Parsing in a hostile world

Darius Blasband
darius@raincode.com
RainCode Corp.
Brussels, Belgium

Abstract

This paper describes a formalization and development effort aimed at automating the production of parsers for languages for which the common classifications (LALR [12][28], LL) – and even the common division of labor between lexical and syntactical analysis – do not apply.

This effort, implemented in tools named DURA and lexyt, aims at real-world applications of parsing technology – applied to software renovation prior to plain compilers – rather than theoretical soundness only. A number of industrial results, around a tool named RainCode, are also described.

1 Introduction

1.1 Inadequate parsing technologies

yacc [25] and lex [30] have been widely taught in universities for more than twenty years now. They have become de facto standards when it comes to parser generation, even if mainly in the way many people remember having used them on toy languages, such as a small Pascal subset or a calculator.

yacc and lex were designed in a time where parsing performance was a major issue, specially in the argument between supporters of automatic vs. manual parser engineering. In order to deliver the best performance possible, yacc and lex have been restricted resp. to LALR and to some forms of regular expressions only. Not all languages, far from it, are LALR. Furthermore, even languages which are formally LALR can only be described by abusively cumbersome LALR grammars and would be far more convenient to express using more liberal grammars. This issue has been addressed by two equally flamboyant papers: [9] addressed the inadequacy of parsing technology in general, while [34] explains how lookahead greater than one are needed in practice, even if not necessary in theory. Up to a point, the languages for which yacc and lex are actually proper solutions are often simple enough not to require an automated parsing technology in the first place (Consider the calculator example mentioned above, for instance). The critical mind might claim that they are solutions for a non-existing problem.

Of course, one might claim that such tools (and the numerous followers [18][36]) are meant to be used for new languages only that can be designed to be LALR (or LL, for that matter) as experience seems to show that such restrictions are not serious when designing a new language.

However, we live in an utterly imperfect world:

- We still have to deal with languages that were designed long before the formalization of such grammar classes had become popular outside the academic area. Unfortunately, these old languages still account for huge amount of legacy source code with huge economic interest today. The fact that these languages are old (and admittedly, ill-designed in numerous ways) does not change anything about the fact that we need parsers for such languages.

- Some languages have evolved from an ad hoc need, when parsing technology was not considered an issue at all. For instance, RPG [23] was originally designed as a report-writing tool, and later evolved to a super-duper bells-and-whistle environment that does everything short of brewing coffee. It is also the case for languages for which parsing only appeared as an afterthought. For instance, ObjectZ [13] is a specification language for which the input format is just LaTeX with a number of ad hoc macros. It is only later that actually parsing ObjectZ specifications seemed to be a sensible idea after all, and one had to live with the intricaties of the LaTeX macros from there on.

- Widely used languages such as C and C++ ap-

1095-1350/01 /$10.00 © 2001 IEEE

pear to be ill-suited to parsing in practice: yacc-based C, and even more so, C++ parsers require extensive twiddling, as these languages cannot be parsed without maintaining extensive semantic information at parse time.

1.2 The shortcomings of yacc and lex

More specifically, the work described in this paper addresses the following shortcomings – or perceived as such – of yacc and lex:

- C as a host language (even if yacc-like tools exist for other languages), does not provide a reasonable level of abstraction for the third millenium. In several circumstances, the author has seen projects start with an illusion of productivity, using a yacc-generated parser, then struggle with the intricaties of C for the implementation of the rest of the project (semantic analysis, type synthesis, etc.) while it would have been far more efficient to use another language, even if the lexer and parser had to be written manually.

  It is also true that yacc and lex support lots of tweaking here and there, in such a way that user actions can alter the tools default behaviour. Of course, this can be seen as an added value, but at the same time, people seem to prefer the quick and dirty patch as user-action rather than question the semantics of the tool itself.

- yacc (and most of its work-alikes, with the notable exception of btyacc [45]) can only generate parsers for LALR grammars. In most cases, the class of grammars is even restricted to LALR/1, while some implementations support LALR/2 [43][32].

- yacc does not produce a parser: it only produces a *recognizer*, that is, an oracle which indicates whether the input is part of the language or not. It is only by attaching ad hoc semantic actions to the operations performed during recognition that one can build a parse tree. This restriction is a mere question of ambition: there is nothing in yacc's underlying technology which prevents it from being used to build parse trees automatically, as some extensions actually did [16].

- Like most parsing methods, even among more recent ones, yacc's parsers are forward-only parsers. The input is processed left to right, and every token is handled once. This has of course the nice property of guaranteeing timely termination, but fails to handle languages such as PL/1 [24] where infinite lookahead is required.

- yacc uses the flat representation $T \Rightarrow \omega$ where $\omega \in \{T \cup N\}*$. Higher-level constructs, such as optional subsequences or repetitions must be translated to yacc's primitive form before a parser can be generated.

Most of the issues listed above have more to do with convenience than theoretical feasibility. For instance, expressivity is sometimes discarded as essentially irrelevant: if yacc's basic format can describe the same languages the higher level operators do, why bother?

Well, we bother because we deal with the real world. We bother because compilers (and more generally, language processing tools) are complex pieces of software, and the burden of the additional work required by yacc comes on top of lots of other, more important things we must care for. We bother because some of the grammar rules we deal with would require dozens of intermediate rules if expressed using yacc. In short, expressivity and convenience are and remain essential issues for the professional compiler writer. If they were not, we would not even need yacc's flattened form: we could do everything using Chomsky's normal form.

1.3 An alternate approach

DURA is a tool which produces a working parser from a cycle-free grammar without hidden left recursions [33]. Despite the apparent similarities, DURA is radically different from yacc:

- Instead of being made of a scaffolding of two languages (one for for the grammar specification – yacc –, one for the semantic actions in the host language – for instance, C) with possibly cumbersome interfaces between them, DURA is built by extending YAFL [6]. The grammar information is fully integrated in YAFL source modules, thereby avoiding impedance mismatch altogether. Besides, YAFL provides a number of high-level facilities (inheritance, extensive genericity, static typing) specially suited for the development of language processing tools.

- Parsing performance is not as crucial as it used to be. [5] shows how parsing in gcc [42] is a minor performance concern, when compared to the sophisticated algorithms used by the code generation phase. Performance is even less of an issue in the context of software renovation.

- While suffering from bad press, backtracking is one of the ways to go to solve lookahead and local ambiguity problems. It has a worst case exponential behaviour, but this worst case does not occur in practice. Languages we commonly deal with are at most locally ambiguous. Backtracking is then a tool to deal with exceptional cases.

- Convenience in the expression of the grammar is an important issue, even at the cost of some level of performance. It makes development faster, and maintenance much easier, specially if one does not have to pay excessive attention to LALR or LL grammar restrictions.

- One of the way to make the parser more convenient is to have it generate complete parse trees automatically, rather than rely on user actions at reduction time to do so[1].

2 DURA and lexyt

2.1 The lexical specification's input format

The lexical specification as recognized by lexyt is somehow similar to lex's, except for the fact that no semantic action can be attached to a specification, just a symbolic code:

```
QuoteCharCode (\'\'[0-9]+\'\')|(\"\"[0-9]+\"\")
CrossCharCode \#[0-9]+
LParent       \(
RParent       \)
```

2.2 The syntactical specification's input format

DURA's input format and semantics are significantly different from yacc's:

- Instead of being stored in an external grammar file, the grammar rules are embedded in YAFL [6] classes – in the common object-oriented sense – and express a convenient morphism between classes and grammar rules. Classes with embedded grammar information are referred to as *non-terminal classes*. This integration as an extension to the YAFL programming language is performed by using YAFL's compiler support facility[2].

  Non-terminal classes inherit from a common base class which includes a number of methods that can be redefined in order to influence the parsing process. For instance, a "Commit" method is called when the non-terminal is being reduced, and can even abort the reduction and force backtracking to retry another interpretation of the input, for instance if some user condition is not fulfilled.

- Inheritance at the class level translates to implicit grammar rules, using a mechanism named *reverse inheritance*. For instance, when an *IfStatement* class inherits from a *Statement* base class, a grammar rule *Statement⇒IfStatement* is added to the grammar.

- The grammar rules can be expressed using a wealth of higher-level operators: possibly empty or non-empty repetitions with separators, disjunctions, optional parts, etc.

```
NONTERMINAL CLASS IfStatement;
  INHERITS Statement;

  GRAMMAR
    IfKeyword LParent TheCondition RParent
      TheStmt SemiColon?
      (ElseKeyword TheElseStmt)?;

  VAR
    TheCondition: Expression;
    TheStmt, TheElseStmt: Statement;

END IfStatement;
```

- Grammatical sub-parts can be attached to attributes directly in the grammar, so that after reducing the grammar rule attached to a non-terminal class, an instance of this class can be built by assigning the grammar subparts directly to attributes. In case of repetitions, the target attribute must be of a collection class, making use of one of YAFL's generic collection classes, providing far more convenient and controlled access than lists linked together using anonymous pointers.

[1] The difference between yacc and DURA, in this regard, is similar to the difference between the two standard XML parsing methods, namely SAX [21] which is event-driven, and DOM [47] which builds a generic parse tree one can query.

[2] The YAFL compiler is boostrapped, and by the systematic use of abstract factories [17], one can easily extend the compiler using inheritance and delegation.

The parse tree produced by DURA is fully typed: it is a scaffolding of the various non-terminal classes; as opposed to generic parse tree's such as the ones produced by DOM [47] and SableCC [15][16].

2.3 Backtracking

2.3.1 Lexical backtracking

Lexical backtracking is the ability to have multiple lexical interpretations of the input, that can be tested successively until one yields a valid parse tree.

lexyt is the lexer generator coupled with DURA. In order to better control the effects of unwanted backtracking, it defines four levels of backtracking:

Level 0 does not backtrack at all: the entire input stream supports a single lexical interpretation. The input can only be divided into lexemes in one way, and every one of these lexemes supports a single interpretation only. This is exactly lex's behaviour.

Level 1 also supports a single division of the input into words, but every word can hold several lexical interpretations. This backtracking level describes PL/1 [24] lexical model accurately, where things such as

```
IF IF = THEN THEN THEN = ELSE ELSE ELSE = IF
```

are valid. There is only one way the input can be divided into lexemes, but some words might have to support several possible interpretations. For instance, in the example listed above, `IF` can refer to the reserved word "IF" or to a user-defined identifier.

Level 2 qualifies a lexer where multiple divisions of the input can be equally valid, but where at most one lexeme with a given token can be returned for any position in the input. Explained as such, it all seems pretty obscure, but it shows easily on a real world example:

```
DO10I = 1,20
```

The fortran [1] example above executes down to statement labeled 10, varying the index variable "I" from 1 to 20. This statement is amazingly similar to the assignment:

```
DO10I = 1.20
```

which gives the value 1.20 to a variable (declared implicitly in some cases) named "DO10I".

Given the loop statement above as input, a lexer with backtracking level 2 will first return a lexeme as an identifier token, with the longest possible match: "`DO10I`". If, by backtracking, the lexer is asked to return a different interpretation of the input at the same position, it will not return the prefixes "`DO10`" or "`DO1`" as identifiers but rather "`DO`" as the DO fortran keyword. The fact that, after having returned a lexeme as a given token, such a lexer does not return prefix lexemes with the same token characterizes lexers with backtracking level 2.

Level 3 lexers have no restriction at all, and since they have truly exponential behaviours in most real-world examples, they have not found a single practical use so far.

The differences between the various forms of lexers, as described above, is formalized in [7]. Implementation of lexical backtracking is straightforward, and it induces little distributed fat[3]: as long as one does not backtrack, lexers generated by lexyt have performance levels similar to those of lex.

2.3.2 Syntactical backtracking

A backtracking parser as generated by DURA can be described as an LR parser where, given a lookahead string[4], a number of actions can be taken, including at most one shift operation and zero or more reduce operations. These operations are ordered in a list – shift (if present) first, followed by reduces (if any) – and will be tried in sequence until parsing ultimately succeeds, or until all possible actions have been tested without success.

When all possible actions have been tested in a state s_i – or if no action was possible at all, which is of course strictly equivalent even if intuitively different – the parser undoes the operation o_s which had lead to state s_i, goes back to a state s_{i-1}, and tries the

[3] Distributed fat denotes a situation where a feature has a negative effect on the performance of a system, even if it is not used. For instance, nested procedures in Pascal have no distributed fat – calling a non-nested procedure is as fast as can be – while multiple inheritance, as provided by C++, has distributed fat, as late binding gets slightly more complex even if multiple inheritance is not used, as classes cannot determine statically whether they are ever used in a MI context or not.

[4] The lookahead string is of length 1 in terms of lexemes in our current implementation. In theory, it could be extended to longer lookahead strings, but it would barely be worth the trouble, since backtracking can deal with lack of determinism due to excessively short lookahead strings, and combining strings with more than one lexeme with lexical backtracking as described above would make things even more complex.

operation following o_s in the list of operations that can be performed when in state s_{i-1}. It is a rather simple backtracking mechanism. It is the generalization of yacc's behaviour which chooses at most one possible operation in every state, given a lookahead string.

Undoing an LR operation as described above to go from state s_i back to state s_{i-1} is very simple: the top of the LR stack indicates whether the last operation was a shift or a reduce.

- In case of a shift, the lexer is restored in its state before the shift: position, length of the current token (in case of a lexer with backtracking level 2 or 3).

  If coupled with a backtracking lexer (level 1, 2 or 3), just before unshifting the current lexeme, the lexer is asked for another possible interpretation of the input at the current position. If there is such an alternate interpretation, the unshift is cancelled, and the parser attempts to advance using this alternate interpretation. Otherwise, the unshift is confirmed, and the lexer is positioned back to the previous position in the input stream.

- If case of a reduce, the slice of the LR stack which was reduced is put back, thereby replacing the synthetic element generated by the reduction.

Synthetically, while plain LR parsers support a shift and a reduce operation, DURA-generared parsers also support an "undo" operation, which, depending on the last operation performed on the LR stack, turns into an "unshift" or an "unreduce".

2.3.3 Restricting backtracking

Backtracking's negative influence on performance can be reduced by restricting it in places where one knows that backtracking would not make any sense:

- At the lexical level, beside the ability to define more of less stringent lexers by using the various backtracking levels as explained above, one can also control backtracking by excluding some reinterpretations. For instance, in the case of COBOL, a directive such as:

  ```
  $EXCLUDE EndIfKeyword EndKeyword
  ```

  ensures that after having found the "END-IF" keyword, the lexer will not consider the "END" keyword as a valid alternate interpretation at the same position in the input. In other words, after having found an "END-IF" keyword at the current position, and subsequently failed to parse the input with this lexeme, the directive above indicates that backtracking to "END" must not be tried, even if "END" is a valid prefix for "END-IF".

- At the syntactical levels, classes can be marked with the CUT and SNIP directives.

  The CUT directive in a class indicates that the non-terminal instances of this class ought to be interpreted as checkpoints: whenever the reduction has taken place, whatever precedes this commitment point is considered valid and will not be challenged. In other words, when a parser attempts to unreduce a grammar clause that has been marked as CUT, it stops looking for a way to match the input with the grammar, and starts the error recovery mechanism.

  The SNIP directive in a class indicates a local CUT: the non-terminal class instance of this class, once analyzed successfully, should not be unreduced to search for another valid interpretation of the input. However, what has been recognized *before* the grammar clause(s) can be challenged for another interpretation. When backtracking encounters a snipped grammar rule, the grammar rule is ignored entirely, and backtracking resumes *before* the beginning of the grammar rule.

Very typically, the CUT directive can be used to commit non-recursive constructs (such as functions in C or paragraphs in COBOL), while SNIPs are generally used in constructs that can be nested, such as statements, for instance.

The CUT and SNIP pragmas are convenient, as they can improve a parser's performance by an order of magnitude when recognizing an input as invalid. On the down side, they require a more intimate knowledge of DURA's parsing algorithm, and can cause a parser to refuse a correct input if misused.

2.4 Pragmatics

As shown in [2], the difference between LALR and SLR [11] is minor in terms of the class of languages they cover, but LALR require more sophisticated data structures and is significantly more complex to implement than SLR.

In the case of a forward-only parser, the extra work is definitely worth the trouble, as there are examples in C which require LALR and for which SLR is not sufficient. DURA has been restricted to SLR only for the sake of simplicity, as the difference between LALR and SLR will, at most, cost a few percent of additional

backtracking, and do not hamper the parser's ability to recognize the input in any way.

3 Industrial exploitation

3.1 Performance

Performance was not the major focus of the project, as convenience, expressivity and productivity were considered more important. However, the performance of the DURA-generated parsers is perfectly adequate: it is an order of magnitude better than other source code reengineering tools [8], and it has even proven good enough for more conventional parsing tasks, such as plain compilers and interpreters. For instance, the COBOL parser – which undoes about 20% of its operations through backtracking – parses over 2000 lines of source per second on a vanilla PC of 2001, including preprocessing and parse tree building[5].

3.2 Raincode

DURA is the kernel technology behind RainCode [37]. RainCode is a generic source code manipulation tool, which serves as high-level platform to implement a number of facilities, including documentation generation, computing metrics and building inventories, quality control – by ensuring that the code complies to project-wide, company-wide or industry-wide standards – or even transformations, including the translation to another programming language or to another dialect of the same language.

RainCode works by reading a source file, building a complete parse tree and providing a special-purpose scripting language to perform high-level operations on this parse tree. As the data structures one can access from within this scripting language are direct mappings from the non-terminal classes in the grammar, not being restricted to LALR or a similar constraint is a major asset in making the parse tree usable.

There are versions of RainCode for a number of languages, including COBOL, PL/1, C&C++, Java, Natural, Informix 4GL, etc. and its parsers have parsed several million lines of various origins and languages.

RainCode is structured in such a way that the only thing that differs from version to version is the parser itself: the parse tree is fed to a generic machinery which, using YAFL's reflexive API to query the nodes attributes and methods, navigates through the parse tree using the scripting language. The scripting language is also parsed using DURA-generated parsers. The parsing performance is good enough – not significant when compared to the number of other things that have to take place before starting to run a script – giving a hint that the DURA parsing technology's performance is adequate for lightweight languages (typically, LL or almost LL) where its backtracking machinery is not required, but where its ability to yield a fully typed parse tree is nice to have.

[5]It is true however, that performance figures related to parsing technologies are hard to compare, as hardware evolves constantly, and as the implementation language (YAFL for DURA, c for yacc, Lisp for [38]) dominates the performance issue more than the underlying theoretical foundations.

3.3 Maintenance

The level of abstraction to express the grammar, together with the lack of LL or LALR constraint, allows for increased efficiency: parsers for complex legacy languages can be developed in a matter of days, and adapting our COBOL parser to take a new set of extensions into account requires at most a couple of days.

Very typically, RainCode for COBOL is based on a single parser while there are numerous mutually incompatible dialects of COBOL, including preprocessor-based extensions such as SQL [3] and CICS [22]. The COBOL grammar used by RainCode for COBOL is the union of all grammar rules found for every COBOL dialect we've encountered so far, and lexical backtracking ensures that what is considered a reserved word in one dialect can be used as plain identifier in another. EXCLUDE (See 2.3.3) clauses are used to ensure that words such as `MOVE`, which are reserved in *all* COBOL dialects, can never backtrack to a plain user-defined identifier. Having a unique parser for all COBOL dialects is based on the (optimistic?) assumption that, while the dialects are mutually exclusive, we shall never encounter an input which can be recognized differently (with significantly different semantics) using two different dialects.

4 Related works

4.1 Backtracking parsers

Just as DURA, btyacc [45] aims at parsing *locally ambiguous languages*, where a local non-determinism can be solved by considering a longer slice of the input, as opposed to *globally ambiguous languages*, where the input can yield more than one parse tree. Error recovery is also similar: btyacc also remembers the most advanced position in the input to start recovering at that stage.

btyacc is also similar to DURA in the sense that its backtracking facility is *automatic*, while Lark's [20] and ANTLR's [35] backtracking is controlled by the programmer, who must indicate how to explore paths before taking a decision about the proper parsing action to take. However, Despite the similarities, DURA and btyacc are also very different in some respects:

- btyacc is based on yacc, and shares some of its shortcomings, such as lack of higher-level grammatical constructions (repetitions, for instance) and no built-in support for building the parse tree. Besides, it does not support any form of lexical backtracking.
- Performance. As long as there is no conflict, a btyacc-generated parser's performance is roughly equivalent to yacc's, which is expected to be significantly better than a DURA-generated parser. We expect the behavior in case of conflict – that is, when backtracking must be used – to be similar to DURA's, except for the intrinsic differences between a parser in C and the equivalent YAFL implementation.

  btyacc's performance is equivalent to plain yacc's in absence of conflict, except for the difference between yacc's deterministic behaviour when facing a shift-reduce conflict, and btyacc's, which should start its backtracking machinery when facing *any* conflict.

4.2 Generalized parsing

Earley[14] is generally considered as the first reasonable generalized – in the sense that it applies to any cycle-free context-free grammar – parsing method, but it has been soon superseeded by GLR. The two approaches have been shown essentially equivalent by [41].

GLR parsers [29] (often referred to as Tomita [44] parsers in the literature) are mere generalizations of LR [27][2] parsers, where, in case of a conflict, the various possible evolutions of the LR stack are maintained simultaneously and synchronized on reduces, yielding parse bushes as opposed to plain parse tree. GLR parsers must take *hidden left recursions* into account [33]. The same restriction applies to DURA [7]: it is of course no coincidence since a DURA parser is little more than the single-threaded depth-first walkthrough of a Tomita parser DAG.

One of the major advantages of GLR lies in its ability to build reasonably compact parse forests while parsing. DURA does not need this ability, since it aims at inputs for which at most a single parse tree can be produced at the end. GLR is based on an optimistic assumption, namely that large parts of the input can be analyzed with a plain LR parser, without requiring LR stack cloning. Practical grammars are seldom ambiguous enough to justify the heavy machinery required by an Earley parser. In [44], Tomita claims that his algorithm produces parsers that are 5 to 10 times faster than the corresponding Earley parser for realistic grammars. Rekers [38] supports this statement based on experimental data, and suggests that Earley would behave better on excessively ambiguous grammars only.

DURA takes this optimistic view on the world even further: not only can a plain LR parser handle most of the input, but in case of conflicts, we won't have to go very far to bump into a problem if we selected the wrong path.

Another experience of using GLR parsing for programming languages was conducted by Wagner and Graham [48]. They take full advantage of GLR's ability to build forests rather than trees to parse isolated fragments of code which are ambiguous in absence of the enclosing context. They also explain how this technique can be used for batch (that is, not incremental) parsers together with *tree filters* as described in [26] for post hoc disambiguation.

4.3 Scannerless parsing

Scannerless parsing is a technique where the lexical definition is merged into the syntactical definition, taking advantage of the fact that context-free languages are a superset of regular languages. The resulting parser, rather than taking a stream of higher-level tokens as input, deals directly with characters.

Scannerless parsing was first introduced by Salomon and Cormack [39][40] by integrating lexical analysis in a yacc grammar, and has been investigated further by Visser in [46], which performs the same operation with a GLR parser. He raised numerous issues in the process:

- While the classes of languages clearly allow the integration of the lexer in the parser, some of the facilities one takes for granted in lexical analysis have to be somehow simulated when plugged direcly into the parser. [46] lists two such facilities, namely, *Longest Match* rule, which ensures that the longest possible token is returned and the *Prefer Literals* rule, where one can reduce the level of ambiguity by explicitly excluding some of the possible interpretations of the input. Both

of these behaviors are naturally required by real world lexers, and are provided by lex[30], lexyt, and most lexical analyzer generators.

- When mapping an existing lexical specification made of regular expressions into a scannerless parser, performance is guaranteed to be linear while recognizing a lexeme by an LR automaton. [39] shows how a yacc-based lexer can be much faster than its lex counterpart, explained as follows: pushing an element (on the LR stack) is much faster than a table lookup in the lex transition tables which requires at least an integer multiplication and possibly some action to handle table compression. This argument is weaken by the fact that lex is notoriously slow, and lexers generated using GLA[19] or RE2C[10] would have been much harder to beat.

  Furthermore, this argument applies to yacc, *but does not translate seamlessly to its numerous extensions.* If pushing an element onto a stack is faster than a table lookup, pushing an element on multiple simultaneous stacks (perhaps implemented as a DAG[6] [44]) might not equally efficient.

4.4 The real drawback of forward only generalized parsing methods

In his original paper, even Earley himself acknowledged that his parsing method, while quasi-linear in many realistic cases, would see its performance controlled by very large constant factors. In practice, however, static computation of sets of Earley items (just as yacc and DURA statically compute sets of LR items) can lead to performance levels which are roughly comparable with LALR parsers [31]. Similar optimizations have been applied to GLR parsers [4], at the cost of increasing the number of grammar rules significantly.

Given these efforts, one can wonder, how come we still use or even consider simple LALR tools such as yacc? If the performance is roughly the same, why do we still bother using tools that require a major understanding of the restrictions imposed by the class of language to be used proficiently?

We believe the reason goes far beyond issues related to classes of grammars only. It has to do with the ability to trigger actions when given constructs have been recognized. Both Earley and GLR parsers only support forward movement on the input, and consider the various possible interpretations of the input simultaneously. Hence, adding some form of user-defined processing while parsing is cumbersome: if some context data structure must be used, it must be replicated whenever an alternate path is considered. Then, when two different paths are reconciled in a single state, the corresponding data structures must be merged, and one can think of examples where this merge operation is simply not possible.

More generally, yacc and lex were designed with the ability to tweak the generated parsers and lexers manually. Languages such as C and C++ cannot be parsed[7] at all without such tweaking, as the type names must be maintained in a symbol table while parsing. Allowing such tweaking in a GLR parser raises numerous logistics issues, up to a point where one can question the relevance of the simultaneous analysis of the various acceptable interpretations of the input.

5 Further work

Positional languages such as most assembly languages and RPG [23] are usually not complex, but cannot be described efficiently using plain terminals and non-terminals. We are currently working on an extension of the way grammars are commonly formalized to be able to represent these languages, while the ultimate goal is of course to be able to generate matching parsers.

6 Summary and conclusions

DURA and lexyt are part of a parser generation toolbox for complex and ill-designed languages, specially suited for software renovation. They are of little theoretical value except perhaps for the formalization of the different lexical backtracking levels and revese inheritance for grammar rules, but they offer a number of interesting features, namely:

- Automatic tree building
- No serious grammar restriction
- Extended expressivity to define the grammar
- Close integration in the host language.

[6] Directed Acyclic Graph

[7] At least, in the sense of producing *one* parse tree as opposed to a set of trees, where the – hopefully unique – one is selected afterwards using semantic analysis.

Acknowledgements

The author wishes to thank Raymond Devillers, Jean-Philippe Bernardy, Laurent Férier, Bernard Rasson, Jean-Christophe Réal and Lionel Ferette for having proofread early versions of this paper, and for having contributed numerous improvements and corrections. Besides, the anonymous referees have also contributed a number of insightful comments that the author has attempted to reflect in the final version of this paper.

References

[1] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener, editors. *Fortran 95 Handbook: Complete ISO/ANSI Reference.* Scientific and Engineering Computation. MIT Press, Cambridge, MA, USA, November 1997.

[2] Aho, Sethi, Ullman. *Compilers: principles, techniques and tools.* Addison-Wesley, 1986. ISBN 0-201-10194-7.

[3] Information technology – Database languages – SQL, 1992. Document Number: ISO/IEC 9075:1992.

[4] John Aycock and R. Nigel Horspool. Faster generalized LR Parsing. In *Compiler Construction 1999*, volume 8, pages 32–46, 1999.

[5] Didier Barzin. GNU-YAFL. Master's thesis, Université Libre de Bruxelles, 1998.

[6] Darius Blasband. *The YAFL Programming Language, 2nd edition.* PhiDaNi Software, 1994.

[7] Darius Blasband. *Automatic analysis of ancient languages.* PhD thesis, Université Libre de Bruxelles, 2000.

[8] M.G.J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of LNCS:235–2575, 1996.

[9] M.G.J. van den Brand, A. Sellink, and C. Verhoef. Current Parsing Techniques in Software Renovation Considered Harmful. *International Workshop on Program Comprehension, Ischia, Italy*, june 1998.

[10] Peter Bumbulis and Donald D. Cowan. Re2c: A more versatile scanner generator. *LOPLAS 2(1-4)*, pages 70–84, 1993.

[11] Frank DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460, July 1971.

[12] Frank DeRemer and Thomas J. Pennello. Efficient computation of LALR(1) lookahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, October 1982.

[13] Roger Duke, Paul King, Gordon Rose, and Graeme Smith. The object-Z specification language - version 1. Technical Report 91-1, Software Verification Research Centre; The University of Queensland, Queensland 4072, May 1991.

[14] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13/2, 1970.

[15] Etienne Gagnon. The SableCC home page. URL=http://www.sable.mcgill.ca/sablecc/.

[16] Etienne Gagnon. SableCC, and Object-Oriented compiler framework. Master's thesis, School of Computer Science, McGill University, Montreal, 1998.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, Reading, MA, 1995.

[18] GNU's Not Unix! The Bison home page. URL=http://www.gnu.org/software/bison/bison.html.

[19] Gray, R.W. γ-GLA - A generator for lexical analyzers that programmers can use. In *USENIX Conference Proceedings*, pages 147–160, 1988.

[20] Josef Grosch. Lark - An LR(1) Parser Generator With Backtracking. Technical report, CoCoLab - Datenverarbeitung, april 1998.

[21] IEEE Computer Society. SAX – The Simple API For XML. URL=http://www.computer.org/internet/v2n3/xml-sax.htm.

[22] International Business Machines. CICS. URL=http://www-4.ibm.com/software/ts/cics/.

[23] International Business Machines. RPG. URL=http://publib.boulder.ibm.com/pubs/html/as400/online/v4r5eng.htm.

[24] International Business Machines Corp. *OS and DOS PL/1 Language Reference Manual*, 1981.

[25] S. C. Johnson. YACC — Yet another compiler - compiler. Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, N.J., 1975.

[26] Paul Klint and Eelco Visser. Using Filters for the Disambiguation of Context-free Grammars. Technical Report P9426, Programming Research Group, University of Amsterdam, december 1994.

[27] Donald E. Knuth. On the translation of languages from left to right. *Information and control*, 8:607–639, 1965.

[28] Bent Bruun Kristensen and Ole Lehrmann Madsen. Methods for computing LALR(k) lookahead. *ACM Transactions on Programming Languages and Systems*, 3(1):60–82, January 1981.

[29] Bernard Lang. Deterministic techniques for efficient non-deterministic parsers. In *Proceedings of the Second Colloquium on Automata, Language and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer-Verlag, 1974.

[30] M. E. Lesk. Lex — a lexical analyzer generator. Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, N.J., 1975.

[31] Philippe McLean and R. Nigel Horspool. A Faster Earley Parser. In *Compiler Construction 1996*, volume 6, pages 281–293, 1996.

[32] Mortice Kern Systems. MKS Lex and YACC. URL=http://www.mks.com/products/interop/ly/.

[33] Nozohoor-Farshi R. Handling of Ill-designed Grammars in Tomita's Parsing Algorithm. In *International Workshop on Parsing Technologies (IWPT'89)*, pages 182–192. Carnegie Mellon University, Pittsburgh, Pa., 1989.

[34] Terence J. Parr. LL and LR Translator Need K. *ACM SIGPLAN Notices*, 31(2), February 1996.

[35] Terence J. Parr and Russell W. Quong. ANTLR: A Predicated- LL(k) Parser Generator. *Software - Practice and Experience*, 25(7):789–810, july 1995.

[36] Vern Paxson. Flex: a fast scanner generator. URL=ftp://ftp.uu.net/packages/gnu/flex/flex-2.5.4a.tar.gz.

[37] RainCode. RainCode home page. URL=http://www.raincode.com.

[38] J. Rekers. *Parser Generation for Interactive Environments.* PhD thesis, University of Amsterdam, 1992.

[39] D.J. Salomon and G.V. Cormack. Scannerless NSLR(1) parsing of programming languages. *SIGPLAN notices*, 24(7):170–178, 1989.

[40] D.J. Salomon and G.V. Cormack. The disambiguation and scannerless parsing of complete character-level grammars for programming languages. Technical Report 95/06, Department of Computer Science, University of Manitoba, Winnipeg, Canada, 1995.

[41] Klaas Sikkel. *Parsing schemata: a framework for specification and analysis of parsing algorithms.* Springer-Verlag, 1997. ISBN 3-540-61650-0.

[42] Richard M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, Inc., 1.39 edition, January 1991.

[43] Thinkage YAY parser generator home page. URL=http://www.thinkage.on.ca/shareware/yay.

[44] Masaru Tomita. *Efficient parsing for natural language.* Kluwer Academic Publishers, 1986.

[45] Vadim Maslov, Chris Dodd. BTYacc – Backtracking yacc – home page. URL=http://www.siber.com/btyacc/.

[46] Eelco Visser. Scannerless Generalized LR Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, august 1997.

[47] W3C. The Document Object Model. URL=http://www.w3.org/DOM/.

[48] T.A. Wagner and S.L. Graham. Incremental analysis for real programming languages. *SIGPLAN notices*, 32(5):31–43, 1997. Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).