Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 69 (2007) 76–85
Science of Computer Programming
www.elsevier.com/locate/scico

Proofs and pedagogy; science and systems: The grammar tool box

Adrian Johnstone*, Elizabeth Scott

Department of Computer Science, Royal Holloway, University of London, United Kingdom

Received 1 November 2005; received in revised form 19 September 2006; accepted 16 January 2007
Available online 12 October 2007

Abstract

GTB (the Grammar Tool Box) is the tool that underpins our investigations into generalised parsing. Our goal is to produce a system that supports systematic investigation of various styles of generalised parsing in a way that allows meaningful comparisons between them in a repeatable and easily accessible fashion whilst also allowing: (i) new theoretical ideas to be generated and explored; (ii) production quality parsers to be generated and (iii) humane pedagogy. GTB comprises a language (LC) with various kinds of built-in grammar and automata related objects, and a set of black-box methods written in C++ that provide implementations of grammar transforms, automata construction algorithms, parsing and recognition algorithms, and a variety of visualisation aids. In this paper we focus on the overall rationale for the GTB framework; the GTB design goals; and some detailed operational flows that are supported by GTB.

Keywords: Context-free grammar; Generalised parsing; Tool design; Tool application

The late Roger Needham famously said that research should be done with a shovel, not tweezers [20], and the computing community both academic and industrial has certainly taken this maxim to heart. However, as archaeologists know, once past the surface layers, using a shovel is dangerous because important details may be obscured; in which case only partial or even incorrect theories may be achievable. In reality, the devil hides in the details.

Computing as an academic subject has three main aspects: a purely mathematical one in which machines are seen as the physical realisation of formal results; a scientific one in which the emphasis is on controlled, repeatable experimentation in the pursuit of observable truth; and an engineering facet in which the delivery of cheap, fast and reliable systems is paramount. These three are often in tension with each other, but in all cases the degree to which we document our motivation and design decisions can be critical to wider acceptance of a technology. As in any scientific domain, proofs, scientific observations and engineered artefacts all need their broader context (as well as their core features) explained. Proofs and pedagogy; science and systems: this is what we should be doing.

Sadly, computing is not like the established sciences in this respect. We see very few genuinely repeatable studies in the literature, and thus very few repeated experiments, making it hard to distinguish a well-founded consensus view from mere fashion. To be fair, computing does not always lend itself easily to controlled experimentation. The field sits at a nexus between mathematics, engineering, psychology and perhaps sociology. Some of our phenomena are

* Corresponding author. Tel.: +44 (0) 1784 443425; fax: +44 (0) 1784 439786.
E-mail addresses: a.johnstone@rhul.ac.uk (A. Johnstone), e.scott@rhul.ac.uk (E. Scott).

0167-6423/$ - see front matter
doi:10.1016/j.scico.2007.01.016

most appropriately studied using the traditional armoury of scientific method and applied mathematics but others, such as human computer interaction, require a much broader approach.

Our primary problem domain is the parsing of general context-free grammars, an area that is particularly amenable to scientific analysis. Parsing and the related topics of programming language syntax and translator design were once central interests of the computing research community. In the most general sense this is still true since almost everything we do with computers is mediated by languages of some sort. However, the discovery of deterministic context-free parsing techniques whose space complexities were tractable for 1970's era minicomputers, and whose coverage included languages sufficiently close to human notations that they allowed comfortable programming languages to be designed, effectively caused the computing community to lose interest in parsing. It is as if eighteenth century engineers, having perfected wrought iron and demonstrated its effectiveness in bridge and ship building gave up on metallurgy and did not bother to discover steel and modern alloys. In computing, just sufficiently good seems to be good enough, and the community quickly moves on seeking the next revolution.

Apart from prospecting worked out seams for the new steel there are four reasons to attack parsing anew: (i) Moore's law conveniently halves constants of proportionality for us every eighteen months; and so general algorithms with quadratic or even cubic complexity may, with time, become attractive; (ii) new applications arise in non-synthetic languages (in particular in natural languages and biological sequence analysis), (iii) by accident or design programming languages that are inadmissible by deterministic techniques sometimes become popular and (iv) academic integrity. Reasons (i)–(iii) might reasonably appear on a grant application but in truth we are particularly motivated by (iv): in terms of completeness and the history of central ideas within computing it is surprising that we still do not know the asymptotic complexity of context-free parsing. Valiant's algorithm [30] establishes a connection between context-free parsing and Boolean matrix manipulation, and thus provides a sub-cubic bound (with impractical constants of proportionality). Lillian Lee's paper [19] shows that the relationship is bi-directional, but still the actual bound is elusive; and the practicalities of engineering such algorithms into real systems have only been explored in a piecemeal fashion.

We feel that context-free parsing will be *done* only when there exists an elegant framework in which the known algorithms appear naturally, allowing their inter-relationships to be seen, and from which general properties such as an overall complexity bound can be derived. (We have in mind here a parallel with Linear Algebra which is truly done in this sense.) Sikkel's parsing schemata [25] allow comparisons to be made between some algorithms, but has not produced insights that attack the bounds problem.

1. The grammar tool box

Proofs stand eternal, but tools become obsolete, so it is perhaps not surprising that we do not see many archival quality tool-based projects: the TEX system with its associated documentation (both technical and motivational) is one of the rare examples.

With the development of GTB our goal is to provide a tool that embodies well-engineered implementations of theoretically sound parsing and translation algorithms and to do this in a way that supports all constituencies: that is theoreticians developing algorithms; engineers needing both meaningful measurements of comparative performance and production quality generated parsers; and non-specialists who want to learn about the algorithms. As we shall discuss in more detail below, GTB comprises a set of *black-box* algorithms written in C++ along with an interpreted language, LC, which can be used as a prototyping language for new algorithms as well as a scripting language for translation tasks composed from black-box methods. LC allows the user to develop their own algorithms and to implement faithfully theoretical descriptions of existing algorithms. The black-box methods allow naïve users to study standard algorithms and engineers to benchmark algorithms.

GTB is a work in progress and is likely to remain so for some time. In particular, the incorporation of semantic actions and the generation of production quality translators is still at a rudimentary stage. The discussion in this paper focuses mainly on the black-box modules and the use of LC to access them. The black-box modules break down parser generation and the actual running of recognisers and parsers into discrete steps that reflect our theoretical treatments. In general, black-box modules come in three flavours: (i) a version that implements the published algorithm in as direct a fashion as possible; (ii) a pedagogic version which incorporates (sometimes voluminous) trace information that can be read directly or passed to an animation tool that displays the construction of, for instance, the run-time data structures for Tomita's GLR algorithm; and (iii) a tightly engineered version that uses highly optimised data structures to produce a fast implementation suitable for production use.

2. Is parsing trivial?

Deterministic parsing had been a mainstay of the undergraduate curriculum for thirty years. The current standard text [1] was originally published in 1977 and, whilst not completely trivial, deterministic parsing almost certainly does not need either a new treatment or a new implementation. Generalised parsing, however, is a different matter. The research literature has a poor track record for correctness and completeness: both Earley's parser [4] and Tomita's initial recogniser [28] contain serious theoretical flaws [6] and the accepted 'fix' for Tomita's algorithm [21] introduces significant inefficiencies. The introduction of backtracking and lookahead into deterministic algorithms has also been a fertile source of erroneous claims of generality [9]. There are very few attempts at a proof of correctness for any generalised parsing algorithm, which must certainly go some way to explaining these problems.

In addition to the errors, treatment of parsing algorithms is often *partial* in the sense that the process of extracting some or all derivations from the (potentially infinite) set of available derivations is glossed. A cubic recogniser does not always naturally extend to a cubic parser in the case where multiple derivations are possible.

A further area which requires care is the process of introducing 'short-cuts' into a parsing algorithm that close off some potential derivations during the parse: examples include backtracking algorithms that commit to a single putative parse and the reject reductions in Visser's variant of GLR parsing [31]. The effect of such on-the-fly pruning of the derivation forest is often equivalent to taking the set difference of two context-free languages (CFLs), and sadly CFLs are not closed under set difference. The practical effect is that it becomes exceedingly hard to reason about the exact language accepted by such an algorithm.[1]

In addition to these theoretical infelicities, there exist few published implementations of general parsing algorithms and almost no attempts to consider competing algorithms side-by-side in a meaningful way other than by checking clock time on a small set of examples. It is thus genuinely hard for tool designers to decide how to implement generalised parsing. The recent introduction of a 'generalised' parsing capability into Bison is a case study on how not to advance the state of the art: Bison is probably the most widely used parser generator and so in principle many people now have access to generalised parsing. However, the algorithm as implemented merely splits stacks when non-determinism is encountered leading to potentially exponential consumption of memory. To counteract this, mechanisms are provided for aborting putative parses. In a private communication, the author told us that he had not looked at the literature for GLR parsing, despite the GNU claim that a GLR algorithm has been added to Bison.

GTB will act as a repository, giving implementers access to correct, efficient and well-engineered algorithms that can, in addition, be easily related to the underpinning theoretical framework.

3. The interplay between theory and practice

GTB is a general framework in which we can explore the similarities between algorithms and automata. We hope that some of these insights will help towards the long-term goal of a natural overall theoretical framework for parsing. In this tradition, apart from Sikkel's work, we should also note Graham and Harrison's work [5] which gives a unified account of the CYK and Earley algorithms in terms of a *recognition matrix*, and which manages to address a variety of implementation issues in a manner which is nevertheless machine independent.

Our own experience is that jointly addressing both theoretical issues and engineering detail in the design of a general tool forces us to take a broad view from which useful simplifications flow. For instance, our separation of Knuth-style items into the primitive notions of slot and follow set allowed for an elegant formulation of an NFA building algorithm which generates LR(0), SLR(1) and LR(1) NFAs simply by selecting no lookahead, so-called left-hand side lookahead or right-hand side lookahead corresponding to the instance level follow sets. The method applies immediately to LR(k) NFAs if the corresponding k-follow sets are constructed. We then found a further parameterisation which specifies *unrolling* of NFAs. When this is performed on a suitable grammar (i.e. one without embedded recursion) the unrolled NFA generates the IRIA automaton discussed in [15]. We now view Knuth-style LR automata and the RIGLR IRIA as special cases of this general NFA construction function.

The modular nature of GTB has been especially helpful in the construction of automata associated with RIGLR parsing. For these to be of finite size, a grammar must be pre-processed to remove embedded recursion. The process is illustrated in the right-hand flow of Fig. 1. A Grammar Dependency Graph (GDG) is generated from a grammar by

[1] For a treatment that neatly sidesteps the problem for backtracking parsers by defining a new class of languages, see Chapter 6 of [2].

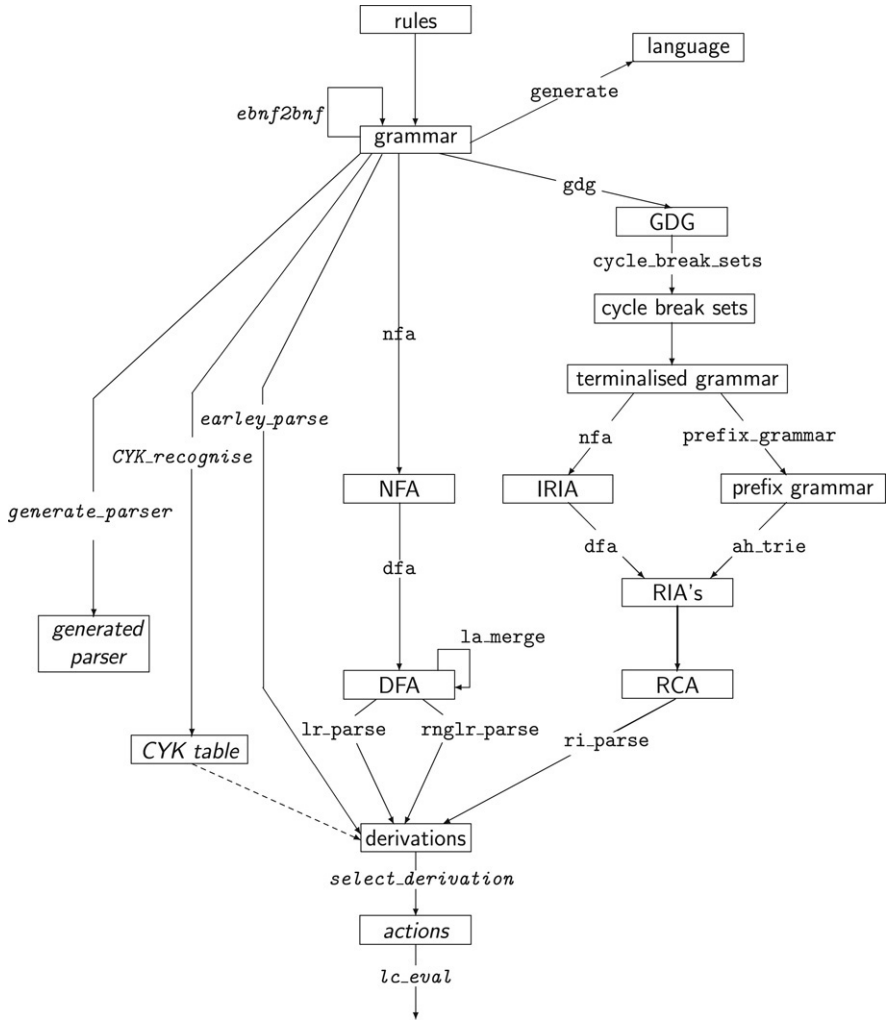

Fig. 1. Some GTB flows.

creating a node labelled for each non-terminal and adding an edge from node A to B if non-terminal B appears in a rule of A. These edges are labelled with the left and right contexts. Embedded recursion corresponds to a cycle within the GDG that shows both left and right non-empty contexts. We can use the visualisation aids in GTB to manually 'terminalise' instances of non-terminals until a *terminalisation set* which suppresses all such cycles has been found, but it turns out that finding minimal terminalisation sets is hard: in fact cycle breaking in graphs, the feedback arc set problem, is known to be NP-complete [16].

Of course, just because a problem is NP-complete we need not assume that every instance of the problem is intractable. In [8] we show that small terminalisation sets can be automatically constructed for real programming language grammars. We were able to develop the implementation and the theory hand-in-hand because GTB's representation of graphs is general and is supplemented by a library of high level graph algorithms such as Tarjan's Strongly Connected Component algorithm [27]. Code re-use allowed us to focus on the high level aspects of the problem whilst prototyping: having found two workable approaches we were able to improve run-times by three orders of magnitude by discarding the general graph representation and replacing it with tightly optimised C code. As well as optimising the code we also discovered strategies for pruning the search space of terminalisations. We see here feedback from theory into engineering and back into theory: there is an analogy here with the way that experimental and theoretical physicists interact. The engineering and theoretical traditions in computing have rather different expectations and often use different notations and, for that matter, notions. Bridging the gap can be stressful, but also productive.

4. GTB implementation and use

GTB is written in C++ to ensure broad portability and performance. An implementation in Java was contemplated, but since one of our aims is to provide performance figures for production translators we were clear that only a compiled language would be suitable. The underpinnings are derived from our earlier RDP tool [10] — the libraries for handling sets, graphs and symbol tables have been carried over and extended, and early versions were implemented using RDP itself.

4.1. The LC language

GTB contains its own dynamically typed object oriented programming language (LC) which manipulates objects held in named variables through either built-in black-box methods or methods written in LC.

LC is a dynamic object oriented language with a very small core whose interpreter is correspondingly small. We have two main design goals: (i) to produce a prototyping language with Smalltalk-like flexibility which can in principle allow algorithms published using set-theoretic levels of description to be fairly directly implemented (without the obscuring the level of clerical detail necessary in a conventional procedural programming language) and (ii) to provide good support for semantic actions, both for parsers from within GTB and for *generated* parsers; that is parsers like those constructed by YACC or RDP which run independently of the parser generator.

This latter is of course the normal production situation but is a significant barrier to parser generators in a world in which many programming languages are in use. We should like to be able to generate parsers in at least Java, C++ and C# as well as ML and some other more exotic languages; and we do allow semantic actions to be written in the target language. However, a more portable approach is to write actions in LC, and then either interpret them in the running translator or (with some restrictions on the use of dynamic aspects of LC) to translate LC actions into the target language before compilation of the translator. This is why we want the core-LC interpreter to be small: in principle a copy of it will be included with any generated translator and an LC interpreter must be written for each new target language; or alternatively a translator from LC to the target language must be written.

Although much of the inspiration for LC comes from Smalltalk's elegant execution paradigm, we use a more conventional syntax where method activation looks like a function call in a procedural language and which supports infix notation for dyadic methods. We also maintain a strict separation between the interpreter and the running LC program: it is not possible in LC to access or change the internal state of an interpreter. Although Smalltalk's everything-is-an-object philosophy is attractive and elegant it incurs a high run-time cost especially in the evaluation of control flow (although some implementations such as Timothy Budd's Little Smalltalk make concessions to efficiency by optimising some control messages).

Space, and the rather rudimentary current state of development of LC's class libraries, precludes an extensive description here.

4.2. Black-box methods

The GTB tool provides a number of built-in classes which directly represent various kinds of automata and grammar related objects. Fig. 1 shows the relationship between many of the built-in classes and black-box methods used to implement standard algorithms. Each rectangular box corresponds to a built-in object class and the edge labels are the method names that generate them. Usually method parameters are also needed to steer the construction process: we have omitted them from the figure to avoid clutter. The central and rightmost paths correspond to traditional (Knuth-style) LR bottom-up parsing and Reduction Incorporated parsing respectively.

GLR parsing is an extension of LR parsing to general grammars. GTB has an implementation of the original algorithm given by Tomita [29] and of the more efficient RNGLR algorithm [24]. Reduction incorporated parsing was introduced by Aycock and Horspool [3] with the goal of reducing stack activity in a GLR parser. The RIGLR algorithm [23] admits all context-free grammars, including those with hidden left recursion. Both GLR and RI approaches essentially compute all the traversals of a pre-constructed automaton on a given input string. The automata are constructed in stages, and these stages are reflected in Fig. 1. For the RNGLR algorithm a DFA is constructed from an NFA via the standard subset construction (using the GTB method `dfa`). For the RIGLR algorithm the grammar must first be terminalised to remove self-embedding (this is done with the aid of the GTB method `cycle_break_sets`),

then either a trie or an intermediate reduction incorporated automaton (IRIA) is constructed. For full details see [23]. The Earley and CYK algorithms operate directly on the grammar and do not require the pre-construction of an automaton.

We now briefly discuss the methods shown on the diagram which fall into four broad classes. A fuller description can be found in the GTB tutorial guide.

Grammar generation and manipulation. Grammar objects are built from rules by calling the `grammar` method on the non-terminal which is to be the start symbol of the grammar. The method pre-computes a variety of grammar attributes including FIRST and FOLLOW sets and closure sets for automaton construction. It also generates a grammar-specific enumeration that allocates unique natural numbers to each terminal, non-terminal, grammar slot[2] and DFA state.

This enumeration is fundamental to the efficiency of many internal operations since it allows bit-vector sets to represent sets of, say, DFA states or DFA states united with sets of slots. It also allows us to support polymorphism cheaply: a method may be passed a natural number which could be a particular terminal, a non-terminal or a slot, and it can discover the type of an object by checking against thresholds held in the grammar object. So there is no need for a separate type indicator field to be supplied.

Other built-in methods include *ebnf2bnf* which converts a grammar from extended BNF to simple BNF and methods to convert a grammar to two-form or Chomsky Normal Form.

Language generation. The `generate` method takes a grammar and produces sentences or sentential forms from the associated language. By default these strings are printed, but they may also be built into a trie structure for use with the RIGLR parsing algorithm.

Automaton generation. We follow Dick Grune's pedagogic approach [6] in which the standard Knuth-style LR(0) automaton is described using an NFA that is then passed through the subset algorithm [1] to produce the classical LR DFA. By changing the parameters in the `nfa` method, SLR(1), LR(1) or IRIA automata (used for reduction incorporated parsing) can be generated. We also provide the method `la_merge` which generates LALR automata by merging LR(1) DFA states.

For the RIGLR algorithm we first construct a Grammar Dependency Graph, cycles in which correspond to self-embedding in the grammar. The method `cycle_break_sets` identifies the cycles and returns terminalisations that ensure that the terminalised grammar has no self-embedding. Then an automaton is constructed either via an 'unrolled' NFA or as an Aycock and Horspool-style trie.

Parsing, recognition and translation. The parsing methods output diagnostics giving a trace of the parse and, in the case of the RNGLR algorithm, the associated graph structured stack (the data structure at the heart of Tomita-style GLR algorithms) is also output.

Parsing methods, such as `rnglr_parse` and `ri_parse` produce derivation forests in the form of Shared Packed Parse Forests. These encode a (potentially infinite) set of derivations in a graph data structure that may be traversed to extract the individual derivations. In general, the user must write a *select_derivation* function to extract the particular derivation(s) in which they are interested, after which the semantic actions decorating the rules used in a derivation may be concatenated into an LC program and passed to a fresh instance of the LC interpreter *via* the *lc_eval* function.

In addition to the main methods, LC has methods that allow objects to be written to a text file, rendered graphically (usually via a text file to be displayed by Georg Sander's VCG utility), saved to a file and restored. LC has a built-in variable called `gtb_verbose` which may be used to increase the level of trace output produced by GTB during automaton construction and during parsing.

LC is polymorphic and this is broadly exploited to allow intermediate steps in a construction to be telescoped. For instance the construction of an LALR(1) parse table requires the augmentation of the grammar, the intermediate construction of an LR(1) non-deterministic finite automaton (NFA), its conversion to a deterministic finite automaton (DFA) and the merging of states in the DFA which have matching lookaheads. Each of these steps can be explicitly specified, with the intermediate value assigned to a variable which allows subsequent manipulation (such as graphical

[2] In GTB parlance a *slot* is a position between two elements of a grammar rule. A slot roughly corresponds to the notion of LR(0) item in the literature (that is a grammar position without a lookahead set). We think of, say, an SLR(1) item as an ordered pair (S, L) where S is a slot and L is a lookahead set.

rendering). However, simply calling an LR parser with a grammar and a string causes all steps to be automatically performed, calling the LR parser with an NFA causes the later steps only to be performed and so on.

4.3. Using GTB

It is the nature of GTB that it is constantly under development as applications require. Thus the precise syntax and structure of GTB input scripts is likely to evolve. However, in this section we give a flavour of GTB use with an example in the current syntax.

The grammar rules are listed at the top of the file. Terminals are singly quoted, the empty string is denoted by #, and there may be only one rule for each non-terminal. At the time of writing the rules must be in BNF but an extension to allow EBNF rules is being developed. The required GTB methods are then listed, in execution order, enclosed in parentheses. Comments can be included in the script between `(* *)` bracket pairs.

```
(* An Example *)
S ::=  E ';' .
E ::= E '+' T | T .
T ::= '0' | '1' .
X ::=  'a' ~X 'b' | A 'a' A | X 'a' .
A ::=  'a' ~B | 'a' ~B ~B | # .
B ::=  B A | # .

(
write["\n" gtb_version " processing '" An Example "' on " date_time "\n\n"]
this_grammar := grammar[S]
this_nfa := nfa[this_grammar lr 1]
this_dfa := dfa[this_nfa]
this_lalr := la_merge[this_dfa]
gtb_verbose := true
lr_parse[this_lalr "0+1;"]
gtb_verbose := false

write[this_lalr]
render[open["dfa.vcg"] this_dfa]

this_derivation := rnglr_parse[dfa[nfa[grammar[S] slr 1]] "0+1;"]
render[open["gss.vcg"] this_derivation]

next_grammar := grammar[X tilde_enabled]
terminalise_grammar[next_grammar terminal]
ah_ex := ah_trie[next_grammar]
ri_recognise[ah_ex "aaaabb"]
write["\n" CPU_time " CPU seconds elapsed\n\n"]
)
```

The method `this_grammar:=grammar[S]` creates a grammar object whose start symbol is S. The next three method calls build an LALR table for `this_grammar` by first building an NFA, then using the subset construction to build the LR(1) DFA and then merging appropriate states. The method `lr_parse` executes a parse of the string $0 + 1$ using the LALR table. Setting `gtb_verbose:=true` causes information to be output to the screen as the parse proceeds. At each step the current stack, current state, current input symbol and the next action are listed. For example, the last step in this example is

```
Stack: [67] (12 'S') [71]
State 71, input symbol 2 '$', action 18 (R[2] R37 |1|->13 Accepting)
******: LR parse: accept
```

The method `write[this_lalr]` causes a textual representation of the LALR table to be written to the screen. This allows the user to see, for example, that rule `R[2]` referred to in the above output is indeed the reduction $S' ::= S\cdot$, the augmented start rule. The method `render[this_dfa]` allows the LR(1) DFA to be viewed graphically *via* VCG, as shown in Fig. 2.

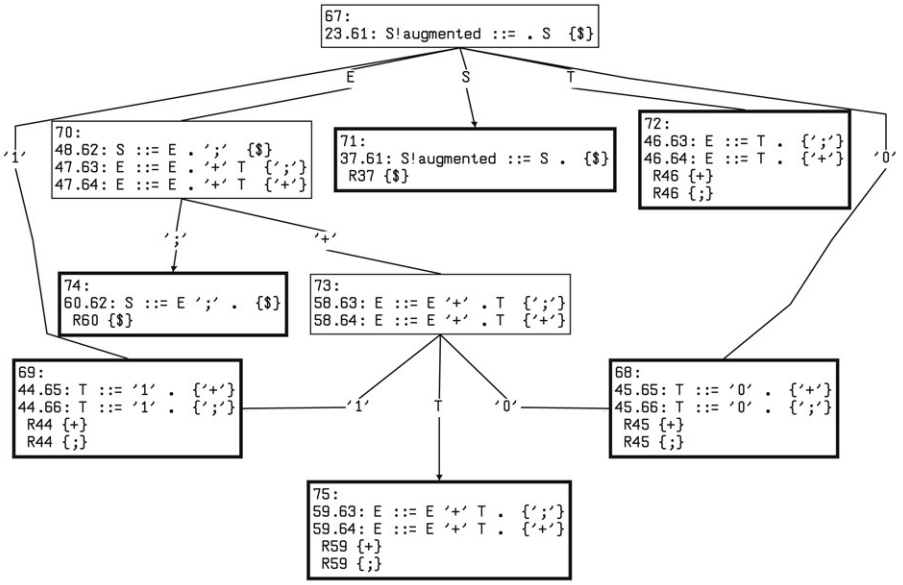

Fig. 2. LR(1) DFA generated by example script.

GTB supports the grammar analysis and automaton construction needed to carry out Aycock and Horspool-style GLR parsing. In this approach recursion must be removed from the grammar by terminalising instances of recursive non-terminals, then ultimately replacing these instances with calls to sub-automata. GTB can automatically detect and terminalise instances of embedded recursion (see the tutorial manual for more details) or the user can specify the non-terminal instances to be terminalised using the ~ notation. In normal usage GTB ignores ~ annotations to allow the same script to be used for all types of application. To incorporate the ~ annotations the grammar is created with the `tilde_enabled` flag set. The `terminalise_grammar` method turns annotated non-terminals into terminals (mutating `next_grammar` so these instances will now be treated as terminals by the other GTB methods). The method `ah_trie` builds the required automata and `ri_recognise` performs the parse.

5. Grammars for standard programming languages

A particularly curious aspect of the computing community's attitude to parsing and programming language design is the mismatch between programming language standards and real-world tools. Even the ISO standard for Pascal features a grammar which is not admitted by a deterministic parser generator although Pascal is well-known as a language whose structure is dictated by the needs of predictive (LL(1)) parsing. The C++ grammar is an uneasy mix of top-down and bottom-up idioms with a little genuine ambiguity added.[3] Historically it has been unusual for the grammar in a programming language standard to be directly used in a real compiler: the Java community has at least attempted to reconcile the needs of a language standard focussed on semantics with the needs of the translator designer by including two grammars, one of which is suitable for an LALR parser generator. Sadly these grammars contain errors.

Grammars are objects that need to be designed, engineered for efficiency, enhanced over time and curated. At the very least we should have a repository of grammars from programming language standards and tools which can turn them into executable recognisers and translators. We think the lack of general purpose generalised parsers has precluded this in the past, and we propose to build a collection of such standards-based grammars to go with GTB. At present we have grammars for ANSI-C, ANSI-C++, Java (language specifications 1, 2 and 3), Pascal, Modula-2, Oberon and for the IBM VS-COBOL grammar extracted by Ralf Lämmel and Chris Verhoef [18] whose ambitious and very welcome approach to tool support for grammar engineering is described in [17].

[3] See pages 68–69 of [26] for some insight into how this came about.

We have already carried out several investigations using GTB, on both the grammars for Pascal, C and COBOL and on grammars that demonstrate boundary case behaviour such as ambiguity and supra-cubic order. In particular, we have compared Tomita's original algorithm, and Farshi's modification of it, with the RNGLR algorithm [24,7]; we have compared the Reduction Incorporated GLR algorithm with the other GLR algorithms [23,11]; we have developed resolved right nullable tables and considered their application to RNGLR and LR parsing [22]. We have also made comparative studies of these algorithms when running with a variety of types of LR tables and discussed some of the phenomena that underlie the space and time complexities of these parsers [14,15,13]. More recently we have reported on approaches to the removal of embedded recursion in grammars, a necessary precursor to automaton construction for RIGLR parsers [12,8]. For the results of these investigations we refer the reader to the corresponding publications.

6. Concluding remarks

A preliminary version of GTB was first released in 2000. The current release (version 2.5) supports our scientific and teaching work, but is not completely ready for production use since the handling of semantic actions is still being developed. In this version, some of the functionality shown in italics in Fig. 1 is disabled: in general we only release features that we expect to remain stable in future versions. Some of the functionality still requires manual intervention and auxiliary tools are available to perform grammar-to-grammar transformation and to display animations of running parsers.

The tool has broad functionality which can be rather daunting to new users. We have constructed a substantial tutorial guide, aimed at final year computing undergraduates and those who understand the basics of translator design without necessarily knowing anything of the internals of even deterministic parsers. In a sense the tutorial comes first: when read in conjunction with a running version of GTB it constitutes an executable text book through which a rather full understanding of GLR, RNGLR and RIGLR parsing may be achieved. GTB and its tutorial documentation may be downloaded from

http://www.cs.rhul.ac.uk/research/languages/

by selecting GTB from the tools section.

Supplementary data

Supplementary data associated with this article can be found, in the online version, at doi:10.1016/j.scico.2007.01.016.

References

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers: Principles Techniques and Tools, Addison-Wesley, 1986.
[2] Alfred V. Aho, Jeffrey D. Ullman, The Theory of Parsing, Translation and Compiling, volume 1 — Parsing of Series in Automatic Computation, Prentice-Hall, 1972.
[3] John Aycock, Nigel Horspool, Faster generalised LR parsing, in: Compiler Construction, 8th Intnl. Conf, CC'99, in: Lecture Notes in Computer Science, vol. 1575, Springer-Verlag, 1999, pp. 32–46.
[4] J. Earley, An efficient context-free parsing algorithm, Ph.D. Thesis, Carnegie Mellon University, 1968.
[5] Susan L. Graham, Michael A. Harrison, Parsing of general context-free languages, Advances in Computing 14 (1976) 77–185.
[6] Dick Grune, Ceriel Jacobs, Parsing Techniques: A Practical Guide, Ellis Horwood, Chichester, England, 1990. See also: http://www.cs.vu.nl/~dick/PTAPG.html.
[7] A. Johnstone, E. Scott, Generalised reduction modified LR parsing for domain specific language prototyping, in: Proc. 35th Annual Hawaii International Conference On System Sciences, HICSS02, IEEE Computer Society, IEEE, New Jersey, 2002.
[8] Adrian Johnstone, Elizabeth Scott, Automatic recursion engineering of reduction incorporated parsers, Science of Computer Programming (submitted for publication).
[9] Adrian Johnstone, Elizabeth Scott, Generalised recursive descent parsing and follow determinism, in: Kai Koskimies (Ed.), Proc. 7th Intnl. Conf. Compiler Construction, CC'98, in: Lecture Notes in Computer Science, vol. 1383, Springer, Berlin, 1998, pp. 16–30.
[10] Adrian Johnstone, Elizabeth Scott, `rdp` — An iterator based recursive descent parser generator with tree promotion operators, SIGPLAN Notices 33 (9) (1998).
[11] Adrian Johnstone, Elizabeth Scott, Generalised regular parsers, in: Gorel Hedin (Ed.), Compiler Construction, 12th Intnl. Conf, CC'03, in: Lecture Notes in Computer Science, vol. 2622, Springer-Verlag, Berlin, 2003, pp. 232–246.
[12] Adrian Johnstone, Elizabeth Scott, Recursion engineering for Reduction Incorporated parsers, in: John Boyland, Gorel Hedin (Eds.), Proc. 5th Workshop on Language Descriptions, Tools and Applications, LDTA2005, Elsevier, 2005 (Also in Electronic Notes in Theoretical Computer Science).

[13] Adrian Johnstone, Elizabeth Scott, Giorgios Economopoulos, Evaluating GLR parsing algorithms (in press).
[14] Adrian Johnstone, Elizabeth Scott, Giorgios Economopoulos, Generalised parsing: Some costs, in: Evelyn Duesterwald (Ed.), Compiler Construction, 13th Intnl. Conf, CC'04, in: Lecture Notes in Computer Science, vol. 2985, Springer-Verlag, Berlin, 2004, pp. 89–103.
[15] Adrian Johnstone, Elizabeth Scott, Giorgios Economopoulos, The grammar tool box: A case study comparing GLR parsing algorithms, Electronic Notes in Theoretical Computer Science 110 (2004) 97–133.
[16] R.M. Karp, Reducibility among combinatorial problems, in: R.E. Miller, J.W. Thatcher (Eds.), Complexity of Computer Calculation, Plenum Press, 1972, pp. 85–103.
[17] P. Klint, R. Lämmel, C. Verhoef, Towards an engineering discipline for grammarware, in: ACM TOSEM, May 30 2005 (in press). Online since July 2003, 47 pages.
[18] R. Lämmel, C. Verhoef, Semi-automatic grammar recovery, Software—Practice & Experience 31 (15) (2001) 1395–1438.
[19] Lillian Lee, Fast context-free grammar parsing requires fast boolean matrix multiplication, Journal of the ACM 49 (2002) 1–15.
[20] Roger Needham, Tom Omitol, ACM Fellow profile, ACM SIGSOFT 26 (1) (2001) 7–10.
[21] Rahman Nozohoor-Farshi, GLR parsing for ϵ-grammars, in: Masaru Tomita (Ed.), Generalized LR Parsing, Kluwer Academic Publishers, Netherlands, 1991, pp. 60–75.
[22] Elizabeth Scott, Adrian Johnstone, Reducing non-determinism in right nulled GLR parsers, Acta Informatica 40 (2004) 459–489.
[23] Elizabeth Scott, Adrian Johnstone, Generalised bottom up parsers with reduced stack activity, The Computer Journal 48 (5) (2005) 565–587.
[24] Elizabeth Scott, Adrian Johnstone, Right nulled GLR parsers, ACM Transactions on Programming Languages and Systems (in press).
[25] Klaas Sikkel (Ed.), Parsing schemata, in: Texts in Theoretical Computer Science, Springer-Verlag, 1993.
[26] Bjarne Stroustrup, The Design and Evolution of C++, Addison-Wesley Publishing Company, 1994.
[27] Robert E. Tarjan, Depth-first search and linear graph algorithms, SIAM Journal on Computing 1 (2) (1972) 146–160.
[28] Masaru Tomita, Efficient Parsing for Natural Language, Kluwer Academic Publishers, Boston, 1986.
[29] Masaru Tomita, Generalized LR Parsing, Kluwer Academic Publishers, Netherlands, 1991.
[30] L. Valiant, General context-free recognition in less than cubic time, Journal of Computer and System Sciences 10 (1975) 308–315.
[31] Eelco Visser, Syntax definition for language prototyping, Ph.D. Thesis, University of Amsterdam, 1997.