Eco: A Language Composition Editor

Lukas Diekmann and Laurence Tratt

Software Development Team, Informatics, King's College London
`http://lukasdiekmann.com/` `http://tratt.net/laurie/`

Abstract. Language composition editors have traditionally fallen into two extremes: traditional parsing, which is inflexible or ambiguous; or syntax directed editing, which programmers dislike. In this paper we extend an incremental parser to create an approach which bridges the two extremes: our prototype editor 'feels' like a normal text editor, but the user always operates on a valid tree as in a syntax directed editor. This allows us to compose arbitrary syntaxes while still enabling IDE-like features such as name binding analysis.

1 Introduction

At its most flexible, language composition gives programmers the ability to use multiple programming languages within a single file (e.g. in this paper we compose HTML, Python, and SQL). Editing composed programs has previously required choosing between two extremes: parsing-based approaches are familiar to programmers, but are either inflexible or prone to ambiguity; whereas SDEs (Syntax Directed Editors) have neither problem, but are insufferably awkward to use [13]. Recent work (e.g. [12,18]) has somewhat ameliorated the limitations of both extremes, but the divide between them, and the inevitable trade-offs, have long been assumed fundamental.

In this paper, we present a fundamentally new approach to editing composed programs which aims for the best of both worlds: it has the 'feel' of parsing-based approaches with the generality of syntax directed editors. The core of our approach is to extend an incremental parser with the new notion of *language boxes*.[1] Incremental parsers parse text as the user types, continuously updating a parse tree. In our approach, when editing a program in language X, one can insert at any place a language box for language Y and edit inside the box (in language Y) or outside the box (in language X). Each box has a separate incremental parser that maintains its own parse tree. Language boxes thus allow arbitrary syntaxes to be composed together without the loss of flexibility or ambiguity problems of traditional text-based approaches. Language boxes may contain any number of language boxes, and can be nested arbitrarily deep. Unlike syntax directed editors, our approach provides a user experience that is virtually identical to a traditional text editor. If only textual languages are used, the only noticeable

[1] Our 'language boxes' should not be confused with the modular language definition concept of the same name from [19].

difference while editing – and a small one at that – is when entering or exiting a language box. The only significant difference from traditional editors is that *Eco* has to save files out as a tree structure rather than as a traditional source file to avoid (re)parsing problems.

Since most programming is currently done in text, our main focus has been on finding a good solution to the long-standing problem of editing textual programs. However, language boxes are not restricted to textual languages: each language box has its own editor which need not be based on parsing – or text – at all.

Our approach is embodied in a prototype language composition editor *Eco*. *Eco* allows users to define composed languages and edit programs against those composed languages. As well as extending an incremental parser with language boxes, we have also added the ability to parse indentation based languages, and to incrementally create ASTs (Abstract Syntax Trees) from parse trees (allowing to easily implement a simple name binding analysis). The version of *Eco* described in this paper can be downloaded from:

`http://soft-dev.org/pubs/files/eco/`

This paper's contributions are as follows:

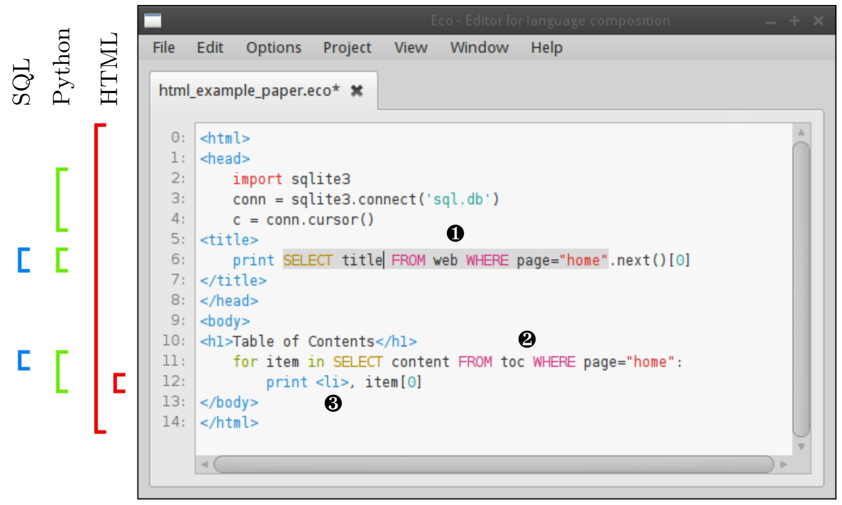
1. We extend an incremental parser with language boxes.
2. We show that the resulting editor is useful for textual language composition.
3. We extend the parser to incrementally parse indentation-based languages.
4. We extend the parser to incrementally create ASTs as well as parse trees.
5. We show that language boxes allow the composition of textual and non-textual languages.

An earlier version of this work, with a simple version of language boxes only, was published in workshop form [6]. This paper extends the concept substantially, including new techniques such as incremental parsing of indentation based languages, and incremental ASTs.

This paper is structured as follows. We first introduce the paper's running example (Section 2) before exploring the existing extremes in language composition editing (Section 3). We then introduce Wagner's incremental parser and our implementation of it (Section 5) before introducing language boxes (Section 6). We then extend the incremental parser to parse indentation-based languages (Section 7) and to incrementally create ASTs (Section 8). Finally, we briefly explain how *Eco* supports name binding and non-textual languages (Section 9).

2 Running example

We use as our running example a composition of HTML, Python, and SQL, leading to the construction of a flexible system equivalent to 'pre-baked languages' like PHP. In essence, we show how a user can take modular languages, compose them, and use the result in *Eco* as shown in Figure 1. We outline how this example composition is defined and used from the perspective of a 'normal' end-user; the rest of the paper is devoted to explaining the techniques which make this use case possible, as well as explaining how important corner cases are dealt with.

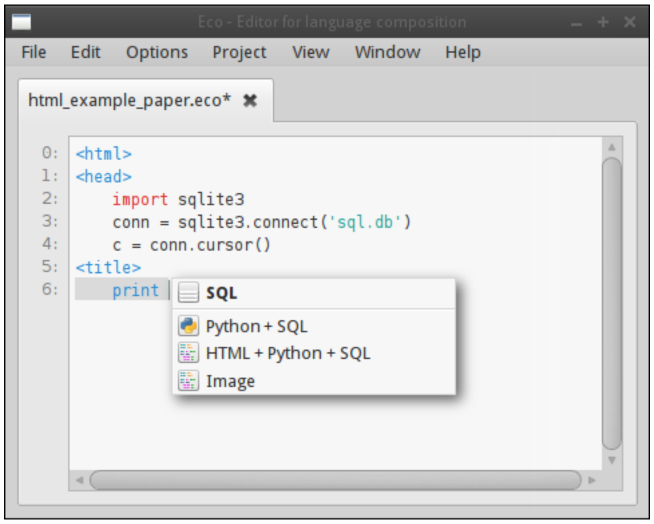


```
<html>
<head>
    import sqlite3
    conn = sqlite3.connect('sql.db')
    c = conn.cursor()
<title>
    print SELECT title FROM web WHERE page="home".next()[0]
</title>
</head>
<body>
<h1>Table of Contents</h1>
    for item in SELECT content FROM toc WHERE page="home":
        print <li>, item[0]
</body>
</html>
```

Fig. 1: *Eco* editing a composed program. An outer HTML document contains several Python language boxes. Some of the Python language boxes themselves contain SQL language boxes. Some specific features are as follows. ❶ A highlighted (SQL) language box (highlighted because the cursor is in it). ❷ An unhighlighted (SQL) language box (by default *Eco* only highlights the language box the cursor is in, though users can choose to highlight all boxes). ❸ An (inner) HTML language box nested inside Python.

When an end-user creates a new file in *Eco*, they are asked to specify which language that file will be written in. Let us assume that they choose the composed language named (unimaginatively) HTML+Python+SQL which composes the modular HTML, Python, and SQL languages within *Eco*. Although users can write whatever code they want in *Eco*, this composed language has the following syntactic constraints: the outer language box must be HTML; in the outer HTML language box, Python language boxes can be inserted wherever HTML elements are valid (i.e. not inside HTML tags); SQL language boxes can be inserted anywhere a Python statement is valid; and HTML language boxes can be inserted anywhere a Python statement is valid (but one can not nest Python inside such an inner HTML language box). Each language uses our incremental parser-based editor.

From the user's perspective, their typical workflow for a blank document is to start typing HTML exactly as they would in any other editor: they can add, alter, remove, or copy and paste text without restriction. The HTML is continually parsed by the outer language box's incremental parser and a parse tree constructed and updated appropriately within the language box. Syntax errors are highlighted as the user types with red squiggles. The HTML grammar is a standard BNF grammar which specifies where Python+SQL language boxes are

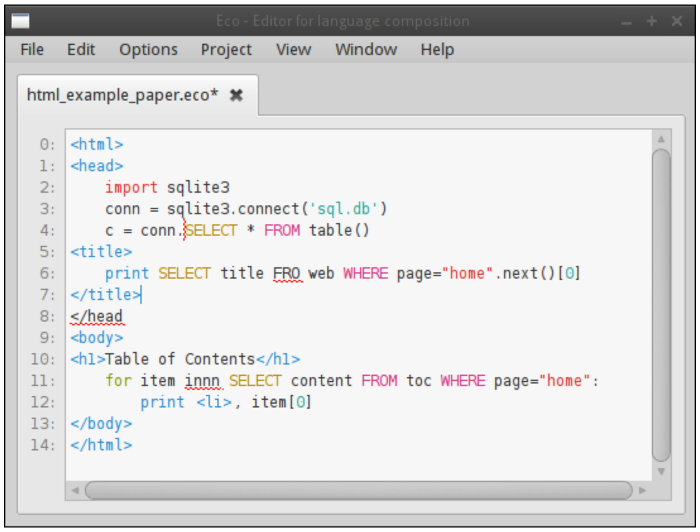


```
<html>
<head>
    import sqlite3
    conn = sqlite3.connect('sql.db')
    c = conn.cursor()
<title>
    print
```

Fig. 2: Inserting a language box opens up a menu of the languages that *Eco* knows about. Languages which *Eco* knows are valid in the current context are highlighted in bold to help guide the user.

syntactically valid by referencing a separate, modular Python grammar. When the user wishes to insert Python code, they press Ctrl+L, which opens a menu of available languages (see Figure 2); they then select Python+SQL from the languages listed and in so doing insert a Python language box into the HTML they had been typing. The Python+SQL language box can appear at any point in the text; however, until it is put into a place consistent with the HTML grammar's reference to the Python+SQL grammar, the language box will be highlighted as a syntax error. Note that this does not affect the user's ability to edit the text inside or outside the box, and the editing experience retains the feel of a normal text editor. As Figure 3 shows, *Eco* happily tolerates syntactic errors – including language boxes in positions which are syntactically invalid – in multiple places.

Typing inside the Python+SQL language box makes it visibly grow on screen to encompass its contents. Language boxes can be thought of as being similar to the quoting mechanism in traditional text-based approaches which use brackets such as ⟦ ⟧; unlike text-based brackets, language boxes can never conflict with the text contained within them. Users can leave a language box by clicking outside it, using the cursor keys, or pressing Ctrl+Shift+L. Within the parse tree, the language box is represented by a token whose type is Python+SQL and whose value is irrelevant to the incremental parser. As this may suggest, conceptually the top-level language of the file (HTML in this case) is a language box itself. Each language box has its own editor, which in this example means each has an incremental parser.



```
 0: <html>
 1: <head>
 2:     import sqlite3
 3:     conn = sqlite3.connect('sql.db')
 4:     c = conn.SELECT * FROM table()
 5: <title>
 6:     print SELECT title FRO web WHERE page="home".next()[0]
 7: </title>
 8: </head
 9: <body>
10: <h1>Table of Contents</h1>
11:     for item innn SELECT content FROM toc WHERE page="home":
12:         print <li>, item[0]
13: </body>
14: </html>
```

Fig. 3: Editing a file with multiple syntax errors. Lines 6, 8 and 11 contain syntax errors in the traditional sense, and are indicated with horizontal red squiggles. A different kind of syntax error has occurred on line 4: the SQL language box is invalid in its current position (indicated by a vertical squiggle).

At the end of the editing process, assuming that the user has a file with no syntax errors, they will be left with a parse tree with multiple nested language boxes inside it as in Figure 1. Put another way, the user will have entered a composed program with no restrictions on where language boxes can be placed; with no requirement to pick a bracketing mechanism which may conflict with nested languages; with no potential for ambiguity; and without sacrificing the ability to edit arbitrary portions of text (even those which happen to span multiple branches of a parse tree, or even those which span different language boxes).

Eco saves files in a custom tree format so that, no matter what program was input by the user, it can be reloaded later. In the case of the HTML+Python+SQL composition, composed programs can be exported to a Python file and then executed. Outer HTML fragments are translated to print statements; SQL language boxes to SQL API calls (with their database connection being to whatever variable a call to `sqlite3.connect` was assigned to); and inner HTML fragments to strings. All of the syntactically correct programs in this paper can thus be run as real programs. For the avoidance of doubt, other syntactic compositions, and other execution models of composed programs are possible (see e.g. [1]) and there is no requirement for *Eco* compositions to be savable as text, nor executed.

3 Parsing and syntax directed editing

In this section we briefly explain the two extremes that bound the overall design space that we work within.

3.1 Parsing-based approaches

While there are many possible approaches to parsing text, three approaches can be used as exemplars of the major categories: LR, generalised, and PEG parsing.

Due to Yacc's predominance, LR-compatible grammars are commonly used to represent programming languages. Indeed, many programming language grammars are deliberately designed to fit within LR parsing's restrictions. Unfortunately, composing two LR grammars does not, in general, result in a valid LR grammar [17]. One partial solution to this is embodied in Copper which, by making the lexer lazy and context-sensitive, is able to allow many compositions which would not normally seem possible in an (LA)LR parser [21]. However, this requires nested languages to be delineated by special markers, which is visually obtrusive and prevents many reasonable compositions.

Generalised parsing approaches such as [24] can accept any CFG (Context Free Grammar), including inherently ambiguous grammars. Ambiguity and programming language tools are unhappy bedfellows, since the latter can hardly ask of a user "which parse of many did you intend?" Unfortunately, ambiguity, once allowed through the door, is impossible to eject. Two unambiguous grammars, when composed, may become ambiguous. However, we know that the only way to determine CFG ambiguity is to test every possible input; since most CFGs describe infinite languages, determining ambiguity is undecidable [4]. Although heuristics for detecting ambiguity exist, all existing approaches fail to detect at least some ambiguous grammars [23]. Furthermore, scannerless parsers – those which intertwine tokenization and parsing, and which are the most obviously suited for language composition – introduce an additional form of ambiguity due to the longest match problem [20].

PEGs (Parsing Expression Grammars) are a modern update of a classic parsing approach [8]. PEGs have no relation to CFGs. They are closed under composition (unlike LR grammars) and are inherently unambiguous (unlike generalised parsing approaches). Both properties are the result of the *ordered choice* operator e_1 `/` e_2 which means "try e_1 first; if it succeeds, the ordered choice immediately succeeds and completes. If and only if e_1 fails should e_2 be tried." However, this operator means that simple compositions such as `S ::= a / ab` fail to work as expected, because if the LHS matches `a`, the RHS is never tried, even if it could have matched the full input sequence. To make matters worse, in general such problems can not be determined statically, and only manifest when inputs parse in unexpected ways.

In summary, when it comes to language composition, parsing approaches are either too limited (LR parsing), allow ambiguity (generalised parsing), or are hard to reason about (PEG parsing). While approaches such as Copper [21] and

Spoofax [12] have nonetheless been used for some impressive real-world examples, we believe that such issues might limit uptake.

3.2 Syntax directed editing

SDE works very differently to traditional parsing approaches, always operating on an AST. AST elements are instantiated as templates with holes, which are then filled in by the user. This means that programs being edited are always syntactically valid and unambiguous (though there may be holes with information yet to be filled in). This side-steps the flaws of parsing-based approaches, but because such tools require constant interaction with the user to instantiate and move between AST elements, the SDE systems of the 70s and 80s (e.g. [22]) were rejected by programmers as restrictive and clumsy [13].

More recently, the MPS editor has relaxed the SDE idiom, making the entering of text somewhat more akin to a normal text editor [18]. In essence, small tree rewritings are continually performed as the user types, so that typing $\boxed{2}$, $\boxed{\text{Space}}$, $\boxed{+}$, $\boxed{\text{Space}}$, $\boxed{3}$ transparently rewrites the `2` node to be the LHS of the `+` node before placing the cursor in the empty RHS box of the `+` node where `3` can then be entered in. This lowers, though doesn't remove, one of the barriers which caused earlier SDEs to disappear from view. Language authors have to manually specify all such rewritings, a tedious task. Furthermore, the rewritings only affect the entry of new text. Editing a program still feels very different from a normal text editor. For example deleting nodes requires great care and special actions. Similarly, only whole nodes can be selected from the AST. For example, one can not copy `2 +` from the expression `2 + 3` on-screen.

Put another way, MPS is sometimes able to hide that it is a SDE tool, but never for very long. The initial learning curve is therefore relatively steep and unpalatable to many programmers.

4 The outlines of a new approach

Our starting hypothesis is that language composition needs an editing approach which can marry SDE's flexible and reliable approach to constructing ASTs with the 'feel' of text editing. In part due to MPS's gradual evolution from a pure SDE to an approach which partially resembles parsing, we decided to start from a parsing perspective and try and move towards SDE. Doing so implicitly rules out any approach which can accept ambiguous grammars. Since the largest class of unambiguous grammars we can precisely define is the LR(k) grammars [14] they were the obvious starting point.[2] In the following sections, we show how one can take an incremental parser which accepts LR grammars and extend it with the notion of language boxes.

[2] Though note there are unambiguous grammars that are not contained within LR(k).

5 Incremental parsing in *Eco*

Traditional parsing is a batch process: an entire file is fed through a parser and a parse tree created. Incremental parsing, in contrast, is an online process: it parses text as the user types and continually updates a parse tree. A number of incremental parsing algorithms were published from the late 70s [9] to the late 90s, gradually improving efficiency and flexibility [16,7]. The last major work in this area was by Wagner [25] who defined a number of incremental parsing algorithms. We use his LR-based incremental parser which has two major benefits: it handles the full class of LR(k) grammars; and has formal guarantees that the algorithm is optimal. In this section, we give a brief overview of our implementation of Wagner's algorithm.

As with other parsing approaches, our implementation consists of both an incremental lexer and incremental parser. We represent both lexer and grammar with notations that are roughly similar to Yacc. Lexer rules are considered in the order in which they are defined to avoid longest-match ambiguities. Grammars are defined in BNF notation.

Both the lexer and the parser operate on a parse tree. Parse tree nodes are either *non-terminals* (representing production rules in the grammar) or *tokens* (representing terminal symbols). Non-terminals are immutable and have zero or more ordered child nodes. Tokens have an immutable type (e.g. 'int') and a mutable value (e.g. '3'). The minimal parse tree consists of three special nodes: a *Root* non-terminal; and *BOS* (Beginning of Stream) and *EOS* (End of Stream) terminals (both children of *Root*). All nodes created from user input are (directly or indirectly) children of *Root* and are contained between *BOS* and *EOS*.

When the user types, the incremental lexer first either creates, or updates, tokens in the parse tree. The lexer considers where the cursor is in the tree (i.e. where the user is typing) and uses look-ahead knowledge stored in the surrounding tokens to work out the affected area of the change. Newly created tokens are then merged back into the tree. In the simple case where a token's value, but not its type, was changed, no further action is needed. In all other cases, the incremental parser is then run to update the parse tree correctly. All nodes on the path from the changed token to the root of the tree are marked as changed. The incremental parser then starts at the beginning of the tree and tries to reorder the parse tree. Assuming the user's input is syntactically valid, non-terminals are created or removed, as appropriate. The parser tries to reuse non-changed sub-trees as is. Since non-terminals are immutable, sub-trees which can't be reused must be recreated from scratch or cloned from existing nodes.

Syntactically incomplete programs lead to temporarily incorrect parse trees. In such cases, the incremental parser typically attaches tokens to a single parent. When the user eventually creates a syntactically valid program, the tree is rewritten (an example for this can be seen in Figure 4).

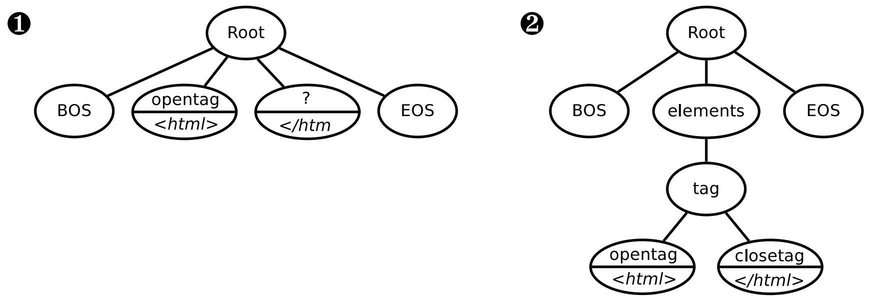


Fig. 4: Parse trees in the process of editing. Non-terminals are represented by ellipses with a name. Tokens are represented by ellipses with a horizontal line; the token's type is above the line; its value below the line. ❶ A parse tree in the process of editing and currently syntactically incorrect. The incremental lexer is able to tell that `</htm` can not be part of the previous token, but is currently unsure what the type of this token should be. The parser is thus not able to order the tokens into a correct tree. ❷ After further editing, the input is syntactically correct. The incremental lexer has been able to determine the type of the `</html>` token and the incremental parser has been able to update the parse tree, inserting appropriate non-terminals as specified by the grammar.

5.1 Whitespace

In most programming languages, whitespace (which, from this paper's perspective, also includes comments) is only important inasmuch as it separates other tokens. Traditional lexers therefore consume and discard whitespace. This is unacceptable in our approach, as we need to maintain whitespace in the parse tree to accurately render the user's input (see Section 6.3). We therefore adopt, with small variations, one of Wagner's suggestions for whitespace handling.

When an *Eco* grammar sets the `%implicit_whitespace=true` flag, the grammar is automatically mutated such that references to a production rule `ws` are inserted before the first, and after every, terminal in the grammar. Although the user can define `ws` to whatever they want, a common example of what is added to the grammar and lexer is as follows:

```
ws ::= TABSSPACES
     |

TABSSPACES : [ \t]+
```

Note that the user need not handle newlines as *Eco* handles those separately (see Sections 6.3 and 7).

Although the resulting parse tree records `ws` nodes (which are used for rendering and for ensuring cursor behaviour works as expected), they soon clutter visualizations of parse trees to the point that one can no longer see anything else. In the rest of this paper, we therefore elide `ws` nodes from all parse trees.

6 Language boxes

Language boxes allow users to embed one language inside another (see Section 2). Language boxes have a type (e.g. HTML), an associated editor (e.g. our extended incremental parser), and a value (e.g. a parse tree). By design, language boxes only consider their own contents ignoring parent and sibling language boxes. We therefore define the notion of the CST (Concrete Syntax Tree), which is a language box agnostic way of viewing the user's input. Different language box editors may have different internal tree formats, but each exposes a consistent interface to the CST. Put another way, the CST is a global tree which integrates together the internal concrete syntax trees of individual language boxes.

In the rest of this section, we examine the characteristics, and consequences, of language boxes.

6.1 Language modularity

To make language boxes practical, languages need to be defined modularly. *Eco* allows users to define as many languages as they wish. Languages are defined modularly, and may have several sub-components (e.g. grammar, name binding rules, syntax highlighting). For example, a language L which uses the incremental parser editor will contain a BNF grammar which can reference another language M by adding a symbol `<M>` to a production rule.

In most cases, we believe that users will want to avoid hard-coding references to different languages into 'pure' grammars. We therefore allow grammars to be cloned and (during initialisation only) mutated automatically. The most common mutation is to add a new alternative to a recently loaded grammar. For example, if we have a reference to `python` and `sql` languages, we can create a reference from Python to SQL by executing `python.add_alternative("atom", sql)`.

6.2 Language boxes and incremental parsing

Language boxes fit naturally with the incremental parser because we use a property of CFGs which is rarely of consequence to batch-orientated parsers: parsers only need to know the type of a token and not its value. In our incremental parser approach, nested language boxes are therefore treated as tokens. When the user inserts an SQL language box into Python code, a new node of type `SQL` is inserted into the parse tree and treated as any other token. From the perspective of the incremental parser for the Python code, the language box's value is irrelevant as is the fact that the language box's value is mutable. Language boxes can appear in any part of the text, though, in our example, an SQL language box is only syntactically valid in places where the Python grammar makes a reference to the SQL grammar. Nested language boxes which use the incremental parser have their own complete parse trees, as can be seen in Figure 5.

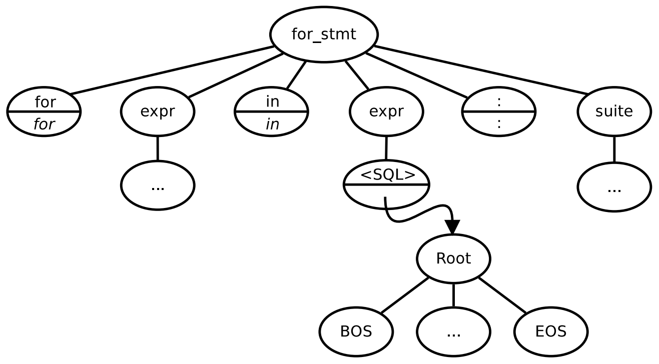


Fig. 5: An elided example of an SQL language box nested within an outer Python language box. From the perspective of the incremental parser, the tree stops at the SQL token. However, we can clearly see in the above figure that the SQL language box has its own parse tree, which thus forms part of the wider CST.

6.3 Impact on rendering

While language boxes do not have any impact on the incremental parser, they do have a big effect on other aspects of *Eco*. One obvious change is that they break the traditional notion that tokens are n characters wide and 1 line high. Language boxes can be arbitrarily wide, arbitrarily high, and need not contain text at all. *Eco* cannot simply store text 'flat' in memory and render it using traditional text editing techniques. Instead, it must render the CST onto screen. However, efficiency is a concern. Even a small 19KiB Java file, for example, leads to a parse tree with almost 19,000 nodes. Rendering large numbers of nodes soon becomes unbearably time-consuming.

To avoid this problem, *Eco* only renders the nodes which are currently visible on screen. *Eco* treats newlines in the user's input specially and uses them to speed up rendering. Similar to Harrison [10], *Eco* maintains a list of all lines in the user's input; whenever the user creates a newline, a new entry is added. Each entry stores a reference to the first CST node in that line and the line's height. Entries are deleted and updated as necessary. Scanning this list allows *Eco* to quickly determine which chunks of the CST need to be rendered, and which do not. Even in our simple implementation, this approach scales to tens of thousands LoC without noticeable lag in rendering.

6.4 Cursor behaviour

In a normal editor powered by an incremental parser, cursor behaviour can be implemented as in any other editor and stored as a *(line#, column#)* pair. We initially took this approach for *Eco*, but it has an unacceptable corner-case: nested language boxes create 'dead zones' where it is impossible to place the cursor and to enter further text.

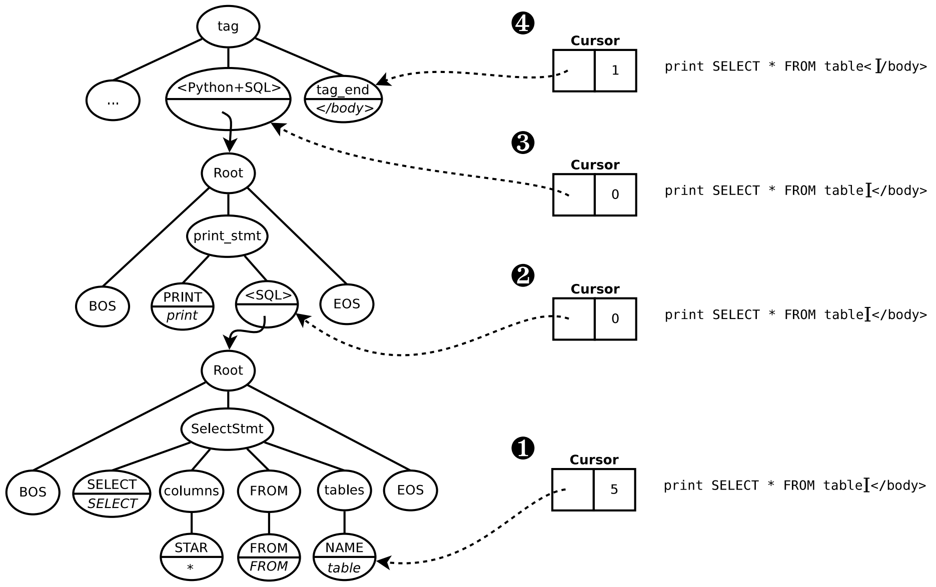


Fig. 6: *Eco*'s cursor behaviour in a program nesting SQL inside Python inside HTML. The cursor is stored as a (node, offset) pair. ❶ In normal program editing, the cursor behaves exactly like any other editor. Typing with the cursor at this position will enter text into the SQL language box right after the `table` token. ❷ After pressing Ctrl+Shift+L, the cursor attaches itself to the current node's language box (`<SQL>`). Typing with the cursor at this position will insert text into the Python+SQL language box between the tokens `<SQL>` and `EOS`. ❸ After pressing Ctrl+Shift+L again, typing will insert text into the HTML outer language box (after the Python+SQL language box, and before the `</body>` token). ❹ Assuming the cursor was as in position ❶ and the user pressed →, the cursor will be moved to this position.

Our solution is simple: *Eco*'s cursor is relative to nodes in the CST. In textual languages, the cursor is a pair *(node, offset)* where *node* is a reference to a token and *offset* is a character offset into that token. In normal usage, the arrow keys work as expected. For example, when the cursor is part way through a token, → simply increments *offset*; when *offset* reaches the end of a token, → sets *node* to the next token in the parse tree and *offset* to 1. ↑/↓ is slightly more complex: *Eco* scans from the beginning of the previous / next line, summing up the width of tokens until a match for the current x coordinate is found.

At the end of a nested language box, pressing → sets *node* to the next token after the language box while setting *offset* to 1 as described above. This means that if two language boxes end at the same point on screen, *Eco* will seemingly skip over the outer of the two boxes, making it impossible to insert text at that point. If instead the user presses Ctrl+Shift+L, the cursor will be set to the current language box token itself instead of the first token after the language box (since language boxes are tokens themselves, this adds no complexity to *Eco*). When the user starts typing, this naturally creates a token in the outer language box. In this way, *Eco* allows the user to edit text at any point in a program, even in seemingly 'dead' zones (see Figure 6 for a diagrammatic representation).

6.5 Copy and paste

Eco allows users to select any arbitrary fragment of a program, copy it, and paste it in elsewhere. Unlike an SDE, *Eco* does not force selections to respect the underlying parse tree in any way. Users can also select whole or partial language boxes, and can select across language boxes. *Eco* currently handles all selections by converting them into 'flat' text and reparsing them when they are pasted in. This seems to us a reasonable backup solution since it is hard to imagine what a user might expect to see when a partial language box is pasted in. However, we suspect that some special-cases would be better handled separately: for example, if a user selects an entire language box, it would be reasonable to copy its underlying tree and paste it in without modification.

7 Indentation-based languages

Indentation-based languages such as Python are increasingly common, but require more support than a traditional lexer and parser offer. Augmenting batch-orientated approaches with such support is relatively simple, but, to the best of our knowledge, no-one has successfully augmented an incremental parser before. In this section we therefore describe how we have extended an incremental parser to deal with indentation-based languages.

The basic problem can be seen in this simplified Python grammar fragment:

```
if    ::= IF expr : suite
suite ::= NEWLINE INDENT stmts DEDENT
stmts ::= stmts NEWLINE stmt
        | stmt
```

and an example code fragment using it:

```
if a > 0:
    a = 0
print a
```

We can not simply parse this text and consume all whitespace, as in most languages. Instead, line 2 should generate `NEWLINE` and `INDENT` tokens before the `a` token and a `DEDENT` token after the `0`. The process to create these tokens must be mindful of nesting: if a `while` statement is nested at the end of an `if`, two `DEDENT` tokens must be generated at the same point. Note that indentation related tokens are solely for the parser's benefit and do not affect rendering. Whitespace is recorded as per Section 5.1 and rendered as normal.

```
def calc_indentl(l):
    if prev(l) == None:
        l.indentl = 0
    elif prev(l).wsl == l.wsl:
        l.indentl = prev(l).indentl
    elif prev(l).wsl < l.wsl:
        l.indentl = prev(l).indentl + 1
    else:
        assert prev(l).wsl > l.wsl
        prevl = prev(prev(l))
        while prevl != None:
            if prevl.wsl == l.wsl:
                l.indentl = prevl.indentl
                return
            elif prevl.wsl < l.wsl:
                break
            prevl = prev(prevl)
        mark_unbalanced(l)
```

Fig. 7: The indentation level calculation algorithm.

7.1 Incrementally handling indentation

Eco lexers that set `%indentation=true` use our approach to incrementally handling indentation. We insert an additional phase between incremental lexing and parsing which looks at changed lines and inserts or removes indentation related tokens as appropriate. To make this possible, we extend the information stored about each line in *Eco* (see Section 6.3) to store the leading whitespace level (i.e. the number of space characters) and the indentation level. These notions are separated, because the same indentation level in two disconnected parts of a file may relate to different leading whitespace levels (e.g. in one `if` statement, 2 space characters may constitute an indentation level; in another, 4 space characters). For example, the following is valid Python:

```
if x:
  y
if a:
    b
```

However, the following fragment is *unbalanced* (i.e. the file's indentation is nonsensical) and should be flagged as a syntax error:

```
if x:
    a
  b
```

For the purposes of this paper, it is sufficient to consider changes to a single line, though *Eco* itself generalises this to simultaneous changes on multiple lines. When a line l is updated, there are two cases. If l's leading whitespace level has not changed, no further recalculations are needed. In all other cases, the indentation level of l, and all lines that depend on it, must be recalculated; indentation related tokens must then be added or removed to each line as needed. Dependent lines are all non-empty lines after l up to, and including, the first line whose leading whitespace level is less than that of l, or to the end of the file, if no such line exists.

We can define a simple algorithm to calculate the indentation level of an individual line l. We first define every line to have attributes `wsl` – its leading whitespace level – and `indentl` – its indentation level. `prev(l)` returns the first non-empty predecessor line of l in the file, returning `None` when no such line exists. The algorithm is shown in Figure 7. There are 4 cases, the first 3 of which are trivial, though the last is more subtle:

1. Lines 2–3: If `prev(l) == None` then l is the first line in the file and its indentation level is set to 0.
2. Lines 4–5: If `prev(l).wsl == l.wsl` then l is part of the same block as the previous line and should have the same indentation level.
3. Lines 6–7: If `prev(l).wsl < l.wsl` then l opens a new block and has an indentation level 1 more than the preceding line.
4. Lines 9–16: If `prev(l).wsl > l.wsl` then either l closes a (possibly multi-level) block or the overall file has become unbalanced. To determine this we have to search backwards to find a line with the same leading whitespace level as l. If we find such a line, we set l's indentation level to that line's level (lines 12–14). If no such line is found (line 11), or if we encounter a line with a lower leading whitespace level (lines 15–16), then the file is unbalanced and we need to mark the line as such (line 18) to force *Eco* to display an error at that point in the file.

In practise, this algorithm tends to check only a small number of preceding lines (often only 1). The worst cases (e.g. an unbalanced file where the last line is modified and all preceding lines are checked) are $O(n)$ (where n is the number of lines in the file).

Each time a line has been affected by this process, we need to check whether the indentation related tokens in the parse tree match the line's current state. If they do not, the tokens in the parse tree need to be updated appropriately (i.e. the old tokens are removed and replaced). If a line is marked as unbalanced, it requires a single `UNBALANCED` token; otherwise, we compare a line with its first non-blank predecessor and calculate the correct number of `INDENT` / `DEDENT` tokens. Once the parse tree has the correct number of tokens, we rely on the incremental parser to reorder the tree appropriately.

8 Abstracting syntax trees

Eco's CST allows it to fully render a program on-screen. Because of this, it contains details that make analysis of the CST painful. For example, we would like to define analyses such as the names in scope in a program (which we can then use to highlight undefined variables, and to code complete names; see Section 9.1) on a tree which abstracts away irrelevant detail. *Eco* therefore maintains an AST which provides a simplified view of the user's data. Different language editors map from the CST to the AST in different ways. Since some editors' data may be non-abstractable, formally the AST contains a non-strict subset of the data in the CST.

In this section, we explain how this relates to the incremental parser. Parse trees in our approach are an extreme example of the pain of a detailed CST:

their nesting is partly dictated by the LR parser, and is often very deep; they contain irrelevant tokens, which are necessary only for the parser or to make the language more visually appealing to users; and child nodes are ordered and only accessible via numeric indices. Instead, one would prefer to work with an AST, where the tree has been flattened as much as possible, with irrelevant tokens removed, and with child nodes unordered and addressable by name.

We first describe the simple (relatively standard) rewriting language *Eco* uses to create ASTs from parse trees. We then describe the novel technique we have developed to make AST updates incremental.

8.1 Rewriting language

The simple rewriting language we use to create ASTs from parse trees is in the vein of similar languages such as TXL [5] and Stratego [3]. In essence, it is a pure functional language which takes parse trees as input and produces ASTs as output. Each production rule in a grammar can optionally define a single rewrite rule. AST nodes have a name, and zero or more unordered, explicitly named, children. The AST is, in effect, dynamically typed and implicitly defined by the rewrite rules.[3]

An elided example from the Python grammar is as follows:

```
1   print_stmt ::= PRINT                {Print(stmts=[])}
2                | PRINT stmt_loop      {Print(stmts=#1)}
3
4   stmt_loop  ::= stmt_loop stmt       {#0 + [#1]}
5                | stmt                 {[#0]}
6
7   stmt       ::= expr                 {#0}
8                | ...
9
10  expr       ::= VAR                  {Var(name=#0)}
11               | ...
```

AST constructors are akin to function calls. Expressions of the form `#n` take the nth child from the non-terminal that results from a grammar's production rule. Referencing a token uses it as-is in the AST (e.g. line 10); referencing a non-terminal uses the AST sub-tree that the non-terminal points to. For example, `Var(name=#0)` means "create an AST element named `Var` with an edge `name` which points to a `VAR` token" and `Print(stmts=#1)` means "create an AST element named `Print` with an edge `stmts` which points to the AST constructed from the `stmt_loop` production rule". A common idiom is to flatten a recursive rule (forced on the grammar author by the very nature of LR grammars) into a list of elements (lines 4 and 5). Note that a rewrite rule can produce more than one AST node (e.g. line 1 produces both a `Print` node and an empty list node).

8.2 Incremental ASTs

All previous approaches of which we are aware either batch create ASTs from parse trees or use attribute grammars to perform calculations as parsing is per-

[3] This is not an important design decision; the AST could be statically typed.

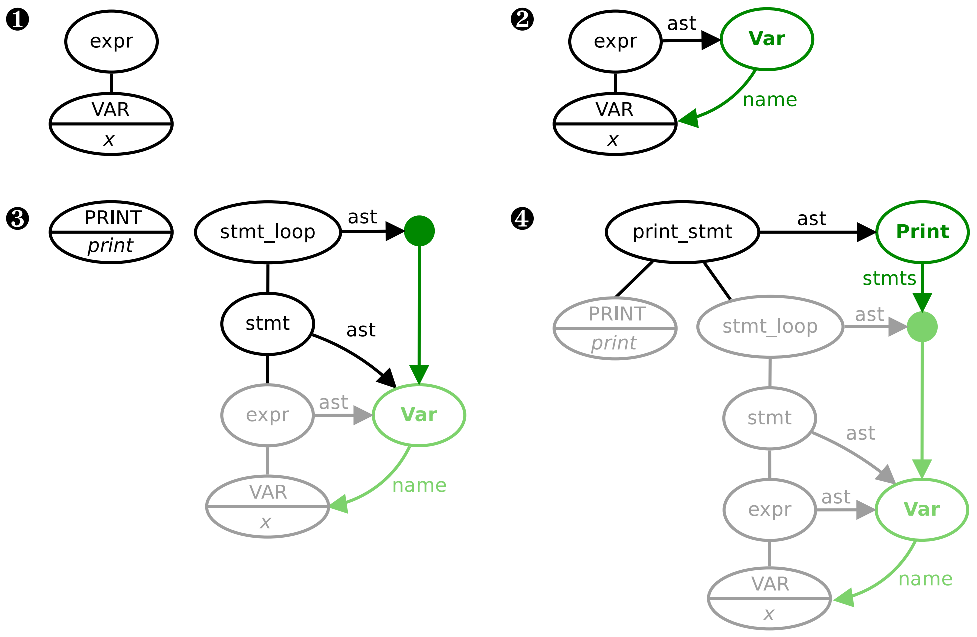


Fig. 8: Incremental AST construction, with the parse tree shown in black and the AST in green. Subtrees that have been reused are in grey / light green. ❶ After typing the input `x`, the incremental parser creates this parse tree fragment. ❷ After the `expr` non-terminal is created, the rewrite language is run on it creating an `ast` reference to an AST node `Var`. ❸ After changing the input to `print x`, the incremental parser starts to update the parse tree and the associated AST as shown in this in-process fragment. The `stmt` production's rewrite rule simply references whatever AST node its child produces, so `stmt`'s `ast` reference is the existing `Var` node. `stmt_loop` however wraps its contents in a list (the green circle). ❹ The final parse tree and AST. The `print` production rule creates a `Print` AST element with a child `stmts` which is a list containing a `Var` node.

formed (e.g. [2]). In this subsection, we explain how Wagner's incremental parser can be easily extended to incrementally create ASTs.

Our mechanism adds a new attribute `ast` to non-terminals in the parse tree. Every `ast` attribute references a corresponding AST node. The AST in turn uses direct references to tokens in the parse tree. In other words, the AST is a separate tree from the parse tree, except that it shares tokens directly with the parse tree. Sharing tokens between the parse tree and the AST is the key to our approach since it means that changes to a token's value automatically update the AST without further calculation. Altering the incremental parser to detect changes to tokens would be far more complex.

In all other cases, we rely on a simple modification to the incremental parser. Non-terminals are created by the parser when it reduces one or more elements from its stack. Every altered subtree is guaranteed to be reparsed and, since non-terminals are immutable, changed subtrees will lead to fresh non-terminals being created. We therefore add to the parser's reduction step an execution of the corresponding production rule's rewrite rule; the result of that execution then forms the `ast` reference of the newly created non-terminal. We then rely on two properties that hold between the parse tree and AST trees. First, the AST only consists of nodes that were created from the parse tree (i.e. we do not have to worry about disconnected trees within the AST). Second, the rewrite language cannot create references from child to parent nodes in the AST. With these two properties, we can then guarantee that the AST is always correct with respect to the parse tree, since the incremental parser itself updates the AST at the same time as the parse tree. Figure 8 shows this process in action.

This approach is easy to implement and also inherits Wagner's optimality guarantees: it is guaranteed that we update only the minimal number of nodes necessary to ensure the parse tree and AST are in sync.

9 Other features

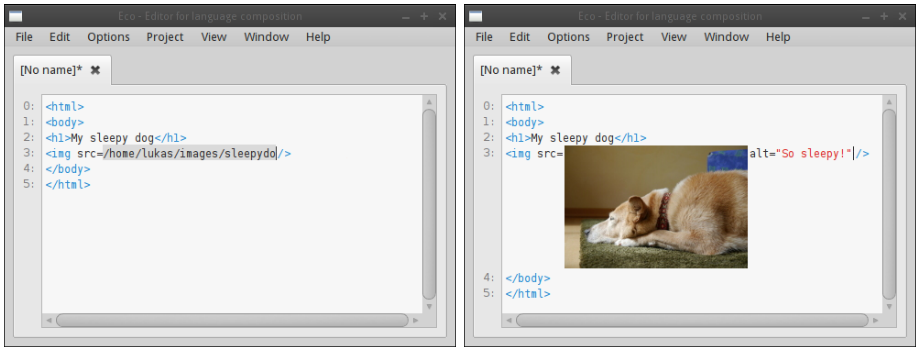
9.1 Scoping rules

Modern IDEs calculate the available variable names in a source file for code completion, and highlight references to undefined names. We have implemented (a subset of) the NBL approach [15] which defines a declarative language for specifying such scoping rules. This runs over the AST created by Section 8. References to undefined variables are highlighted with standard red squiggles. Users can request code completion on partially completed names by pressing Ctrl + Space. Code completion is semi-intelligent: it uses NBL rules to only show the names visible to a given scope (e.g. variables from different methods do not 'bleed' into each other). We needed to make no changes to the core of *Eco* to make this work. We suspect that other analyses which only require a simple AST will be equally easy to implement.

9.2 Non-textual languages

Although this paper's main focus has been on textual languages, language boxes liberate us from only considering textual languages. As a simple example of this, the HTML language we defined earlier can use language boxes of type `Image`. Image language boxes reference a file on disk. When an HTML file is saved out, they are serialised as normal text. However, the actual image can be viewed in *Eco* as shown in Figure 9. Users can move between text and image rendering of such language boxes by double-clicking on them. The renderer correctly handles lines of changing heights using the techniques outlined in Section 6.3.

As this simple example may suggest, *Eco* is in some senses closer to a syntactically-aware word processor than it is a normal text editor. Although


Eco - Editor for language composition
File Edit Options Project View Window Help
[No name]*
Eco - Editor for language composition
File Edit Options Project View Window Help
[No name]*

```
<html>
<body>
<h1>My sleepy dog</h1>
<img src=/home/lukas/images/sleepydo/>
</body>
</html>
```

```
<html>
<body>
<h1>My sleepy dog</h1>
<img src=         alt="So sleepy!"/>
</body>
</html>
```

Fig. 9: An example of a non-textual language in *Eco*.

we have not explored non-textual languages in great detail yet, it is easy to imagine appropriate editors for such languages being embedded in *Eco* (e.g. an image editor; or a mathematical formula editor).

10 Conclusions

In this paper we presented a new approach to editing composed programs, which preserves the 'feel' of normal text editors, while having the power of syntax directed editors. The core of our approach is a traditional incremental parser which we extended with the novel notion of language boxes. We showed how an incremental parser can naturally incrementally create ASTs, allowing us to build on modern IDE features such as name binding analysis. All this is embodied in a prototype editor *Eco*, which readers can download and experiment with.

We divide possible future work into two classes. First are 'engineering issues'. For example, the incremental parser stops rewriting the tree after the first syntactic error, which can make editing awkward. Various solutions (e.g. [25,11]) have been proposed, and we intend evaluating and adjusting these as necessary. Second are 'exploration issues'. For example, we would like to embed very different types of editors (e.g. spreadsheets) and integrate them into the *Eco* philosophy. It is for the most part unclear how this might best be done.

Acknowledgements: This research was funded by Oracle Labs. Edd Barrett, Carl Friedrich Bolz, Darya Kurilova, and Samuele Pedroni gave insightful comments on early drafts. Michael Van De Vanter gave invaluable advice on editor technologies.

References

1. Edd Barrett, Carl Friedrich Bolz, and Laurence Tratt. Unipycation: A case study in cross-language tracing. In *VMIL*, pages 31–40, Oct 2013.

2. Marat Boshernitsan. Harmonia: A flexible framework for constructing interactive language-based programming tools. Master's thesis, University of California, Berkeley, Jun 2001.
3. Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1–2):52 – 70, 2008.
4. David G. Cantor. On the ambiguity problem of backus systems. *J. ACM*, 9(4):477–479, Oct 1962.
5. James R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190 – 210, 2006.
6. Lukas Diekmann and Laurence Tratt. Parsing composed grammars with language boxes. In *Workshop on Scalable Language Specifications*, Jun 2013.
7. Manuel Vilares Ferro and Bernard A Dion. Efficient incremental parsing for context-free languages. In *International Conference on Computer Languages*, pages 241–252, 1994.
8. Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL*, pages 111–122, Jan 2004.
9. Carlo Ghezzi and Dino Mandrioli. Incremental parsing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):58–70, 1979.
10. Michael A Harrison and Vance Maverick. Presentation by tree transformation. In *Compcon*, pages 68–73, Sep 1997.
11. Fahimeh Jalili and Jean H Gallier. Building friendly parsers. In *POPL*, pages 196–206, Jan 1982.
12. Lennart C.L. Kats and Eelco Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *OOPSLA*, pages 444–463, Oct 2010.
13. Amir Ali Khwaja and Joseph E. Urban. Syntax-directed editing environments: Issues and features. In *SAC*, pages 230–237, Feb 1993.
14. Donald Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, Dec 1965.
15. Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In *SLE*, pages 311–331. Oct 2013.
16. Warren X Li. A new approach to incremental LR parsing. *J. Prog. Lang.*, 5(1):173–188, 1997.
17. Rohit J. Parikh. On context-free languages. *J. ACM*, 13(4):570–581, Oct 1966.
18. Vaclav Pech, Alex Shatalin, and Markus Voelter. JetBrains MPS as a tool for extending Java. In *PPPJ*, pages 165–168, Sep 2013.
19. Lukas Renggli, Marcus Denker, and Oscar Nierstrasz. Language boxes. In *SLE*, pages 274–293, Oct 2009.
20. D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) parsing of programming languages. *SIGPLAN Not.*, 24(7):170–178, Jun 1989.
21. August Schwerdfeger and Eric Van Wyk. Verifiable composition of deterministic grammars. In *PLDI*, Jun 2009.
22. Tim Teitelbaum and Thomas Reps. The Cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, Sep 1981.
23. Naveneetha Vasudevan and Laurence Tratt. Detecting ambiguity in programming language grammars. In *SLE*, pages 157–176, Oct 2013.
24. Eelco Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, Sep 1997.
25. Tim A. Wagner. *Practical Algorithms for Incremental Software Development Environments.* PhD thesis, University of California, Berkeley, Mar 1998.