# Can Deterministic Context Free Grammars Catch-up in Defining Current Programming Languages?

Dave Bone

December 1, 2014

**Abstract**

This paper is a first in a series questioning why deterministic context-free grammars and current bottom-up parsers are not making it any easier to define and compile current-day computer languages like C++. It discusses limitations drawn from my LR(1) compiler / compiler — yacco2 — at the time of thesis development. It introduces the ideas to do parallel parsing, with the companion papers describing the extended Backus-Naur Form (EBNF) language, and sampled solutions to the limitations raised.

## 1 Introduction

To set the tone of the paper, the following quote from Stroustrup's book "The Design and Evolution of C++" published in 1994 [?] illustrates the frustrations in using bottom-up compiler / compilers with their limitations.

> In 1982 when I first planned Cfront, I wanted to use a recursive descent parser because I had experience writing and maintaining such a beast, because I liked such parsers' ability to produce good error messages, and because I liked the idea of having the full power of a general-purpose programming language available when decisions had to be made in the parser. However, being a conscientious young computer scientist I asked the experts. Al Aho and Steve Johnson were in the Computer Science Research Center and they, primarily Steve, convinced me that writing a parser by hand was most old-fashioned, would be an inefficient use of my time, would almost certainly result in a

> hard-to-understand and hard-to-maintain parser, and would be prone to unsystematic and therefore unreliable error recovery. The right way was to use an LALR(1) parser generator, so I used Al and Steve's YACC[Aho,1986].
>
> For most projects, it would have been the right choice. For almost every project writing an experimental language from scratch, it would have been the right choice. For most people, it would have been the right choice. In retrospect, for C++ and me it was a bad mistake. C++ was not a new experimental language, it was an almost compatible superset of C — and at the time nobody had been able to write an LALR(1) grammar for C. The LALR(1) grammar used by ANSI C was constructed by Tom Pennello about a year and a half later — far too late to benefit me and C++.

In January 1997, ACM SIGPLAN Notice's article "Programming Languages: Past, Present, and Future" [**?**] where Peter Trott interviews a variety of computer scientists intimate in language processing, Dr. Stroustrup again restates his frustration with bottom-up processing and its over-exaggerated expressive powers (I apologize if this is an overstatement). In 1998 my thesis's concluding chapter concurred with Dr. Stroustrup?s sentiments. While using yacco2 [**?**] to compile itself, I slammed into the deterministic context-free grammar limitations. As I was still convinced that LR(1) style of compiling is the way to go, particularly when it?s the most powerful deterministic grammar recognizer [**?**], the concluding remarks of my thesis described limitations to be explored within the domain of parallel parsing to cure its ills.

Before delving into the LR limitations experienced, I would like to comment lightly on hand-coded recursive decent parsing. I'm all for it when it's the only technique at hand to get the job done. My reservation is its freedom to do what you want, how you want, and when you want within the author's programming language. Let me rephrase this, the compiler writer can use whatever programming discipline to achieve his goals. One is not bound to the limitations of a compiler / compiler, the grammars it accepts, nor the limitations of its emitted run-time environment. There is a one-to-one relationship between the author and language used with no indirectness to the compiled code effected by the grammar. A grammar, on the other hand, is translated by a compiler / compiler into another language which then gets compiled. Depending on the type of compiler / compiler being used, a LL type recognizer produces recursive decent code while a LR recognizer produces table driven code which makes it very difficult for the programmer to tweak when grammar limitations are to be overcome. This code tweaking should not be tolerated but when a tool has limitations, it might be the last measure to complete the job at hand.

As this paper's intent is to ask questions, let me put forth this thought "How easy is it for one to understand the flow-control of a hand-coded recursive decent compiler?". Usually, the program is accompanied by the railroad type diagrams as in "Pascal User Manual and Report" by Kathleen Jensen and Niklaus Wirth. Though these diagrams are very helpful, their one weakness is that they are not mechanically verified. Grammars are verified by a compiler / compiler. Each person has his own preferences to writing a compiler. My leanings are towards the tools supporting deterministic context free grammars and their verification process. This series intent is to develop a parsing paradigm to eliminate the constraints that led to the above quote by Dr. Stroustrup and my own experiences with the original yacco2's grammar with its nested syntax-directed C++ code. Now with approximately four years experience using the parallel parsing paradigm, this paper begins the journey into the LR parsing shortcomings and the parallel parsing paradigm as a solution.

Plan of the paper: Section **??** discusses the limitations using yacco2. The limitations are general manifestations also exhibited by other LR recognizers. Section 3 discusses the source of the context free grammars limitations. Section 4 introduces the framework required to support the multi-parsing concept. Section 5 concludes the paper with lead-ins to the companion papers to follow.

## 2 Limitations to Bottom-up Parsing

The shortcomings come from experiences using my LR compiler / compiler. Though the points raised are not exhaustive, I believe they represent well the general problems experienced by others using LR compiler / compilers and their accepted grammars. The points are in no significant order of appearance in the list. Some limitations are generalizations about grammars while others are implementation / facilities needed when using the grammars.

### 2.1 Ambiguity Resolution is Hard to Correct

When struck with an ambiguous grammar and the explicative(s) usually muttered, how easy is it to find the source of the ambiguity and to fine-tune the ambiguity or better yet to get rid of it? In bottom-up parsing, there are two types of ambiguous situations produced by a grammar's finite automaton:

- reduce / reduce
- shift / reduce

The following examples are to re-enforce to the reader the necessity in simplifying ambiguity resolution. I know gentle reader that you don't need this remainder; the examples give the common source of ambiguity — mixed contexts that surprisingly compete equally for resolve. They fall into the following contexts:

- grammars defining subset / superset language features. I use the term grammars in the sense of grammatical expressions within one grammar.
- nested grammars
- ambiguous language features

Example one is the mixing together subset and superset grammars. An example is keywords being a subset of identifiers defined in a language. For most implementations, the compiler uses the grammar defining an identifier and does post processing by table lookup to see whether the identifier is a keyword. Though this illustrates the problem, you might wonder is this really a problem particularly when the superset grammar suffices — strange that a grammar snippet is used to define the keywords [**?**] but cannot be incorporated into the overall grammar. Apart from efficiency issues whereby each identifier formed must be checked by a table lookup as to its terminal type, does this situation not ask for a better way to resolve the mixing of grammatical expressions into a grammatical environment without ambiguity? Is there a way to break them up, use them as separate entities within the parsing environment and yet have no uncertainty about their parsed results?

Example two is the defining of multiple language entities that are nested such as all the legitimate character sequences allowed within a C++ literal: ex. "abc\xab\n". There are the regular characters sequences, octal and hexadecimal escape sequences, and character escape sequences expressing the likes of line feed, bell, tab, etc. The principal grammar now contains other grammatical expressions to refine the language accepted; the various escape sequences are such a refinement. Mix them as one grammar and ambiguity shows up as a gorgon. Again post processing on the literal is usually done after using a coarser grammar. But can this be refined using multiple grammars without interference from ambiguity? Are there assurances that their findings (accepted input) are unique and well behaved as reported to the pushdown automaton?

The last example deals with language features that are inherently ambiguous. Is there a C++ grammar out there that handles templates, exceptions, and run-time-type identification [**?**][item 37.11]? Typically a coarser grammar is used to define a larger language with semantic routines used to complete the verification of the accepted string. Deterministic context-free grammars are very expressive but become weak in processing quasi context sensitive environments. The normal recourse is to use a dis-ambiguity strategy within the pushdown automaton, or modify the grammar to accept a larger language and to legitimize its accepted input by post-process semantic routines. Can this limitation be eliminated or minimized?

An ambiguous grammar can be difficult to resolve particularly if there are many rules that have been recognized in a long grammatical sentence. The common prefixes of grammatical phrases being reduced can be challenging to pinpoint the source of the problem when the grammar is large. Figuring out who contributed to the lookahead sets when merges have occurred within the lookahead graph can be vexing when faced with correcting the grammar to make it kosher.

Ambiguity reporting in my eyes is good. I mumble like everyone when such a situation occurs. It takes on the same context as an error message from a traditional compiler. Unfortunately, the correction of it can be a bit of a challenge though witness convoluted template errors from C++ which can produce the same perplexity. Regardless of the algorithm used to compile a grammar into its finite automaton: LR(0), SLR(1), LALR(1), LR(1), LR(k), the expressive difference between them is not very significant [**?**, Horning:9]. To the grammar writer, somehow ambiguity needs a different approach for its resolution. YACC [**?**] defaulted to a simple set of rules to ease the ambiguous situation as a last resort by favouring shifting instead of reducing. Other research uses an on-demand technique of "k look-ahead" in resolving the ambiguity [**?**]. This does not completely solve ambiguity but at least gives another way out of the dilemma. Are there other ways to resolve ambiguous grammars with simplicity and elegance? Are there grammatical contexts and constructs that can bailout ambiguity without the Dr. Watson type investigations needed to unravel a convoluted ambiguous grammar? The generic question being raised is "how can one fine tune a grammar with possible extensions to the EBNF language and not jeopardize the LR(1) constraint to help resolve ambiguity?".

## 2.2 Monolithic Grammars Are Difficult to Understand and Maintain

To prepare the reader, look at any textbook that uses a grammar to describe the language. As Java and C++ are two popular languages currently in use, references [**?**] and [**?**] each demonstrate the size of the grammar with their substantially large number of rules. I put forth to you that if this were program code (which they are), would they be considered good examples of coding practices as viewed from a computer science perspective — structured programming or object-oriented principles? Large grammars are hard to understand and even harder to maintain when corrections or language features are to be added.

## 2.3 Grammar Modularity and Reusability

Where are the little grammars making up a larger one? As high-level procedural languages use procedures, functions, classes, messaging, to break a problem up into smaller pieces, so, should not grammars have the same ability? Under current practice, there is one heavy-weight grammar expressing the language to be recognized. Can this be simplified?

Where is the re-usability of a monolithic grammar? Grammatical expressions are not entities unto themselves but just phrases inside a grammar. Somehow monolithic grammars should be broken down into smaller pieces and assembled into a cohesive whole with all the parts participating together. If grammars were re-useable, they could be published just as algorithms are. For example, the grammar used to describe C++ style comments should be an off-the-shelf piece of code that can be dropped into one's grammar defining a new language. In fact when I first was developing yacco2's grammar, I used the comment grammar described in Java to implement my own C++ type comments until I rewrote it using the parallel parsing facilities to simplify and make it more runtime efficient. Not to sound too pithy, reuse adds more worth to its definition due to others not having to re-invent the same thing: published material has many eyes and ears to proofread it — ahh open source...

Modularity and re-usability go hand-in-hand. The programming of grammars should follow the same refinements as high-level programming languages with their coding practices.

## 2.4 Grammar Debugging

How can one properly test out a monolithic grammar? Lots of regression tests, but this begs the point. Monolithic grammars are too complex to debug properly let alone to resolve when ambiguous extensions to the finite automaton are employed. Here comes that echo again, grammars should be small stand-alone units that can be translated into finite automaton and tested individually before being assembled into a larger grammar environment. The assembly process should respect their stand-alone definitions. Modern languages emit callable functions / procedures, and data sections, that get assembled and connected by a linker. A similar process should be applied to the finite automata.

Newer compiler / compilers are improving the resolution of ambiguous grammars but how well are they in the actual debugging of them? By shrinking the grammar down into a smaller comprehensible pieces of code, not only is it easier to understand their functionality within a stand-alone context but also easier to verify their accepted languages. Current code debuggers as in Microsoft Visual Studio can be used to watch, trace and trap the finite automaton's control flow as executed by the pushdown automaton. Pushed further, should not the grammar and its runtime environment have tracing facilities which can be turned on / off statically or dynamically within various contexts? This becomes more a requirement in light of the simultaneous execution of parsing threads.

## 2.5 Error Signaling / Processing Not Integrated In Grammars

Error processing is another dimension in the use of grammars. I am not talking about error correction and backtracking techniques used to repair the faulty input string but the simple reporting facility to the pushdown automaton. Somehow error signaling should be incorporated into the grammar's vocabulary and productions. C++'s 'try' and 'catch' language constructs are variants in dealing with error processing; where are the programming constructs to support error processing within the grammar context? Can extensions to the grammar / pushdown automaton be made to provide this capability?

## 2.6 How to Parse a Different Language Within a Language?

To illustrate, the EBNF language defining the grammar and its associated syntax-directed code is such a situation. Other examples are assembler code embedded in C++ using a #pragma facility. The question becomes "how far does one verify the secondary language by use of a grammar?". This becomes a problem of how can one distinguish the boundaries between these two languages such that the grammars used do not interfere with one another. Boundary overflow can lead to some interesting errors. Look at the current state of template processing and the error messages cascaded by your C++ compiler. Verbosity and too many indirect levels of processing leads to some interesting guesses by the programmer to resolve errors caused by the instantiated template.

### 2.7 LR Tables can be Weighty

Well this situation jumped at me when I was developing a translator to retarget Oregon Pascal to HP Pascal on VMS. Not to bog the reader down, a complete compiler was written to accept the Pascal language with its extensions like "loophole and ref" and inhouse pre-processing constructs. The translator consisted of approximately 80 individual grammars. Within the project, three things were identified that contributed to major code bloat using C++: the dynamic runtime building of template tables, class hierarchy, and the shift / reduce / lookahead sets.

Depending on the number of terminals defined, the lookahead sets can get quite large. Shift tables can also contribute to the automaton's bulk. Shifting has a double cost: not only in the shift of the terminal but then in the reduce that eventually follows. There are times when a wild terminal shift facility — implicit shifting of terminals — is wanted instead of an explicit shift. It is easier to program and it shrinks dramatically the tables generated. This led to the following investigation:

- What means is there within the grammar where these sets can shed their weight and allow for the fine tuning of their contents within the LR(1) constraints.
- Where can a wild-shift facility be integrated into the grammar which protects against the Pacman-like appetite of consuming all input tokens. The reduce operation must be protected from this type of shift overrun.

## 3 Eureka? Source of Limitations

The source of the problem is a monolithic grammar tends to become a context sensitive issue. There are just too many contexts being mixed into one grammar where their boundaries overlap producing ambiguity. This overlap becomes a scoping issue: that is, local entities — grammatical expressions — all share in the global space of the grammar. It is this one-scope-for-all that causes problems. So given that grammatical expressions be defined and protected in their on local scope, how can they partake in the global parsing environment? The requirement within the global context becomes one of deterministically controlling the grammatical expressions under some governance.

The multi-threading paradigm provides such an environment. To wit, an Operating System runs processes as separate entities. It controls each process, their resources both globally and locally; the scheduling and running of the processes are managed by the fielding of the clock's interrupts. Now, threads are sub-processes that run within the process? environment. The process shares its resources across its threads. The control of threads is done by the Operating System with special facilities like thread spawning, semaphores, critical regions, spin locks, monitors, etc. So by mapping the starting parse as a process and the individual grammars as threads, the problem of mixed grammatical contexts under a global roof has been eliminated. Effectively, grammatical threads can be thought of as recursive-decent procedures that run in parallel non-deterministically. What a mouthful. The non-deterministic part is most interesting because deterministically both launching of the threads and in the arbitrating on their accepted languages must take place. Governship is used in arbitrating between non-deterministic outcomes. By use of predicate logic at specialized syntax-directed control points throughout the grammar, the outcome of mutiple parsing can be controlled.

The following sub-sections introduce the ideas to be expanded upon in future papers. The points expressed are general and are portable to other Operating environments with similar support. In fact yacco2 is running under VMS, NT, Sun Operating systems using different thread libraries.

## 3.1 How to Introduce Threads Into a Grammar

Threads provide the separate run environments needed to execute individually each grammar's automaton: it houses their own run autonomy. Other competing parsing threads are cocooned by the Operating System separation of run spaces. Below outlines the requirements to support threads within a grammar:

- A thread operator must be added to the EBNF language, which gets incorporated into the emitted finite automaton.

- The EBNF language must support separate grammatical definitions of thread entities. Each thread grammar is compiled separately and finally assembled together using the standard language linker of the Operating system.

- A lookahead expression must be added to the EBNF language to support the grammatical thread. This expression acts as the end-of-input terminal(s) to the grammar. It allows the grammatical thread's lookahead sets to be fine tuned according to its own boundaries and context of use.

- A syntax-directed code arbitrator construct is needed in the EBNF language to resolve between two or more competing grammatical threads both accepting an input phrase. Each arbitrator construct within a grammar is associated with the individual state configuration of the finite automaton that launches the competing threads. There can be many arbitrator constructs within a grammar, which are deterministic in their execution.

All language extensions must fall under the same ambiguity constraints of a deterministic context-free grammar.

## 3.2 Specialized Terminals and Runtime Requirements

Constant terminals are needed to overcome the code bloat of tables, wild terminal shift facility, calling threads, and an explicit shifting out of an ambiguous context. They are never part of the string being recognized by the grammar. They are internal terminal constants used by the pushdown automata runtime library. My thesis classifies terminals into 4 groups: error symbols, raw characters, lr constant symbols, and the normal terminals. Some of the specialized terminals have their own grammatical phrases while others can be mixed throughout the grammar's phrases:

- Invisible shift terminal — used to expicitly shift out of an ambiguous context(s) within a grammar. It is explicitly programmed by the grammar writer.

- Wild shift terminal

- 'All terminals' terminal — represents all terminals defined in the grammar's alphabet. Used to shrink the size of a lookahead set.

- Parallel thread terminal — introduces the parallel thread phrase within a grammar

For multi-threaded parsing to take place, the following concepts need to be implemented. This support expands the normal pushdown automata runtime environment:

- The detection of threads to-be-run in the finite automaton tables by the pushdown automaton
- A dispatching mechanism to spawn threads by the pushdown automaton
- A thread-monitor for the starting and ending of different thread sessions along with the dispatching of the appropriate arbitrator module on the accepted token queue from each session of launched threads.
- A communication protocol to support the dispensing of tokens to the launched threads and reporting of results from the grammatical threads to the control monitor.
- Support for nested parallel parsing. This comes about when a parsing thread spawns its own threads which can spawn other threads...
- Thread support within the Operating System along with its appropriate mutual exclusion locking mechanism.
- A ranking of conditional parses, if present, within the current state's configuration. This is expanded in subsection **??**.

## 3.3 Arbitration Between Competing Threads

In this framework, the individual threads all participate co-operatively with the eventual control being given back to the its launching control monitor. But how does one resolve two or more competing grammars each successfully parsed and reported back to their parent control monitor? Each successful parse is just a restatement of the subset/superset problem. The solution is quite simple — have an arbitrator judge who should win. Taking this thought further, by adding an arbitration construct to the EBNF language, the author of the grammar now has complete control over who wins and losses by syntax-directed code. By extending arbitration support to the pushdown automaton, the code gets executed within the contexts defined by the finite automata.

Within this context, nested arbitration must be supported due to parsing threads spawning their own threads. There are now multiple control monitors administrating their own parse environment. Let's look at this situation from a high level altitude. To begin, there is a parsing process that spawns threads. At this point, the start token boundary is fixed for all future threads spawned by this process; it is the first of the tokens in the to-be-parsed token stream from which the to-be-launched threads parse from. This start position can be anywhere along the token stream. Now the process's parse is put on hold while waiting for its threads to report back. Each thread is consuming the lookahead tokens at a different rate with its concluding token boundary being possibly different than its brethren parsing threads. Now if the spawned threads also spawns threads, the same thing happens: the spawning thread is put on hold, launches its control monitor, and lets its threads continue the parse at the token position within the input stream of the launching thread.

Across this parsing spectrum, there really is only one parse going on arrived at in a deterministic way to consume and judge what token sequence wins. Each parse thread defines the viable prefix to-be-parsed from the current token input. Each sequence accepted can be different with variable length viable prefixes parsed. Nested threads are just part of the viable prefix of its spawning thread. Threads spawning threads are also put on hold waiting for the results from its spawned threads. Arbitration takes place within each put-on-hold process / thread. All of this is happening simultaneously across the parallel processing spectrum. With these successfully competing threads, the languages accepted range from the very general to the most specific. To distinguish between competing accepted parses, arbitration is required along with a proper re-setting of the lookahead token boundary for the process's pushdown automata to continue. The winning acceptance provides the lookahead token position within the input token stream to continue parsing.

## 3.4 Conditional Parsing Within the Pushdown Automata

The term conditional parsing relates to multiple attempts at parsing the same input using a pre-established order of different parse entities: parallelism, regular parsing. When one entity fails, the following parse entity will be tried. Effectively the pushed down automaton is being extended into a context sensitive ranking of activities. Depending on the context within the pushdown automaton's stack, the first activity matching the ranked conditions gets run.

One can look at the proposed extensions into conditional parsing as a kind of multi-tracking. It is not true 'back' tracking because the parse stack is not unwound with try-to-find-a-successful stack configuration to continue parsing with possible error correction. It is the 'potential parsing' of multiple contexts at the singular point within the pushdown automaton's stacked state configuration. These potential viable prefixes get executed in a sequential order: parallel context followed by a regular parse with conditions if the parallel context was parsed unsuccessfully or not present in the configuration state of the finite automata. The following list ranks the conditional order of activities tested by the pushdown automaton:

- Is parallel parsing present? If so then do parallel parsing. This is a shift operation if successful.
- Regular parse, if parallel parsing was not successful or not present:
    1. Possible shift operation. This is the regular shift operation done on the current token input.
    2. Possible invisible shift operation. This is a meta-terminal defined in the grammar's Terminal alphabet acting as a constant symbol. It is invisible to the input tokens. If present in the automaton's current configuration, it is shifted. It is used in the grammar to resolve ambiguity.
    3. Possible 'all terminal' shift operation. It becomes a catch all shift facility which can also be used for error processing.
    4. Possible reduce operation on the current input token. This is the normal reduce operation of a bottom-up parser.

Is parallelism and conditional parsing really backtracking? Yes it is — but, backtracking takes on many forms and repeated parsing 'back and forth' along the input stream or parse stack is definitely backtracking. Competing threads evaluate the same input in parallel to arrive at some result; conditional parsing tests the current parse configuration for internal contextual refinements: specialized terminals. Though this is minimized backtracking, I call it optimized forward-tracking. There is no backtracking taking place on the pushdown automaton's parse stack or a resetting of the token-to-be-parsed somewhere along the input token stream. The parse stack is not popped to determine where to restart the parsing process to produce a successful acceptance. Parsing gets done only in a forward progression using parallel determinism.

## 3.5 Making Parallel Parsing Efficient

As this subject is quite complex in determining runtime statistics, I will just raise some generic optimizations needed to improve performance:

- Only run threads that have the potential to complete
- Have a thread manager that can quickly launch threads
- Support multiple readers of a mutex protected token dispenser allowing multiple threads simultaneous access to fetching of their own tokens.
- Increase the number of cpus to support true parallel processing and possibly use compiler / cpu optimizations for enhanced multi-threading performance

The first improvement is to publish the first set of each thread. This allows the thread dispatcher to determine dynamically at-thread-launch time if the thread has the potential to parse successfully where the current token is in the thread's first set. To achieve this, post processing on all the threads's first sets must be done with a specialized first-set linker; this is due to the transient closure property of a thread calling another thread out of its first set in a nested type environment. Finally the compiler is created using a traditional linker to bind all its first-sets and assorted other objects into a runable program.

The next optimization is to keep a global map of worker threads already spawned with run type statuses: waiting for work, busy, exit. Statuses allows the thread launcher to determine whether it needs to create multiple copies of the same thread when used in a nested way: a thread can call another thread which can eventually lead to a nested calling sequence. This optimization creates only the threads that are needed in a just-in-time fashion. It also allows for the re-cycling of their use when they have completed their parse by optimizing out the start / run / stop cycle per called thread.

The last optimization is to take advantage of multiple cpus to run threads in parallel. By use of multiple readers on the token fetching, each thread can fetch its next token asynchronously. The token dispenser is globally accessible to all threads: this eliminates the use of a single queue and its loss of parallelism.

## 4 Conclusion

Apart from the computer languages normally associated with compiler / compiler use, as one gets more into the Internet, its supported languages are starting to flourish at an astounding rate. Visible are languages like HTML and XML with its family of languages: schema grammars, style sheet language, query language. Now look at the underlining protocols driving the Internet: HTTP, ARP, SMTP, FTP, SOAP are just a few protocols that are languages needing a parser. Scripting languages of various functionalities, SQL for relational databases, report writers, code-morphing environments to emulate machine type behavior, search engines, type setting languages are other activities using languages. From this varied list, it becomes rather obvious that the compiler / compiler is a necessity and not a teaching tool of applied grammar theory. Unfortunately various compiler / compiler implementations impede their general use. Recursive descent rules the roost using hand-coded techniques.

Lets muse on 3 decades of deterministic context free grammars and their compilers / compilers. Though I'm not intimate in the use of YACC and LEX [**?**], from the literature and language interest groups, YACC's duration has set the bar for others to pass but so far its still leading the pack with its look-alikes. Other compiler / compilers have improved the development cycle but have not substantially extended the capabilities of the recognizer or the ease in which the grammar is developed to define a language. This says a lot for the LEX - YACC combination with its ambiguity override facility. Published grammars like C++ and JAVA are LALR(1) ambiguous but use YACC's dis-ambiguity rules to generate their finite automata. Language developers still vent their frustration in not being able to refine the grammar(s) to their satisfaction due to the constraints of YACC or worse due to ambiguous languages like C++. As expressed in this paper, it is the constraints of a monolithic deterministic context free grammar that limits the compiler writer.

To conclude, a multi-threaded parallel parsing framework was introduced along with language extensions in the EBNF language to solve the short failings described. The second paper in this trilogy will detail the EBNF language, the syntax-directed code constructs, and the parallel-parsing operators needed. Like culinary activities, tasting of one's cooking tells how well the recipe turned out. The third paper provides such a sampling of parallel-parsings to this paper's questions. Hopefully this series will demonstrate to the reader that the answer to the proposed titled question is a definite yes. Of course, everything is said in a context be it free, parallel, or otherwise ;}.

## References

[1] Bauer, F. L., and Eickel, J. *Compiler Construction - An Advanced Course*. Springer-Verlag, New York Heidelberg Berlin, 1976. 2nd edition: J. J. Horning, LR Grammars and Analysers.

[2] Bone, D. A syntax-directed compiler/compiler emitting lr(1) object-oriented code. Master's thesis, Concordia University, Montreal, Que, Canada, Sept. 1998.

[3] Cline, M. C++ faq lite. www.parashift.com. Copyright 1991 - 2003 Marshall Cline: item 37 Is there a yacc-able C++ grammar?

[4] Gosling, J., Joy, B., and Steele, G. *The Java$^{tm}$ Language Specification*. Addison-Wesley Publishing Company, Reading, Massachusetts, U.S.A., 1996.

[5] Johnson, S. C. Yacc - yet another compiler-compiler. Tech. rep., AT&T Bell Laboratories, Murray Hill, N. J., 1975. Technical Report 32.

[6] Knuth, D. E. On the translation of languages from left to right. *Information and Control 8(6)* (1965), 607–639.

[7] Lesk, M. E. Lex - a lexical analyzer generator. Tech. rep., AT&T Bell Laboratories, Murray Hill, N. J., 1975. Technical Report 39.

[8] Stroustrup, B. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, U.S.A., 1997. 3rd edition.

[9] Stroustrup, B. *The Design and Evolution of C++*. Addison-Wesley Publishing Company, Reading, Massachusetts, U.S.A., 1997. 3rd edition.

[10] Trott, P. Programming languages past, present, and future. *ACM Sigplan Notices 32(1)* (Sept. 1997). What do you consider the most significant contribution to compiling?

[11] Wong, R. W., and Parr, T. J. Ll and lr translators need k > 1 lookahead. *ACM Sigplan Notices 31(2)* (Feb. 1996).