A Set of Tools to Teach Compiler Construction

Akim Demaille Roland Levillain Benoît Perrot
EPITA Research and Development Laboratory (LRDE)
14/16, rue Voltaire
F-94276, Le Kremlin-Bicêtre, France
akim@lrde.epita.fr roland@lrde.epita.fr benoit@lrde.epita.fr

ABSTRACT

Compiler construction is a widely used software engineering exercise, but because most students will not be compiler writers, care must be taken to make it relevant in a core curriculum. Auxiliary tools, such as generators and interpreters, often hinder the learning: students have to fight tool idiosyncrasies, mysterious errors, and other poorly educative issues. We introduce a set of tools especially designed or improved for compiler construction educative projects in C++. We also provide suggestions about new approaches to compiler construction. We draw guidelines from our experience to make tools suitable for education purposes.

Categories and Subject Descriptors

K.3.2 [**Computers & Education**]: Computer & Information Science Education—*Computer Science Education*; D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers, Parsing, Run-time environments*

General Terms Design, Management

Keywords Compiler Design, Object Oriented Programming, Educational Projects, Design Patterns, Tools.

1. INTRODUCTION

In [9] we introduced "yet another compiler construction project": the Tiger project. Contrary to its peers (e.g., [2, 4, 5, 6]) its primary objective is *not* to implement a compiler, since "students will (most likely) never design a compiler" [8]. In the terms of [17], our approach is "software project" oriented: the actual goal is to teach undergraduate students Object-Oriented Programming (OOP), C++, design patterns, software engineering practices, long run team work, etc. The scale is another significant difference: 250 students making about a dozen submissions in groups of four — more than 2000 (proto-)compilers to assess each year.

We used off-the-shelf tools when available. They often proved to be inadequate for students: cryptic and/or incomplete error messages, lack of conformity with the coding standard we demand of students, and from time to time, impossible automation. They were frequently pointed out as sources of confusion during the end-of-the-year debriefings with the students. So over the years, to improve their educative value and/or to ease the management of such a large scale project, we improved several of these tools and created new ones taking student advice and requests into account. In [9] we skimmed over these tools; reviewers and readers asked for more details. This paper addresses their demand. A small survey on compiler construction project Web sites shows that some obsolete tools (e.g., Yacc) are still widely used, and that some teachers still evaluate code correction by hand. This paper also aims at discouraging this.

The contributions of this paper are to introduce tools tailored to students and tools facilitating evaluation of student projects, to highlight some rarely used features that enhance learning, to show new teaching techniques enabled by these tools, and to try to specify what features an auxiliary tool should have to be suitable for education.

Figure 1 presents the compiler, Section 2 component generators, and Section 3 pedagogical interpreters, whose interest is discussed in Section 4. Section 5 concludes.

2. GENERATED COMPILER COMPONENTS

Some compiler components are understood well enough to be generated from high-level descriptions (Figure 1). Section 2.1 covers parser generation. The parser builds an AST, whose generation is exposed in Section 2.2. Section 2.3 presents an assembly code generator generator. They all produce C++ code that comply with strict standards.

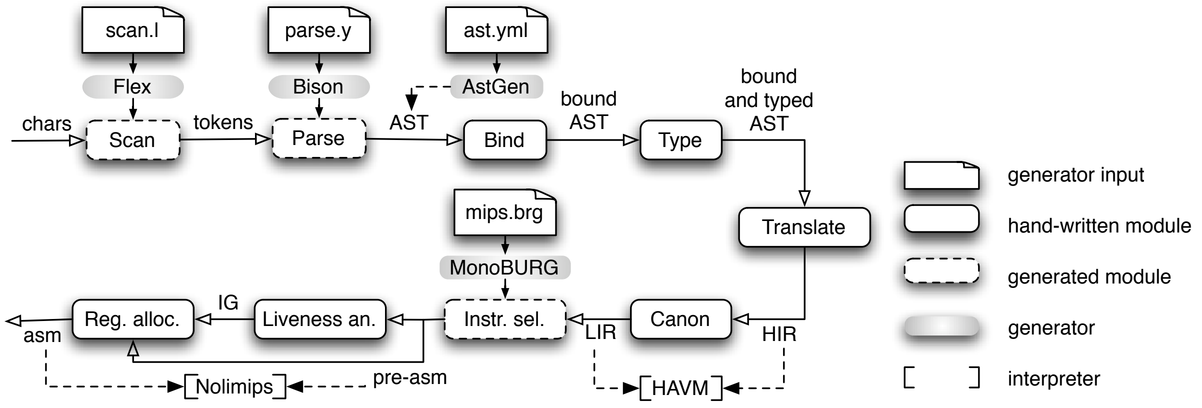
2.1 Parser Development

The Yacc [11] parser generator lacks many features proposed by modern alternatives (e.g., [13]). Our curriculum covers LALR(1) parsing because it is still widely used. To address its shortcomings, we have extended Bison, the GNU Yacc implementation [7], and improved the educative value it lacked.

2.1.1 LALR(1) Automaton Development

LALR(1)'s speed come with a cost: hard-to-solve conflicts. Educators often demonstrate how to address some conflicts. The Tiger language provides two syntactic constructs that bear enough similarity to confuse LALR(1):

```
/* Create a table of 51 integers initialized to 0. */
int_array [51] of 0;
/* Access cell #42 of the array. */
my_array [42];
```



The scanner (sans-serif names refer to the module names) breaks the input stream of characters into a stream of tokens, from which the parser builds an Abstract Syntax Tree (AST). Uses are bound to their declaration in accordance with scoping rules. The type checker verifies that the AST is correct. Several stages (Translate, Canon) perform multiple simple translations to High- and Low-level Intermediate Representations (IRs). Then instruction selection produces pseudo-assembly using an unlimited register set. Liveness analysis produces an Interference Graph (IG) used by the register allocator to produce the final assembly code.

Figure 1: The Tiger Compiler main modules

Both start with `"id" [ exp ]` ; it is only the following `"of"` that allows one to decide. The corresponding (natural) grammar is not LALR(1), nor even LR(k):

```
   exp: "id" "[" exp "]" "of" exp | lvalue
lvalue: "id" | lvalue "[" exp "]"
```

To help students understand and solve the conflict, we made Bison provide a detailed textual presentation of the automata, as illustrated below. New material is highlighted.

```
   state 0         0 $accept: . exp $end
 2                 1 exp: . "id"  "["  exp  "]"  "of"  exp
                   2    | . lvalue
 4                 3 lvalue: . "id"
                   4       | . lvalue  "["  exp  "]"
 6   "id"    shift , and go to state 1
    exp      go to  state  2
 8  lvalue   go to  state  3

10 state 1        1 exp: "id" .  "["  exp  "]"  "of"  exp
                  3 lvalue: "id" .               [$end, "[", "]"]
12   "["       shift , and go to state 4
     "["      [reduce using rule 3 (lvalue)]
14 $default   reduce using rule 3 (lvalue)
```

The old presentation had poor pedagogical value: too concise, and often significantly different from the traditional exposure in class for no good reason. Firstly, the automaton had no initial rule (rule 0) from which the automaton is derived, and had a different termination scheme. It was the source of recurrent questions from students. Secondly, Bison presented only the "core" of the "item sets" (the list of "pointed rules"), which is sufficient for trained users, but confuses students (e.g., only line 1 would have been reported for state 0). In the case of the initial state (lines 1–5) the item set was not actually given at all! Only actions were reported, out of the blue. Thirdly, the lookaheads (as in line 11) were not reported. This is of tremendous help in some cases to understand why LALR(1) fails.

In addition to the generation of the textual description of the automaton, we added graphical representations generations. While these representations are barely usable to debug real life parsers, they do help novices deal with conflicts in small scale grammars, or to check automaton computations performed by hand. They are also a valuable help to prepare lecture notes.

2.1.2 Parser Actions

Parsers generated by Yacc/Bison execute user-provided actions when a rule is "recognized". These actions may manipulate so-called *semantic values* (which attribute grammar practitioners actually call *synthesized attributes*) associated to the various symbols. Unfortunately, to refer to the semantic values one must use their index in the rule:

```
exp: " identifier " "[" exp "]" "of" exp
//$$    $1        $2   $3  $4  $5   $6
{ $$ = new Array (@$, new Type (@1, $1), $3, $6) }
```

This is hard to read and error-prone, yet Yacc is still widely used. Under the intensive use by students, accepted shortcomings had to be addressed. We modified Bison to allow the user to name the symbols.

To produce good error messages, the compiler needs to keep track of the location of the symbols. Automatic location tracking was implemented: `@$`, `@1`, `@2`, etc. denote the location of each symbol. The following excerpt from the Tiger parser demonstrates it, together with named symbols.

```
res=exp: type=" identifier " "[" size=exp "]" "of" init=exp
{ $res = new Array(@res, new Type(@type, $type), $size, $init) }
```

2.1.3 The Parser Execution

To ensure that students correctly manage the memory, we promote the use of checkers [9]. They have revealed a well-known fact: it is impossible to properly reclaim memory during error recovery or when special actions such as `YYACCEPT` or `YYABORT` are used. This was solved by the introduction of the `%destructor` directive.

During its development, the run-time behavior of a parser may be incorrect. For instance, if the aforementioned conflict is improperly solved, it may fail to recognize the sentence `my_array [my_index]`. Debugging traces (dating back to Yacc) help:

```
   Starting parse
2  Entering state 0
   Reading a token: token ‘‘ identifier ’’ (1.0−1.9: my_array)
4  Shifting token ‘‘ identifier ’’ (1.0−1.9: my_array)
   Entering state 1
6  Reading a token: token ‘[’ (1.10−1.11: )
   Shifting token ‘[’ (1.10−1.11: )
8  Entering state 4
   Reading a token: token ‘‘ identifier ’’ (1.11−1.19: my_index)
10 Shifting token ‘‘ identifier ’’ (1.11−1.19: my_index)
   Entering state 1
12 Reading a token: token ‘]’ (1.20−1.21: )
   Reducing stack by rule 3 (line 9):
14       $1 = token ‘‘identifier’’ (1.11−1.19: my_index)
   −> $$ = nterm lvalue (1.11−1.19: my_index)
16 Stack now 0 1 4
   Entering state 3
18 Next token is token ‘]’ (1.20−1.21: )
   Reducing stack by rule 2 (line 6):
20       $1 = nterm lvalue (1.11−1.19: my_index)
   −> $$ = nterm exp (1.11−1.19: my_index)
22 Stack now 0 1 4
   Entering state 7
24 Next token is token ‘]’ (1.20−1.21: )
   Shifting token ‘]’ (1.20−1.21: )
26 Entering state 9
   Reading a token: Now at end of input.
28 1.22: syntax error, unexpected end of file, expecting ‘‘of’’
   Error: popping token ‘]’ (1.20−1.21: )
30 Error: popping nterm exp (1.11−1.19: my_index)
   Error: popping token ‘[’ (1.10−1.11: )
32 Error: popping token ‘‘identifier’’ (1.0−1.9: my_array)
```

Important data (highlighted above) was missing, making it virtually impossible to spot errors in long inputs. Students were wasting time gaining useless expertise: processing incomplete traces. Bison has been improved in several ways:

- The locations are reported.
- The user can specify how to print semantic values using the new `%printer` Bison directive (e.g., line 3).
- The reductions (e.g., line 13) now detail both the right-hand side symbols (type, location and value) before the reduction takes place, and the computed result.
- Error recovery, as exposed in class, is made explicit (see the "popping" lines below, starting at 29).
- The discarded symbols are freed thanks to `%destuctor`.

2.1.4 Bison Back-ends

The Tiger project teaches C++, which Bison did not support. During the first years we used the regular C output, but it had several shortcomings which denatured the exercise into "how to embed C++ code in a hostile C environment". So we equipped Bison with a multiple back-end architecture and a LALR(1) C++ back-end was developed and used for the Tiger assignments.

Usually, with the help of extended descriptions of automata and run-time traces, students manage to resolve the conflicts such as the one of Section 2.1.1.

Although teaching this process is valuable — LALR(1) is widely used — students should also be given the opportunity to use a GLR parser generator. Generalized LR (GLR) [14] has numerous advantages:

- It can process any Context-Free Grammar (CFG). This is in sharp contrast with LR(1) which supports only deterministic languages, and even sharper with LALR(1).
- It is modular. Because CFGs are closed under union (contrary to (LA)LR(k)), modules can be assembled.
- It supports nondeterministic grammars, such as that of Section 2.1.1, by providing unlimited look-ahead.
- It even supports ambiguous grammars, yielding parse forests instead of simple parse trees.

GLR gracefully addresses complex grammars, or "globally nice" grammars with local complexities. When properly implemented, GLR parsers are as efficient as LALR(1) parsers on LALR(1) grammars. P. Hilfinger provided Bison with an efficient GLR back-end in C, on top of which we built a C++ version. It suffices to change the back-end to make the natural grammar of Section 2.1.1 work: the GLR parser is a working drop-in replacement for the incorrect LALR(1) parser.

This enables new approaches to teaching parser generation. In the first step LALR(1) is used to train students in conflict resolution on a simple grammar fragment, then GLR is used for the full, more complex, possibly locally ambiguous, grammar. Conversely, one may first use GLR to focus on parser generation, and then switch the same grammar to LALR(1) to teach conflict resolution recipes.

2.2 Abstract Syntax Tree Generation

In typical compilers (Figure 1), the parser builds an AST: an intermediate representation of the input program. In OOP, ASTs are straightforwardly implemented as classes composed of "standard" parts (constructors, destructor, visitor hooks, attributes, accessors...). Tools such as Treecc [18] generate AST support (classes and traversals). Some parser generators automate the AST generation [13], and some even derive it from the grammar [15]. These tools are actually *too* convenient: The implementation by hand of AST support is an ideal introduction to OOP and C++. In our curriculum, it is the first C++ assignment. Since students are provided with code with gaps, we needed the classes to be simple, "look" hand-crafted, and be carefully documented. No tool fulfilled these needs, so we wrote an AST generator (which is not provided to the students). It produces AST nodes and basic visitors (abstract visitor, identity visitor, and cloning visitor). Parts declared "teacher only" are not given to the students. This tool helped us change the architecture to address the shortcomings spotted during the yearly debriefings with the students [9]. It also helps us change the assignment from year to year, to diminish the interest of cheating by stealing code from previous classes.

2.3 Code Generator Generation

The back-end of the compiler performs the "instruction selection": The translation from an Intermediate Language (IL) (Section 3.1) to assembly with arbitrarily many registers. It consists in mapping tree patterns into assembler instructions. This well studied topic led to the inception of several code generator generators such as Twig [3] or BURG [10]. Since no suitable tool existed, we used to provide a hand-written code generator, which was hard to maintain and difficult to understand or enhance.

This module was one of the least-interesting parts of the compiler, with poor educative content, which led to dissatisfaction among students. We finally chose to improve an existing (free software) generator to match our (and our students') expectations: Monoburg, the BURG-like generator from the Mono Project [1]. It produces code generators in C using bottom-up rewriting, from an IL grammar. The rules of this grammar express acceptable rewriting patterns from Tree, the IL we use, to assembly:

```
binop: Binop(lhs : exp, rhs : Const)
{ EMIT (ASM.binop (tree−>oper(), lhs, rhs−>value())); }
```

Thanks to Benoît Perrot and Michaël Cadilhac, Monoburg became a standalone project that suits our needs, providing

- C++ features, e.g., namespaces and references;
- named arguments to simplify their manipulation (e.g., *lhs* and *rhs* above, instead of having to reach them from the current top-level node, *tree*);
- modules, thanks to a new `%include` directive;
- easier development and debugging, e.g., `#line` statements linking the generated code and the input file(s).

Using Monoburg was a considerable improvement: some students implemented some assembly optimizations, and even wrote back-ends for other architectures.

3. PEDAGOGICAL INTERPRETERS

As depicted in Figure 1, a compiler is a long pipe. In the Tiger project, students deliver code for virtually every stage. Hence, each stage should produce machine and human readable output, to enable automated checking, and to ease manual debugging. Automated verification is mandatory in the context of large classes: in our case about 70 groups submit a partial compiler every two/three weeks (possibly several times if students with bad results are given additional chances). On the one (front) end, checking an AST is straightforward: pretty-print it back to source code, and compare the result with the input[1]. On the other (back) end, checking the assembly code is simple: merely run it on an actual computer, or on a simulator. Checking intermediate representations is troublesome. We developed Havm (Section 3.1) to check the Intermediate Representations (IRs), and Nolimips (Section 3.2) to check pre- and final assembly code.

3.1 Register-Based Intermediate Language

In most compilers several stages perform simple translations on several Intermediate Languages (ILs) instead of a single jump from the source to the target language. Two classes of IL are common: stack-based languages (such as the Java Byte Code, or MSIL, the IL used in the .Net framework), or register-based languages. While several environments exist for stack-based IL, none was available for register-based IL.

Havm, written by Robert Anisko, is free software available on the Internet. It interprets TREE, a simple high-level register-based IL introduced by Andrew W. Appel [4]. In a later stage, the compiler transforms TREE code into a lower level subset of TREE. Havm is an interpreter for both these high- and low-level ILs—in the latter case it checks that the restrictions are verified.

It features two sets of temporaries. The *special* temporaries correspond to specific purpose registers such as the frame and stack pointers (`fp`, `sp`), and incoming (`i0`, `i1`, etc.) and outgoing (`rv`) argument registers. Using these registers, the student is in charge of the stack management. In typical compilers, register allocation is not performed yet, and therefore registers are not preserved across function calls. To comply with this, Havm provides an *unlimited* set of *plain* temporaries and recursion support by saving and restoring the whole set across calls. The combination of explicit stack frame management and implicit recursion support for registers proved to help introduce students gradually to the implementation of recursion support in actual languages and compilers.

In addition to simple verifications (e.g., uninitialized memory reads), Havm features a trace mode that helps students analyze incorrect executions, and simple performance measurements, to compare programs in terms of efficiency.

3.2 MIPS Assembly Language

The last compiler stages (Figure 1) usually process assembly language. Simple and clean, MIPS is a commonly chosen architecture for education [4, 12, 16]. We are no exception.

B. Perrot introduced Nolimips, a MIPS simulator. It allows one to test compilers on any computer, and provides several features specifically developed for education that other MIPS simulators lack (e.g., SPIM [12], MARS [16]).

Because registers are not allocated at instruction generation (Figure 1), students cannot check their modules at this stage. They have to complete the full compiler before checking, but then errors might come from either the instruction selection, the liveness analysis, or the register allocation. The instructors are not in a better position: the assessment cannot be automated. To address this, Nolimips supports arbitrarily many general purpose registers.

Challenging a complete register allocator for MIPS is also delicate. Since there are 32 registers, writing tests that exercise "spills" (i.e., when no register is left and temporaries have to be allocated on the stack) is very difficult, and a useless skill. Nolimips makes this straightforward: It can downgrade the MIPS architecture by limiting the number of caller-save, callee-save or argument registers. It can also check that callee-save registers are preserved across function calls, as required by the MIPS calling convention.

Programs generally generate side-effects, such as printing or reading characters. Other services typically provided by operating systems (exiting, memory allocation or disposal) may improve the experience as well. These tasks can be carried out in Nolimips through the use of convenient system calls, inspired by the standard C library.

Compared to our previous use of SPIM, Nolimips helped our students diagnose their mistakes much earlier, and even allowed them to uncover bugs. Indeed, some calling convention violations can remain unnoticed. While these features were designed for education, they do make sense in production by making stress tests easier to design.

4. DISCUSSION

4.1 Feedback from the users

Two assistant professors make the lectures, and 20 mentors, students from older classes who enjoyed the project, handle lab sessions and assist the students at any time. These educators reported that from year to year, the interaction with the students is less demanding. Basic questions are less frequently asked since the tools now provide students with more relevant information. The quality of the delivered compilers also increased, in particular thanks to the early control by stricter tools. Automating everything that could be, included gathering the projects and evaluating them, allowed us to spend more time on more advanced issues. Though it is harder to get feedback from the students, since they don't see the evolution of the tools, the

[1]External AST comparison is slightly less simple because there might be modifications such layout changes, syntactic sugar removal, routine inlining and so forth. We actually compare that the operations are idempotent: the pretty-printed AST should remain unchanged if parsed and pretty-printed again.

mentors report that the tools led more students to success than before with less assistance. Besides, some improvements were prompted by students and mentors.

4.2 Software tools for education

What makes a tool suitable for education purposes? In our experience it provides several distinctive features:

Low learning curve. While practitioners are used to complex and possibly inconsistent interfaces, instructors don't have the leisure to spend time on them. Students need to focus on the actual issues. This requirement is at the origin of most changes in Nolimips, the one-directive change to move from LALR(1) to GLR with Bison, etc.

Fidelity to theory. Tools should obey as strictly as possible to the theory. The previous lack of initial rule in Bison disrupted students.

Complete context. Providing as much context as possible in errors or debugging traces also helps novices to locate and understand a problem. The changes in Bison's automaton report, location tracking (which benefits Bison too), and debugging traces fall into this category.

Fill the gap. Large pieces of software that cannot be checked before completion are a daunting task for students. The ability to run Intermediate Representations (IRs) thanks to Havm and Nolimips enables most groups to succeed where only a few groups used to. As far as the authors know, no industrial strength compiler exercises IRs this way.

Paranoid checks. "Useless" constraints can bring out existing bugs more easily. For instance, Nolimips provides means to augment the register pressure artificially to stress the register allocation.

Abundant and relevant warnings Report valid but dubious constructs. Activate warning flags and teach students to pay attention to them.

Added freedom. As emphasized in [9], this project must be doable by a majority of students. This demoralizes the best ones who think they've done no better than the majority. So any such project should provide optional challenges. Thanks to GLR and Monoburg some students did work that went much further than the requirements.

Compliance Tools should comply with the standard we demand of students. For instance, it is unacceptable to ask students to use generated code that leaks memory.

Auxiliary educative tools also help instructors by (i) freeing them from interactions with students about basic problems, (ii) improving the whole quality allowing one to focus on more interesting issues, and (iii) automating parts of the assessment. As an example, from one year to the following, we doubled the number of groups whose parser passes more than 97% of our tests, we tripled the number of correct ASTs, and doubled the average grade for the binding stage.

5. CONCLUSION

Addressing shortcomings in the tools students use improves the learning by making it more efficient. The tools we changed or introduced for compiler construction project gave them more freedom to experiment, more leisure to understand the core issues, more opportunities to find their mistakes by themselves. They also helped us automate parts of the management of the project. Following the guidelines we propose should help to improve other student project frameworks.

Acknowledgments

Olivier Gournet designed Figure 1. Stephen Frank, Benoît Sigoure, and Alexandre Borghi proofread this paper.

6. REFERENCES

[1] Mono home page. `http://www.mono-project.com`.

[2] A. Aiken. Cool: A portable project for teaching compiler construction. *ACM SIGPLAN Notices*, 31(7):19–24, July 1996.

[3] A. W. Appel. Concise specifications of locally optimal code generators. Technical Report CS-TR-080-87, Princeton University, Dept. of Computer Science, Princeton, New Jersey, February 1987.

[4] A. W. Appel. *Modern Compiler Implementation in C, Java, ML*. Cambridge University Press, 1998.

[5] J. Aycock. MBL: A language for teaching compiler construction. Technical Report 1995-574-26, Department of Computer Science, University of Calgary, 1995.

[6] D. Baldwin. A compiler for teaching about compilers. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education (SIGCSE'03)*, pages 19–23, Reno, Nevada, USA, February 2003.

[7] R. Corbett, R. Stallman, and P. Hilfinger. Bison: GNU LALR(1) and GLR parser generator, 2003. `http://www.gnu.org/software/bison/bison.html`.

[8] S. Debray. Making compiler design relevant for students who will (most likely) never design a compiler. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 341–345. ACM Press, 2002.

[9] A. Demaille. Making compiler construction projects relevant to core curriculums. In *Proceedings of the Tenth Annual Conference on Innovation and Technology in Computer Science Education (ITICSE'05)*, pages 266–270, Universidade Nova de Lisboa, Monte da Pacarita, Portugal, June 2005.

[10] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG–fast optimal instruction selection and tree parsing. Technical Report CS-TR-1991-1066, 1991.

[11] S. C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979. AT&T Bell Laboratories Technical Report July 31, 1978.

[12] J. R. Larus. SPIM S20: A MIPS R2000 simulator. Technical Report TR966, Computer Sciences Department, University of Wisconsin–Madison, 1990.

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

[14] M. Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.

[15] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[16] K. Vollmar and P. Sanderson. MARS: An education-oriented MIPS assembly language simulator. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education (SIGCSE'06)*, pages 239–243, Houston, Texas, USA, March 2006. ACM Press.

[17] W. M. Waite. The compiler course in today's curriculum: Three strategies. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education (SIGCSE'06)*, pages 87–91, Houston, Texas, USA, March 2006. ACM Press.

[18] R. Weatherley. Treecc, the Tree Compiler-Compiler. `http://www.southern-storm.com.au/treecc.html`, 2002.