Extending Old Compiler Tools with Meta-Tools

John Aycock
Department of Computer Science
University of Calgary
2500 University Drive N.W.
Calgary, Alberta, Canada T2N 1N4
Phone: +1 403 210 9409, Fax: +1 403 284 4707
Email: aycock@cpsc.ucalgary.ca

***Abstract*— There are many tradeoffs involved in choosing between a new, more powerful software tool and an older, more established one. The best way to handle this problem may be to make the old tool more powerful through the use of meta-tools. Compiler tools suffer from this exact problem – we present YETI, a meta-tool that provides a framework for transforming and enhancing Yacc specifications. YETI can generate new Yacc specifications which automate common tasks, enhancing programmer productivity.**

***Index Terms*— Tools, Meta-tools, Parsing, Yacc.**

I. Introduction

How can a transition be made from old, very well-established software tools to new, more powerful ones? Practically speaking, there are a number of issues that may be involved:

- Software developers are unfamiliar with the new tools, and may need training or retraining.
- Training materials may not be readily available for the new tools.
- The new software tools may not be universally available on all platforms of interest.
- The new tools may be under active development, presenting a moving target for software developers.
- Support and the future existence of the new tools may be questionable.

This tool-transition problem arises in the area of compiler tools. The most ubiquitous parser generation tool, Yacc [1], [2], dates back *a quarter of a century*.[1] In the late 1970s, computing resources were much more limited, resulting in an engineering tradeoff – using a weaker parsing algorithm in exchange for reduced space requirements. Yacc (and the GNU incarnation, Bison) is now available for most any system imaginable, generations of compiler writers have been trained using it, and it has influenced the design and implementation of other parser tools.

We are now at a point computationally where more powerful, easier-to-use, more resource-intensive parsing algorithms can be feasibly used. There are a variety of new parser tools incorporating these algorithms [4]–[8], but they suffer the tool-transition problem described above, especially against a tool as old and entrenched as Yacc.

[1] [3, page 99] has a first-hand account of the problem that led to Yacc's inception.

Perhaps the solution to the tool-transition problem is not training, not tool stability, not even using the new tools at all. Perhaps we should be looking at new, more powerful ways of using existing tools rather than adopting new ones.

We have taken this approach with Yacc. In the remainder of this paper we describe YETI – Yacc-Enhancing Tool Infrastructure – which allows Yacc to be used in more productive ways. Effectively, YETI is a meta-tool, which automatically generates code to enhance an existing Yacc specification. Programmers are still using Yacc, avoiding many training problems, and YETI is a portable, lightweight tool whose output is not reliant on YETI's continued existence.

Starting with a brief Yacc tutorial, we present YETI's architecture, two different transformations that YETI is currently able to perform, then related work and possible applications.

II. A Yacc Primer

Yacc is a parser generator tool for compiler writers. It takes a Yacc specification as input, and produces C code as output that implements a table-driven parser. The output code from Yacc is compiled by the programmer and linked in with the compiler they are building.

Yacc specifications have three sections, separated by %% symbols:

```
declarations
%%
grammar rules
%%
C code
```

Declarations simply declare information about symbols used in the grammar rules. The grammar rules themselves specify what inputs to the generated parser are valid; the grammar rules may have code snippets embedded within them to perform actions at certain points during parsing. C code in the third section is copied to the generated output file untouched.

III. YETI: Yacc-Enhancing Tool Infrastructure

YETI transforms existing Yacc specifications by automatically inserting new code into them. The architecture of YETI is shown in Figure 1:

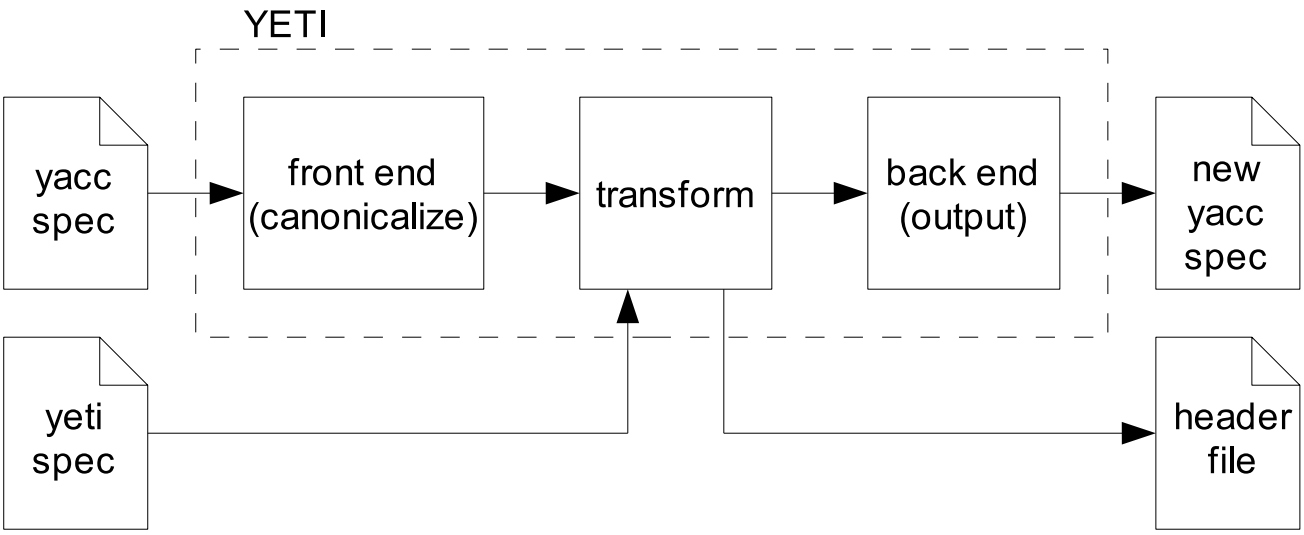

Fig. 1. YETI architecture.

- Yacc specifications are read in and then canonicalized by the front end. Converting into a canonical form simplifies later parts of YETI.
- An additional specification file may be required, depending on the transformation the user has selected. If necessary, this additional specification file is read directly by the transformation engine.
- The new, transformed Yacc file is output by YETI's back end. The back end is decoupled from the transformation process, allowing different output formats to be supported by YETI. Current back ends output either the transformed Yacc specification or just the grammar (useful for extracting the grammar for use in different parsing tools).

YETI is written in Python, an object-oriented scripting language, and consists of approximately 2100 lines of code.

IV. Transformation Examples

Currently, YETI supports two transformations, both of which automatically add code to the Yacc specification to construct an abstract syntax tree (AST). This is a common task, and is usually done by the programmer laboriously writing repetitive code in the Yacc specification to manually construct the AST.

We will use the Yacc specification in Figure 2 as a running example; it parses a list of key/value pairs. Such a grammar would describe flat file databases, such as the domain name resolver configuration shown in Figure 3.

A. Simple AST Construction

Simple AST construction builds an AST using two user-supplied functions:

`AST_TERMINAL`
: Passed the token type, a pointer to the token, and the token's attribute, this function converts a token into a leaf node in the AST.

```
%token ID

%%

list    : list pair
        | pair
        ;

pair    : {flag = 1;} key value
        ;

key     : ID ;
value   : ID ;

%%
```

Fig. 2. Sample Yacc specification.

```
domain cpsc.ucalgary.ca
nameserver 127.0.0.1
nameserver 192.168.100.1
```

Fig. 3. Sample input file with key/value pairs. Yacc would see six "ID" tokens for this file.

`AST_NONTERMINAL`
: This function constructs an interior AST node. Its arguments are the node type, its arity, and an array of child nodes.

YETI's transformation inserts code into each grammar rule to call these functions appropriately. A naïve user implementation of these two functions, which constructs the AST nodes exactly as requested by YETI-generated code, results in the construction of a parse tree (also called a concrete syntax tree). Figure 4 shows such a tree built for the input in Figure 3.

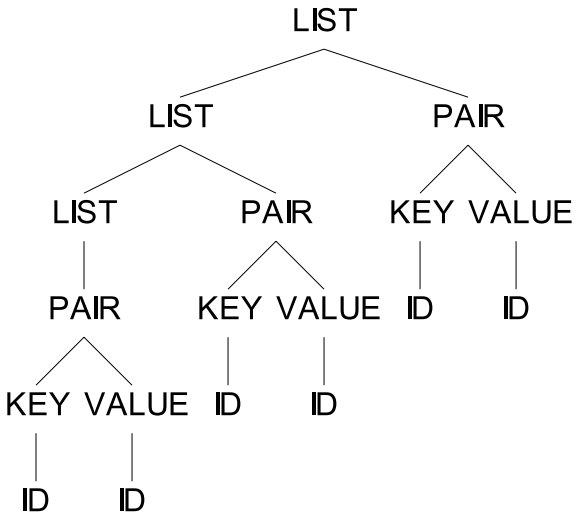

Fig. 4. Parse tree built by simple AST construction.

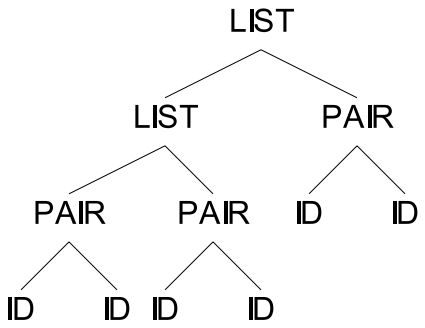

Fig. 5. Flattened AST, produced with simple AST construction modifications.

A user may make simple changes to their two AST-building functions that mold the AST from the parse tree. For example, `AST_NONTERMINAL` could be modified so that it doesn't construct a new node if only one child is present. This type of modification avoids creating some extraneous AST nodes, and results in the much-flattened tree in Figure 5.

Part of the Yacc output from YETI is shown in Figure 6. Besides the calls to the two AST-building functions, YETI has inserted appropriate declarations for the grammar, and preserved the user-specified code that was in the input specification – such an action might be required to send lexical feedback to the scanner, which is outside the scope of YETI's transformation. The `AST_ROOT` function is called at the root of the tree to allow the tree to be processed by the user's code in an application-specific manner. All the data structures YETI uses, such as AST nodes and token types, are user-defined for maximum flexibility.

This method of simple AST construction has been used for over four years in our SPARK toolkit [8] and has proved simple but effective. Among other things, we have used it to build ASTs for a bytecode decompiler, and for transformation specifications in YETI itself.

```
%{
    static AST_TYPE *_ast_kids[3];
%}
%union {
    AST_TOKEN_TYPE AST_TOKEN_TAG;
    AST_TYPE *ast_node;
}
%type <ast_node> key
%type <ast_node> list
%type <ast_node> pair
%type <ast_node> value
%token <AST_TOKEN_TAG> ID
%start list
%%
list: list pair {
    _ast_kids[0] = $1;
    _ast_kids[1] = $2;
    $$ = AST_NONTERMINAL(
              2, 2, _ast_kids
         );
    AST_ROOT($$);
} ;

...

pair: {flag = 1;} key value {
    _ast_kids[0] = $2;
    _ast_kids[1] = $3;
    $$ = AST_NONTERMINAL(
              3, 2, _ast_kids
         );
} ;
key: ID {
    _ast_kids[0] = AST_TERMINAL(
                       5, ID, $1
                   );
    $$ = AST_NONTERMINAL(
              1, 1, _ast_kids
         );
} ;

...

%%
```

Fig. 6. Partial output for simple AST construction.

```
list/2   $1 +($2)
list/1   LIST ($1)
pair/3   PAIR ($1, $2)
key/1    $1
value/1  $1
```

Fig. 7. Sample AST shaping specification.

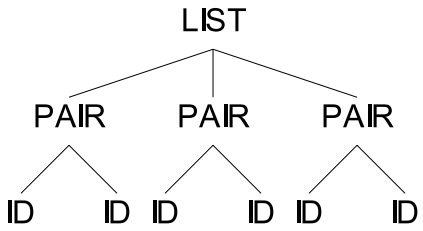

Fig. 8. AST produced with AST shaping.

```
...

%%
list: list pair {
    $1 = AST_APPEND($1, $2);
    $$ = $1;
    AST_ROOT($$);
} ;
list: pair {
    _ast_kids[0] = $1;
    $$ = AST_NONTERMINAL(
             1, 1, &_ast_kids[0]
         );
    AST_ROOT($$);
} ;

...
```

Fig. 9. Partial output for AST shaping.

B. AST Shaping

AST shaping is a somewhat newer concept, only a year and a half old, but we have used the method in many substantial projects already: Yacc files in YETI, an assembler, a compiler, and a grammar utility. It allows a more concise, higher-level, finer-grained description of AST construction than is easily achievable with simple AST construction.

YETI uses a separate specification file to describe how AST shaping should be performed. A specification file for the running example in shown in Figure 7, which associates a shaping rule with each rule in the original grammar. We use a Prolog-inspired syntax for the association – `list/2` identifies the grammar rule for "list" with two items on its right-hand side, for instance. Our experience is that this is usually sufficient to uniquely identify a grammar rule, but an extended mechanism is supported too:

```
list/2 with list
```

and

```
list with list
```

both specify the same grammar rule, allowing a distinguishing symbol on the grammar rule's right-hand side to be named.

The AST shaping rules can take one of six forms:

1) New AST node. Given the node type to build, calls `AST_NONTERMINAL` to create a new node, as in simple AST construction.
2) Existing AST node. Used for passing subtrees without modification or, if applied to a token, automatically creates a leaf AST node by calling `AST_TERMINAL`.
3) Append to existing AST node. Extremely useful for constructing lists in an AST, such as representing a list of statements or a list of arguments.
4) NULL. For AST designs that call for a NULL placeholder.
5) Function call. Invokes a user-specified function to construct the AST node, effectively supplying an escape mechanism for shaping too complicated to be described by the shaping rules.
6) Arbitrary code. The ultimate escape mechanism, which just copies the supplied code directly into the output.

AST shaping rules are processed recursively, so tree shapes of arbitrary complexity can be described. Items in a grammar rules's right-hand side are referenced using `$1`, `$2`, `$3`, and so on. Figure 7's shaping rules are very straightforward: `ID` tokens are made into leaf nodes; key/value pairs are hung off new `PAIR` nodes; one `LIST` node is made initially for the first key/value pair, and subsequent pairs are appended to that node. Figure 8 shows the resulting AST for the input in Figure 3, and Figure 9 shows sample output from YETI.

The ease with which an AST can be built with AST shaping, compared to manual construction or even simple AST construction, has made it our method of choice for new projects.

V. Related Work

Automatically generating code is a well-known technique for software development [9]–[11]. Strictly speaking, even tools like Yacc are meta-tools, since they generate code for another tool (the C compiler), but this distinction is usually overlooked.

While there are some systems that enhance parser tools in some respect (e.g., [12]), the closest work to ours is MetaTool™ [13]–[15].[2] As an application generator, their tool was much more ambitious (and heavyweight) than ours, and was not intended to address tool-transition problems. Ox [16] is another related tool, for working with attribute grammars; in contrast, our tool is a much more general framework.

A different approach is to change the legacy tool itself. Recent versions of Bison have included experimental support

[2]MetaTool was formerly known as STAGE, and the work is now carried on as *ivy*meta*™.

for a more powerful parsing algorithm. We would argue, however, that a new parser tool is effectively created by doing so, presenting the same tool-transition problem as before.

VI. Possible Applications and Conclusions

The above transformations really just begin to demonstrate what can be done with YETI. YETI can be viewed as a general framework for applying transformations to Yacc specifications; new transformations can be added to YETI in the same way that plug-ins can be added to Eclipse [17]. Some possible applications include:

- Converting legacy Yacc specifications to new/different forms, for use in reengineering tools.
- Automatically inserting debugging code as an aid to development.
- Inserting code to graphically display Yacc's operation, for use in training.
- Enhancing the power of the parsing algorithm Yacc uses, making it easier to use, without changing Yacc itself.
- Applying transformations to the grammar automatically.
- Adding support for attribute grammars, which would allow more of the compilation process to be specified at a high level.

Our experience with YETI suggests that it is definitely possible to breathe new life into old tools. Adoption of newer tools is not always feasible, and the meta-tool approach can provide enhanced programmer productivity with an existing tool.

Acknowledgment

Thanks to Shannon Jaeger and Rob Walker for commenting on an early version of this paper. This work was funded in part by the Natural Sciences and Engineering Research Council of Canada.

References

[1] S. C. Johnson, "YACC — yet another compiler compiler," *UNIX Programmer's Manual, 7th Edition*, vol. 2B, 1978.
[2] J. R. Levine, T. Mason, and D. Brown, *Lex & Yacc*, 2nd ed. O'Reilly & Associates, 1992.
[3] P. H. Salus, *A Quarter Century of UNIX*. Addison-Wesley, 1994.
[4] A. van Deursen, "Introducing ASF+SDF using the λ-calculus as example," in *Executable Language Definitions*. PhD thesis, University of Amsterdam, 1994.
[5] F. W. Schröer, "The ACCENT compiler compiler, introduction and reference," German National Research Center for Information Technology, Tech. Rep. 101, June 2000.
[6] P. T. Breuer and J. P. Bowen, "The PRECC compiler compiler," in *Proceedings of the UKUUG/SUKUG Joint New Year 1993 Conference*, 1993, pp. 167–182.
[7] S. McPeak, "Elkhound: A fast, practical GLR parser generator," University of California, Berkeley, Tech. Rep. UCB/CSD-2-1214, Dec. 2002.
[8] J. Aycock, "Compiling little languages in Python," in *Proceedings of the 7th International Python Conference*, 1998, pp. 69–77.
[9] J. Herrington, *Code Generation in Action.* Manning, 2003.
[10] J. Bentley, "Little languages," in *More Programming Pearls*. Addison-Wesley, 1988, pp. 83–100.
[11] R. S. Pressman, *Software Engineering: A Practioner's Approach*, 3rd ed. McGraw-Hill, 1992.
[12] B. Trancón y Widemann, M. Lepper, and J. Wieland, "Automatic construction of XML-based tools seen as meta-programming," *Automated Software Engineering*, vol. 10, pp. 23–38, 2003.
[13] J. C. Cleaveland, "Building application generators," *IEEE Software*, vol. 5, no. 4, pp. 25–33, July 1988.
[14] I. L. Sindelar, "Specification-driven tool technology," in *Proceedings of the Sun User Group 1990 Conference*, 1990, pp. 209–219.
[15] T. T. Wetmore, "MetaTool compiler generator," 1990, Usenet, `comp.compilers` 90-10-038.
[16] K. M. Bischoff, "Ox: An attribute-grammar compiling system based on Yacc, Lex, and C: User reference manual," 1993.
[17] *Eclipse Platform Technical Overview*, Object Technology International, Inc., Feb. 2003.