Generation of Incremental Parsers for Modern IDEs

Christoph Höger choeger@cs.tu-berlin.de

Technische Universität Berlin

Abstract. Tools in modern IDEs (*Integrated Developments Environment*) usually work on models derived from the abstract syntax tree. Because those models should be validated and reconciled in real time, parsing the source code is subject to strict limitations in runtime and memory. In this paper we provide a Java based approach to extend a special LR Parser with methods for incremental re-parsing a given text. We also show that while incremental parsers cannot be faster than batch-parsers in general (that is: for any given grammar), our implementation reaches $\mathcal{O}(log(n))$ in the median. We will conclude by showing some optimizations on the approach and measuring their influence on the runtime of the parser.

1 Introduction

This paper will start with a general overview or the parsergenerator that was extended to generate incremental parsers and a motivation for incremental parsing in general. Afterwards we will present the general algorithm, its implementation and some optimization. Finally we will give an example on how the incremental parser works on an example grammar.

1.1 Prerequisites and Conventions

In this paper we will talk about a common parsing technique called *shift-reduce* parsers. We assume the reader is familiar with those kind of parsers and formal languages in general.
As a notational convention when it comes to grammars, Terminals will be written *underlined*, words as small latin characters and Nonterminals big latin characters. A parser configuration is written as a tuple

$$([S_1, \ldots, S_n\$], [\underline{t_1}, \ldots, \underline{t_k}\$], [T_1, \ldots, T_n\$])$$

with S being the states on the state stack, $\underline{t}$ the terminals on the input stack (that may be written as words) and T the tokens on the token stack. The current stacks top is marked by a \$. Otherwise we use common mathematical notations.

1.2 Related Work

The automated creation of various parsers and their classification in terms of formal languages can be found in [3]. General informations about parsing and compiler techniques are described in depth in [2]. An optimal algorithm for incremental LR parsers is shown in [6].

1.3 PaGe

PaGe (**Pa**rser **Ge**nerator) is an implementation of an experimental LR parser generator developed at the TU-Berlin [5]. PaGe differs from classic LR parsers in two ways: First of all the state machine is not built by using the well known Sets of Items (as seen in e.g. [3]), but by transforming the grammar to directly represent the language of the viable prefixes. Secondly there are two kinds of special symbols added to the grammar: Action symbols and Pseudo Terminals. Both are used to keep track of reductions: An Action Symbol represents a distinct reduction to a Nonterminal while a Pseudo Terminal represents the productions that follow after a reduction had been applied.
After adding Action Symbols to every production in the grammar it is transformed into the so called Normalform where every production has the either the form: $N \rightarrow \alpha Z_i$ or $Z_j \rightarrow ⓝ$ where α is a Terminal or Nonterminal from the original grammar and ⓝ is the Action Symbol that was introduced for the reduction of the Nonterminal N. Note, that this transformation does not change the language of the grammar, because of the way the Z symbols are introduced: A rule $N \rightarrow \alpha\beta ⓝ$ would become $N \rightarrow \alpha Z_1$ with the two rules $Z_1 \rightarrow \beta Z_2$ and $Z_2 \rightarrow ⓝ$ added to the grammar.
In a next step the grammar is 'unfolded': Every original Nonterminal is replaced

$$
\begin{aligned}
E &\rightarrow V \underline{+} V[A]ⓔ \\
A &\rightarrow \underline{(}E\underline{)}ⓐ \\
V &\rightarrow X \\
&\quad | \ Y \\
X &\rightarrow \underline{x}ⓧ \\
Y &\rightarrow \underline{y}ⓨ
\end{aligned}
$$

Fig. 1. Input grammar G

with the right hand side of every production. This transformation is repeated until there is no original Nonterminal left. Additionally whenever a Nonterminal N is unfolded the original right hand side is left in the rule with N replaced by a Pseudoterminal of the same name. With the rules given above a production $M \rightarrow NZ_k$ would be transformed into $M \rightarrow \alpha Z_1 Z_k$ and $M \rightarrow \underline{N}Z_k$ with $\underline{N}$

being the newly generated Pseudoterminal. Again this transformation does not change the language of the grammar.
When every original Nonterminal has been unfolded, the unfold transformation

$$
\begin{array}{lll}
Z_3 \rightarrow EZ_\# & Y \rightarrow \underline{y}Z_9 & Z_6 \rightarrow \underline{Z_7} \\
Z_\# \rightarrow \oplus & Z_0 \rightarrow \underline{+}Z_{10} & Z_7 \rightarrow \textcircled{a} \\
E \rightarrow VZ_4 & Z_1 \rightarrow \underline{V}Z_{11} & Z_8 \rightarrow \textcircled{x} \\
A \rightarrow \underline{(}Z_5 & Z_2 \rightarrow AZ_{12} & Z_9 \rightarrow \textcircled{y} \\
V \rightarrow \underline{X} & \mid \textcircled{e} & Z_{10} \rightarrow Z_1 \\
\mid Y & Z_4 \rightarrow Z_0 & Z_{11} \rightarrow Z_2 \\
X \rightarrow \underline{x}Z_8 & Z_5 \rightarrow EZ_6 & Z_{12} \rightarrow \textcircled{e}
\end{array}
$$

Fig. 2. Normalized grammar G

is applied repeatedly together with the well known operations for left factorization and the removal of leftrecursion until every production starts with either a (Pseudo)Terminal or Action symbol. From this form the first and only language modification is done: Every rule is cut off after the first occurrence of a Nonterminal, yielding a grammar in which every rule has either the form: $Z_i \rightarrow \underline{t}Z_j$ or $Z_k \rightarrow \textcircled{n}$.
At this point one can easily create a LR parse table from the grammar: Every Nonterminal represents a state and its productions the state transitions. The former yield *shift* transitions and the latter *reduce* operations (the exact operation has been created when the Action symbol was introduced in the first place). The state that is to enter after a *reduce* operation is determined by the productions starting with Pseudoterminals (in those states that are on top of the state stack after the *reduce* was invoked).
Of course one has to populate the Action symbols with their respective lookahead before the parse table can be created but this can be done by examining the rules starting with Pseudoterminals and the rule elements that were cut off before.
Figure 1 shows an example grammar G. The transformation result of the normalization is shown in figure 2. Finally figure 3 shows the parse table as it was derived from the viable prefix grammar.

1.4 Incremental Parsing

In classic applications in compilers or interpreters a parser (called 'batch' parser) is invoked exactly once per input file. The produced AST is then handed over to next stage tools (e.g. semantic analysis) and will not be used again. Anyway it should be clear that in most scenarios the change that was made to an input file between two subsequent parser runs is very small. (Tools like `diff` and `patch` make use of that fact in diverse distributed software development processes.)
So to shorten the time of the parse run it seems natural to re-use the AST from

	$\underline{x}$	$\underline{y}$	$\underline{A}$	$\underline{X}$	$\underline{Y}$	$\underline{E}$	$\underline{(}$	$\underline{)}$	$\underline{+}$	$\underline{\#}$
$Z_{\#}$										⊕
Z_3	Z_8	Z_9		Z_4	Z_4	$Z_{\#}$				
Z_4									Z_{10}	
Z_5	Z_8	Z_9		Z_4	Z_4	Z_6				
Z_6								Z_7		
Z_7								ⓐ		ⓐ
Z_8							ⓧ	ⓧ	ⓧ	ⓧ
Z_9							ⓨ	ⓨ	ⓨ	ⓨ
Z_{10}	Z_8	Z_9		Z_{11}	Z_{11}					
Z_{11}			Z_{12}			Z_5	ⓔ			
Z_{12}								ⓔ		ⓔ

Fig. 3. Parse table

the latest invocation of the parser and only apply the small change to it. This implies that an incremental parser gets two input parameters: The list of input tokens (with the information which tokens represent the change) and an AST that was derived from the same list of input tokens without the change.
For general parsers it needs to be mentioned that the actual advantage of recycling an old AST depends on the grammar and the position of the changed tokens. Theorem 1 gives an example for a situation where incremental parsing yields no advantage at all.

Theorem 1. *The worst-case-complexity of an incremental parser is the same as of a batch parser.*

Proof. Let G_1 and G_2 be two distinct grammars with $G_1 = (N_1, T, P_1, S_1)$, $G_2 = (N_2, T, P_2, S_2)$ and $N_1 \cap N_2 = \emptyset$. The combined grammar $G_3 = (N_1 \cup N_2, T \cup \{\underline{g_1}, \underline{g_2}\}, P_1 \cup P_2 \cup \{S \to \underline{g_1}S_1, S \to \underline{g_2}S_2\}, S)$ will then accept the language of both grammars (with the Terminal $\underline{g_1}$ or $\underline{g_2}$ as first symbol).
If an input $\underline{g_1}e, e \in T^*$ with $e \in \mathbb{L}(G_1) \wedge e \in \mathbb{L}(G_2)$ is changed into $\underline{g_2}e$, no element of the abstract syntax tree could be reused since N_1 and N_2 are disjoint, which means the same operations are needed as if a batch parser was invoked. □

Although there is no general advantage for every given grammar over a batch parser it is clear that the incremental parser will perform much better when the change only affects a single node in the syntax tree. We will call a change *constrained* by a Nonterminal if the difference between the old AST and the new one is limited to a node that was introduced during a *reduce* operation for that Nonterminal. It is obvious that such a Nonterminal exists for every change since every grammar as a special start symbol S.
Though this approach obviously can save some parsing time it needs still another motivation: In a classic setup one could easily think of an incremental parser that would its resulting AST in a temporary file so it could be reused. But since reading and decoding the AST would also take $\mathcal{O}(n)$ with n the count of input tokens (the same runtime class a batch parser would typically need) there

would be no point in recycling any data whatsoever. This changes with modern integrated development environments (IDEs).

1.5 Application in the Eclipse IDE

Current IDEs like the Eclipse Java Development Tools (JDT) support the user with mighty tools for refactoring, error detection and automatic code completion. All those operations need the current input text to be parsed into an AST. Since this AST needs to be updated after each change as fast as possible (in fact this process called *reconciling* is invoked 500ms after each change per default) Eclipse delivers the perfect rationale for the implementation of an incremental parser. In addition the IDE will always know the changed region of the text since it has to be displayed in the first place. This allows a slightly faster lexing process (which basically bisects to the changed place, lexes the new tokens and reuses old tokens when possible).
When applied to an Eclipse editor an incremental parser also needs to fulfill two additional requirements: Firstly it should be as less space consuming as possible. Since every open file is represented as an AST in-memory any additionally bytes per node might render the whole IDE useless for real application scenarios. Secondly and less prominent, the parser should literally re-use the old AST from the root. Since Java is an object oriented language the models that are used inside the Eclipse API are often used to store additional informations and states among them (e.g. the expanded state in a tree view). Re-creating similar objects after every change would mean to loose those extra informations and be very irritating to the user.

2 Implementation of an Incremental Parser

A first naive approach on incremental parsing with a shift-reduce parser would be to save the configuration of the parser whenever a token has been shifted and to associate that configuration with the token. In the next run one could easily get the last unchanged token and re-create the configuration in constant time. Although such an algorithm would be relatively easy to implement and work very fast it has two major drawbacks: It does not cover the AST nodes that where built from tokens *behind* the change and, even worse, it wastes an enormous amount of memory. So this kind of algorithm (as it is described in [1] - although optimized there) would clearly fail our above mentioned objective.

2.1 Basic Concept

Although that idea (called "state matching") is useless for the application in IDEs its review introduces the concept of splitting the AST into nodes that are before the change and nodes behind it. More formally a node is called "before" the change if the action that constructed its right-most leaf removed only tokens

from the token stack that precede the changed tokens in the input list and called "behind" when the tokens succeeded the change for its left-most leaf. Those nodes can be reused as seen from theorem 2.

Theorem 2. *Let* $([S_1, \ldots, S_n\$], [w^{-1}v^{-1}\$], [N_1 \ldots N_n\$])$ *be a configuration that PaGe can enter and* $A \Rightarrow^* v$ *a production from the original grammar (and* w *prefixed with the lookahead for this production), then, if there is a state* S_p *with* $GOTO(S_n, \underline{A}) = S_p$*, PaGe will enter the configuration*
$([S_1, \ldots, S_n, S_p], [w^{-1}], [N_1, \ldots, N_n, A])$

Proof. From the existence of S_p *follows, that* $S_n \Rightarrow AS_p$ *must be part of the normalized grammar (because otherwise the Pseudoterminal* $\underline{A}$ *would not be in the GOTO table for* S_n*).*
Since all following shift/reduce operations and therefore the next states are determined by the lookahead (PaGe is deterministic) and $A \Rightarrow^* v$ *(with* w *being a valid lookahead), it follows that after* $|w|$ *shift operations and a finite amount of reductions (because in any given state there can only be a finite amount of reductions and there are only a finite amount of states on the stack) the parser will finally reduce* v *to* A *and enter the configuration* $([S_1, \ldots, S_n, S_p], [w^{-1}], [N_1, \ldots, N_n, A])$ *after finding* $S_p = GOTO(S_n, \underline{A})$ □

Obviously a third group of nodes (including the root node) exists, in which every token "overlaps" the change and thus has to be parsed for changes. Following this observation [6] presents another approach with no additional space requirements. The algorithm described here is similar to this optimal solution but makes use of PaGe's special features and has a much simpler design since it only processes one change at a time (which is caused by the eclipse environment which will always report changes one after the other).
The splitting of the AST naturally implies a tree-phase algorithm: Firstly recreate the state just before the changed tokens would be shifted by descending the "before" part of the AST, after setting the parser into its initial configuration. Since every node A covers a word v that has not been changed there must be a production $A \Rightarrow^* v$ in the grammar and theorem 2 applies (the validity of the configurations follows directly from the theorem and the fact that we start with the initial configuration that is always valid). So the parser is effectively short-circuited instead of going through all configurations. Secondly to handle the changed part the algorithm only invokes the parser until the constraining node has been restored and finally the algorithm reuses the behind part by ascending the AST from the input tokens.
The only problem that remains unsolved with this algorithm is the calculation of the constraining node. Fortunately this is not necessary when the second and third phase are merged together: The third phase ascension is always applied if possible (that is: the next input tokens are unchanged and their parent AST nodes match a Pseudoterminal that occurs in the *GOTO* set of the current state). So every time a normal parse step was applied successfully and the next Terminal would be shifted, this Terminals ancestor nodes (it has none if it is

affected by the change) are checked for direct reduction and the highest one that completely lies before the change is reused in a short circuit reduction similar to the one in phase one. Afterwards a batch parser step is invoked and so on.
This eliminates the need for an explicit calculation of the constraining node and also allows the recycling of nodes that had been created from a different parser configuration or are children of the constraining node - that way the incremental parser is reuses even more nodes than any state matching parser.

2.2 Corner Cases

Although the above algorithm is correct (it returns the same AST that the batch parser would return), it needs to be refined to correctly handle some corner cases which PaGe can handle in batch mode. The most obvious is the lookahead problem: Since PaGe is a *LR(n)* parser generator, the restriction in theorem 2 to a valid lookahead v can be very important. Since the current PaGe implementation generates lookahead only where needed, one could determine the used lookahead in every step of the incremental parser and include this into the validity check for the re-usage of a node. This method was not yet implemented due to the consideration of the expected additional costs. Instead the change will be expanded by the maximum length of lookahead used. With this simple step it is made sure that no token has changed that was used as a lookahead.
The next corner case that had to be handled was the existence of epsilon productions. Though PaGe's meta grammar does not allow direct epsilon productions, they are still possible by using the optional construct for the right hand side of a production (e.g $A \rightarrow [B]$). This forces the incremental parser to determine if an empty AST node E, that effectively has no Terminals as leafs, is affected by a given change (Just stopping the process as if the change was reached would also work but be very ineffective). Since the incremental parser also can not recreate the configuration that the batch parser had E was pushed the only possible solution is to simulate the batch parser by shifting tokens until a state S_f is reached with $\underline{E} \in GOTO(S_f)$. This is not needed in the second phase since the batch parser will recreate any empty nodes that are not directly reused.
Finally, the error handling mechanisms pf PaGe had to be handled. In the current implementation, PaGe will always skip erroneous tokens and not try to add any. (This is due to the fact that most languages are defined by lists of elements and the parser can easily ignore one of those elements while the AST structure remains correct.) This process is called *recovering*.
Since the incremental parser will get erroneous input quite often, it needs to handle it as well as the batch parser - only faster. To simplify this the first handling is left to the lexer: If there are old syntax errors far (that is: more than the used lookahead) tokens away from the change, the error tokens are hidden from the input. Otherwise the error and the change are simply merged. If the batch parser encounters an error in the second phase the error handling is left to it, since after a successful recovery the incremental parser can simply continue. Obviously there is some room for improvement here: The incremental parser could

check the AST for some sense of what element actually constrains the syntax error, but this remains object to further development.

```
private final void incrementalPhase1(Absy node, Range change) {
/*
 * before the change spot we can simply short circuit the shift/reduce
 * cycle
 */
  if (change.isBehind(node.getRange())) {
    reduce(node);
    skipInput(node);
  } else {
    // do they overlap?
    if (!node.getRange().isBehind(change)) {
      for (Absy child : node.getChildren()) {
        if (child.getRange().getStartToken() >= change.getStartToken())
          break;
        if (child.getRange().getStartToken() >= 0) {
          shiftIntermediateTerminals(child);
          incrementalPhase1(child, change);

          if (child.getRange().getEndToken() >= change.getStartToken())
            break;
        } else {
          /* handle empty productions */
          if (change.isBehind(parser.getInputStack().peek().getRange()))
            handleEpsilonAbsy(change, child);
        }}}}}
```

Fig. 4. Incremental Phase 1

2.3 Implementation and Tests

The incremental parser was implemented as an additional module that can be used with PaGe to not interfere with the current development (Especially to not introduce any new bugs).
Figure 2.2 shows the implementation of the first phase: Nodes that are before the change are reduced, others are descended and their children are handled, if they do are not completely behind the change.
The second phase implementation in figure 2.3 shows how the ascension is handled: After all possible actions are applied (to make sure the next token is really about to be shifted), the highest parent node that can be reused is put on the token stack effectively short circuiting a reduction. For the testing process using JUnit was an obvious choice. The test setup works on the library files from MOSILAB [4], an implementation of the Modelica language. For every source

```
boolean incrementalPhase2(Token token, Range change) throws ParserException {
  applyAllActions();

  Absy highest = getHighestValidParent(parser.getStateStack().peek(),
      token, change);
  if (highest != null) {
    // clean up any already shifted tokens
    for (int i = highest.getRange().getStartToken(); i < token
        .getRange().getStartToken(); i++) {
      parser.getTokenStack().pop();
      parser.getStateStack().pop();
    }
    reduce(highest);
    // remove the input tokens, we don't need anymore
    skipInput(highest);
    return true;
  }
  return false;
}
```

Fig. 5. Incremental Phase 2

file a Test case is created that runs the incremental parser after marking some terminals as changed and compares the output AST with that from the batch parser. For every iteration of that process the change is moved roughly 1% ahead on the input. Although the coverage may be bad since no real change is handled, this method has the benefit of allowing a good benchmark of the incremental parsing for constant changes.

2.4 Runtime and Space Complexity

As shown in Theorem 1 there can be no general advantage for every given grammar (or every given AST). So for any given change the constraining node C the runtime will not go below $\mathcal{O}(l_C)$ with l_C denoting the amount of input tokens covered by C.
Since in the first phase the tree is descended, this phase finishes in $\mathcal{O}(log(n))$ if we assume the AST being balanced with n nodes. Additionally after C has been reconstructed all of its right siblings (and its parents right siblings and so on) can be reused. Since the algorithm always searches for the *highest* reusable parent node in the second phase, the search for the highest valid parent will return nodes closer to the AST root at least all $\mathcal{O}(c_{max})$ steps with c_{max} being the maximum of children a node can have in the AST. Because the maximum distance a node can have in the AST is $\mathcal{O}(log(n)$, this leads to a runtime of $\mathcal{O}(c_{max}log(n)^2)$ for the second phase. Note, that c_{max} is not really a constant due to the existence of lists of AST nodes. Yet this makes the tree more balanced. But for any practical applications (since the parser shall work in IDEs for humans) one can assume

the length of those lists as being limited, so the runtime of $\mathcal{O}(log(n)^2 + l)$ with l being very small for a real programming language is archived.
The algorithm was designed with the target of zero additional space cost in mind. That is archived with one minor exception: In the AST that is produced by PaGe every node has a direct parent reference. Without this reference the AST could still be used, but in the second phase to determine the path from the root node to the next unchanged token additional runtime cost of $\mathcal{O}(log(n)$ would be required, not changing the complexity class but effectively halving the incremental parsers performance in that phase.

3 Optimizations

Although the runtime complexity is near to optimal (see [6]) and only non optimal because of some properties of PaGe or the input grammar, there is still room for some optimizations:

3.1 Sibling Ascension

After the constraining node C has been parsed, the complete right hand side of the AST can be reused by the incremental parser. This means that instead of walking up the tree from the bottom, one could reuse the right hand side siblings of C directly. Then the sibling of Cs parent and so on.
Since C is not calculated explicitly, we implemented this again implicitly by checking if the siblings of the last reused node in phase two can be reused too. Again this has the advantage that also children of C can be reused.
Obviously this speeds up the AST ascension in the second phase. Instead of ascending one level in $\mathcal{O}(c_{max}log(n))$ with this optimization it is done in $\mathcal{O}(c_{max})$. Therefore the overall runtime is $\mathcal{O}(l_C + log(n))$ with Sibling Ascension enabled. Although this seems to be a big performance gain, benchmarks (table 6) did not support that assumption. In fact tested grammars showed a runtime of $\mathcal{O}(l_C + log(n))$ in the above mentioned setup.

3.2 Lists

Another difficulty are the Lists that are used by PaGe: While grammar productions like $A \rightarrow B^*$ are transformed into recursive productions internally the created actions form special tokens that expose a list interface instead of the degenerated trees one would expect. Although the parser works with those lists (they can be reconstructed in the second phase or completely reused in the first phase), there is still some room for optimizations: The deconstruction of a list could work by bisecting it and since it is clear that a list *must* be created in the incremental run, if the first child covers more tokens than the maximum lookahead one could try to split the list in a prefix and suffix part to speed up the recreation. This is not yet implemented since that change might demand changes of the batch parser which was not allowed by the current development

concept.

3.3 Object Reusage

While the algorithm meets all requirements in runtime and space that were mentioned above, its behaviour is suboptimal in the aspect of object reusage: Though all nodes that have no changes are recycled (and some more that changed their place inside the AST), the path from the AST root to the constraining node C is completely recreated even if it is not altered structurally.
Again one does not know this path because of the implicit calculation of C, but it can only consist of nodes that were visited but not reused during the first phase. Since every Action that the parser applies to the stack "knows" how much Tokens it will remove from the Token Stack, it can easily check if the leftmost of these tokens already has a parent and if this parent matches the Class that this action would instantiate. If this is the case, instead of creating a new AST node, that parent can be reused and populated with the new set of child nodes. Since the parser creates the AST bottom up, that way every node that has children in the "before" part of the AST will be reused, if possible.

Tokens	**Batch (ms)**	**Incremental (ms)**	**Optimized (ms)**
4906	74.3	1.9	5.5
5181	88.3	3.3	3.3
7423	105.9	4.5	3.6
8871	141.5	5.7	11.2
10046	131.3	7.4	7.6
10088	148.8	4.9	6.6
11212	181.5	5.0	5.2
14385	178.1	4.1	4.3
15720	188.8	4.4	5.6
17208	236.8	6.7	4.7
19568	227.9	5.8	4.8
20295	263.8	3.5	3.6
21861	259.8	5.5	7.1

Fig. 6. Benchmarking results

3.4 Benchmarks

As mentioned above we tested the incremental parser by marking single input tokens as changed (this can be easily done by deleting their parent reference). Therefore the effect of l_C is minimized, making this benchmark somewhat synthetic (one could consider this a realistic scenario for refactoring operations

where e.g. identifiers are renamed).
The table in figure 6 shows the worst case runtime of the incremental parser and the batch parser against the number of tokens in the file.
To minimize the influence of dynamic optimizations inside the JVM every benchmark was run ten times.
Surprisingly the optimization did not yield a significant advantage but some major performance drawbacks. Obviously the overhead of the implementation surpasses the advantage of ascending the tree in an optimized order. (Also it should be clear that the pure tree ascension is extremely fast compared with the parsing operations and reusable checks).

4 Example

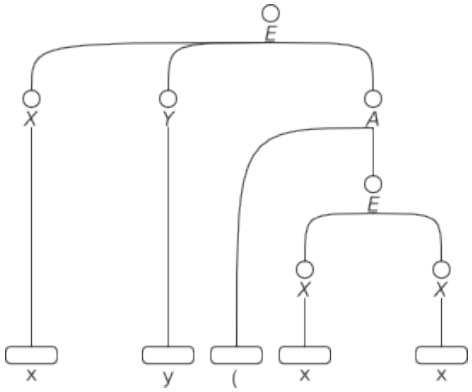

Fig. 7. Example AST for G

As an example for the incremental parser consider the AST for G shown in figure 7. This tree was constructed from the input `x + y (x + x)`. If this input is changed to `x + x (x + x)`, the parser will enter the following configuration after the first phase:

$$([Z_3, Z_4, \$], [\underline{)}, \underline{x}, \underline{+}, \underline{x}, \underline{(}, \underline{+}, \underline{x}\$], [X\$])$$

The parser will then parse the $\underline{+}$ and $\underline{y}$ Terminals, entering:

$$([Z_3, Z_4, Z_{10}, Z_{11}\$], [\underline{)}, \underline{x}, \underline{+}, \underline{x}, \underline{(}\$], [X, \underline{+}, X\$])$$

The next Terminal, $\underline{(}$ is the leftmost Terminal of the A node which can then directly be recycled:

$$([Z_3, Z_4, Z_{10}, Z_{11}, Z_{12}\$], [\$], [X, \underline{+}, X, A\$])$$

And after a final reduction reaching the final state $Z_{\#}$:

$$([Z_3, Z_{\#}\$], [\$], [E\$])$$

In this example the only newly created node would be the new X node, since E could be recycled by the above mentioned object reusage mechanism.

5 Conclusion

The proposed algorithm makes it possible to reconcile models from source code in real time for arbitrary LR grammars. This allows IDE developers to create sophisticated tools without having to struggle with complex tree algorithms.
The benchmarks taken from a real programming language source code reveal that optimizations on the algorithm do not make a notable difference in the time a reconciling operation takes. Therefore the tradeoff of additional complexity and possible sources of errors must be considered when using the parser for a given language.
This algorithm will be implemented for a IDE supporting the MOSILAB language and revised based on the experiences from this application. Afterwards the algorithm may be merged into PaGe.

References

1. Agrawal, R., Detro, K.D.: An efficient incremental lr parser for grammars with epsilon productions. Acta Inf. 19(4), 369–376 (1983)
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986)
3. Kunert, A.: Lr(k)-analyse für pragmatiker (2008), http://www.informatik.huberlin.de/ kunert/papers/lr-analyse/, (de)
4. Nytsch-Geusen, C., Ernst, T., Nordwig, A., et al.: Mosilab: Development of a modelica based generic simulation tool supporting model structural dynamics. In: Schmitz, G. (ed.) Proceedings of the 4th International Modelica Conference, Hamburg, March 7-8, 2005. pp. 527–535. TU Hamburg-Harburg (2005)
5. P.Pepper, M.: Compilergenerator für opal-2 (2008), (unpublished)
6. Wagner, T.A., Graham, S.L.: Efficient and flexible incremental parsing. ACM Transactions on Programming Languages and Systems 20 (1996)