solves the DUPLICATED FORMULAS smell [12].

5.1.1 User interface

This refactoring requires the user to select cell(s) to be refactored, enter the subformula to be extracted and select where the extraction should occur to. Figure 5.3 shows the process as experienced by the user. The user first selects the formulas to be extracted (Figure 5.3a) and clicks the Extract Formula entry in the refactoring context menu (not shown). A side-panel pops out which allows the user to enter the sub-formula to be extracted and where it should be extracted to (Figure 5.3b) and presses the Extract Formula button. In the example the `50%` subformula was extracted to the left, and Figure 5.3c shows the situation after the user has named the new column.

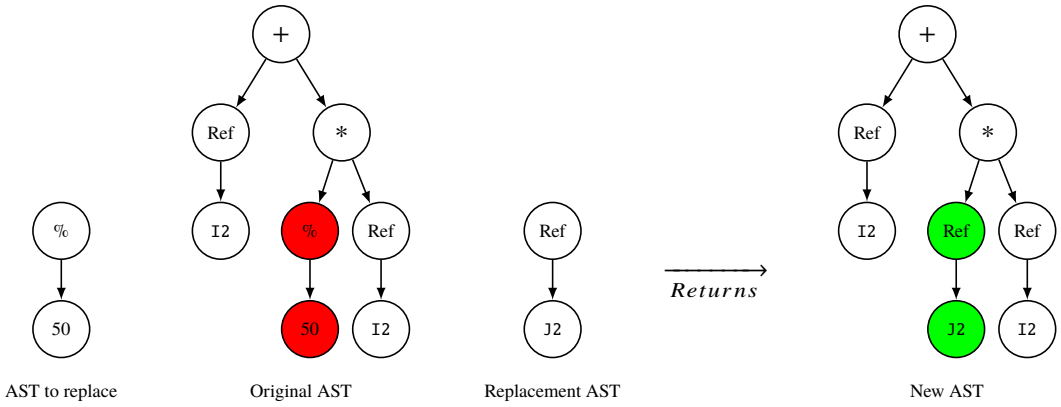


Figure 5.4: AST transformation to implement the EXTRACT FORMULA refactoring

5.1.2 Implementation

This subsection describes the details of the refactoring implementation, which consists of 2 parts: an AST tranformation of the formula, and a modification of the worksheet. The first part operates solely on the formula and refactors it to the desired form, the second part handles actual placement of the formula in the cells and the moving if necessary.

AST Transformation

The AST transformation takes the original AST, the AST to replace and the replacement AST as inputs. Then the original AST is traversed and every occurrence of the AST to replace is replaced by the replacement AST, yielding the new AST, this is illustrated in Figure 5.4. The C# code for the AST replacement can be found in Listing A.2.

This transformation is somewhat similar to the BumbleBee formula transformation rules. However, it is complimentary rather than identical as can be seen by comparing Figure 5.4 and Figure 5.5. It might seem like the transformation rule "`=I2 + [a] * I2`" is suitable for this refactoring. However, using transformation rules BumbleBee would have searched for the "outer" formula, keeping the `[a]` ← `50%` available for the replacement rule. In contrast, this transformation searches for the `[a]` "inner" formula, and replaces it with something different.

Spreadsheet refactoring

The AST replacement is performed on the original formula yielding a new formula, which is assigned to the cell that is being refactored. If multiple cells are refactored at once, the AST replacement is performed on all of them. Formulas with the same original R1C1 formula will have the same new R1C1 formula, so the AST replacement is only performed once per unique R1C1 formula and the resulting formula is re-used.

If the target of the extraction is a single cell, that cell gets assigned the subformula that will be extracted, otherwise if the user wants to extract in a direction, new cells are created in the appropriate direction and all will get assigned the subformula that will be extracted.

The C# code for the spreadsheet refactoring can be found in Listing A.3.

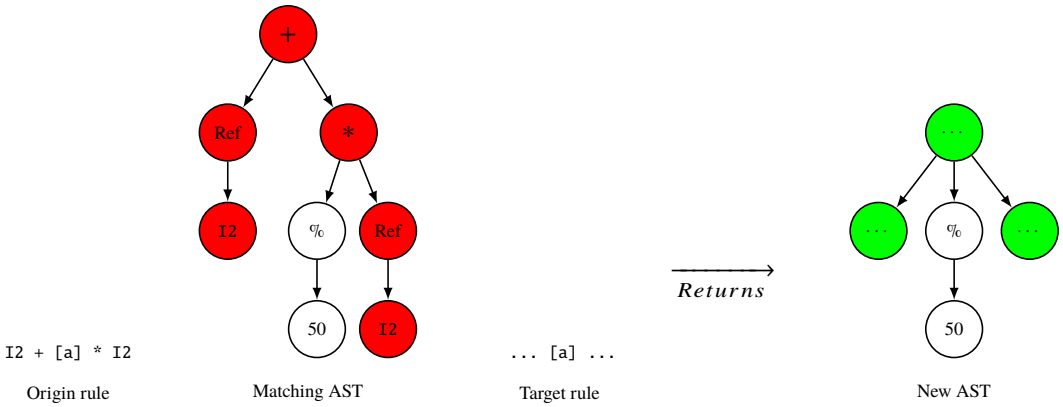


Figure 5.5: BumbleBee transformation rule

5.1.3 Detection of applicability

EXTRACT FORMULA is always applicable to a formula cell, as even a very simple formula like `=A1` still has a component that can be extracted. In this case if `=A1` would be extracted to `B1` the original cell would become `=B1`. This could be repeated endlessly, similar to how one could always extract a method that only consists of a call to another method. Whether it is a good thing to perform this refactoring is dubious, but BumbleBee relies on the user to make this assessment.

5.1.4 Improvements over RefBook's EXTRACT ROW OR COLUMN and EXTRACT LITERAL

Two specialized versions of this refactoring were previously described by Bamade and Dig [4] and implemented in their RefBook tool. RefBook's EXTRACT ROW OR COLUMN and EXTRACT LITERAL refactorings can both be performed by EXTRACT FORMULA. We have chosen to not keep the EXTRACT ROW OR COLUMN refactoring name because it does not fully describe the refactoring (a full row or column does not necessarily have to be extracted) and to keep the name in line with refactoring names in other domains.

The RefBook EXTRACT LITERAL refactoring can put a constant value into a cell and replace the occurrences of it with references to that cell, which can also be achieved with the BumbleBee EXTRACT FORMULA refactoring. In addition this is possible for any constant expression, an expression without references, instead of only for constants.

The BumbleBee EXTRACT FORMULA refactoring has several advantages over Badame's implementation of EXTRACT ROW OR COLUMN. Firstly RefBook does not handle operator precedence. This can be very problematic for this refactoring, because one of the most important properties of a refactoring should be that it does not change the program results. Note that the RefBook authors were aware of this deficiency, and left this as future work. This future work has been performed by the thesis author.

Secondly RefBook can only handle a single row or column, which must have exactly the same R1C1 formula in every cell. It can only extract the subformula to a column to the right of the original range or a row above the original range. BumbleBee can handle arbitrarily shaped ranges, with the only requirement that the subformula to be extracted occurs in all selected cells. Furthermore in addition to extracting to a cell neighboring the original formula cell (up, down, left or right) it can also extract the subformula to a single shared cell location, which is very useful to remove duplication.

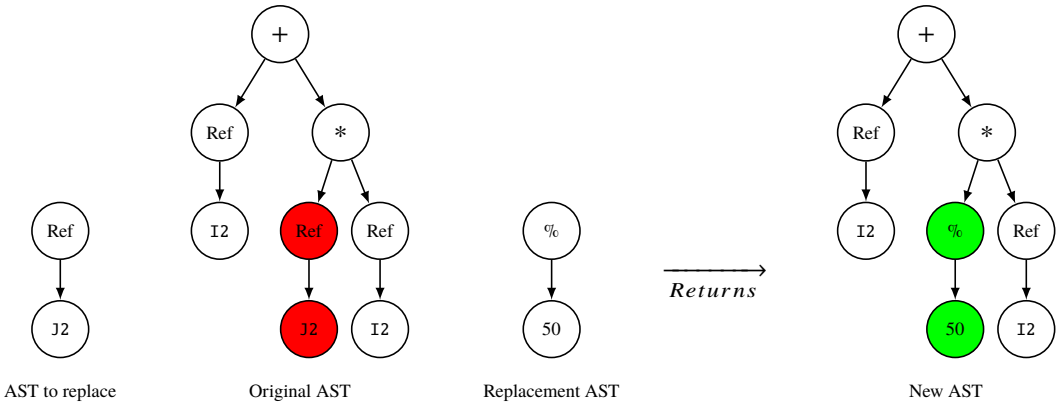


Figure 5.6: The AST transformation for INLINE FORMULA, which is the inverse of the EXTRACT FORMULA transformation in Figure 5.4

5.2 INLINE FORMULA

The goal of the INLINE FORMULA refactoring is to replace all references to a cell with its contents and delete the original cell, and is therefore the inverse of the EXTRACT FORMULA refactoring. For example if `A1` would contain `=1+1` and `B1` contains `=A1*2`, after applying this refactoring to `A1` `B2` would contain `=(1+1)*2`. The main potential use case for this formula is when the contents of a cell are clearer or just as clear as a cell reference. It can also be used to solve the LONG CALCULATION CHAIN smell [12].

While single cell references (e.g. `=SUM(A1,A2,A3)`) can always be inlined, a cell referenced as part of a range (e.g. `=SUM(A1:A3)`) can not always be inlined. If in the previous example `A1` would contain `20`, the first formula would turn out fine: `=SUM(20,A2,A3)`, while the formula `=SUM(20:A3)` is invalid. It might be possible to handle inlining into ranges using array formulas, but the extra complexity this would introduce in the formulas never outweighs the benefit of inlining in the authors opinion. Some formulas might be able to be rewritten, e.g. `=SUM(A1:A3)` could become `=SUM(20,A2:A3)`, but this does not work in every case (e.g. `A1` cannot be inlined into `=A1:A5 3:3`) and thus such behavior has a higher chance to introduce errors and confuse users. For these reasons the implementation does not perform the refactoring if the cell is referenced as part of a range.

For similar reasons, this refactoring cannot be performed on cells which are part of a named range consisting of more than one cell.

5.2.1 User interface

The refactoring is a one-click refactoring that is activated from the cell context menu, no additional user input is normally needed.

If one of the selected cells is referenced as part of a range, the user gets the choice to either abort the refactoring or continue replacing all references where the cell isn't part of a range.

5.2.2 Implementation

If a formula cell D contains a reference to cell P, D is called a dependent of P, and P is called a precedent of D. This refactoring works by first collecting all dependents of the to be inlined cell. This information is provided by Excel, although it could be manually constructed by parsing all formulas and building a dependency graph.

In every dependent a reference to the to be inlined cell is replaced by its contents, using the same AST transformation used by EXTRACT FORMULA as described in Section 5.1.2, with a reference to the cell as the AST to replace and the cell contents as the replacement AST. This is also illustrated in Figure 5.6, which shows the INLINE FORMULA inverse action of the transformation performed in Figure 5.4. If the refactoring is successful, the original cell is deleted.

The refactoring can be performed on multiple cells at the same time. To achieve this the above process is simply repeated.

The C# code for this spreadsheet refactoring can be found in Listing A.4.

5.2.3 Detection of applicability

As described in the introduction of this section, INLINE FORMULA is applicable to all cells which have dependents but are not referenced as part of ranges. For speed purposed however, the BumbleBee refactoring context menu only check whether the cell has any dependents. Doing the full check would introduce significant delay every time the user would right click.

5.3 INTRODUCE CELL NAME

As noted in Chapter 3, a cell or range can have a user defined name. If the user has defined a name for a cell or range, this name can be used in formulas instead of its location.

This is the motivation for the INTRODUCE CELL NAME refactoring which was defined by Badame and Dig [4]. With this refactoring, a user can define a name for a cell, and all references to that cell will be replaced the name.

BumbleBee re-implements this refactoring, with as an additional functionality to name a range of cells and replace all references to that range with the name.

5.3.1 User Interface

Excel has a "Define Name" option in cell the context-menu, but it lacks the functionality to also perform this refactoring. The refactoring context-menu for this option closely mirrors the Excel user interface, and is called "Define and Use Name". After the user clicks it, he enters a new name in a dialog box and the refactoring is performed.

5.3.2 Implementation

This implementation also uses the notion of dependents and precedence, which is explained in Section 5.2.2.

First the cell or range is given the name supplied by the user. Then in every dependent a reference to the now named cell is replaced by its name, using the same AST transformation used by EXTRACT FORMULA as described in Section 5.1.2.

The C# code for this spreadsheet refactoring can be found in Listing A.5.

5.3.3 Detection of applicability

This refactoring shares the detection of applicability with INLINE FORMULA: if the cell or range is referenced anywhere in the spreadsheet

5.3.4 Improvements over RefBook's INTRODUCE CELL NAME

BumbleBee's version incorporates a small improvements over RefBook's refactoring: it not only works on cells, but also on ranges.

5.4 GROUP REFERENCES

In the spreadsheet formula language, some built-in functions have the ability to accept a variable number of arguments, most prominently `SUM`, the most commonly used function [42]. These functions also accept ranges, and thus the formulas `=SUM(A1,A2,A3,A4)` and `=SUM(A1:A4)` are equivalent. The GROUP REFERENCES refactoring assumes spreadsheet users prefer the latter, and merges multiple adjacent cell references into a single range reference. The refactoring can be used to solve the MULTIPLE REFERENCES [12] smell if the referenced cells are adjacent but referred to separately. The refactoring was defined but not implemented by Hermans et. al [12]

5.4.1 User Interface

The refactoring is a one-click refactoring that is activated from the cell context menu, no additional user input is needed.

5.4.2 Implementation

Grouping algorithm

In order to find the best grouping we have to solve the following problem: given a sheet with a certain set of cells selected, what are the ranges that select exactly those cells and do so with a minimum amount of ranges?

It turns out that this is a NP-hard problem, because it has a straightforward translation to a NP-hard version of the Polygon Covering problem [43], specifically covering a rectilinear polygon (the selected cells) with axis-parallel rectangles (ranges), allowing for holes. This allows us to use an approximation algorithm or heuristic. A $O(\sqrt{\log n})$ approximation algorithm for this specific problem has been found by Kumar and Ramesh [44], but implementing this would take a non-trivial amount of effort.

However, rather than implementing a heuristic ourselves, this is currently delegated to Excel which contains this functionality. The algorithm Excel uses for this is unknown.

Spreadsheet Refactoring

The implementation traverses the formula AST, and, for every function with a variable number of arguments it encounters, groups its references by excluding all non-references (e.g. constants) and sending these to Excel to be grouped. The function arguments then are replaced by grouped references and the AST is printed back to the formula cell. References are processed separately depending on their absolute markers, e.g. `A1`,`$A1`, `A$1` and `$A$1`, because grouping references with different markers cannot be done without changing the meaning of the formula.

If multiple cells are selected, the refactoring is repeated for every one.

The C# code for this refactoring can be found in Listing A.6

5.4.3 Detection of applicability

This refactoring will be available to the user if the formula contains two or more references.

5.5 INTRODUCE AGGREGATE

In the spreadsheet formula language, three binary operators have an equivalent aggregate function that accepts any number of arguments: + corresponds to SUM, * to PRODUCT and & to CONCATENATE.

Thus a formula like =A1+A2+A3+A4 can be rewritten to =SUM(A1,A2,A3,A4), which is what the REPLACE AWKWARD FORMULA refactoring, defined by Badame and Dig [4], does. The refactoring is especially useful when combined with the GROUP REFERENCES refactoring, which rewrites it to =SUM(A1:A4).

The author proposes an alternate name for this refactoring, INTRODUCE AGGREGATE, for two reasons. Firstly REPLACE AWKWARD FORMULA does not describe very good what the refactoring does, as more types of "Awkward Formulas" could be thought of which cannot be handled. For example a formula with multiple nested IFs like =IF(IF(IF(...),FALSE,TRUE),A1,B1) could be described as "awkward", but this refactoring does not replace it. Secondly this keeps the name consistent with the INTRODUCE CONDITIONAL AGGREGATE refactoring.

5.5.1 User Interface

This refactoring is a one-click refactoring that is activated from the cell context menu, no additional user input is needed. In the user interface it will always transparently be followed by the GROUP REFERENCES refactoring.

The refactoring shares a menu item with INTRODUCE CONDITIONAL AGGREGATE, as the operator will be changed to a SUMIF if applicable. The menu item is called "Change to SUM or SUMIF" to make it easier to understand for users, because INTRODUCE (CONDITIONAL) AGGREGATE is abstract and contains jargon. While "Change to SUM or SUMIF" does not fully describe the refactoring, this will be the most common use case and easy to understand for spreadsheet users.

5.5.2 Implementation

The implementation traverses the formula AST until it encounters the first operator it can refactor. The left subtree then gets added to a list of arguments for the aggregate function, and the right subtree gets checked to see if it corresponds to the same operator. If it is, the process gets repeated until a non-operator right subtree is encountered, at which point that subtree is the final addition to the argument list. Note that this works because all operators are left-associative in Excel. If operator has other associativities the algorithm would have to be slightly altered.

The C# code for this refactoring can be found in listing A.7.

5.5.3 Detecting applicability

This refactoring will only be offered to the user if the top layer of a formula cell consisting of one of the applicable operators (+,* or &)

5.5.4 Improvements over RefBook's REPLACE AWKWARD FORMULA

The concept of the INTRODUCE AGGREGATE/REPLACE AWKWARD FORMULA refactoring is not complicated, and therefore not many improvements can be made. However, BumbleBee does improve over RefBook in several ways related to this refactoring.

A first improvement in BumbleBee's implementation comes through the underlying parser: the RefBooks parser does not take operator precedence into account, and therefore a formula like `=1 + 2 * 3 + 4` would be refactored into `=SUM(1,2 * 3 + 4)` by RefBook, while BumbleBee refactors it into `=SUM(1,2 * 3,4)`.

A second improvement comes by combining this refactoring with two other refactorings through the user interface. Combining this refactoring with GROUP REFERENCES allows a formula like `=A1+A2+A3+A4` to be rewritten into `=SUM(A1:A4)` instead of just `=SUM(A1,A2,A3,A4)`. Combining this refactoring with INTRODUCE CONDITIONAL AGGREGATE when the spreadsheet data allows for this enables a formula like `=B12+B24+B36` to be rewritten into `=SUMIF("2015", B:B)` instead of just `=SUM(B12,B24,B36)`.

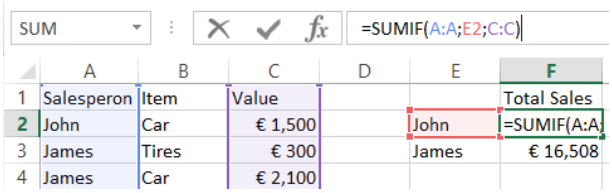


Figure 5.7: An example spreadsheet which uses `SUMIF`

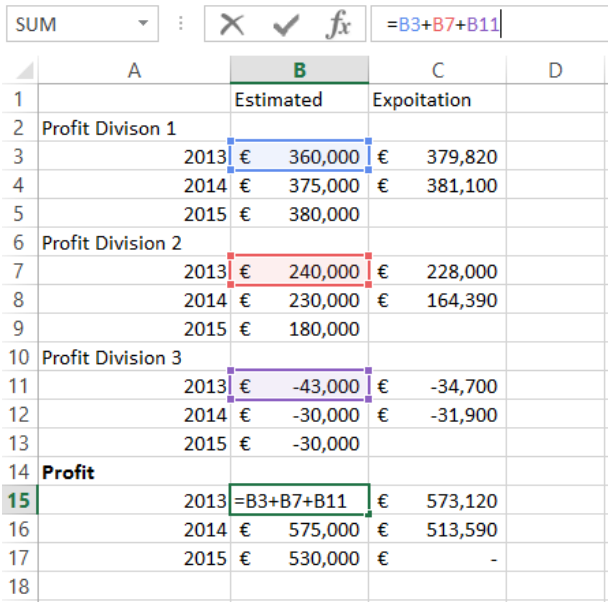


Figure 5.8: An example where `SUMIF` could be used, but is not

5.6 INTRODUCE CONDITIONAL AGGREGATE

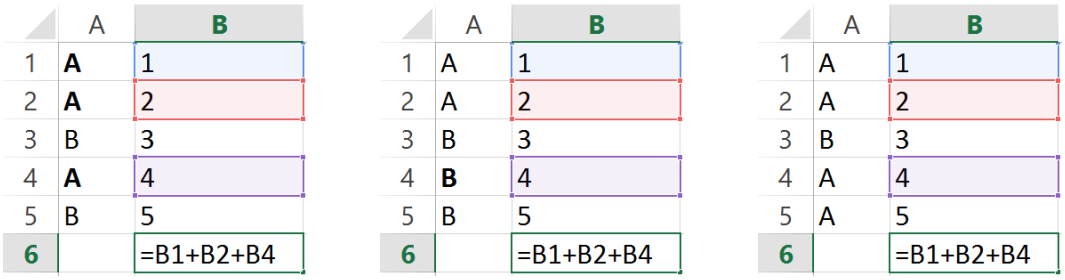
Three aggregate functions also have conditional counterparts: `SUMIF`, `AVERAGEIF` and `COUNTIF`. These functions take two mandatory arguments: **check_range**, **condition** and the optional **operating_range**. If **operating_range** is absent it equals **check_range**, with `COUNTIF` not supporting **operating_range** at all. These functions evaluate the **condition** on every cell of **operating_range**, and if it is met the function's operation is performed on the corresponding cell from **operating_range**.

Examples make this clearer: `=SUMIF(A:A, "> 10")` sums all cells in column `A` that contain a value larger than `10`, and `=SUMIF(A:A, E2, C:C)` sums the cell from column `C` of every row where the cell value from column `A` equals `E2` , as illustrated in Figure 5.7.

The conditional aggregate functions are powerful tools, but they are not always used. Instead what we often encountered is the "Manual" selection of the correct cells and summing these, as illustrated in Figure 5.8. When combining this refactoring with the INTRODUCE AGGREGATE refactoring we can rewrite `=B3+B7+B11` to `=SUM(B3,B7,B11)`, and that to `=SUMIF(A:A, "2015", B:B)`. Because this was the most common pattern we will focus on `SUM` and `SUMIF`, but the refactoring works identical for `COUNT` to `COUNTIF` and `AVERAGE` to `AVERAGEIF`.

5.6.1 User Interface

This refactoring shares a context menu entry with INTRODUCE AGGREGATE.



	A	B
1	**A**	1
2	**A**	2
3	B	3
4	**A**	4
5	B	5
6		=B1+B2+B4

(a) Supported: Column `A` uniquely determines which values are summed.

	A	B
1	A	1
2	A	2
3	B	3
4	**B**	4
5	B	5
6		=B1+B2+B4

(b) Unsupported: No column determines which values are summed.

	A	B
1	A	1
2	A	2
3	B	3
4	A	4
5	A	5
6		=B1+B2+B4

(c) Unsupported: Column `A` does not determine which values are summed, because `B5` is not summed.

Figure 5.9: Example scenarios of INTRODUCE CONDITIONAL AGGREGATE

5.6.2 Implementation

In order to see if a `SUM` can be rewritten to a `SUMIF`, we must determine if there is another range of cells with a value that uniquely identifies the summed cells. In relational database theory, this is a concept known as a *functional dependency*. Several algorithms exist to automatically find functional dependencies, such as Dep-Miner [45], FUN [46] and TANE [47], where FUN has been applied in a spreadsheet context [21].

Due to implementation time constraints, these algorithms were not used, and instead only a very simple but useful scenario is supported: the summed values are in a single row or column, and there is a single row or column that has a single unique value in it which can serve as the condition for `SUMIF`. Figure 5.9a shows such a scenario: column `A` uniquely determines the summed values and the formula can thus be rewritten to `=SUMIF(A:A, "A", B:B)`. Figure 5.9b shows a case where this is not possible: `A4` differs from `A1` and `A2` meaning there is no column that determines the summed values. Figure 5.9b shows another impossible case: here column `A` determines the summed values, but there is an additional row with an identical value in cell `A5` and thus column `A` does not *uniquely* determine the summed values. Additionally, the refactoring is not supported if the summed cells are not in a single row or column.

The C# code that performs the refactoring if the above conditions are met can be found in Listing A.8.

5.6.3 Detection of applicability

This refactoring is activated only as part of the INTRODUCE AGGREGATE refactoring, and therefore shares its detection of applicability. Any further detection is performed within the code that implements the refactoring itself.

5.7 Discussion

5.7.1 Undo and redo functionality

It must be noted that all refactorings have a major deficiency: they cannot be undone with the Excel undo function. This is not a limitation inherent to the refactorings, the previous state could be remembered and some refactorings even have a well-defined inverse refactoring.

The reason that undo and redo functionality is not available in BumbleBee is a technical limitation imposed by Excel: the Excel undo-redo stack is not available to Excel Add-Ins. Instead, as soon as a Excel Add-In changes the Excel spreadsheet file, in fact as soon as it interacts with the internal document model even if it does not change anything, the Excel undo-redo stack gets cleared. In an industrial-strength application this functionality would be essential.

A workaround is possible by manually maintaining an undo stack, however this was deemed outside of the scope of this thesis. This workaround also does not integrate well with Excel and does not allow changes made outside of the Add-In to be undone. As long as Excel keeps this restriction this will always be a severe limitation to any tool that automatically changes spreadsheet files for the user.

5.7.2 Future improvement possibilities

EXTRACT FORMULA could benefit from more UI support. If the user could select the formula to be extracted from within the Excel formula editor the user interface could be streamlined. Another possibility would be to determine candidate subformulas that the user could extract, and present these in a list. Generating a list of all subformulas is fairly trivial, but this list can become very large, especially in the longer formulas that would benefit from this refactoring. A heuristic which can prune the list of subformulas could improve this experience.

A minor deficiency in the current EXTRACT FORMULA code is that is does not fully handle the commutative operators + and *. An operator is commutative if the order of the operands does not matter: `1 + 2` is equal to `2 + 1`. BumbleBee's equality comparer takes this into account, and tries the reverse order of the operands for + and * if it does not find a match. However, this does not fully solve the problem: the formula `=1 + 2 + 3` is parsed as `=(1 + 2) + 3`, thus if the user wishes to extract `2 + 3` BumbleBee will report an error since it does not encounter `2 + 3` in the AST.

The INTRODUCE CONDITIONAL AGGREGATE automated refactoring currently only supports a very basic scenario. The algorithm only checks if a single column determines another column, and furthermore the algorithm is not proven correct. Both of these problems could be solved by properly determining functional dependencies using an established algorithm. Functional dependencies where two or more columns determine the content of a single column can be supported through the `SUMIFS`, `AVERAGEIFS` and `COUNTIFS` functions. For example `=SUMIFS(C:C,A:A,2015,B:B,"John")` sums the values from column `C` of the rows where column `A` contains `2015` and column `B` contains `"John"`, which corresponds to column `C` being dependent on columns `A` and `B`.

Chapter 6

Evaluation

6.1 Refactorings

The refactorings implemented in BumbleBee were not formally evaluated as part of this thesis. Some of the refactorings implemented in this thesis are based on RefBook [4], which did perform an evaluation, in which it was found that the more people preferred formulas after the EXTRACT FORMULA and INTRODUCE AGGREGATE refactorings were applied, while more people preferred the unrefactored formulas after the INTRODUCE CELL NAME refactoring was applied.

6.2 Parser

In order to evaluate XLParser, it was used to parse all formulas extracted from the two publicly available spreadsheet research datasets, the EUSES corpus [48] and Enron email corpus [42, 49].

A "scantool" not developed by the thesis author was used to extract formulas from these two datasets, which succeeded for 19.601 of the 20.688 spreadsheets. Not all spreadsheets could be read because they were either password-protected or could not be processed by the scantool. A combined 22.310.406 formulas were found in these spreadsheets. Duplicate formulas were filtered on a sheet-level by using the formula's R1C1 notation. This resulted in 1.035.586 unique formulas. 26 formulas were discarded, because they were corrupted by the scan tool, bringing the total to 1.035.558 formulas. The extracted list of formulas is available as part of XLParser in its repository for both the Enron [1] and the EUSES [2] datasets.

Of these formulas, exactly two could not be parsed: `=-NOX, Regi` and `-_SO2, regi`, a successful parse rate of 99.999%. The unparsable formulas are examples of unions without brackets which the parser cannot parse due to reasons outlined in Section 4.4.2.

[1] https://github.com/spreadsheetlab/XLParser/blob/v1.2.1/src/XLParser.Tests/data/enron/formulas.txt

[2] https://github.com/spreadsheetlab/XLParser/blob/v1.2.1/src/XLParser.Tests/data/euses/formulas.txt

It must be noted that both datasets used only contain spreadsheets created before 2005. This means that they do not contain structured table references, as these were introduced in Excel 2007. It thus remains unevaluated how well the parser handles formulas containing these constructs.

Another deficiency is that the scantool extracts formulas as they are stored inside the spreadsheet file. When a formula references an external file, in the spreadsheet file the filename is stored in a separate lookup array and the file is replaced by a numeric index in the stored formula. Thus a formula like `='C:\path\[file.xlsx]Sheet!A1'` would be stored as `=[1]Sheet!A1`. However, Excel Add-Ins like BumbleBee receive the file name and path in the formula when requested from Excel. It is thus possible that the parser cannot handle all formulas formulas containing external references in their non-numeric file reference form. However, a member of the spreadsheet lab who used the parser to process the same two datasets reports using their non-numeric file reference form reports no additional parse errors, except in the case of a file path containing a space, which is a problem that most likely can be solved.

6.2.1 Analysis

For all the formulas extracted from the Enron and EUSES datasets a count was performed of which tokens (terminals, leaf nodes of the parse tree) and which production rules (non-terminal, internal nodes of the parse tree) were used when parsing the formula.

At least one reference was used in 99.2% of the formulas, which shows that references are an integral part of spreadsheets. Arithmetic operators are also very common (59.77%), as is the use of built-in excel functions (35.82%).

Perhaps more interesting are the less commonly used grammatical constructs: Empty arguments, Dynamic Data Exchange calls, Intersections and Unions, Multiple Sheets, Reserved Names and constant arrays. All of these constructs are used in less than 0.05% of all formulas.

The rarity of empty argument, DDE calls, reserved names and multiple sheets can be expected because of their very niche uses. Intersections, unions and constant arrays are powerful constructs, but apparently users either do not know about these, prefer to use other constructs instead or their usage is too niche.

Interestingly, some of these constructs are quite hard to parse. Especially unions and intersections complicate the grammar because of their curious syntax. Because these constructs are so rare this means they are prime candidates to leave out of the grammar if simplification is desired.

Syntax	Example	Unique formulas		Total formulas	
⟨*Formula*⟩	`=1+2`	**1,035,586**		**22,310,406**	
⟨*Reference*⟩	`=E9/E10`	962,783	92.97%	22,131,002	**99.20%**
CELL	`=A5`	951,521	91.88%	22,021,833	98.71%
⟨*FunctionCall*⟩	`=SUM(A5:A22)`	701,626	67.75%	18,944,204	84.91%
⟨*BinOp*⟩	`=H10-H8`	397,580	38.39%	13,333,844	**59.77%**
⟨*Constant*⟩	`=A5+134`	271,585	26.23%	8,731,489	39.14%
EXCEL-FUNCTION	`=SUM(A5:A22)`	264,673	25.56%	7,991,329	**35.82**%
NUMBER	`=(B8/48)*15`	250,085	24.15%	7,849,495	35.18%
⟨*Prefix*⟩	`=Sheet1!B1`	337,727	32.61%	5,599,011	25.10%
⟨*RefFunctionName*⟩	`=SUM(J9:INDEX(J9:J41,B43))`	55,680	5.38%	5,349,237	23.98%
SHEET	`=Sheet1!B1`	303,981	29.35%	5,282,386	23.68%
REF-FUNCTION-COND	`=IF(A1<0,0,1)`	50,171	4.84%	4,872,661	21.84%
⟨*Reference*⟩ ':' ⟨*Reference*⟩	`=SUM(A5:A22)`	184,451	17.81%	3,735,005	16.74%
⟨*UnOpPrefix*⟩	`=+B11+1`	218,397	21.09%	3,289,326	14.74%
STRING	`=IF(AD3<0,"buy","sell")`	56,635	5.47%	2,587,971	11.60%
⟨*NamedRange*⟩	`=SUM(freq)`	20,686	2.00%	1,645,120	7.37%
BOOL	`=IF(AND(R11=1,R14=TRUE),G19,0)`	7,532	0.73%	1,183,798	5.31%
FILE	`=[11]Sheet1!C5`	104,892	10.13%	1,135,185	5.09%
REF-FUNCTION	`=SUM(J9:INDEX(J9:J41,B43))`	9,907	0.96%	778,056	3.49%
SHEET_QUOTED	`=('[2]Detail I&E'!D62)/1000`	33,781	3.26%	325,498	1.46%
UDF	`=SQRT(_eoq2(C5,C4,C6,C7))`	21,352	2.06%	286,210	1.28%
(' ⟨*Reference*⟩ ')'	`=(2*(B29))/(1+B29)`	6,394	0.62%	266,420	1.19%
_xll.	`=_xll.RiskTriang(F9,F7,F8)`	11,922	1.15%	127,348	0.57%
ERROR-REF	`=AVERAGE(#REF!)`	3,477	0.34%	123,447	0.55%
VERTICAL-RANGE	`=COUNT(A:A)`	851	0.08%	55,254	0.25%
FILE '!'	`=[1]!today`	2,040	0.20%	28,448	0.13%
ERROR	`=IF(R14=TRUE,G19,#N/A)`	379	0.04%	27,237	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%**
DDECALL	`=TWINDDE|RSFRec!'NGH2 NET.CHNG'`	3,276	0.32%	3,686	**0.02**%
Intersection	`=Ending_Inventory Jan`	304	0.03%	2,343	**0.01**%
MULTIPLE-SHEETS	`=SUM(Sheet1:Sheet20!I29)`	173	0.02%	1,986	**0.01%**
External UDF	`=[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**%
RESERVED_NAME	`=C23/_xlnm.Print_Area`	70	0.01%	276	**0.00**%
FILE MULTIPLE-SHEETS	`=SUM([2]Section3A:formulas!B11)`	4	0.00%	189	**0.00**%
⟨*ConstantArray*⟩	`=FVSCHEDULE(1,0.09;0.11;0.1)`	15	0.00%	19	**0.00**%

Table 6.1: Frequency of tokens and production rules in all parsed formulas

Chapter 7

Conclusion

While extending the BumbleBee spreadsheet refactoring tool it was found that the existing spreadsheet formula parser was insufficient to support refactoring without the risk of introducing errors. To solve this problem, the existing parser was improved, named XLParser and publicly released as open-source with a binary[1], online demo[2], source code[3] and EBNF grammar available. The parser is geared towards spreadsheet research and therefore is highly compatible with the official language and produces parse trees which are suitable for further analysis and manipulation. The parser was evaluated on over a million formulas from two datasets and parsed 99.999%. BumbleBee was extended with six refactorings: extract formula, inline formula, introduce cell name, group references, introduce aggregate and introduce conditional aggregate. inline formula, group references and introduce conditional aggregate were not implemented before and extract formula and introduce cell name improve upon previous implementations.

[1] https://github.com/spreadsheetlab/XLParser/releases
[2] http://xlparser.perfectxl.nl/demo
[3] https://github.com/spreadsheetlab/XLParser

Bibliography

[1] Raymond R Panko. Facing the problem of spreadsheet errors. *Decision Line*, 37(5):8–10, 2006.

[2] Raymond R Panko. What we know about spreadsheet errors. *Journal of End User Computing*, 10:15–21, 1998.

[3] Felienne Hermans and Danny Dig. Bumblebee: A refactoring environment for spreadsheet formulas. In *Proceedings of FSE'14*, pages 747–750, 2014.

[4] Sandro Badame and Danny Dig. Refactoring meets spreadsheet formulas. In *Proceedings of ICSM'12*, pages 399–409. IEEE, 2012.

[5] Efthimia Aivaloglou, David Hoepelman, and Felienne Hermans. A grammar for spreadsheet formulas evaluated on two large datasets. In *Proceedings of SCAM '15*, 2015.

[6] William F Opdyke. Refactoring: A program restructuring aid in designing object-oriented application frameworks. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[7] Robert S Arnold. *An introduction to software restructuring*. IEEE Computer Society Press, Washington, DC, 1986.

[8] Martin Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 1999.

[9] Jácome Cunha, João P Fernandes, Hugo Ribeiro, and João Saraiva. Towards a catalog of spreadsheet smells. In *Proceedings of ICCSA'12*, pages 202–216. Springer, 2012.

[10] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting and visualizing inter-worksheet smells in spreadsheets. In *Proceedings of ICSE'12*, pages 441–451, 2012.

[11] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting code smells in spreadsheet formulas. In *Proceedings of ICSM '12*, 2012.

[12] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering*, 20(2):549–575, 2014.

[13] Jocelyn Paine. Ensuring spreadsheet integrity with model master. *arXiv preprint arXiv:0801.3690*, 2001.

[14] Jocelyn Paine, Emre Tek, and Duncan Williamson. Rapid spreadsheet reshaping with excelsior: multiple drastic changes to content and layout are easy when you represent enough structure. In *EuSpRIG 2006*, 2006.

[15] Gregor Engels and Martin Erwig. Classsheets: automatic generation of spreadsheet applications from object-oriented specifications. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 124–133. ACM, 2005.

[16] Martin Erwig, Robin Abraham, Steve Kollmansberger, and Irene Cooperstein. Gencel: a program generator for correct spreadsheets. *Journal of Functional Programming*, 16(03):293–325, 2006.

[17] Jan-Christopher Bals, Fabian Christ, Gregor Engels, and Martin Erwig. Classsheets-model-based, object-oriented design of spreadsheet applications. *Journal of Object Technology*, 6(9):383–398, 2007.

[18] Jácome Cunha, Joost Visser, Tiago Alves, and João Saraiva. Type-safe evolution of spreadsheets. In *Fundamental Approaches to Software Engineering*, pages 186–201. Springer, 2011.

[19] Jácome Cunha, Jorge Mendes, Joao Saraiva, and Joao Paulo Fernandes. Embedding and evolution of spreadsheet models in spreadsheet systems. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 179–186. IEEE, 2011.

[20] Jácome Cunha, Jorge Mendes, João Saraiva, and Joost Visser. Model-based programming environments for spreadsheets. *Science of Computer Programming*, 96:254–275, 2014.

[21] Jácome Cunha, João Saraiva, and Joost Visser. From spreadsheets to relational databases and back. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 179–188. ACM, 2009.

[22] Robin Abraham and Martin Erwig. Inferring templates from spreadsheets. In *Proceedings of the 28th international conference on Software engineering*, pages 182–191. ACM, 2006.

[23] Jácome Cunha, Martin Erwig, and Joao Saraiva. Automatically inferring classsheet models from spreadsheets. In *Proceedings of VL/HCC'10*, pages 93–100. IEEE, 2010.

[24] Power utility pak version 7. `http://spreadsheetpage.com/index.php/pupv7/home`. Retrieved 02-10-2015.

[25] Asap utilities for excel 5. `http://www.asap-utilities.com/`. Retrieved 02-10-2015.

[26] Daniel W Barowy, Dimitar Gochev, and Emery D Berger. Checkcell: data debugging for spreadsheets. In *Proceedings of OOPSLA'14*, pages 507–523. ACM, 2014.

[27] Jácome Cunha, João Paulo Fernandes, Jorge Mendes, and João Saraiva. Mdsheet: A framework for model-driven spreadsheet engineering. In *Proceedings of ICSE'12*, pages 1395–1398. IEEE Press, 2012.

[28] Unknown. Excel formula parsing. `http://ewbi.blogs.com/develops/2004/12/excel_formula_p.html`. Retrieved 03-10-2015.

[29] Daniel Ballinger. Investigation into excel syntax and a formula grammar. http://homepages.ecs.vuw.ac.nz/~elvis/db/Excel.shtml. Retrieved 03-10-2015.

[30] Libreoffice. https://www.libreoffice.org. Retrieved 03-10-2015.

[31] Calligra. https://www.calligra.org/. Retrieved 03-10-2015.

[32] Gnumeric. https://www.gnumeric.org/. Retrieved 03-10-2015.

[33] Microsoft. Microsoft by the numbers. https://news.microsoft.com/bythenumbers/ms_numbers.pdf, 2015.

[34] Apache foundation. Apache openoffice.org download statistics.

[35] Libreoffice: What's new. https://people.gnome.org/~michael/data/2015-05-02-libreoffice-whats-new.pdf. Retrieved 21-10-2015.

[36] A bridge to the cloud: Google cloud connect for microsoft office now available to early testers. http://googleforwork.blogspot.nl/2010/11/a-bridge-to-cloud-google-cloud-connect.html. Retrieved 21-10-2015.

[37] Felienne Hermans. Excel turing machine. http://www.felienne.com/archives/2974.

[38] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. A user-centred approach to functions in excel. In *ACM SIGPLAN Notices*, volume 38, pages 165–176. ACM, 2003.

[39] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*. Addison wesley, 2 edition, 2006.

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

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

[42] Felienne Hermans and Emerson Murphy-Hill. Enron's spreadsheets and related emails: A dataset and analysis. In *Proceedings of ICSE'15*, 2015.

[43] William J Masek. Some np-complete set covering problems. *Unpublished manuscript*, 1979.

[44] VS Anil Kumar and H Ramesh. Covering rectilinear polygons with axis-parallel rectangles. *SIAM Journal on Computing*, 32(6):1509–1541, 2003.

[45] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Efficient discovery of functional dependencies and armstrong relations. In *EDBT*, volume 1777, pages 350–364. Springer, 2000.

[46] Noel Novelli and Rosine Cicchetti. Fun: An efficient algorithm for mining functional and embedded dependencies. In *Database TheoryICDT 2001*, pages 189–203. Springer, 2001.

[47] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal*, 42(2):100–111, 1999.

[48] Marc Fisher and Gregg Rothermel. The euses spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–5. ACM, 2005.

[49] Bryan Klimt and Yiming Yang. The enron corpus: A new dataset for email classification research. In *Machine learning: ECML 2004*, pages 217–226. Springer, 2004.

Appendix A

Code listings

Listing A.1: XLParser Print method (simplified)

```
public static string Print(this Node node) {
  // Print token values
  if(node is Terminal) return node.Token.Text;

  // Select is C#'s map function
  var ch = node.ChildNodes.Select(Print).ToList();

  switch(node.Type()) {
    case "ArrayFormula":
      return "{=" + ch[0] + "}";
    case "FunctionCall":
      if(node.IsBinaryOperation()) {
        return ch[0] + " " + ch[1] + " " + ch[2];
      }
      if(node.isNamedFunction()) {
        return String.Join("", ch) + ")";
      }
      // some more conditions
      break;
    // More cases for every node type
  }
}
```

Listing A.2: Formula AST replacement (simplified)

```
/* Context contains the workbook and worksheet of a node */

public static Node Replace(Node subject, Node search, Node replace, Context csub,
    Context csearch, Context crepl) {
  // Check if the subject matches search
  if(Equals(subject, search, csub, csearch)) {
    // We can return the replacement.
    // Moveto handles changing reference prefixes if necesarry
    return MoveTo(replace, crepl, csub);
  }

  // No match, if we are at a leaf node, simply return the leaf node
  if (subject.ChildNodes.Count == 0) return subject;

  // Otherwise continue the replacement on the child nodes
  return new Node() {
    Type = subject.Type(),
    // Select is C#'s map
    ChildNodes = subject.ChildNodes.Select(child => Replace(child, search, replace,
        csub, csearch, crepl))
  };
}

public static bool Equals(Node p1, Node p2, Context c1, Context c2) {

  // RemoveNonEqualityAffectingNodes removes things like brackets,
  // which do not affect the equality of nodes
  p1 = RemoveNonEqualityAffectingNodes(p1);
  p2 = RemoveNonEqualityAffectingNodes(p2);

  // Qualify adds workbook and worksheet prefix to all references, so that
  // equality isn't affected by whether or not these are supplied in the original
      formula
  p1 = c1.Qualify(p1);
  p2 = c2.Qualify(p2);

  return p1.Type() == p2.Type()
    // Compare the token values if these are tokens
      && (p1 is Terminal && p1.Token.ValueString == p2.Token.ValueString)
      // Compare child count
      && p1.ChildNodes.Count == p2.ChildNodes.Count
      // Check if all children are equal
      && p1.ChildNodes.Zip(p2.ChildNodes).All((ch1, ch2) => Equals(ch1, ch2, c1,
          c2));
}
```

Listing A.3: Extract formula refactoring (simplified)

```
public void ExtractFormula(Range applyto, Location moveto, Node subformula) {

  /** Check if all applyto cells contain subformula **/

  /** Check if target cell is empty **/

  /** Check if subformula contains any non-absolute references **/

  // Set the target cell to the subformula
  moveto.Formula = subformula.Print();
  // and get its parsed address reference
  var replacementAST = moveto.Address().Parse();

  // Apply the refactoring once per unique R1C1 formula
  foreach (var uniqueR1C1 in applyto.Cells.GroupBy()(c => c.FormulaR1C1)) {
    var prototype = uniqueR1C1.First();
    var AST_or = prototype.Parse();

    prototype.Formula = Replace(AST_or, subformula, replacementAST,
        /*...*/).Print();

    foreach(var cell in uniqueR1C1) {
      cell.FormulaR1C1 = prototype.FormulaR1C1;
    }
  }
}

public void ExtractFormula(Range applyto, Direction dir, Node subformula) {

  /** Check if all applyto cells contain subformula **/

  /** Insert new cells in the appropriate direction **/

  /** Set all new cells to contain the subformula formula **/

  // Apply the refactoring once per unique R1C1 formula
  foreach (var uniqueR1C1 in applyto.Cells.GroupBy()(c => c.FormulaR1C1)) {
    Cell prototype = uniqueR1C1.First();
    Node AST_or = prototype.Parse();
    Node replacementAST = prototype.Offset[dir].Address().Parse();

    prototype.Formula = Replace(AST_or, subformula, replacementAST,
        /*...*/).Print();

    foreach(var cell in uniqueR1C1) {
      cell.FormulaR1C1 = prototype.FormulaR1C1;
    }
  }
}
```

Listing A.4: Inline Formula Refactoring (simplified)

```
public void InlineFormula(Cell toInline) {
  // Dependendants are gotten from Excel
  var dependents = toInline.dependents;
  if(dependents.Count == 0) {
    // Abort, no dependants
  }

  Node AstToRepl = Parse(toInline.Address);
  Node AstReplacement = Parse(toInline.Formula);

  // Check if this cell is part of any named ranges with more than one cell
  if(toInline.Names.Any(name => name.Cells.Count > 1)) {
    // Abort
  }

  // AST representation of the names
  var names = toInline.Names.Select(Parse);

  foreach(var dependent in dependents) {
    Node AstOriginal = Parse(dependent.Formula);

    // Abort if to be inlined cell is references as part of a range
    if(CellContainedInRanges(toInlineAddress, AstOriginal)) {
      // Abort
    }

    // Replace references to the cell with the value
    var AstNew = Replace(AstOriginal, AstToRepl, AstReplacement, /*...*/);
    foreach(var name in names) {
      AstNew = Replace(AstNew, name, AstReplacement, /*...*/);
    }

    dependent.Formula = AstNew.Print();
  }

  toInline.Delete();

}
```

Listing A.5: Introduce Cell Name (simplified)

```
public void IntroduceName(Range toName, string name) {
    // Check if name already exists
    if(toName.Workbook.Names.Contains(name)) {
      // Abort
    }

    toName.Name = name;

    var dependents = toInline.dependents;
    if(dependents.Count == 0) {
      // Abort, no dependants
    }

    var AstToRepl = Parse(toName.Address);
    var AstReplacement = Parse(name);

    foreach(var dependent in dependents) {
      var AstOriginal = Parse(dependent.Formula);

      var AstNew = Replace(AstOriginal, AstToRepl, AstReplacement, /*...*/);

      dependent.Formula = AstNew.Print();
    }

}
```

Listing A.6: Group References Refactoring (simplified)

```
public Node GroupReferences(Node formula) {
  // Get all nodes representing varargs functions
  var targets = formula.AllNodes().Where(IsVarargsFunction);

  foreach(Node function in targets) {
    // split arguments that can be grouped from those than can't
    var toGroupArguments = function.arguments
      .Where(node => node.IsCellOrRange());
    var ignoredArguments = function.arguments
      .Where(node => !node.isCellOrRange());

    var grouped = new List<Node>();

    // Several characteristics define whether references can be grouped
    foreach(var sheetGroup in SplitByWorksheet(toGroupArguments)) {
      foreach(var toGroup in SplitByAbsoluteMarkers(sheetGroup)) {
        grouped.Add(GroupUsingExcel(toGroup));
      }
    }

    function.arguments = ignoredArguments.Concat(grouped);
  }

  return formula;
}
```

Listing A.7: Introduce Aggregate (simplified)

```
public Node IntroduceAggregate(Node formula) {

  // Precondition: formula is a function
  if(!formula.IsFunction()) {
    // Abort
  }

  // Precondition: formula is an operator that has an aggregate equivalent
  var op = formula.GetFunctionName();
  if(!AggregateEquivalents.ContainsKey(op)) {
    // Abort
  }

  var arguments = new List<Node>();
  var current = formula;

  // Gather arguments while the right subtree remains the same operator
  while(current.GetFunctionName() == op){
    arguments.Add(current.LeftArgument);
    current = current.RightArgument;
  }
  arguments.Add(current.RightArgument);

  return new Function(AggregateEquivalents[op], arguments);
}

private static Dictionary<string,string> AggregateEquivalents =
  new Dictionary<string,string>() {
    { "+", "SUM" },
    { "*", "PRODUCT" },
    { "&", "CONCATENATE" }
  };
```

Listing A.8: Introduce Conditional Aggregate (simplified)

```
public void IntroduceConditionalAggregate(Cell subject) {

  Node function = Parse(subject.Formula);

  // Check if we can perform the refactoring
  if(!IsTargetFunction(function)) {
    if(function.IsFunction() && function.FunctionName == "+") {
      // Rewrite + to SUM
      function = IntroduceAggregate(Ast);
    } else {
      // Abort
    }
  }

  // Check if all arguments are references to a single cell
  var arguments = function.arguments;
  if(argument.Any(arg => !IsSingleCellReference(arg))) {
    // Abort
  }

  // Logic for all argument in a single row is ommited for brevity,
  // it is identical but transposed.

  // Check if all arguments are in a single column
  var summedColumn = arguments.First().Select(cell => cell.Column);
  if(arguments.Select(arg => arg.Column).Any(col => col != summedColumn)) {
    // Abort
  }

  var summedRows = argument.Select(cell => cell.Row).ToList();

  // Get all non-empty columns in the worksheet
  var columns = GetNonEmptyColumns(subject.Worksheet);

  // Find a functional determiner
  // See the listing on the next page for the code of FindDeterminerColumn
  var determiner = FindDeterminerColumn(columns, summedRows);

  if(determiner == null) {
    // Abort, there are no candidate columns
  }

  // Create the SUMIF(Column, Value, SummedColumn)
  var AstNew = new Function(function.FunctionName + "IF", new List<Node>() {
      new ColumnRange(determiner.Item1),
      new AstString(determiner.Item2),
      new ColumnRange(summedColumn)
    }
  );

  subject.Formula = AstNew.Print();
}
```

Listing A.9: FindDeterminerColumn method

```
private static Tuple<string,string> FindDeterminerColumn(IEnumerable<Column>
    columns, IEnumerable<Row> summedRows) {
  // Return the first column that is a determiner of the summed column
  return columns.Select(column => {
    // Check if there is a value that is the same in all
    // the corresponding rows of the column
    string candidateValue = column[summedRows.First()].Value;

    if(summedRows.Any(row => column[row.ID].Value != candidateValue)) {
      return null;
    }

    // Value is the same in all corresponding rows
    // Now check if it is different in all other rows
    if(column.rows.Except(summedRows)
      .Any(cell => cell.Value == candidateValue)) {
      return null;
    }

    // We found a candidate column
    return Tuple.Create(column.ID, candidateValue);
  })
  .Where(found => found != null)
  .FirstOrDefault();
}
```

Appendix B

A Grammar for Spreadsheet Formulas Evaluated on Two Large Datasets

The following is a verbatim copy of the paper as it was published in the proceedings of SCAM 2015.

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, which 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 is compatible with the spreadsheet formula language, is compact enough to feasibly implement with a parser generator, and produces parse trees aimed at 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 uncommon 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, spreadsheets have been the topic of research since the nineties [4]. Recent research has often focused on analyzing and visualizing 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 involve formula parsing. 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 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.

To make a grammar suitable for this goal, 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 most common and well known formula structures and implemented it using a parser generator. Then 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. We used 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 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 in the datasets.

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 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).

978-1-4673-7529-0/15 © 2015 IEEE SCAM 2015, Bremen, Germany

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 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.

B. Sheet and External References and DDE

By default references point to cells or ranges in the same sheet as the formula, but 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 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`.

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 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`.

III. Spreadsheet Formula Grammar

For previous and ongoing research the authors needed a grammar for Microsoft Excel spreadsheet formulas with 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) 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: first, it is over 30 pages long and contains hundreds of production rules and thus fails requirement 3. Second, because of the detail of the grammar 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 1(a): the relatively simple formula `SUM(B2,5)` results in a 37-node tree with a depth of 18 nodes.

For these reasons 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 ambiguity discussed in Section III-F, we implemented this grammar using a LALR(1) parser generator. The present ambiguity can be solved by defining operator precedence (section III-D) and manually resolving conflicts (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. Characters are defined as unicode characters x9,xA,xD and x20 and upwards.

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 (e.g. '%', '!') are directly defined in the production rules in Figure 2 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

TABLE I: Lexical tokens used in the grammar

Token Name	Description	Contents	Priority
BOOL	Boolean literal	TRUE | FALSE	0
CELL	Cell reference	$? [A-Z]+ $? [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	(Any entry from the function list[1]) \(	5
FILE	External file reference	\[[0-9]+ \]	5
HORIZONTAL-RANGE	Range of rows	$? [0-9]+ : $? [0-9]+	0
MULTIPLE-SHEETS	Multiple sheet references	(($\square_2$+ : $\square_2$+)| ' ($\square_3$ | '')+ : ($\square_3$ | '')+ ')) !	1
NR	Named range	[A-Z_\\][A-Z0-9_.$\square_1$]*	-2
NR-PREFIXED	Named range which starts with a string that could be another token	(TRUE | FALSE | [A-Z]+[0-9]+) [A-Z0-9_.$\square_1$]+	3
NUMBER	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	'\[[0-9]+ \] ($\square_3$ | '')+ '!	5
REF-FUNCTION	Excel built-in reference 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	($\square_2$+ | ' ($\square_3$ | '')+ ') !	5
STRING	String literal	" ([^ "] | "")* "	0
UDF	User Defined Function	(_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 ' * [ ] \ : / ?

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

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 of 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]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.

C. Syntactical Analysis

The complete production rules of our grammar in Extended BNF syntax are listed in Figure 2. 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 1(b).

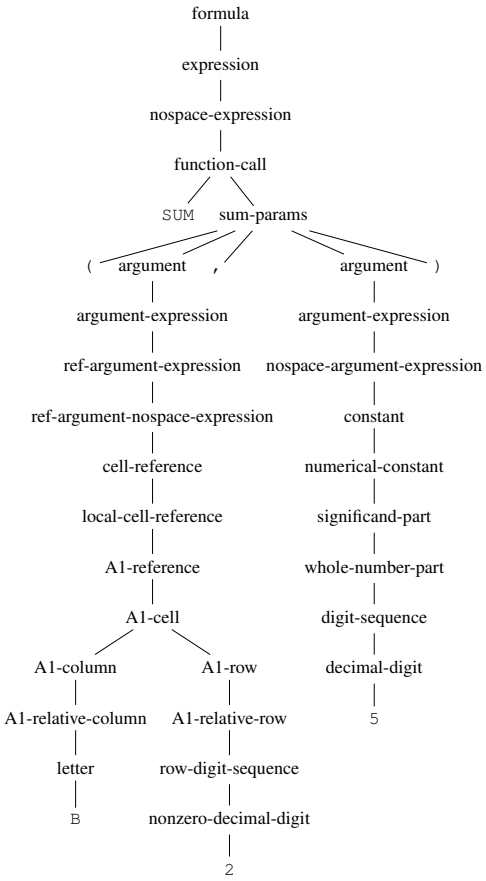
⟨*Formula*⟩ and ⟨*Reference*⟩ are the two most important production rules. These are also illustrated as syntax diagrams, with most production rules expanded, in Figures 3 and 4.

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 II-C), or reserved names (`=_xlnm.Print_Area`). Function calls invoke actual named (built-in or user defined) functions or operators applied to one or more formulas.

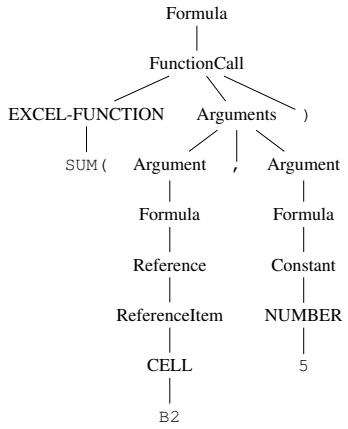
The ⟨*Reference*⟩ rule covers all types of referencing expressions, which are diverse. The simple case of a reference to a cell range can be expressed in any of the following ways:

```
  SUM(A1:A2)              = SUM(A1,A2)
= SUM(Sheet!A1:A2)        = SUM((A1,A2))
= SUM(Sheet!A1:(A2))      = SUM(A1:A2:A1)
= SUM('Sheet'!A1:A2)      = SUM(A1:A2 A:A)
= SUM(namedRangeA1A2)
```

The ⟨*Reference*⟩ rule, as shown in Figure 4, supports internal (in the same or in different sheets), or external single cell



(a) Microsoft Excel parse tree, constructed based on reference [14]



(b) Parse tree produced using the grammar defined in this paper

Fig. 1: Parse trees for formula `SUM(B2,5)`

references, cell ranges, horizontal and vertical ranges, named ranges and reference-returning, built-in or user-defined, functions.

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 as listed in Table II.

E. 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 $5 \cdot a$.

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) ⟨*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.

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 detrimental to ease of implementation (Design goal 3).

⟨*Start*⟩ ::= ⟨*Constant*⟩
| '=' ⟨*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

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

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

⟨*NamedRange*⟩ ::= NR | NR-PREFIXED

⟨*Union*⟩ ::= '(' ⟨*Reference*⟩ { ',' ⟨*Reference*⟩ } ')'

⟨*ConstantArray*⟩ ::= '{' ⟨*ArrayColumns*⟩ '}'

⟨*ArrayColumns*⟩ ::= ⟨*ArrayRows*⟩ { ';' ⟨*ArrayRows*⟩ }

⟨*ArrayRows*⟩ ::= ⟨*ArrayConst*⟩ { ',' ⟨*ArrayConst*⟩ }

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

Fig. 2: Production rules

TABLE II: Operator precedence in formulas

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

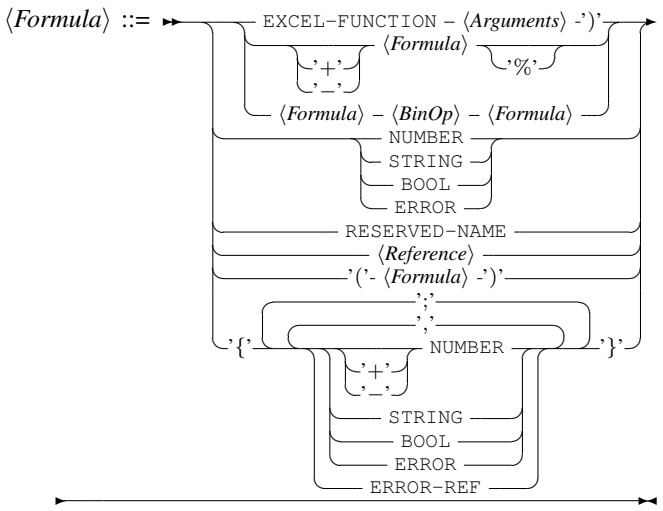


Fig. 3: Syntax diagram of the ⟨*Formula*⟩ production rule with most production rules expanded

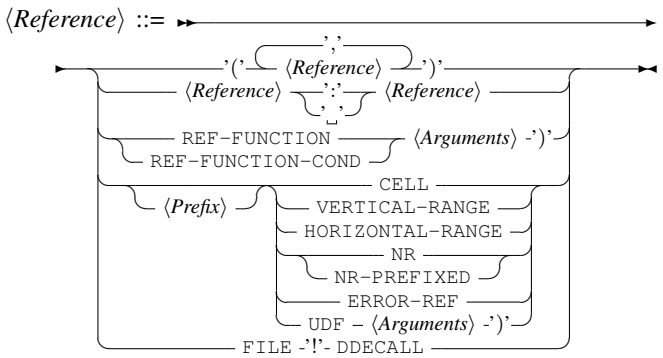


Fig. 4: Syntax diagram of the ⟨*Reference*⟩ production rule with most production rules expanded

*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 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 IV-B) and, in the datasets we used, all but two were with parentheses (Section IV-A).

IV. Evaluation

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

The grammar is implemented using the Irony parser generator framework[2] and the resulting parser, named XLParser, is available for download[3]. An online demo is also available.[4]

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[5]. 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 use it for parsing a total of 1,035,586 unique formulas. These originate from 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 in the Gembox library. In total, the 19,601 spreadsheets that were processed from the two datasets include 22,310,406 formula cells with 1,035,586 unique formulas —89,266 from the EUSES and 946,320 from the Enron dataset.

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 SSD.

Out of the 1,035,586 unique formulas from the two datasets that were used as input to the parser, 1,035,558 (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 the resulting parse trees are suited for analysis and manipulation, having only 19 types of non-terminal nodes.

A. Unparsable Formulas

The 28 formulas that were not parsed using the grammar defined in Section III 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 III-G2.
- `=+Ë%` was included in an Enron file that we assume to be either corrupt or another type of binary file, as the file is indecipherable.
- 25 formulas that are not returned correctly from the Gembox library. For example our tool reads and attempts to parse the formula `IF(=7,AVERAGE(C4:C11),0)` and fails, but in reality the formula is `IF(B8=7,AVERAGE(C4:C11),0)` which can be parsed. All these 25 cases are parsed successfully when we manually provide them as input to the parser.

B. Grammar Analysis

In this section we describe an analysis of the formulas in the datasets and measure the frequency of their characteristics. We also identify potentially smelly grammatical constructs and rare syntactical cases.

1) Formulas and Functions: Table III shows how frequently each of the production rules occurs in the formulas of the two datasets. Jointly, 84.91% of the formulas include a function call. Built-in value-returning functions are invoked by 35.82% of the formulas. A significant amount of formulas (286,210 or 1.28%) invoke user-defined functions—e.g., `=[1]!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 0.57% of the formulas.

Operators are used in 66.74% of the formulas, with binary operators being the most common ones, appearing in 59.77% of the formulas. Analyzing the utilization of constants, we find that 39.14% of the formulas contain at least one; more than one third (35.18%) of the formulas contain a number and 11.60% are formulas that contain text. Reserved names are uncommon, with 271 occurrences of the `_xlnm.Print_Area` and 5 occurrences of `_xlnm.Database`.

[2]https://irony.codeplex.com/

[3]https://github.com/PerfectXL/XLParser

[4]http://xlparser.perfectxl.nl/demo

[5]http://www.gemboxsoftware.com/

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

Syntax	Example	Unique formulas		Total formulas	
⟨*Formula*⟩	`=1+2`	**1,035,586**		**22,310,406**	
⟨*Reference*⟩	`=E9/E10`	962,783	92.97%	22,131,002	**99.20%**
CELL	`=A5`	951,521	91.88%	22,021,833	98.71%
⟨*FunctionCall*⟩	`=SUM(A5:A22)`	701,626	67.75%	18,944,204	**84.91%**
⟨*BinOp*⟩	`=H10-H8`	397,580	38.39%	13,333,844	**59.77%**
⟨*Constant*⟩	`=A5+134`	271,585	26.23%	8,731,489	**39.14%**
EXCEL-FUNCTION	`=SUM(A5:A22)`	264,673	25.56%	7,991,329	35.82%
NUMBER	`=(B8/48)*15`	250,085	24.15%	7,849,495	**35.18%**
⟨*Prefix*⟩	`=Sheet1!B1`	337,727	32.61%	5,599,011	**25.10%**
⟨*RefFunctionName*⟩	`=SUM(J9:INDEX(J9:J41,B43))`	55,680	5.38%	5,349,237	**23.98%**
SHEET	`=Sheet1!B1`	303,981	29.35%	5,282,386	23.68%
REF-FUNCTION-COND	`=IF(A1<0,0,1)`	50,171	4.84%	4,872,661	**21.84%**
⟨*Reference*⟩ ':' ⟨*Reference*⟩	`=SUM(A5:A22)`	184,451	17.81%	3,735,005	**16.74%**
⟨*UnOpPrefix*⟩	`=+B11+1`	218,397	21.09%	3,289,326	14.74%
STRING	`=IF(AD3<0,"buy","sell")`	56,635	5.47%	2,587,971	**11.60%**
⟨*NamedRange*⟩	`=SUM(freq)`	20,686	2.00%	1,645,120	**7.37%**
BOOL	`=IF(AND(R11=1,R14=TRUE),G19,0)`	7,532	0.73%	1,183,798	5.31%
FILE	`=[11]Sheet1!C5`	104,892	10.13%	1,135,185	**5.09%**
REF-FUNCTION	`=SUM(J9:INDEX(J9:J41,B43))`	9,907	0.96%	778,056	**3.49%**
QUOTED-FILE-SHEET	`=('[2]Detail I&E'!D62)/1000`	33,781	3.26%	325,498	1.46%
UDF	`=SQRT(_eoq2(C5,C4,C6,C7))`	21,352	2.06%	**286,210**	**1.28%**
(' ⟨*Reference*⟩ ')'	`=(2*(B29))/(1+B29)`	6,394	0.62%	266,420	1.19%
_xll.	`=_xll.RiskTriang(F9,F7,F8)`	11,922	1.15%	127,348	**0.57%**
ERROR-REF	`=AVERAGE(#REF!)`	3,477	0.34%	123,447	**0.55%**
VERTICAL-RANGE	`=COUNT(A:A)`	851	0.08%	55,254	**0.25%**
FILE '!'	`=[1]!today`	2,040	0.20%	**28,448**	**0.13%**
ERROR	`=IF(R14=TRUE,G19,#N/A)`	379	0.04%	27,237	**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 ranges	`=SUM(I8:K8:M8)`	367	0.04%	**8,581**	**0.04%**
DDECALL	`=TWINDDE|RSFRec!'NGH2 NET.CHNG'`	3,276	0.32%	**3,686**	0.02%
Intersection	`=Ending_Inventory Jan`	304	0.03%	**2,343**	0.01%
MULTIPLE-SHEETS	`=SUM(Sheet1:Sheet20!I29)`	173	0.02%	**1,986**	**0.01%**
Prefixed right ref. limit	`=SUM('Tot-1'!$B8:'Tot-1'!B8)`	147	0.01%	**1,501**	**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%
RESERVED_NAME	`=C23/_xlnm.Print_Area`	70	0.01%	**276**	0.00%
FILE MULTIPLE-SHEETS	`=SUM([2]Section3A:formulas!B11)`	4	0.00%	189	0.00%
⟨*ConstantArray*⟩	`=FVSCHEDULE(1,0.09;0.11;0.1)`	15	0.00%	**19**	0.00%

Regarding function arguments, spreadsheet systems allow empty arguments (e.g. `=SUM(,E35,E37)`) but this is rarely done—in only 0.05% of the formulas. Unions are found in only 385 formulas, e.g. `=LARGE((F38,C38),1)`. All occurrences were arguments of the `LARGE` and `SMALL` functions—these two 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. 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 array formulas 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 two datasets.

2) References: 99.2% of the formulas in the two datasets contain at least one ⟨*Reference*⟩, and 25.10% of these contain a reference that is not local, since it includes a ⟨*Prefix*⟩. External file references exist in almost 5.09% of the formulas. 16.74% of the formulas include a reference to a `':'` separated cell range. Named ranges exist in 7.37% of formulas. 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 reference errors are more than four times as common as all other types of errors combined—the `ERROR` token exists in 0.12% of the formulas.

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

File-only external references
: External references are normally in the form `[File]Sheet!Cell`. In 28,448 formulas (0.13%), however, the sheet is not specified, e.g. `=[2]!LastTrade`. These are cases of references to either external named ranges or external UDFs.

Multiple sheet references
: 1,986 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 189 formulas, the reference is to external files.

References to external UDFs
: 855 formulas (0.004%) contain references to external UDFs, for example `=[1]!SheetName()`.

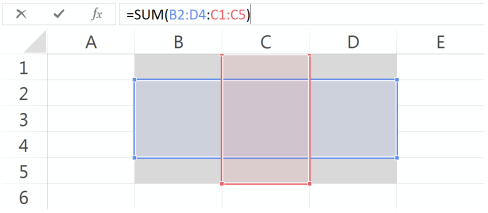
Prefixed right limits
: 1,501 formulas (0.01%) include a reference with a prefix in the right limit, e.g. `=SUM('Deals'!F9:'Deals'!F16)`. In all cases this prefix is identical to the first one, as continuous ranges spanning across multiple sheets are not supported by Excel. Still, this syntax is supported.

A special case in the grammar are the functions that always return references, namely the `INDEX`, `OFFSET` and `INDIRECT` functions 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)`. These reference returning functions are relatively common: they are found in 3.49% of the formulas, with the most common one being the `INDEX` (in 2.51% of formulas) and the least common one being the `INDIRECT` (in 0.2%). 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 like this 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.

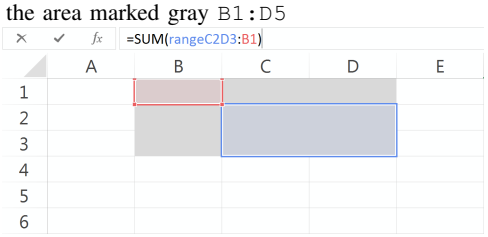
Another rare case of references are the dynamic data exchange links, which were found in 3,686 formulas. These take the form of `=Program|Topic!Arguments`, e.g. `=Database|TableA!Column1`, and are used in Windows versions of Microsoft Excel to receive data from other applications.

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

By *complex ranges* we mean ⟨*Reference*⟩s that include more than two or different types of `':'` separated ⟨*ReferenceItem*⟩s.



(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`

Fig. 5: Examples of references to complex ranges

An example is range `B2:D4:C1:C5`, illustrated in Figure 5a. 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 5b. 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. Complex ranges are rare: 8,581 formulas (0.04%) include complex ranges in the Enron dataset, and they are all defined using three cell locations.

The *intersect operator* is included in this discussion because it is `␣`, a single whitespace. An advantage of this approach is that it enables more natural language definition of intersections, e.g. `=SUM((Total_Cost Jan):(Total_Cost Apr.))`. However, we find this representation inconsistent to the grammar, because whitespace does not carry meaning in the rest of the language. Other spreadsheet systems do not use whitespace for this operator, either by using an alternative like LibreOffice which uses `!` or simply not supporting it. In the two datasets intersection operations are rare, as they are found in only 2,343 formulas (0.01%).

V. Discussion and Limitations

The currently defined formula grammar is able to parse 99,99% of the 1,035,586 unique formulas in the EUSES and the Enron datasets. In this section, we discuss a variety of issues that affect its applicability and suitability.

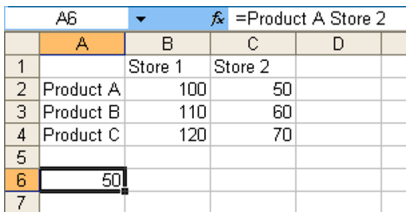


Fig. 6: A natural language formula in Excel 2003

A. Dialects

While other spreadsheet programs (e.g. Numbers, LibreOffice, Google Sheets) have generally adopted the Excel formula syntax, there are slight differences between programs and even Excel versions. The grammar has been designed as a generically as possible and has been enriched to include all syntactical features found in the two datasets. Both datasets, however, contain spreadsheets created in, or converted to, the Excel 2007 format. This limits the grammar support for language elements 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 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 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 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 6 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 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 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.

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 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 incorporate any of these limits.

D. 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.

VI. Related Work

Efforts to reverse-engineer language characteristics based on existing artifacts have been successful for other languages, including COBOL [16] and C, C++, C# and Java [17].

Most related to our research on the spreadsheet formula language 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, unions, named ranges and operator precedence. An extension of the same grammar was used to refactor formulas by Hermans and Dig [9].

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 [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 also analyze spreadsheet formulas but do not detail which grammar or analysis method they use.

VII. Conclusion

In this paper we (1) present a grammar for spreadsheet formulas, (2) evaluate it against over one million unique formulas, successfully parsing 99.99%, and (3) use it to analyze the formulas in the dataset, measure the frequency of their characteristics and find uncommon grammatical cases.

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[6] is published as open-source software and an online demo is also available.[7]

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 whether two context-free grammars are equivalent is undecidable, but in practice several techniques have been successfully used for this purpose [20], [21].

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 *Proc. of 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] ——, "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 *FSE 2014*, 2014, pp. 747–750.

[10] S. Badame and D. Dig, "Refactoring meets spreadsheet formulas," in *Proc. of ICSM 2012*. 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.

[12] B. Klimt and Y. Yang, "The enron corpus: A new dataset for email classification research," in *Machine Learning: ECML 2004*. Springer Berlin Heidelberg, 2004, vol. 3201, pp. 217–226.

[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] ——, "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 *Proc. of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications. Springer verlag*, 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 *Proc. of 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 *Proc. of ICSME'14*. IEEE Computer Society, 2014, pp. 111–120.

[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.

[6] https://github.com/PerfectXL/XLParser

[7] http://xlparser.perfectxl.nl/demo

End of appendix B