Context Free General Top Down Parsing in Cubic Time (Unfinished Draft)

Arnold Lankamp

February 10, 2011

Abstract

In this article we will describe our general top-down parsing algorithm. Our intent with this algorithm is to develop something that is both easy to comprehend and performs well on any grammar. The user should not be required to refactor a grammar to enable the parser to complete within an acceptable amout of time. Additionally, it should not be hard to grasp what is going on under the hood, nor should it be complicated to create an implementation of this algorithm.

It sounds fairly ambitious, but we managed it. We created a general parsing algorithm that is both easy to understand and can compete with any other general parsing algorithm out there in terms of scalability and performance.

1 Introduction

If the abstract did not catch your attention yet, here are some highlights:

- Worst-case cubic space and time bounds with respect to the length of the input string.
- Worst-case linear scaling relative to the size of the grammar.
- Linear performance on LL and LR grammars.
- No performance hit for rules that are not left factored.
- No penalty for being scannerless.
- Low stack activity.
- Generating or hand crafting the recognizer / parser code is trivial.
- Parse traces are easy to visualize.

For our parsing algorithm we choose for a top-down approach, as it is more in-line with human thinking then bottom-up. This makes the algorithm and implementation easier to comprehend. Ultimately it should also make its usage simpler, since grammars can more easily be converted to input for the parser; the translation is one-on-one and does not require any extra work, unlike what is necessary when generating parse tables. Nor will it ever explode in size, which parse tables potentially could. Lastly, top-down parsers can also generate understandable error reports more easily, which is a very welcome feature.

However, general top-down parsers do have the image of being slower than general bottom-up parsers. Mostly because they tend to perform poorly on non-left-factored grammars. Additionally, they usually try to recover from parse errors by back-tracking, which does not scale well, especially in absence of memorization. Our algorithm and implementation do not suffer from any of these problems.

We call our parser, the Scannerless General Top Down Binary Forest Parser or SGTDBF.

2 Recognizer

First we will discuss the algorithm for the recognizer, for simplicity's sake. Further on in this article we will explain how the recognizer can be extended to become a parser.

2.1 Graph

To represent the parallel stacks we use a directed cyclic graph, acustom to GLR's Graph Structured Stack. In our case we generate a stack node with a unique identifier for every item in the grammar. For example; if take the following grammar: $S ::= AB \mid BA$, $A ::= a$, $B ::= b$, we would assign these identifiers as follows: $.S(-1)$, $.AB(0)$, $A.B(1)$, $.BA(2)$, $B.A(3)$, $.a(4)$ and $.b(5)$. These identifiers are needed to be able to handle the sharing of graph nodes correctly, so regardless whether or not comparable items exist, they should not be merged. Each stack node consists of this identifier, edges back to their 'parents' in the graph, information about its 'right neighbour' in the production and the location in the input string it is associated with. Each stack node is unique for each identifier, location combination.

2.2 Basic algorithm

The basic idea behind the algorithm is fairly straightforward. In general terms this is how it works:

1. Expand the left most node of every production that did not match anything yet on all stacks, until no stacks can be expanded anymore (i.e. they all have a terminal at the bottom).
2. Match all nodes on the bottom of the stacks to the input. Reduce the ones that match, remove the ones that do not.
3. Reduce the nodes on all stacks that matched something and have them queue the 'next' node in the production they are part of where applicable, until no reductions are possible anymore.
4. If there are nodes queued for expansion go to 1.
5. If the end of the input has been reached and we have at least one derivation for one of the start symbols we are successful; otherwise recognition failed.

2.3 Pseudocode

In this section we will present and discuss the recognizer's psuedocode. Note however that this psuedocode already contains the edge related optimizations listed in section 5.2. This was done since, besides improving performance, they simplify the implementation of the algorithm slightly and they enable the recognizer to achieve most of its advertised performance guarantees. We will not go into detail about how these optimization work or why they are correct here; please refer to the indicated section for more details. Other optimizations were not merged in with the code here, since they either complicate things too much or can be implemented in multiple ways; a decision we would like to leave for the implementor.

2.3.1 Main

```
main(){
  toExpandSet.add('startNode');
  expand();

  while(hasMoreStacksToReduce(toReduceStore)){
    expandedSet.clear();
    reducedStore.clear();
    toReduceSet = getStacksToReduce(toReduceStore);

    do{
       reduce();
       expand();
    }while(isNotEmpty(toReduceSet));
  }

  if(endOfInputHasNotBeenReached()) error;
}
```

This is the main function of the recognizer. It starts with building the stacks for the first level, by queuing the 'start node' and expanding it. Once it has done so it will keep alternating between reducing and expanding until there are no more stacks left alive. At the start of each iteration we get the appropriate **toReduceSet** from the **toReduceStore**. The **toReduceStore** is a collection that contains one **toReduceSet** for each level. At the end of each iteration we check whether or not we need another iteration in the current level or need to shift to the next one. In case we shifted we need to discard all level specific data. This behaviour is necessary to be able to handle nullables properly.

Also note that after each iteration all graph nodes that are no longer reachable through any of the nodes in the **toReduceStore**, can be discarded.

2.3.2 Expand

```
expand(){
  while(node <- toExpandSet){
    if(node.isTerminalOrEpsilon()){
      if(node.match(input)){
        toReduceStore.add(node);
      }
    }else{
      if(cachedEdges.contains(node.sort)){
        edgesSet = cachedEdgesMap.get(node.sort);
      }else{
        edgesSet = createAndCacheEdgesSet(node.sort);

        for(childNode <- getAlternatives(node)){
          childNode = childNode.initialize(location);
          childNode.setEdgesSet(edgesSet);
          toExpandSet.add(childNode);
        }
      }
      edgesSet.addEdge(node);

      if(reducedStore.contains(node.sort)){
        toReduceStore.add(node);
      }
    }
  }
}
```

This is the expand loop. It will iterate over the **toExpandSet** until there are no more stacks queued for expansion. While expanding we check the type of the node we are handling. If the node is a terminal or an epsilon it needs to match the input (epsilons naturally always match). If it matches, it is added to appropriate **toReduceSet** in the **toReduceStore**; otherwise we discard it, causing the stack it was associated with to die of. If the node is a non-terminal, we check if there is a cached set of edges available for that non-terminal sort. If there is, we queued the alternatives for that non-terminal sort before; since this is the case, we merge the current stack with the already expanded ones by adding an edge to the cached **edgesSet** for this non-terminal sort. If a cached **edgesSet** is not available, we have not encountered a non-terminal of this sort yet and need to expand it. We to this by retrieving its alternatives and queueing the first 'child' of each alternative for expansion, by adding them to the **toExpandSet**. We also create and cache a set of edges for this non-terminal sort, to which an edge to the current node will be added. Each of the 'child' nodes will receive a reference to this **edgesSet**, which may be updated with additional edges later on in the expansion process.

Finally we check whether or not there have been nullable reductions for the current node's non-terminal sort. This can occur because we to multiple iterations in this same level to be able to handle nullables properly. If such a nullable reduction is present in the **reducedStore**, we queue the current node for reduction.

2.3.3 Reduce

```
reduce(){
  while(node <- toReduceSet){
    if(node.lastInProduction()){
      for(edgeLevel <- node.edgesPerLevel){
        if(reducedStore.notContains(edgeLevel.getOne().sort)){
          for(edges <- edgeLevel){
            toReduceSet.add(parent);
          }

          reducedStore.add(node.sort);
        }
      }
    }else if(node.hasNext()){
      next = node.next.initialize(location);

      if(toExpandSet.notContains(next)){ // Sharing
        next = toExpandSet.get(next);
      }else{
        toExpandSet.add(next);
      }
      next.addEdges(node.edges);
    }
  }
}
```

This is the reduce loop. It will iterate over the **toReduceSet** until there are no more stacks queued for reduction. When a node is being reduced, one of two things can happen. Either it is the last node in the production and we need to queue this node's 'parents' for reduction (if not reduced of queued for reduction already), by adding them to the **toReduceSet**. Otherwise we need to move to the 'next' node in the production, queue this 'next' node for expansion and transfer all edges from this node to it. In case the 'next' node has already been scheduled for expansion we need to retrieve the equivalent 'next' node from the **toExpandSet** and add the edges to that node instead, for sharing purposes.

The edges are grouped by level. This level indicates the start location of the node the edge points to. To check whether or not the parents of a node need to be queued for reduction, we get one of the edges from each **edgeLevel** and check if the node this edge points to has been reduced yet or not. If a node has not been reduced yet, we queue all of the nodes that are pointed to from edges in that **edgeLevel** for reduction; otherwise we do not. This will work correctly since every node containing a non-terminal of the same sort always has exactly the same parents in the same level. Note that for the recognizer this behaviour is not a necessity, but an optimization (which reduces the number of edge visits by a factor between one and the number of non-terminals in the grammar). However for the parser this behaviour is required to achieve proper sharing in the resulting trees (in an uncomplicated and fast way).

The **reducedStore** is a dual layered data structure. The first layer is indexed by the **start position** of the node and the second layer by the sort name of the non-terminal in the node. Since the recognizer algorithm assumes it is level synchronized, the **reducedStore** can contain a variant for every non-terminal

for each possible substring up until the current location. This **reducedStore** is necessary to keep track of which nodes have or have not been reduced yet.

Note that the first layer of the **reducedStore** needs to be implemented as an array or table with O(1) look-up time; otherwise it is not possible to guarantee worst-case cubic time complexity.

2.4 Correctness

In this section we will go into detail on some special cases that deserve some extra attention.

2.4.1 Left recursion

First of all, left recursion. Normal top-down parsers cannot handle grammar rules containing left recursion, since they can keep expanding left recursive rules indefinitely, preventing implementations of these kind algorithms from terminating. Because we introduce sharing into the graph, which leads to automatic terminalization of all these rules, this undesirable behaviour is prevented from manifesting. By terminalization we mean that the expansion phase converges to a point at which each stack has a terminal at the bottom and thus cannot be expanded further. For example, if we take the grammar:
$S ::= A$
$A ::= Aa \mid a$
We would expand these rules in the following way:

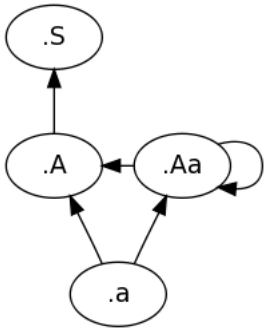

Figure 1: Graph representation of an expanded left-recursive grammar.

As can be seen a cycle is added to the graph, since $.Aa$ is, in a way, a child of itself. To see how left recursive grammars work in action see section 2.6.2.

2.4.2 Cycles

Cycles present another issue that needs to be handled. Similarly to left recursion, this problem is automatically fixed by the sharing that is introduced into the graph. Instead of infinitely expanding the circularly dependent production rules, a cycle will be added to the graph and the expansion process will halt. The reduction process will also terminate, since it will only perform the reduction for each node once.

2.4.3 Nullables

Nullables also deserve some attention. Nullables do not pose any difficulties in most cases; however when a production contains two or more consecutive nullables, hidden left recursive or otherwise, problems arrise. These problems are sharing related. First we will describe what would happen in the naïve / broken implementation.

Consider the following grammar:
$S ::= AA$
$A ::= C$
$C ::= \epsilon$
Input = <empty>
After the first expansion and reduction phase, we matched both C and the first A in the production $S ::= AA$ at location 0. Next we move to the second A in the production $S ::= AA$; when we expand this A node we notice C has already been expanded at the current level (0), causing recognition to stop. Since there is no work left to do the recognizer will terminate with an error, as we failed to produce any derivations for the input string.

In our algorithm we address this issue by checking whether or not the non-terminal sort associated with the node we are currently attempting to expand has nullable results in the current level. If this is the case we queue this node for reduction. Additionally, we also update the appropriate set of edges (as described in section 2.3.2). This is necessary, since the children of the current node were already expanded in a previous iteration in the same level; this means the current stack needs to merge with the stacks of these children, so no future (non-nullable) reductions for these children will go missing. We will keep alternating between expansion and reduction until no more work can be done for the current level (i.e. there are no more nullable reductions); once this is the case, we shift to the next level.

2.4.4 Termination

One of the main questions that some people will have on their minds is: "does it terminate?". Ofcourse it does. The reason for this fairly obvious and can be even explained in one sentence; since we move forward to the next level once all work is done in the current level and we share all nodes within each level, causing expansion and reduction to converge to a point where no more work can be done in a level, we reach the end of the input string at some point.

2.4.5 Breadth-first vs depth-first

Another requirement of the algorithm, of which the reason may not be immediately obvious, is that it needs to be level synchronized. We will demonstrate why the depth-first sibling does not work correctly without modification and why this modification would break the cubic time bound, by means of an example. Consider the following grammar: $S ::= SSS \mid a \mid aa$. Now imagine we are somewhere in the middle of recognizing an input string containing a number of a's; currently we are recognizing $SS.S$, which matches with one a. The preceding items that belong to this production, $.SSS$ and $S.SS$ both recognized one a as well. Causing $S(S(a), S(a), S(a))$ to be reduced at this location. We progress a bit more and are now recognizing $.SSS$ at a location that is one character

earlier in the input string as the other $.SSS$ we recognized before; this time it matches aa. Since we now end up at a $S.SS$ node at a location where it was already recognized, we do not progress further; this node needs to be shared, otherwise we would do duplicate work, breaking the cubic time bound. However, in the scheme we are using, sharing this node means we stop recognizing the current derivation, preventing reductions from being executed. Which in this case will mean that we fail to recognize the derivation $S(S(aa), S(a), S(a))$ at the current location. If we would be using a different grammar it could even cause recognition to fail entirely because of this behaviour.

This problem could be solved by tracing forward through the production, gathering any potential missing derivation(s) and executing reductions where necessary. However this would cause the algortihm to become unbound polynomial in time, since it would lead us to 'touch' nodes $N^{(productionLength+1)}$ times in the worst-case.

In our opinion the solution for this problem is both to complicated and, above all, unnecessary. Since our algorithm is breadth-first, we will never have stacks running ahead of others, eliminating the possibility of this behaviour popping-up. We do need extra iterations within the same level to handle nullables, but this is only a problem for the parser, which needs to keep track of hidden right recursion. We will cover the handling of hidden right recursion in the parser in section 3.3.

2.5 From grammar to code

Converting a grammar to a recognizer or parser, either by hand writing or generating it, is relatively straight forward. All that is needed is a direct translation from the grammar rules to either functions or a table like data structure. Basically the recognizer just needs to know what alternatives are associated with each left-hand-side. In case we are generating code, this would mean that we need one function per non-terminal sort, which contains logic that informs the recognizer about what alternatives should be expected.

Since the mapping between the original grammar and the code or table is one-on-one, it is possible to implement or edit it by hand without much effort. Another advantage is that, in combination with the top-down-ness of our algorithm, it makes tracing errors in a grammar easier; if you would like to know why something does not match at a certain position, you can just go through it with a debugger to see what happens (your degree of success naturally depends on the amount of stacks that are alive at the moment you are trying to observe, but at least the possibility to do so exists).

2.6 Example traces

To illustrate how the recognizer works in action, we constructed some example traces using various grammars to give an impression.

2.6.1 Straight forward

First we will take a look at a simple non-ambiguous grammar:
$S ::= AB$
$A ::= a$
$B ::= b$
input = ab

1. expand $.S$
2. expect $.AB$
3. expand $.AB$
4. expect $.a$; $.a$ matches
5. reduce $a.$ and follow edge to $.AB$
6. move from $.AB$ to $A.B$
7. expand $A.B$
8. expect $.b$, $.b$ matches
9. reduce $b.$ follow edge to $A.B$
10. reduce $AB.$ and follow edge to $.S$
11. parse for $S.$ is complete

This one is easy to follow and does not really need any additional explanation.

2.6.2 Left recursive

Next up is a grammar containing a left-recursive rule:
$S ::= A$
$A ::= Aa \mid a$
input = aaa

1. expand $.S$
2. expect $.A$
3. expand $.A$
4. expect $.Aa$ and $.a$; $.a$ matches
5. expand $.A \Rightarrow .A$ shared
6. reduce $a.$ and follow edges to $.A$ and $.Aa$
7. reduce $A.$ and follow edges to $.S$
8. parse for $S.$ is incomplete and is discarded
9. move from $.Aa$ to $A.a$; $A.a$ matches
10. reduce $Aa.$ and follow edges to $.A$ and $.Aa$
11. reduce $A.$ and follow edges to $.S$
12. parse for $S.$ is incomplete and is discarded
13. move from $.Aa$ to $A.a$; $A.a$ matches
14. reduce $Aa.$ and follow edges to $.A$ and $.Aa$
15. reduce $A.$ and follow edges to $.S$
16. parse for $S.$ is complete
17. move from $.Aa$ to $A.a$; $A.a$ does not match since EOI has already been reached

As one can see, when expanding $.A$ for the second time (at 5), sharing is detected causing a cycle to be added to the graph. Reductions of $Aa.$ follow the edge back to $.Aa$ which will move to $A.a$; this will lead to one a being matched at each 'iteration' and ultimately consuming all of them.

2.7 Worst-case complexity

We are designing a parsing algorithm that is intented to scale as well as is possible, regardless of the input grammar. For this reason, the recognizer must not break the cubic time bound. Here we will prove that we remain within this bound.

Since the graph only contains nodes that we expect to 'encounter', there are at most $O(N)$ of them; where N is the number of characters in the input string. Each node only has one edge to each of its possible parents per level and there are N levels, so there are at most $O(N)$ edges per node. In the worst-case there are N times $O(N)$ nodes that match a substring that ends at the 'current' level. When moving to the 'next' node in the production, all edges of these nodes need to be carried over. Since these sets of edges need to be merged, the time this operation needs to complete is equal to the number of levels before (and including) the 'current' level (so N at most). So this operation will execute in $O(N^3)$ time at most in the worst-case.

Reducing consumes $O(N^3)$ time in the worst-case as well. Each of the $O(N^2)$ edges is associated with one reduction. The check whether or not each of the parents still needs to be reduced, is an operation that completes in $O(1)$ time. However we do need to visit the edge to this parent to be able to determine who this parent is. This happens N times per edge at most. So each edge is visited $O(N^3)$ times at most.

This means that the algorithm remains within cubic time bounds in worst-case scenarios.

3 Parser

3.1 Parse forest

To represent the parse forest we use a format that was especially designed for this parser, to ensure worst-case behaviour remains within cubic space and time bounds. We call it 'deflattenized', for lack of a better description. While its purpose is similar to binarized SPPFs, its implementation is somewhat different.

The parse forest consists of nodes. Every node in the forest contains a result and a set of prefixes. Each results represents a substring for a certain symbol. In case this symbol is a non-terminal, this result contains one or more references to alternative representations of the substring it denotes. If the symbol is a terminal, it will just represent that specific terminal. The set of prefixes contained in the node hold all possible alternatives for the preceding item in the production. (Naturally this set is empty for the node containing results for the first item in the production). A prefix only consists of a reference to a node. If one traces all possible paths through this representation of a production to the start, you will get all alternatives for this production of the substring it represents.

Note however, that the parser forest contains an exact representation of what the parser recognized. This means it can contains cycles.

3.1.1 Example

To give an idea of what a typical parse forest would look like we will give an example using the following grammar:
$S ::= AAA$
$A ::= a \mid aa$
input = $aaaa$

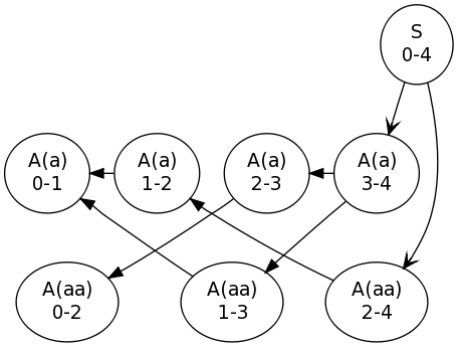

Figure 2: A visual representation of the parse forest, showing all alternatives for the given input string. The numbers indicate the start and end position of the matched substring.

In the figure above we see the parse forest. It illustrates how the parse results are stored in memory.

To show how we can obtain all possible derivations from this parse forest, we will list them in the table below; which relates each path through the forest to a derivation.

Derivation	Forest path
S(A(a),A(a),A(aa))	A0-1 ← A1-2 ← A2-4
S(A(a),A(aa),A(a))	A0-1 ← A1-3 ← A3-4
S(A(aa),A(a),A(a))	A0-2 ← A2-3 ← A3-4

Table 1: A flattened representation of the parse forest, listing all derivations.

3.2 Psuedocode

Augmenting our recognizer with parse tree construction code is trivial. Only a few minor adjustments are needed. Relevant pieces of code, containing changes or additions, are highlighted in italic.

3.2.1 Main

```
main(){
  toExpandSet.add('startNode');
  expand();

  while(hasMoreStacksToReduce()){
    expandedSet.clear();
    reducedStore.clear();
    toReduceSet = getStacksToReduce();

    do{
       reduce();
       expand();
    }while(isNotEmpty(toReduceSet));
  }

  if(endOfInputHasNotBeenReached()) error;

  return resultStore.get('startNode');
}
```

The only things that need to be changed in the main function to transform the recognizer into a parser are the returning of the final result (if not in error) and replacing the **reducedStore** by **resultStore**. The **resultStore** is similar to the **reducedStore**, except that it also contains the result associated with the non-terminal sorts in the store.

3.2.2 Expand

```
expand(){
  while(node <- toExpandSet){
    if(node.isTerminalOrEpsilon()){
      if(node.match(input)){
        toReduceStore.add(node);
      }
    }else{
      if(cachedEdges.contains(node.sort)){
        edgesSet = cachedEdgesMap.get(node.sort);
      }else{
        edgesSet = createAndCacheEdgesSet(node.sort);

        for(childNode <- getAlternatives(node)){
          childNode = childNode.initialize(location);
          childNode.setEdgesSet(edgesSet);
          toExpandSet.add(childNode);
        }
      }
      edgesSet.addEdge(node);

      if(reducedStore.contains(node.sort)){
        toReduceStore.add(node);
      }
    }
  }
}
```

The expansion code does not need to change a lot, since it has limited interation with parse results. The only change is related to the **reducedStore**, which has been replaced by the **resultStore**. In this context they serve a similar function as in the recognizer; to check whether or not there are nullable reductions available for a certain non-terminal sort. We will discuss the **resultStore** in more detail further on in this section.

3.2.3 Reduce

```
reduce(){
  while(node <- toReduceSet){
    if(node.lastInProduction()){
      for(edgesLevel <- node.edgesPerLevel){
        if(resultStore.contains(edgesLevel.getOne().sort)){
          result = resultStore.get(edgesLevel.getOne().sort);
        }else{
          result = createResult(edgesLevel.getOne());
          for(edges <- edgesLevel){
            toReduceSet.add(parent);
          }
        }
        result.addAlternative(node.prefixes, node.results);
      }
    }else if(node.hasNext()){
      next = node.next.initialize(location);

      if(expandedSet.notContains(next)){ // Sharing
        next = expandedSet.get(next);
      }else{
        toExpandSet.add(next);
      }
      next.addEdges(node.edges);
      next.updatePrefixes(node.prefixes, resultStore.get(node));
    }
  }
}
```

The reduce code does need to be modified. Basically two things need to be added.

First of all the storing of results. We need to create (at most) one **result** per sort of non-terminal for each matched substring; no more then that. In the recognizer code we used the **reducedStore** for a similar purpose. Here it is replaced by the **resultStore**, which basically is the same thing, but with results associated with the non-terminal sorts in the store.

Secondly we need to add **prefixes** to the nodes that are being queued for expansion when we are moving to the **next** node in the production. We do this by creating a new prefix using the prefix of the current node and its result and adding it to the **next** node. We add the prefixes to graph nodes for convenience reasons; regardless of where the result(s) of the node end, they will always share the same prefix. By adding the prefixes to the graph nodes they are easy to retrieve; all these nodes need is a collection that holds all the prefixes for it, grouped by the start location of the production.

3.2.4 Results

```
createResult(node){
  result = allocateResult(node.startLocation, node.sort);
  resultStore.add(result);
  return result;
}
```

We also need a function for creating **results**. Every time a result is created we need to add it to the **resultStore**. The **start position** and non-terminal **sort** of the given node are used as keys. Like the **reducedStore** the **resultStore** is a dual layered datastructure, where the first layer is indexed using the **start position** of the node and the second layer by the non-terminal **sort** the node represents.

3.3 Correctness

Since the conversion of our recognizer to a parser is so trivial, the parser is as correct as the recognizer. The only difference, as mentioned before, is the handling of hidden right recursion.

3.3.1 Hidden right recursion

The reason for complications related to hidden right recursion is that multiple iterations within the same level are possible. This could cause results to 'go missing'. The only thing that needs to be done to fix this, is to check whether or not the next node is the last node in the production and is a nullable with results, in case the current node is also a nullable. If this is the case we need to construct the necessary result nodes and add them to their corresponding parents. This is a fairly general description of how to solve it, since the best way to handle this issue may differ per implementation. All that is needed is that the implementor is aware of this issue and takes it under consideration.

3.4 Worst-case complexity

We looked at the worst-case behaviour of the recognizer in section 2.7 and proved that is does not break the cubic time bound. For the parser to be able to make the same guarantee, the size of the parse forest must be, at most, cubic in the length of the input. The reason for this, is that it is impossible to construct a greater then cubic parse forest in a cubic amount of time. Here we will prove that we do not break this space bound.

Every node in the tree is identified by the substring its contained result represents and its prefixes. There can be only one result per substring, this means there are at most $O(N^2)$ results. The prefix sets are identified by the start and end position of the substring they represent, so there are at most $O(N^2)$ of them. Each prefix set can contain up to N different prefixes, which all denote the same substring; at most one per location (before the current location) in the input string. So there are at most $O(N^3)$ prefixes. The number of unique nodes is determined by multiplying the number of results by the number of prefix sets. However, since a prefix set can only be matched in a node, together with a result that starts at the same position as the prefix ends each prefix set can

be associated with $O(N)$ different results at most. Hence the number of unique nodes is limited to $O(N^3)$, making the parse forest $O(N^3)$ worst-case. Note that in the implementation all prefix sets are shared regardless of the end position of the node they are matched to; this has no effect on worst-case behaviour, but does improve performance and decreases memory usage.

3.4.1 Worst-case statistics

We have already theoreticly proven that the recognizer / parser will be $O(N^3)$ in the worst-case. In this section we will show that in practice this is indeed the case.

We will give an overview of the most relevant parser statistics for the following grammar:
$S ::= SSS \,|\, SS \,|\, a$
input = $a * 2$ to $a * 10$, $a * 50$, $a * 100$, $a * 200$, $a * 300$, $a * 400$ and $a * 500$

Input length	Graph nodes	Edges	Edge visits	Results	Prefixes	Forest nodes
2	9	7	6	3	1	2
3	13	10	15	6	4	8
4	17	13	31	10	10	20
5	21	16	56	15	20	40
6	25	19	92	21	35	70
7	29	22	141	28	56	102
8	33	25	205	36	84	168
9	37	28	286	45	120	240
10	41	31	386	55	165	330
50	201	151	42926	1275	20825	41650
100	401	301	338351	5050	166650	333300
200	801	601	2686701	20100	1333300	2666600
300	1201	901	9045051	45150	4499950	8999900
400	1601	1201	21413401	80200	10666600	21333200
500	2001	1501	41791751	125250	20833250	41666500

Table 2: Worst-case parser statistics (for the fully optimized version).

In the table above we can see how the different components in the parse forest increase with respect to the length of the input string. Note however that all optimizations mentioned in chapter 5 were enabled while gathering these statistics; most notably edge sharing (5.2) and graph prefix-sharing (5.3.1).

The number of graph nodes and number of edges scale linear with respect to the length of the input string. The number of edge visits remains within the cubic bound and is only slightly higher then the number of result nodes. The 'extra' visits are for failed parse results, one for each location in the input string squared. This means the number of edge visits is as low as they can possibly be; meaning we can conclude that an algorithm with less stack activity is impossible to concieve.

Looking at the parse tree related statistics we can see that the number of results increases by N for every extra character in the input string. Which is because the number of results is equal to the number of different substrings in

the input, in this case. As can be observed, the number of prefixes and number of nodes remains within cubic bounds for this worst-case example.

3.4.2 Flattening

Optionally it is possible to output a flattened version of the parse forest at the user's request. Naturally the size of this parse forest will be unbound polynomial relative to the length of the input, in the worst-case. Although, in practice this unlikely to happen; moreso since filtering can be done during parsing and flattening, making it improbable that many ambiguities remain in the final tree, in the general case. We will discuss filtering in chapter 6.

One might wonder why, in case one flattens the parse forest afterwards, using the 'deflattenized' version as internal representation would be advantageous. The reason for this is that it enables us to guarantee that both the recognition and parse phases will always be cubic with respect to the length of the input in the worst-case. If we would not use the 'deflattenized' representation we would not be able to make this guarantee for the parse phase. The consequence of this is that it may make the parsing of input for certain grammars unfeasible. While it may be feasible to flatten the final parse result in these cases. The main reason that this is a possibility is because of the stravation of incomplete parse results; either ones that died because they failed to match the input further down the line or because they were filtered. In both cases it will prevent their results from being added to the parse forest as alternatives. This means the final parse result will likely have less nodes in it then there are constructed in total while parsing, reducing the amount of nodes the flattener has to touch.

4 Extended capabilities

To improve the usefulness of the parser on real grammars a few additional features were added. Namely support for lists, optionals and location related features is added. In this chapter we will discuss how these features can be intergrated into our algorithm and implementation.

4.1 Lists

List, separated lists and optionals are handled in a similar way, because in essense they are the same. However their semantics are slightly different; separated lists are lists with extra symbols between their elements and optionals are star-lists with one element at most; from here on out we will refer to all of these simply as 'lists'.

Often lists are implemented by adding extra 'virtual' rules to the grammar. For example $S ::= A+$, $A ::= a$ can be parsed like a 'normal' grammar, if we add the rule $A+ ::= AA+ \mid A$. The advantage of this approach is that the parser does not need to be modified. On the other hand, we always get a binarized version of the list as a result, that uses these imaginary productions, which may not be what we wanted. Additionally, because an extra rule is inserted into the grammar we need more graph nodes to be able to parse the list.

Our approach involves special casing the handling of lists, by introducing additional types of graph nodes that contain knowledge about how they need to

be expanded. For example to expand the star-list $A*$, it would queue an ϵ and a A non-terminal graph node; this A graph node has a 'next' pointer to itself and is marked as last node in the 'production' (see figure below). One could view this construction as a graph node with a kind of dynamically growing production as its child. Each time an A is recognized the list is reduced and the next A is pushed on the stack to be recognized; causing the next element to be appended to the list.

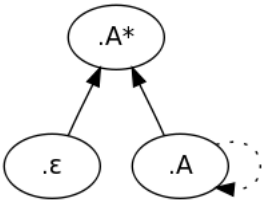

Figure 3: Graph representation of a expanded star-list; the solid arrows represent edges and the dotted arrows 'next pointers' in the production.

These list children can be shared just like any other graph node. Nothing special needs to be done for this. Since this is the case, worst-case time and space bounds do not change by the introduction of this feature.

Exactly the same principle is used for separated lists and optionals. We will not go into detail on those, since one can easily imagine how these work.

4.2 Location

Similar to how lists are implemented, there are also specialized graph nodes for location related indicators. We have the following varieties:

1. Start-of-line
2. End-of-line
3. At-column(X); where X indicates the column number

Basically these are epsilons that only match when we are at a specific location in the input. So they do not consume input and act more like filters then anything else. They can be very useful when parsing certain languages.

5 Optimizations

The basic algorithm is fairly straightforward and relatively simple to implement. The naïve implementation will respect worst-case cubic time and space bounds, however adaptations can be made to improve its overall efficiency.

5.1 General

5.1.1 Matching

A minor performance improvement can be made by matching on entire literals at once, instead of on induvidual characters. This decreases the amount of necessary graph nodes and thus reduces stack activity, offsetting the performance overhead being scannerless brings along completely.

5.1.2 Look-ahead

A more obvious optimization is adding support for look-ahead restrictions. By adding simple checks to the (generated) parser code, we can prevent unnecessary work; if a certain alternative will never match, we do not need to expect it. This both improves performance and will make recognizing / parsing deterministic for LL(k) grammars. When the prefix-sharing optimization (see section 5.3.1) is enabled, this will also be the case for LR(k) grammars.

5.1.3 Breadth-first

Finally we would like to note that (eventhough it is mandatory for the correctness of our algorithm) a breadth-first general parsing algorithm, or breadth-first implementation of a general parsing algorithm is generally more efficient then a depth-first version. This is because recognizing / parsing can be approached in a level synchronized way, which enables the possibility of handling certain things more efficiently; this mainly has a positive effect on memory usage.

5.2 Edge related

Generally parsers store results on edges in the graph. If we would refrain from doing this and store parse results elsewhere (in a table with constant look-up time), edges could remain pointers. At first sight this may not be an advantage, however we would be able to share edges among graph nodes and it opens up numereous other opportunities for optimization.

5.2.1 Expansion

First of all, by caching 'expected children' for every type of node per level a substantial performance increase can be achieved. Additionally, this will ensure linear scaling for non-left-factored grammars. The reason for this is the following; for example, if we take the grammar rules:
$S ::= E$
$E ::= E + E \mid E - E$
The expansion at the first level would look like this:

1. $.E \rightarrow .E + E$
2. $.E + E$ edges $= \{.E\}$
3. $.E \rightarrow .E - E$
4. $.E - E$ edges $= \{.E\}$
5. $.E + E \rightarrow .E + E$
6. $.E + E$ edges $= \{.E, .E + E\}$
7. $.E + E \rightarrow .E - E$
8. $.E - E$ edges $= \{.E, .E + E\}$
9. $.E - E \rightarrow .E + E$
10. $.E + E$ edges $= \{.E, .E + E, .E - E\}$
11. $.E - E \rightarrow .E - E$
12. $.E - E$ edges $= \{.E, .E + E, .E - E\}$

This example clearly demonstrates quardratic behaviour. If we would cache the edges set of one E, we could reuse it for any other E in the same level and enable us to update it with additional edges by reference. This is possible since

nodes are guaranteed to have the same edges if they are 'expected' by the same 'parent(s)'. This will make the expansion look like this:

1. $.E \rightarrow .E + E$
2. $.E \rightarrow .E - E$
3. E edges $= \{.E\}$
4. $.E + E \Rightarrow E$ edges $+= .E + E$
5. $.E - E \Rightarrow E$ edges $+= .E - E$

Now expansion completes in linear time. An additional benefit is that all the edge sets are shared between the children of the different E's, saving memory. It will reduce the worst-case number of edges to $N * \mathit{numberOfSorts}$. Originally, without sharing, this would have been $N * \mathit{numberOfProductions}^2$, so this is a massive improvement. One other benefit of this optimization is that it lessens the need for the left-factoring of grammars. In absence of look-ahead filtering expansion performance should be on par with the non-factored equivalent of the grammar; in cases where look-ahead information is used, it may be close enough to remove its necessity.

Also note that this optimization is not only useful in case we have numereous non-factored productions. It also causes the set of edges to be shared between different alternatives associated with the same left-hand-side. So even if we would share the prefixes of all the alternatives (see 5.3.1), we would still gain something.

5.2.2 Reduction

As mentioned before, since every graph node that contains the same non-terminal sort always has the same children if they are in the same level, it is sufficient to initially just follow the first edge in the edge set associated with that level. In case the node this edge points to has already been reduced, nothing needs to be done for this node or any of the other edges in this set (except record the alternative's results, in case we are parsing and not just recognizing); otherwise all other edges in this set need to be followed as normally would have been the case, to queue their associated nodes for reduction. This reduces the number of edge visits significantly. Additionally, it enables us to quickly determine whether or not the parents of a node still need to be reduced or not, without having to execute any additional sharing checks.

5.3 Graph

5.3.1 Prefix sharing

In many grammars productions exist that start with the same symbols. There is no reason to do duplicate work for these symbols. For example, if we take the grammar rule: $S ::= E + E \mid E - E$. Both E's at the start of these productions will always be derived exactly the same way for the same substring(s) and thus they are equal. Because of this, the prefixes of these two productions may as well be merged; i.e. by converting the rule into $S ::= E\,(+\,E \mid -\,E)$. It is trivial to modify our algorithm to support this. Simply by allowing every node in the graph to have more then one 'next' node and assigning the same id to both E's (to indicate they must be shared), the desired result can be achieved.

Naturally the merged prefixes of grammar rules can be arbitrarily long and are not restricted to just two partially equal rules; as long as a rule's prefix overlaps with another rule it can be merged, regardless whether or not it already has been merged with another rule. This optimization ensures we can recognize and parse LR grammars in linear time. The reason for this is that there will always be one stack at most on which things match, all alternative stacks are guaranteed to die of after visiting each level, thus each visit of a level completes in $O(1)$ time; this leads to the conclusion that the total recognize / parse time for any LR grammar will be $O(N)$, where N is the length of the input.

One may wonder why merging anything other then the prefixes of productions is not supported. While in theory sharing the postfixes of production is also possible, this is more complicated to implement (for reasons we will not elaborate on here) and opportunities to apply this optimization rarely occur in reality. For these reasons we decided not to add this feature to our recognizer / parser. Merging blocks of symbols that are not located at either the beginning or end of productions will never be possible when using our algorithm, since this may lead to incorrect results. For example if we had shared the B of the following alternatives: $S ::= ABC \mid DBE$ we would also end up with derivations for ABE and DBC, which is obviously undesirable.

6 Filtering

As is common in general parsing, ultimately you often end up with an ambiguous parse forest. In this chapter we will discuss a number of features we have implemented to filter the trees in this forest. These features are all optional and have no dependencies on eachother.

6.1 Follow restrictions

Follow restrictions work as a kind of look-ahead after the production; if the production we are trying to reduce matches the specified character(s) after the current location in the input, the reduce fails. They are mainly, though not exclusively, used to indicate eagerness.

6.2 Rejects

Reject filtering is intended for the removal of trees that are not considered to be correct at a certain location in the grammar. One could think of using it for things like filtering alternatives for identifiers that match keywords in the language, for example.

Any alternative of a production can be marked as reject. If this reject alternative matches, all other alternatives this reject belongs to are discarded. In our current implementation, rejected nodes are partially removed from the graph at parse time and partially during post-parse filtering / flattening. Removing rejected stacks while parsing prevents unnecessary work from being performed; however it can be expensive to do in case the node at the end of a rejected alternative's edge is already reduced. This is the reason why it is not completely done at parse time.

In our current implementation reject rules cannot be nested, since this may lead to incorrect behaviour. Consider the following grammar: $S ::= A(r) \mid a$, $A ::= B(r) \mid a$, $B ::= a$. The reject alternatives are marked with (r). In this case the order in which reductions are done will determine the final parse result. For example, we could follow the edges from $.a$ to $.A$ and then from $.A(r)$ to $.S$. This will cause $S.$ to be rejected, failing the parse. On the other hand, if we would have followed the edge from $.a$ to $.B$ first and then from $.B(r)$ to $.A$ we would end up with a successful parse, since A gets rejected, preventing the edge from $.A(r)$ to $.S$ from being followed. So if one writes a grammar that contains nested rejects it may, or may not do what was intended.

It is possible to resolve this problem, however this would require a second pass over the tree. Additionally it would prevent us from being able to remove parses that will be filtered in the end, before exploring them further. So it would impact performance in more then one way. This is the main reason why we decided not to support nested reject rules.

6.3 Priorities and associativity

Priorities and associativity restrictions are implemented as 'don't nest' relations. For example, if we take the grammar rule $E ::= E * E > E + E$, this means that the production $E + E$ cannot be a child of $E * E$ on either side of the production. $E ::= E + E\,\{left\}$ on the other hand means that the production $E + E$ only cannot be a child of itself on the right side of the production; similarly for $\{right\}$, but the other way around. If declared as $\{nonassoc\}$ it cannot be nested on either side of itself.

The current implementation handles priority and associativity filtering completely at parse time. While reducing, it checks whether or not the reduction is allowed for the edge we want to follow; in case it is, it is stored in a result node that is identified by the sort name of the non-terminal of the node the edge points to and the set of 'don't nest' relations associated with this node. This is because the result needs to be shared between all non-terminals with the same sort and 'don't nest' relations. For example consider the following grammar rule: $E ::= E * E\,\{left\} > (E + E \mid E - E)\{left\}$. Here $.E + E$ and $.E - E$ would be derived in the same way and thus should share their parse results, just as $E + .E$ and $E - .E$, since their 'don't nest' relations are identical.

The main advantage of handling priority and associativity filtering at parse time oppossed to implementing it as post-parse filter is that it is more efficient. We prevent the construction of results that will be thrown away later on, saving memory. Additionally it will prevent unnecessarilly exploring certain parses, because we can determine they will be filtered earlier on, increasing performance.

There is a down-side however. Handling priority and associativity filtering at parse time, disables the edge visit reduction optimization discussed in section 5.2.2, since we now always need to visit each of the node's outgoing edges at every reduction to check whether or not the nesting is allowed. There is a tradeoff here and it will depend on the grammar whether or not the loss of this optimization can be compensated by what we gain by doing priority and associativity filtering at parse time, opposed to implementing it as a post-parse filter. In our opinion this should be the case for most 'real' grammars.

6.4 Actions

Each grammar rule can have a number of actions associated with it. These actions can be used to disambiguate non-context-free ambiguities (like C typedefs) or filter alternatives that cannot be disambiguated using priorities. I will not go into detail about this feature, since actions are outside of the scope of this article. The option exists to either intergrate the execution of action into the parser or to apply them as a post-parse filter during, or after, flattening; depending on the requirements or preference of the user. However their implementation is not trivial, because of their interaction with ambiguities and the parse forest.

7 Benchmarks

In this chapter we will have a look at the performance of the current Java implementation of our algorithm.

Benchmarks were executed on a machine with the following specifications:

CPU	Intel Q6600
Memory	8 GB DDR-800
OS	Fedora Core 12
JRE	Sun 1.6.0_13 (32-bit)
JRE options	-Xmx1800m

Measurements are listed in terms of CPU-time (system + user time) and are gathered using Java's build in management tool. Before performing the benchmarks, the recognizer / parser code was executed a number of times so Java's JIT compiler has the opportunity to optimize.

7.1 Worst case

The obvious candidate for a benchmark is the following worst-case grammar:
$S ::= SSS \,|\, SS \,|\, a$
input = $a * 50$ to $a * 500$ at 50 character intervals.

Input chars	Recognize time	Parse time
50	4	6
100	18	36
150	60	130
200	154	332
250	322	672
300	596	1206
350	996	2018
400	1534	3184
450	2228	4776
500	3090	6880

Table 3: Worst-case performance scaling; times are in milliseconds.

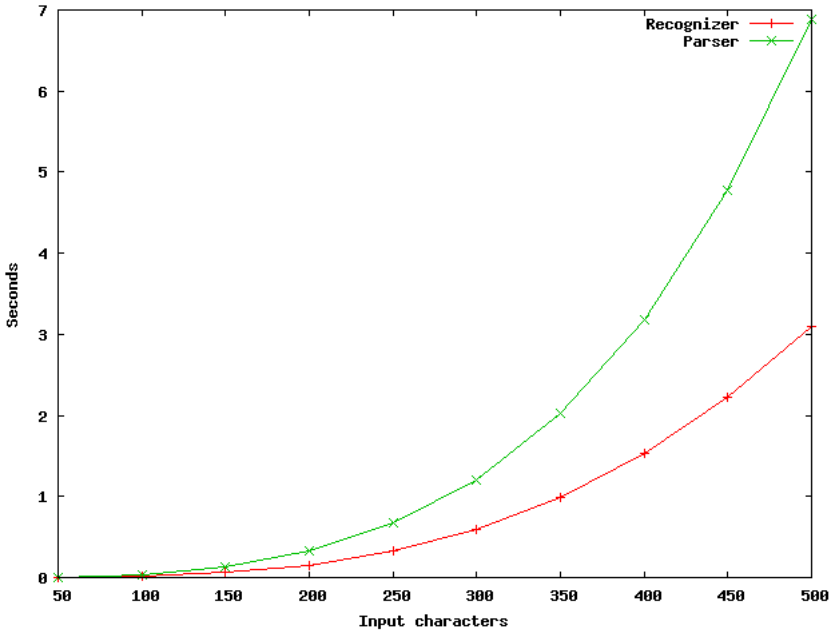

Figure 4: Worst-case recognizer and parser performance scaling.

Note however that the recognizer is optimized for speed and the parser for a balance between memory usage and speed (meaning a faster implementation is possible at the cost of increased memory footprint). Regardless of this, our recognizer and parser implementations clearly demonstate cubic worst-case behaviour, as expected. Looking at the times, our implementation seems very efficient in worst-case scenarios.

7.2 Grammar factoring

Apart from worst-case behaviour in terms of the number of ambiguous parse results, it is also interesting to look at how well we perform on worst-case grammars without ambiguous input. Here we will compare the performance of our parser between different version of a grammar; a non-factored non-prefix-shared version, a non-factored prefix-shared version and a left-factored version. The original grammar is the following:
$S ::= E+$
$E ::= a \mid E + E \mid E - E \mid E * E \mid E / E \mid E > E \mid$ ... 25 more like it ...
input = $a * 50000$, $a * 100000$, $a * 150000$ and $a * 200000$

It contains lots of left-recursion and is highly ambiguous; the input, on the other hand, is not ambiguous. Normally one would expect non-linear performance when parsing a string for this grammar. However, as can be seen in the graph below this is not the case. Performance seems to scale perfectly linear regardless of how the grammar is factored.

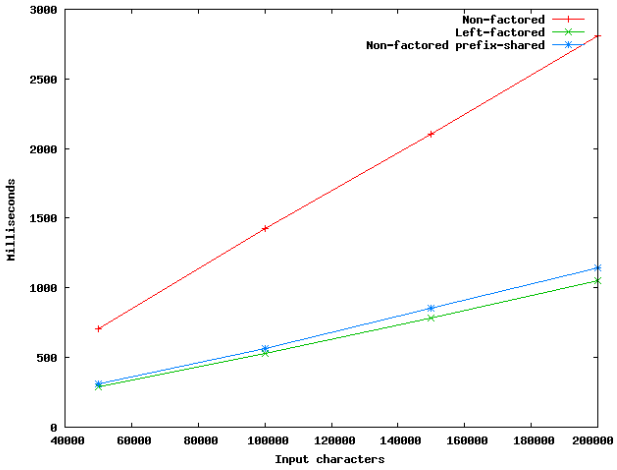

Figure 5: Grammar factoring related parse time scaling; without look-ahead filtering.

Note that the parser is about twice as slow on the non-factored non-prefix-shared version of the grammar. This is because both the number of graph nodes as the number of edge visits is significantly higher for this case. Regardless of this, parsing performance still scales linear with respect to the length of the input, due to the optimization discussed in section 5.2.1.

The non-factored prefix-shared and the left-factored versions of the grammar perform almost similar. The main reason that the parser using the left-factored version of the grammar performs slightly better is that our current implementation needs to do a little extra work to detect the sharing. However, if required, it is reasonably simple to construct an implementation of our parsing algorithm that does not suffer from this overhead. Additionally, the refactoring of the grammar also removes all ambiguities from it. Counterary to what one might think, this does not have an impact on performance in this specific case. The reason for this is that look-ahead filtering was disabled for the purposes of this benchmark. If one would enable look-ahead filtering, parsing the left-factored version would become deterministic, since it falls within the LL(1) class of grammars.

7.3 Versus non-general

We also liked to know how we compare when pinned against non-general parsers, since general parsers have the image of being inferior in terms of performance, compared to 'normal' LL or LR parsers.

We used the following basic LR grammar for our benchmark:
$S ::= E$
$E ::= E + F \mid F$
$F ::= a \mid (E)$
input = '$A ::= a \mid a + (A)$' from length 100003, to length 1000003 at 100000 character intervals.

We compared our performance results with JavaCup, which was used in combination with JFlex. We selected JavaCup, since it is one of the more

widely known and used parser generators that produce LALR parsers in Java.

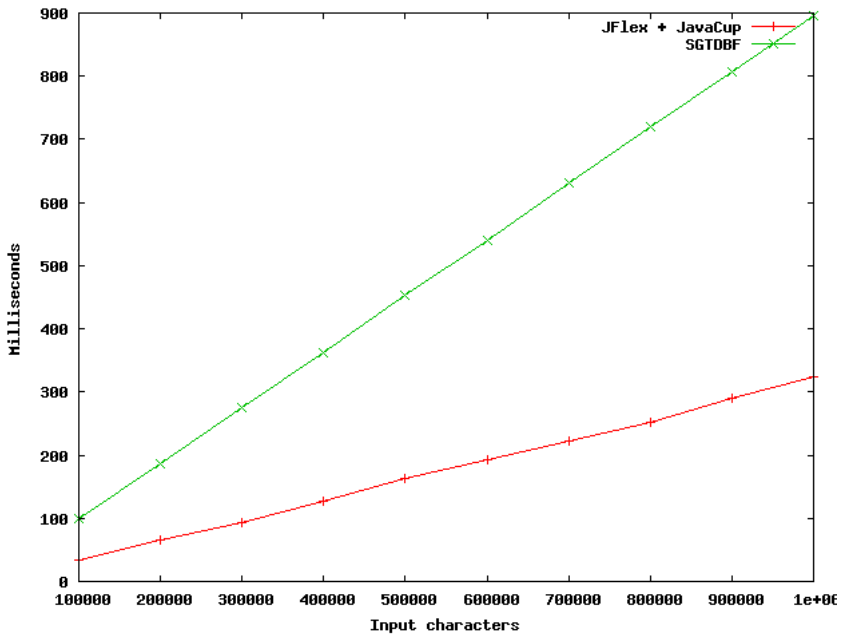

Figure 6: A parsing performance comparison with a LALR parser.

As expected JavaCup performs better. For this specific grammar, JavaCup consistently parses the given input about three times as fast. This seems like a lot, however it is a fairly reasonable result. Especially considering we are using a graph, since we need to be fully general, instead of the application's stack.

7.4 Realistic cases

Benchmarks on artificial grammars are nice for getting a point across or to indicate certain guarantees can be met, however they generally do not offer any insight about performance in realistic cases. Here we will list a number of statistics related to parsing actual source files.

Note that, for now, we just list SDF2 files; a grammar definition format which is mildly ambiguous before filtering. In the future more results will be added for other languages.

Filename	Input characters	Throughput (char/sec)	Throughput including filtering & flattening (char/sec)
ATerm.def	2393	341857	217545
C.def	10552	363862	245395
SDF2.def	18054	368448	234467
Java.def	30983	360267	232954

Table 4: Average parser throughput.

One does have to take into account that our current implementation is written in Java. If we would re-implement it in C, a performance increase of at least three to five fold is to be expected. Putting our throughput at around 1 million characters per second, while having all filters enabled, in the SDF2 case.

Regardless of this, compared to the parser we previously used, a C implementation of scannerless GLR, we achieved a performance gain of roughly 40% (i.e. our implementation parses the same file in 0.6 the amount of time), in this specific case.

8 Prototype

The development of this algorithm was approached in a rather unorthodox way. Rather then starting with the design of the algorithm and making an implementation for validation and testing purposes afterwards, we started with implementing a basic top-down recognizer, extending it to a parser, followed by testing, profiling and optimization; gradually we improved this implementation and its algorithm, until it matched our requirements. By using this approach we are able to get more direct feedback about issues and scalability and performance bottlenecks. Additionally opportunities for optimization will be highlighted that may not be immediately apparent or which may be overlooked when looking at the problem from a purely algorithmic point of view.

The main purpose of this project has always been to create a working, usable general parser implementation. The development of a new algorithm was secondary but required, since no suitable alternative was available for our purposes.

Our first implementation is written in Java. The reason we chose this language for our prototype was that is was both required for our current project and gave us the opportunity to easily change and extend it. The down side is that it makes it harder to make a comparison with other general parser implementations, which are mainly written in C and thus have a major performance advantage.

9 Future work

Numereous things can still be improved or changed. Mostly these involve optimizations or implementation improvements. We will not go into detail, but just list a number of ideas instead.

- Prefix sharing can be implemented more efficiently by moving more logic to the generator, so parts of productions do not have to be merged dynamically any more.
- We can pre-construct a data structure which contains sort to alternatives mappings in combination with their look-ahead information, so we can obtain relevant alternatives more efficiently.
- Nullable derivations or trees can be pre-computed or dynamically cached, improving efficiency.
- Matching could be handled more efficiently (more bottom-up like, so we 'touch' less terminals).
- Pre-computed information about priority / associativity restrictions can be used to handle filtering more efficiently.
- Some rejects can be statically computed and used as a kind of 'disallowed' look-ahead. This prevents alternatives of which we know they will die in advance from being explored.
- Multi-core / processor support is relatively easy to add and may be interesting to explore as possible performance booster for certain cases.

Finally, it would be interesting to create a C implementation of our parsing algorithm, both to see how far we can push the parser's throughput and to be able to make a broader and more accurate performance comparison with implementations of other (general) parsing algorithms. We expect to do fairly well in such a showdown.

10 Conclusion

In this article we described our parsing algorithm, discussed optimizations that can be applied to it and gave an impression of its capabilities. To summarize, we developed a general parsing algorithm that is both easy to comprehend and scales well, regardless of the input or grammar used. The best of both worlds, combined into one beautiful, elegant solution.

11 References

TODO:
-GSS
-LL
-LR
-SPPF (and binarized)
-GLR
-SGLR (and ASF+SDF-Meta)
-JFlex
-JavaCup
-SDF2