A Backtracking LR Algorithm for Parsing Ambiguous Context-Dependent Languages

Adrian D. Thurston and James R. Cordy

School of Computing
Queen's University
Kingston, ON, Canada
{thurston, cordy}@cs.queensu.ca

Abstract

Parsing context-dependent computer languages requires an ability to maintain and query data structures while parsing for the purpose of influencing the parse. Parsing ambiguous computer languages requires an ability to generate a parser for arbitrary context-free grammars. In both cases we have tools for generating parsers from a grammar. However, languages that have both of these properties simultaneously are much more difficult to parse. Consequently, we have fewer techniques. One approach to parsing such languages is to endow traditional LR systems with backtracking. This is a step towards a working solution, however there are number of problems. In this work we present two enhancements to a basic backtracking LR approach which enable the parsing of computer languages that are both context-dependent and ambiguous. Using our system we have produced a fast parser for C++ that is composed of strictly a scanner, a name lookup stage and parser generated from a grammar augmented with semantic actions and semantic 'undo' actions. Language ambiguities are resolved by prioritizing grammar declarations.

1 Introduction

To successfully parse modern programming languages such as C, Java and C# requires an ability to handle context dependencies. We must lookup the meaning of an identifier to determine what kind of symbol we are dealing with before we proceed to parse the identifier. The established practice for dealing with this problem is the "lexical feedback hack." During forward parsing, semantic actions are responsible for maintaining lookup tables. The lexical analyzer is then responsible for querying the type of an identifier before sending it to the parser. We have many tools which support this method of parsing.

A different language classification criteria is ambiguity. To parse context-free languages that are ambiguous requires an ability to pursue at least one parse given multiple potential parses. Again, we have many tools for generating parsers from ambiguous grammars. The techniques available to use include GLR, Earley parsing, and generalized recursive descent.

Languages that are both context-dependent and ambiguous are considerably more difficult to parse than languages with just one of these properties. They require that our facilities for considering alternatives accommodate our need to maintain and query global state. For example, we may pursue one potential parse, ma-

nipulating global state while doing so, only to discover that we have made the wrong guess and we must give up and try an alternative parse. Before we can however, our manipulation of the global state must be abandoned. Since the nature of these manipulations are determined by the semantics of the language to be parsed, they must be programmed by the user of the parser generator. They cannot be declared by the user and generated automatically by the parser generator.

C++ is an example of a language that has context dependencies and which is ambiguous. We very rarely find C++ parsers to be generated from a grammar.

One approach to parsing these languages is to convert a standard LR parser into a backtracking parser. Examples of such systems include BtYacc [4], Basil [20], Ratatosk [15] and Lark [7]. There are a number of advantages to pursuing a backtracking LR approach. The parser will inherit the speed of LR parsing, the simplicity and power of the bottom-up semantic action model, and the ease-of-use of backtracking, giving us a natural ability to handle ambiguities.

There are two problems with simply endowing an LR parser with backtracking which make it difficult to apply the approach to the parsing of ambiguous context-dependent languages. Existing backtracking LR systems only backtrack at the level of parsing. They do not elevate backtracking to the level of semantic actions. We therefore cannot backtrack over any attempted parse that has modified the global state in preparation for handling context dependencies. Secondly, with these systems it is difficult or impossible for the user to specify which potential parses should be preferred when the grammar contains ambiguities.

In this work we have solved these two problems. The forward parsing phase is free to manipulate the global state because our backtracker invokes semantic undo actions during backtracking. Semantic undo actions can be used to revert the effects of the forward phase. Secondly, we have devised a method of ordering the attempts of conflicting actions to achieve a user-controlled and predictable parse of an ambiguous grammar.

This approach has been successfully used to write a grammar-based C++ parser in a straightforward manner. Reduction actions are free to change the global data structures that are used to determine the type of identifiers. This may entail pushing a namespace to the declaration stack, or inserting a class name into a dictionary. Immediately below a reduction action which modifies global state, the disciplined programmer is responsible for implementing the reverse of the reduction action, for example popping the declaration stack or removing an item from a dictionary. This allows the parser to correctly backtrack.

At strategic points, such as statement boundaries, parse trees may be committed and non-reversible actions executed. In these non-reversible actions, which we call final actions, the user may perform permanent tasks such as constructing an AST or printing the result of the parse. Finally, rules for resolving C++ language ambiguities are implemented by ordering mutually ambiguous productions in the order in which they should be tried.

In the next section we discuss the various grammar-based approaches to generating parsers for ambiguous context-dependent languages, including existing backtracking LR systems. In Section 3 we describe our enhancement to backtracking LR which allows ambiguities and context dependencies to co-exist. In Section 4 we describe our enhancement which puts control of the backtracking strategy in the hands of the user. In Section 5 we show how our parsing algorithm can be applied to C++.

2 Related Work

2.1 GLR

The generalized LR parsing method [10] is one approach to parsing ambiguous context-dependent languages. Due to inherent parallelism, use of GLR relies on post-processing of the parse trees.

In the course of building a parse table, standard LR parser generators will emit an error upon discovering shift-reduce or reduce-reduce conflicts. Some parser generators may choose one action by default and produce code that runs, but does not necessarily work as intended. Others generators may simply fail to proceed.

GLR parser generators will accept any grammar and will always produce a working parser regardless of the number of conflicts contained in it. At run time, the generated parser will take conflicts in stride; when encountering multiple actions on a single arc of the parse table, it will simultaneously take all actions. From then on all potential parses are parsed in lockstep. Since parsing in lockstep requires multiple stack instances, much research as gone into managing a shared stack which conserves space and computation time, making the approach much more practical.

The GLR method can be applied very successfully to the parsing of ambiguous languages, but we experience problems when we introduce context dependencies. The need to maintain type information while concurrently pursuing multiple parses requires that we also maintain multiple copies of the global data structures which store the type information.

It may be possible to extend the idea of automatic parse forest sharing to the global context dependency state. After all the parse tree is itself a global state. No work in this area is known. However, if we consider that the structure of the global state information is dependent on the language being parsed, it seems doubtful that automatic sharing of context dependency information is a task that can be moved to the parser generator. C++ has a unique and complicated namespace structure, accompanied by many nontrivial name lookup rules. Lookup of template specializations requires an implementation of the type system. Models of the C++ namespace have been produced [18], though these exist only for the purpose of human understanding.

Rather than attempt to parse in a single pass, the common approach to parsing ambiguous context-dependent languages with a GLR parser is to attempt all possible parses irrespective of context dependencies. The tokenizer will yield tokens which simultaneously carry multiple types [1] and the GLR parser will acquire all possible interpretations. Following the parse, or at strategic points, disambiguation rules [11, 21] eliminate the parse forests which are illegal according to context dependency rules.

2.2 Generalized Top-Down

Generalized top-down parsing with full backtracking is a very flexible parsing method that can be applied to ambiguous languages. When provisions are made for handling left recursion a wide range of parsing tasks can be implemented. The TXL [2] programming language, a language designed for prototyping and manipulating language descriptions, tools and applications, contains a parser which implements the generalized top-down parsing method. It allows the definition of arbitrary context-free grammars, according to which it will parse input using a top-down parser with full backtracking. This parsing strategy has been shown to be very useful in language design and software renovation tasks.

A key advantage of this method is that it puts the user in control of the parsing strategy when the grammar is ambiguous. The preferred order in which to attempt to parse mutually ambiguous alternatives can be specified locally. This is advantageous for grammar composition tasks in software engineering [3]. The innermost backtracking strategy makes it easy for the user to predict the result of the parse.

Definite clause grammars (DCGs) [17] are a syntactic shorthand for producing parsers with Prolog clauses which represent the input with difference lists. Prolog-based parsing is a very expressive parsing technique which can be considered a generalized top-down parsing method. Prolog's backtracking guarantees that all possible grammar derivations are tested. Prolog clauses may be embedded in grammar productions acting as syntactic or semantic predicates.

It is conceivable to enhance a generalized top-down parser with semantic actions for maintaining global state and semantic undo actions for reverting changes to the global state when the parser must backtrack, though work in this area is not known to us.

The primary disadvantage of this parsing approach is that it can result in very long parse times. Full backtracking often induces redundant reparsing when grammar alternatives contain common prefixes. Packrat parsing [6] attempts to solve this problem by memoization of parse trees.

Another approach to improving the performance of generalized top-down parsing is described in [9]. In this work, grammars which have the follow-determinism property exhibit improved parsing performance because the follow sets can be used to prune the search space.

ANTLR [16] is another parsing tool which manages the tradeoff between parsing power and performance. Parsers generated by ANTLR normally have the LL(k) property, with $k > 1$. In recent versions it supports LL(*) which allows k to roam and eliminates the need to explicitly set k. Since many languages in use have LL properties, this is often sufficient. However, for cases when it is not, ANTLR is able to revert to a generalized top-down method. Should the LL(*) method fail, the parser automatically enters into a full backtracking mode with memoization.

2.3 Backtracking LR

Like GLR parser generators, a backtracking LR parser generator will accept any grammar and will always emit a working parser. Upon encountering a conflict, the run time system will try the first action, remembering the choice that was made, and continue parsing in a single thread. Later on, should a parse error be encountered it will undo its parsing up to the most recent choice point, then try the next possibility.

Where a standard top-down parser with full backtracking will revert to the innermost choice point with respect to the grammar, a backtracking LR parser will revert to the rightmost, topmost choice point with respect to the LR stack. Such a strategy will eventually try all possible parses.

The primary advantage of backtracking LR parsers is that they retain the speed of LR parsing when the grammar is deterministic. If backtracking can be kept to a minimum, the cost of processing nondeterministic grammars need not be prohibitive. Also, a backtracking parser will alway yield a single parse on a single pass. Yielding a single parse is a usual requirement of programming language processors.

Merrill [14] describes changes made to Yacc to support backtracking for the purpose of parsing C++. Power and Malloy [19] suggest using a backtracking LR parser for the initial deployment of a C++ grammar due to the language's incompatiblity with standard LR parsing and the ease with which backtracking LR allows complicated grammars to be deployed from their specification.

Backtracking LR is not without is drawbacks. Some problems come with the territory. In [10], it is shown that it is easy to write a grammar that exhibits exponential behaviour when given to a backtracking LR parser generator. Our parsing method also succumbs to such grammars. Users must themselves guard against producing poorly performing grammars. Also, hidden left recursion causes us problems and must be avoided.

Other problems we aim to fix in this work. An inability to backtrack over semantic actions which modify global state and an inability to control the parsing of ambiguous constructs are two problems which make it difficult to apply backtracking LR in practice. These problems are discussed in more detail in the following sections.

2.4 BtYacc and Basil

BtYacc is a backtracking LR parser generator derived from Berkeley Yacc. When a BtYacc parser proceeds without encountering any conflicts, regular reduction actions are executed. Since no backtracking is possible these actions can have side effects. We refer to these as final actions. When the parser encounters a conflict in the parse tables, it goes into trial parsing mode where it stops executing final actions, but continues to execute a second class of actions, which are specified differently in the grammar. We refer to these as trial actions. Since the reductions that the trial actions are associated with may be undone, and when this happens there is no way to revert the effects of these actions, trial actions cannot have any side effects.

When facing a shift-reduce conflict a BtYacc parser will always choose to shift first, then reduce. This choice of action ordering makes it difficult for the user to control the relative priority of mutually ambiguous productions. For example, when a BtYacc parser built with the following grammar is given the input `a b`, the policy of shifting first will always yield the

parse `S -> a b`, regardless of the user's intentions.

```
S -> A B          A -> a          (1)
S -> a b          B -> b
```

The Basil parser generator leaves the ordering of conflicts up to the programmer. The ordering of shift and reduce actions is not in any way dependent on the grammar. If one writes an ambiguous grammar, there is more work to be done before a desired parse strategy is attained.

BtYacc allows the programmer to invoke a commit command from reduction actions. When one issues a commit command the entire parse is committed to. This is a coarse-grained solution that is suitable for committing the parse when a particular token is seen.

2.5 Elsa

Elsa [13] is a C++ parser produced with the Elkhound GLR system. Elsa demonstrates a successful application of the post-parse disambiguation approach. It parses input regardless of the meaning of C++ identifiers, then later rejects the returned parse trees which do not satisfy the C++ name lookup rules.

2.6 Keystone

Keystone [12] is a C++ parser written using BtYacc. Keystone suffers from problems related to BtYacc supporting only trial and final actions. Since the effect of trial actions cannot be undone, they are unable to modify the global state and therefore cannot perform tasks such as changing the current name scope so that subsequent parsing can lookup names correctly.

The program in Figure 1 demonstrates why simply supporting trial and final actions is insufficient for parsing C++. The second last line of the example is a declaration of an object `g` of type `E` that is initialized with the value that `f` returns. The last line is a declaration of a function `h` that returns an object of type `E` and has one unnamed parameter of type `D`. To distinguish between these requires examining the meaning of the symbols `f` and `D`. Once the parser arrives at the initial open parentheses, it must enter into a trial parse mode before it can decide upon the nature of the declaration. This trial parsing must continue past an unknown number of tokens until the `f` and `D` symbols are parsed and their meaning deciphered.

```
namespace ns1
{
    template <class T> struct C
    {
        struct D;
        static int f( int )
            { return 0; }
    };
}

namespace ns2
{
    struct E
    {
        E( int ) {}
    };
}

ns2::E g( ns1::C< ns2::E >::f(1) );
ns2::E h( ns1::C< ns2::E >::D );
```

Figure 1: C++ code demonstrating a need to backtrack over global state modifications.

To properly lookup `f` and `D` we must be aware of the qualifying scope, in both cases this is `C`. The scanner cannot perform this task because it is unable to correctly parse template parameters. The class template `C` may indeed have template specializations and the template parameters must be used to look these up. If we try to communicate the qualifying scope using some form of attribute transfer in semantic actions we are foiled by the fact that name lookup happens in the scanning stage, before tokens are passed to the parser. The scanner could cheat and peek at the stack to try and guess the correct context, but it has no sense of what to expect on the stack.

The most sensible and straightforward way to propagate qualification information from the parser to the name lookup stage is for the parser to maintain the qualification information in a global variable and for the lookup stage to consult this variable when needed. But since we are in trial parse mode and any of our

parsing may get undone, we are forbidden from modifying global variables.

We therefore require a parsing model that allows us to parse both in the forward and backwards direction over semantic actions which modify the global state. The first contribution of this work, described in the Section 3 addresses this need.

3 Backtracking Semantic Actions

To permit parsers to backtrack over semantic actions which modify the global state, we have introduced the notion of semantic undo actions. In all, the semantic action known the user of Yacc has been specialized into three types: trial, undo and final actions. Each is appropriate for a particular kind of task.

3.1 Trial Actions

Trial actions are always executed immediately upon a reduction. They are appropriate for executing actions which will affect future parsing. The user is free to make any modifications to the global state which can be reverted. In the context of programming languages, this usually involves tasks such as inserting or deleting dictionary items, attaching or detaching list items, or pushing or popping stack items. Should enabling the reversibility of an action require saving some data, as in the case of popping from a stack, it can be stored in the data element representing the reduced tree node.

Since we may need to unparse the reduced node, we always preserve the children of a reduction. This results in a stack of parse trees. In many applications preserving the entire parse tree is wasteful. In Section 3.4 we describe commit declarations, which give the parser hints as to when it is allowed to free parse tree nodes. When nodes are freed at regular intervals, the cost of preserving reduced data is marginal.

3.2 Undo Actions

Undo actions are used for reverting side effects of trial actions. They are invoked as the parse loop undoes parsing. This happens when the parser encounters an error and must backtrack to the most recent decision. Within the unparsing loop one item is popped from the top of the parse stack. If the node is a token, the token undo action is executed and it is pushed back to the input stream. If the node is a nonterminal, the nonterminal's undo action is executed, the node is discarded and the children of the node are pushed onto the parse stack. In both cases, if the recently popped node contained an alternate action then unparsing terminates and forward parsing resumes, with the initial action to take guided by the previous choice which was stored in the popped node.

3.3 Final Actions

Final actions are executed when a reduction can never be undone. They are free to make irreversible changes to the global state and should perform all work that is not required for the parser to produce a correct parse tree. This could be writing out the result of the parse, building an AST or freeing memory.

The execution of final actions is triggered by commit operations, which are described in the next section. Following a commit operation, if there are no backtracking decision points remaining, then all pending final actions are invoked up to the top of the parse stack and all children underneath the nodes of the parse stack are freed. This reduces the parser's memory usage to that of a standard LR parser.

3.4 Declarative Commit Points

Programming language ambiguities are often localized. For example, in C++ once the parsing of a statement completes, any alternative parses of the statement's tokens can be discarded. Discarding alternatives drastically improves parser performance by eliminating fruitless reparsing, expediting the execution of final actions, reducing the parser's memory usage, and enabling it to report erroneous input in a prompt and accurate manner.

We allow the user to declare localized commit points within a grammar. When the parser arrives at a commit point, it deletes any alternatives within the commit point's scope. Al-

```
orderState( tabState, prodState, time ):
    if not tabState.dotSet.find( prodState.dotID )
        tabState.dotSet.insert( prodState.dotID )
        tabTrans = tabState.findMatchingTransition( prodState.getTransition() )

        if tabTrans is NonTerminal:
            for production in tabTrans.nonTerm.prodList:
                orderState( tabState, production.startState, time )

                for all expandToState in tabTrans.expandToStates:
                    for all followTrans in expandToState.transList
                        reduceAction = findAction( production.reduction )
                        if reduceAction.time is unset:
                            reduceAction.time = time++
                        end
                    end
                end
            end
        end

        shiftAction = tabTrans.findAction( shift )
        if shiftAction.time is unset:
            shiftAction.time = time++
        end

        orderState( tabTrans.toState, prodTrans.toState, time )
    end
end

orderState( parseTable.startState, startProduction.startState, 1 )
```

Figure 2: Ordering shifts and reduces to emulate a generalized top-down strategy.

---

ternatives deeper in the stack are not affected, allowing the user to delete alternatives which are of no interest, while preserving earlier alternatives that are still plausible.

There are two forms of commit points: reduction-based and shift-based. A reduction-based commit point is associated with an entire grammar production. When the production is initially reduced, all alternatives embedded underneath the new nonterminal are deleted. A shift-based commit point is embedded into a production someplace before the end. When the first character of the grammar item to the right is shifted, the grammar items to the left are committed.

The need for shift-based commit points arises due to the block structutre of programming languages. For example, it desirable to be able to commit the signature of a function definition when we begin to recognize the statements contained in it.

```
definition -> type name ( param_list )
  commit { definition_list }
```

Without shift-based commit declarations we would be required to restructure our grammar if we wanted to commit the function signature before entering the body.

Note however, that until a character which follows an entire production is recognized, it is not guaranteed that the production will be reduced, which means that shift-based commit declarations may affect alternate parses. They should therefore be used ahead of relatively unambiguous language constructs.

A commit declaration does not guarantee that the tree underneath it will be freed because earlier alternatives may still exist. To

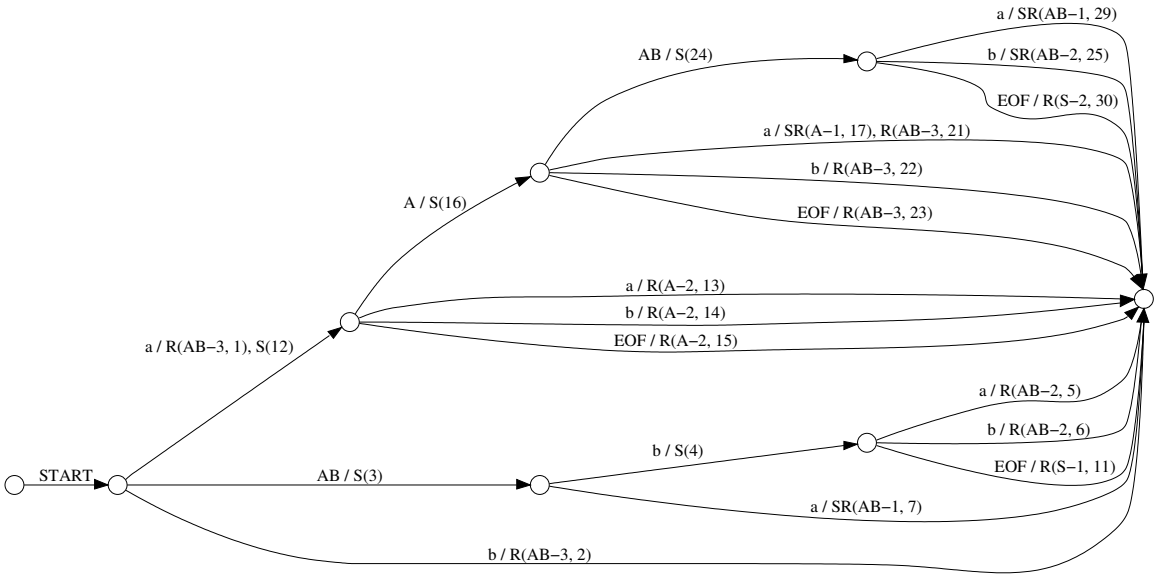

Figure 3: LALR(1) parse tables of grammar (2). The unique timestamp assigned to each action is shown. The resulting action ordering emulates a top-down parsing strategy.

avoid redundant tree traversal, an implementation of this technique should optimize the commit operation such that it only incurs a cost when alternatives exist within the scope of the commit.

4 User-Controlled Parsing Strategy

The second contribution of this work addresses the need for user-controlled parsing of ambiguous language constructs. We have improved the backtracking LR approach by giving users localized, grammar-based control of the order in which conflicting LR actions should be attempted. To accomplish this we have devised an algorithm which traverses the parser's state tables and assigns an ordering to the actions such that a generalized top-down backtracking strategy is emulated. This algorithm is shown in Figure 2. By emulating a generalized top-down approach we give the user the ability to specify mutually ambiguous productions in the order in which the parser should attempt to parse them. We also ensure that the parser will prefer the longest possible match of sequences.

Following the construction of the parser state tables, our algorithm traverses both the state tables and the grammar productions in parallel. The order of the traversal is guided by a top-down interpretation of the grammar productions. We start with a nonterminal as our goal and consider each production of the nonterminal in succession. As we move along a production's right-hand side we recurse on nonterminals before we proceed past them. When we visit a shift transition we assign a time to it if one has not already been given. Following the traversal down the right-hand side of a production, we find the transitions which contain the reduction actions of the production instance and assign a time to each reduction action if one has not already been given.

To limit the traversal and guarantee termination we visit a parse table and production state pair only once. This is accomplished by inserting the dot item of the production state into the parse table state and proceeding with the pair only when the dot item previously did not exist. Note that this yields dot sets identical to those computed during the LR parse table construction.

The following grammar demonstrates the

ability of our algorithm to properly order mutually ambiguous productions and to prefer the longest match of a sequence. The corresponding parse tables are given in Figure 3.

```
S -> AB b          AB -> AB a
S -> a A AB        AB -> AB b     (2)
A -> A a           AB ->
A ->
```

The timestamps assigned by our action ordering algorithm are shown in each transition action. There are two conflict points. The first is in the transition leaving the start state on the input character `a`. This transition will first induce a reduction of `AB`, then a shift. This represents the pursuit of the first production of `S`. The second conflict represents the choice between extending the `A` sequence and beginning the `AB` sequence when we are matching the second production of `S`. In this case the parser first shifts and reduces `A` to pursue a longest match of `A`.

4.1 Parsing Example

Figure 4 shows the run-time behaviour of the backtracking LR parser generated from grammar (2) when run on the input `a b a`. Normally, an LR parser discards the nodes it has popped off of the stack during a reduction. Since we must be prepared to backtrack we preserve these nodes as children of the newly reduced node. These children nodes are shown underneath their parent in the middle column. Though there are none in this example, use of commit declarations to clear the retry points causes these nodes to be freed.

The retry point `(r:2)` is recorded in the reduced node `AB` of the first reduction. When the unparsing loop arrives at this retry point it transfers it to the first input symbol and resumes forward parsing. The forward parsing loop will then read the retry point and shift instead of reduce.

4.2 Out-of-Order Parse Correction

Unfortunately it is possible to find grammars whose mutually ambiguous productions will not be parsed in order. As it turns out, the parse strategy we aim for is reserved only for true top-down parsers. LR parsers attempt to parse common production prefixes in parallel. This allows parsers to run very fast, but it can inhibit us from achieving a top-down strategy because it shuffles the order of backtracking decisions by delaying the branching of productions. For example, consider the following

Action	**Stack**	**Input**
		a b a EOF
reduce	AB(r:2)	a b a EOF
shift	AB(r:2) a	b a EOF
reduce	AB AB(r:2) a	b a EOF
shift	AB b AB(r:2) a	a EOF
reduce	AB AB b AB(r:2) a	a EOF
shift	AB a AB b AB(r:2) a	EOF
ERROR		
unshift	AB AB b AB(r:2) a	a EOF
unreduce	AB b AB(r:2) a	a EOF
unshift	AB AB(r:2) a	b a EOF
unreduce	AB(r:2) a	b a EOF
unshift	AB(r:2)	a b a EOF
unreduce		a(r:2) b a EOF
shift	a	b a EOF
reduce	a A	b a EOF
reduce	a A AB	b a EOF
shift	a A AB b	a EOF
reduce	a A AB AB b	a EOF
shift	a A AB a AB b	EOF
reduce	a A AB AB a AB b	EOF

Figure 4: Parsing of the string `a b a`, according to grammar (2). Since in this case we must be prepared to backtrack, we preserve popped nodes. Commit declarations can be used to clear retry points, which in turn causes these preserved nodes to be freed.

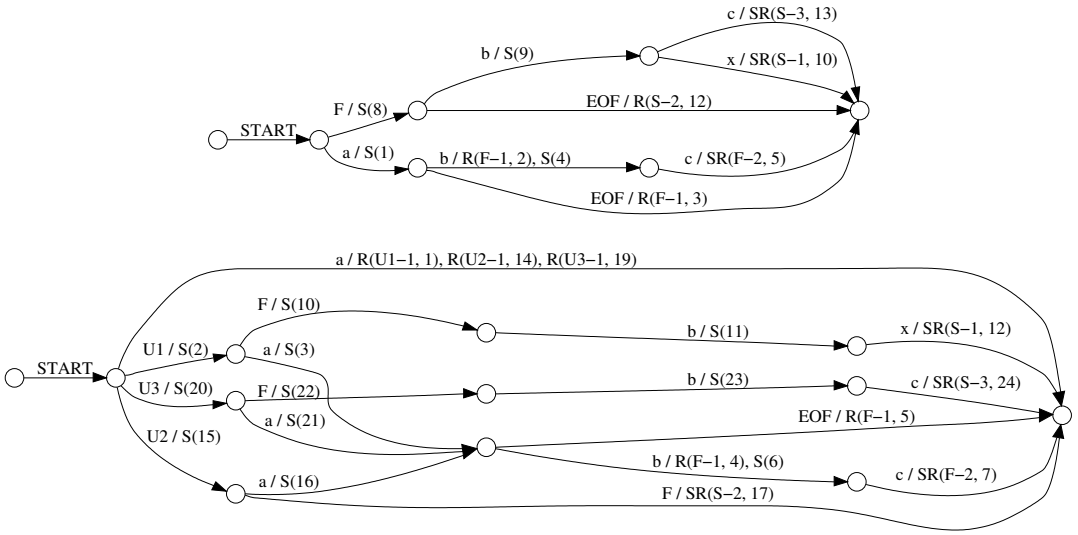

Figure 5: LALR(1) parse tables before and after adding unique empty productions that force the parser to select on the possible derivations of S before selecting on the possible derivations of F.

---

grammar.

```
S -> F b x          F -> a
S -> F              F -> a b c      (3)
S -> F b c
```

When given the input string `a b c`, a generalized top-down parser will attempt the following derivations. Note that it branches on the possible derivations of `S` first, then branches on the possible derivations of `F`.

```
S -> F( a ) b x         fail
S -> F( a b c ) b x     fail
S -> F( a )             fail
S -> F( a b c )         accept
```

Our backtracking LR parser does not yield the same parse. Since all three `S` productions have a common prefix `F`, the prefix will be parsed once for all productions. The parser will branch on the possible derivations of `F` first, then later branch on the derivations of `S`. This out-of-order branching causes an out-of-order parse. When we trace the parser's behaviour, we find that it first reduces `F -> a`, then succeeds in matching `S -> F b c`.

```
S -> F( a ) b x         fail
S -> F( a )             fail
S -> F( a ) b c         accept
```

The offending LR state tables are shown in the first part of Figure 5.

Fortunately we are able to solve this problem easily. When we find our input to be parsed out of order with respect to our grammar, we can force a correct order by introducing unique empty productions at the beginning of the productions which are parsed out of order. The unique empty productions will cause an immediate reduce conflict before any inner productions are reduced, effectively allowing us to force the slower top-down parsing approach in a localized manner. We can change grammar (3) to the following and achieve the same parsing strategy as a top-down parser.

```
S -> U1 F b x    F -> a       U1 ->
S -> U2 F        F -> a b c   U2 ->     (4)
S -> U3 F b c                 U3 ->
```

The second part of Figure 5 shows the LR tables after forcing an initial selection on `S`. An ability to force a branch point is very useful when unioning grammars because it frees us from analyzing how the LR state tables interact. The cost of forcing a branch point lies in increasing the number of states and lengthening parse times. However we do so only locally, and only when required.

4.3 Semantic Conditions and Error Recovery

An advantage of a our approach is that it affords simple implementations of semantic conditions and error recovery. Should a reduction action detect that a parse violates a semantic condition, it can invoke the backtracker and the parser will move on to an alternative parse.

With a simple enhancement to the state table generator in the form of an `any*` token which repeatedly matches input tokens up to a termination point, it is possible to implement a well-known error handling technique. The following error handler consumes input until the input stream and parser go back into a stable state where correct parsing may resume.

```
statement -> U1 for_block
statement -> U1 while_block
...
statement -> U2 any* ;
```

5 Case Study: C++

To validate our ideas we have applied them to the parsing of C++. The C++ language has a reputation of being very difficult to parse using grammar-based techniques. Many C++ compilers use a hand-written recursive descent approach, including GCC, OpenC++ and OpenWatcom.

Our parser is composed strictly of a scanner, a name lookup routine inserted between the scanner and parser, and a grammar. Some productions are accompanied by trial, undo and/or final actions. Backtracking performance is improved with a small number of commit declarations that we associate with C++ declarations, statements and the opening of block structures.

5.1 Use of Semantic Undo Actions

We use semantic undo actions to revert the effects of trial actions which manipulate the C++ name hierarchy and prepare for name lookups by posting name qualifications. An example is given in Figure 6. These empty nonterminals open and close a C++ declaration. They are used to initialize the data structure into which we collect information about the declaration. This information will be used when we record the declaration in the C++ name hierarchy. During forward parsing, we push a fresh instance of the structure to a stack when opening a declaration and pop the structure when closing a declaration. During unparsing we revert these actions.

```
globals {
    Stack<bool> templDecl;
    Stack<DeclarationData> declData;
};

declaration_start:
    try {
        declData.push( DeclarationData() );
        declData.top().init();
        declData.top().isTemplate =
            templDecl.top();
    }
    undo {
        declData.pop();
    };

nonterm declaration_end {
    DeclarationData declData;
};

declaration_end:
    try {
        $$->declData = declData.pop();
    }
    undo {
        declData.push( $$->declData );
    };
```

Figure 6: Semantic actions which wrap declarations.

In all, semantic undo actions are relatively sparse. In our C++ grammar there are 576 productions, 61 of which have semantic undo actions. Many of them are concerned with removing items from dictionaries or popping items from stacks and are similar.

5.2 Resolving Ambiguities

C++ has a number of ambiguities documented in the language standard [5]. These ambiguities can be resolved according to the standard by utilizing the parsing strategy of our backtracking LR algorithm. In the remainder of this section we describe how we have implemented the resolution of each ambiguity.

5.2.1 Ambiguity 1: Section 6.8

There is an ambiguity between declaration statements and expressions statements. To resolve this ambiguity, we follow the rule that any statement that can be interpreted as declaration is a declaration. We program this by specifying the declaration statement production ahead of the expression statement production.

```
struct C {};
void f(int a)
{
    C(a)[5];      // declaration
    C(a)[a=1];    // expression
}

statement: declaration_statement commit;
statement: expression_statement commit;
```

5.2.2 Ambiguity 2: Section 8.2, Para 1

There is an ambiguity between a function declaration with a redundant set of parentheses around the parameter declaration and an object declaration with an initialization using a function-style cast expression. Again, we apply the rule that any program text that can be a declaration is a declaration. Therefore we must prefer the function declaration. The resolution of this ambiguity is handled automatically by our parsing strategy, because parameter specifications are innermost relative to object initializations.

```
struct C {};
int f(int a)
{
    C x(int(a));  // function declaration
    C y(int(1));  // object declaration
}

init_declarator:
    declarator initializer_opt;

declarator:
    ptr_operator_seq_opt declarator_id
    array_or_param_seq_opt;

array_or_param_seq_opt:
    array_or_param_seq_opt array_or_param;
array_or_param_seq_opt: ;

array_or_param:
    '[' constant_expression_opt ']';
array_or_param:
    '(' parameter_declaration_clause ')'
    cv_qualifier_seq_opt exception_spec_opt;
initializer_opt: '=' initializer_clause;
initializer_opt: '(' expression ')';
initializer_opt: ;
```

5.2.3 Ambiguity 3: Section 8.2, Para 2

In contexts where we can accept either a type-id or an expression, there is an ambiguity between an abstract function declaration with no parameters and a function-style cast. The resolution is that any program text which can be a type-id is a type-id. We program this by specifying the productions which derive type-ids ahead of the productions which derive expressions.

```
template<class T> class D {};
int f()
{
    sizeof(int());  // sizeof type-id
    sizeof(int(1)); // sizeof expression

    D<int()> l;   // type-id argument
    D<int(1)> l;  // expression argument
}

unary_expression: KW_Sizeof '(' type_id ')';
unary_expression: KW_Sizeof unary_expression;

template_argument: type_id;
template_argument: assignment_expression;
```

5.2.4 Ambiguity 4: Section 8.2, Para 7

In contexts which accept both abstract declarators and named declarators there is an ambiguity between an abstract function declaration with a single abstract parameter and an object declaration with a redundant set of parentheses. This arises in function parameter lists. The resolution is to consider the text as an abstract function declaration with a single abstract parameter. We program this by specifying abstraction declarators ahead of named declarators.

```
struct C {};
void f(int (C)); // anon function ptr param
void f(int (x)); // variable parameter

parameter_declaration:
decl_specifier_seq param_declarator_opt
parameter_init_opt;

param_declarator_opt: abstract_declarator;
param_declarator_opt: declarator;
param_declarator_opt: ;
```

5.3 Parsing Speed

Our parsing method is competitively fast. Though meaningful timings are difficult to obtain because there are no C++ parsers which perform the exact same amount of work as ours. Admittedly, our parser is not complete; we do just enough work to obtain a nearly-correct parse. We have not implemented the complete type and expression evaluation systems, which both require a considerable effort to implement. These are necessary for looking up template specializations. This in turn affects our ability to properly lookup names in some contexts.

Nevertheless, we give a timing of our untuned and incomplete prototype and a timing of GCC on the same file to give a general sense that our method is suitable for practical tasks. On a 2.4 GHz Intel processor, our parser handles a 1.3 MB preprocessed file belonging to the Mozilla source code repository in 0.154 seconds. On the same file, the g++ 3.3.5 compiler reported that the sum of scanning, parsing and name lookup took 0.980 seconds. This information was obtained with the `-ftime-report` option.

6 Future Work

The problem of detecting out-of-order parses and eliminating them by inserting unique empty productions is a task that we leave up to the user. It would be desirable to have a static analysis which was able to detect out-of-order parses and automatically correct the problem by inserting unique empty productions where appropriate. In initial investigations we found this to be a difficult problem, closely related to the detection of ambiguities in context-free grammars, which has been shown to be an undecidable problem [8].

An alternate strategy for guaranteeing that no out-of-order parses are possible might be to begin by inserting unique empty productions at the beginning of every production, then later eliminate those which are unnecessary. Maintaining in-order parsing may be easier than detecting out-of-order parsing.

When we insert unique empty productions at the beginning of every production we guarantee that no input is parsed out of order. This causes a backtracking LR parser to behave like a generalized top-down parser. This idea was tested with our C++ parser. We inserted unique empty productions at the beginning of every production which did not contain left recursion, direct, indirect or hidden. The resulting parser produced the same output, but performance slowed by a factor of 20, and there was an increase in the number of states by a factor of 2.

If automatic detection of out-of-order parses proves too difficult or unnecessary, it may be worthwhile to pursue methods for analyzing an explicitly specified pair of ambiguous productions for potential out-of-order parses. This would ensure that unique empty productions are added only when necessary.

7 Conclusion

In this work we describe two enhancements to a backtracking LR parsing approach which enable the parsing of languages that are both context-dependent and ambiguous.

We introduce a new class of semantic actions for reverting changes made to the global state, which we call undo actions. These actions are straightforward to program and permit the parser to backtrack over areas of input text which require preparations for handling context dependencies. Declarative commit points can be used eliminate fruitless backtracking and improve performance in a localized manner.

Secondly, we assign an ordering to conflicting shift and reduce actions that causes the parser to emulate the parsing strategy of a generalized top-down parser for many grammars. In cases where common prefixes inhibit the desired top-down strategy, unique empty productions can be inserted at the beginning of productions to force a localized top-down approach. This will guarantee that the parser attempts to parse mutually ambiguous productions in the order in which they are given. Using our method, we can apply a top-down backtracking strategy where needed for resolving ambiguities, while retaining the speed of LR parsing for sections of the grammar which are deterministic.

Acknowledgments

The authors wish to thank Nigel Horspool and Terence Parr for their assistance in understanding the relation of our work to other methods. This work is supported by the Natural Sciences and Engineering Research Council of Canada.

About the Authors

Adrian Thurston is a Ph.D. candidate at Queen's University working in the Software Technology Laboratory under the supervision James Cordy. He completed his M.Sc. also at Queen's and his B.Math (Computer Science) at the University of Waterloo. Adrian's research interests include parsing technology, source transformation and programming languages.

James Cordy is the Director of the School of Computing and Professor of Computing and Electrical and Computer Engineering at Queen's University. From 1995 to 2000 he was Vice President and Chief Research Scientist at Legasys Corporation, a software technology company specializing in legacy software system analysis and renovation. Dr. Cordy is a founding member of the Software Technology Laboratory at Queen's University and winner of the 1994 ITRC Innovation Excellence award and the 1995 ITRC Chair's Award for Entrepreneurship in Technology Innovation for his work there. He serves on a range of software engineering conference committees and has recently co-chaired several conferences and workshops including CASCON 2005. Dr. Cordy is an IBM Faculty Fellow and has been awarded IBM Faculty Innovation Awards in both 2004 and 2005.

References

[1] John Aycock and R. Nigel Horspool. Schrodinger's token. *Software: Practice and Experience*, 31(8):803–814, July 2001.

[2] James R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, August 2006.

[3] Thomas R. Dean, James R. Cordy, Andrew J. Malton, and Kevin A. Schneider. Agile parsing in TXL. *Journal of Automated Software Engineering*, 10(4):311–336, October 2003.

[4] Chris Dodd and Vadim Maslov. Backtracking Yacc, 2006. http://www.siber.com/btyacc/.

[5] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. American National Standards Institute, First edition, September 1998.

[6] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming (ICFP'02)*, pages 36–47, New York, NY, USA, 2002. ACM Press.

[7] Josef Grosch. Lark - An LALR(2) parser generator with backtracking. Technical Report 32, CoCoLab - Datenverarbeitung, September 2002.

[8] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[9] Adrian Johnstone and Elizabeth Scott. Generalised recursive descent parsing and follow determinism. In *Compiler Construction: 7th International Conference (CC'98)*, volume 1383 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[10] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. Generalised parsing: Some costs. In *Compiler Construction: 13th International Conference (CC'04)*, volume 2985 of *Lecture Notes in Computer Science*, page 89, April 2004.

[11] Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San

Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, October 1994.

[12] Brian A. Malloy, Tanton H. Gibbs, and James F. Power. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software: Practice and Experience*, 33(1):19–39, 2003.

[13] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In *Compiler Construction: 13th International Conference (CC'04)*, volume 2985 of *Lecture Notes in Computer Science*, April 2004.

[14] Gary H. Merrill. Parsing non-LR(k) grammars with Yacc. *Software, Practice and Experience*, 23(8):829–850, 1993.

[15] Torben Mogensen. Ratatosk: A parser generator and scanner generator for Gofer, 1993. ftp://ftp.diku.dk/pub/diku/dists/Ratatosk.tar.Z.

[16] Terence J. Parr and Russell W. Quong. ANTLR: A predicated LL(k) parser generator. *Software, Practice and Experience*, 25(7):789–810, 1995.

[17] Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis - A survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13(3):231–278, 1980.

[18] James F. Power and Brian A. Malloy. Symbol table construction and name lookup in ISO C++. In *Proceedings of the International Conference on the Technology of Object-Oriented Languages and Systems (TOOLS'00)*, pages 57–68, November 2000.

[19] James F. Power and Brian A. Malloy. Exploiting metrics to facilitate grammar transformation into LALR format. In *Proceedings of the 2001 ACM Symposium on Applied Computing (SAC'01)*, pages 636–640, New York, NY, USA, 2001. ACM Press.

[20] Michael Spencer. Basil: A backtracking LR parser generator, 2006. http://www.lazycplusplus.com/basil/.

[21] Mark G. J. van den Brand, Jeroen Scheerder, Jurgen Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. In *Compiler Construction: 11th International Conference (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, Grenoble, France, April 2002.