# An Automatic Object-Oriented Parser Generator for Ada

**Martin C. Carlisle**
Department of Computer Science
2354 Fairchild Dr., Suite 6K41
U.S. Air Force Academy, CO 80840-6234
carlislem@acm.org

**ABSTRACT**

Although many parser generator tools (aka compiler compilers) are available, very few create a parse tree. Instead, the user must add actions to the grammar to create the parse tree. This is a tedious, mechanical task, which could easily be automated. We propose a simple scheme to map a grammar to an object-oriented hierarchy, and provide a tool, called AdaGOOP, that creates lexer and parser specifications complete with all of the actions necessary to create the parse tree.

## 1 INTRODUCTION

A large number of tools have been developed to automate lexer and parser generation [Arc93,Con97,FSF99,Joh75,LS75,MF81,Met99,Pax90,etc]. Probably the most well known of these are Lex and Yacc [Joh75,LS75] and their GNU cousins Flex and Bison [FSF99,Pax90]. Although these tools were developed for C programmers, Ada versions (Aflex and Ayacc) were developed at the University of California-Irvine [Arc93].

Using these tools, the programmer provides a set of regular expressions and an LR(1) grammar, and obtains a completely functional lexer and parser. While using the SCATC versions of these tools [Con97] to create an Ada implementation of a software reengineering tool that translated Fortran to object-oriented Ada 95 and Java [SH97], we discovered that we were repeatedly inserting actions into the specifications to create a parse tree, and that these actions could be directly inferred from the grammar itself. As a result, we decided to create a tool, AdaGOOP (Ada Generator of Object-Oriented Parsers), that would insert the actions needed to create the parse tree directly into the scaflex and scayacc input. This tool is distributed in the public domain as a service of the United States Air Force; but it is unsupported and without any warranty. It is currently available at: ftp://ftp.usafa.af.mil/pub/dfcs/carlisle/usafa/adagoop. The Public Ada Library (PAL) mirror is at http://wuarchive.wustl.edu/languages/ada/usafa/adagoop.

## 2 MAPPING FROM GRAMMAR TO OBJECTS

To begin, we create an abstract object, Parseable, that will serve as the root of our hierarchy. (It has no fields.) A production A => B C, where B and C are non-terminals, then corresponds to a node in the parse tree containing pointers to the subtrees derived through B and C. We can then create the following type declaration:

```
type A_nonterminal is new Parseable with record
   B_part   B_nonterminal_ptr:    pointer to B nonterminal
   C_part   C_nonterminal_ptr:    pointer to C nonterminal
end record;
```

To handle all terminal symbols, we create a single object type as follows:

```
 type Parseable_Token is new Parseable with record
    Line           Natural;
    Column         Natural;
    Token_String   String_Ptr;
    Token_Type     Token;    enumeration type of all tokens
 end record;
```

Therefore, if we have A => B terminal1 C, we simply add a pointer to a Parseable_Token as the second field of the record. The parse tree then contains all of the information to completely recreate the original source text, with the exception of comment tokens (which are ignored).

We then must address the issue of multiple productions for a single non-terminal. To accomplish this, we insert an abstract class for the non-terminal, and have concrete classes corresponding to each possible production. For example, if we have A => B terminal1 C | D, then we would have the following tagged type declarations:

```
type A_nonterminal is abstract new Parseable with null record;
type A_nonterminal_Ptr is access all A_nonterminal_Class;
type A_nonterminal1 is new A_nonterminal with record
   B_part   B_nonterminal_ptr:    pointer to B nonterminal
   Terminal1_Part   Parseable_Token_Ptr;
   C_part   C_nonterminal_ptr:    pointer to C nonterminal
end record;
type A_nonterminal2 is new A_nonterminal with record
   D_part   D_nonterminal_ptr:    pointer to D nonterminal
```

```
end record;
```

The tool also will automatically number the fields in the case where a non-terminal or terminal appears more than once on the right hand side of a production.

## 3 USING THE TOOL

AdaGOOP is distributed as Ada95 source code. To use the tool, simply download and compile the source (adagoop.adb contains the main procedure). Since we use the GNAT specific attribute 'Unrestricted_Access, you will need to either use the GNAT compiler, or another compiler supporting this attribute, or modify generate.adb to remove the 6 occurrences of this attribute (e.g., by using global variables).

AdaGOOP takes as input a set of regular expressions and a grammar, and generates inputs to scaflex and scayacc, along with a package specification for the parse tree.

AdaGOOP should be executed as follows:

```
adagoop input_file output_prefix
```

The following files are generated as output of AdaGOOP: output_prefix.l (scaflex input), output_prefix.y (scayacc input), output_prefix_model.ads (parse tree package spec). For example, in the Ada 95 sample folder, running "adagoop ada95.g ada95" generates ada95.l, ada95.y and ada95_model.ads.

Your input file must have the following form (bold face indicates reserved words):

Comments begin with "--" and must appear on a line by themselves. Comment and blank lines may appear anywhere in the input file.

```
token_macros
-- the token_macros section allows you to associate names with regular expressions
-- so they can be used more than once in the token descriptions.
name    regular_expression
name    regular_expression
...
```

tokens

-- the tokens section allows you to associate names with tokens

-- the token_macros may be used as part of the regular expressions

-- if the name is "ignore" (not in quotes), then this token will be skipped and not returned

-- ignore may not be used in the grammar

name  regular_expression

name  regular_expression

...

global_methods

-- specify the name of methods that will be generated in the spec for every tagged type.

name

name

...

grammar

-- Use yacc format to specify the grammar

-- ignore may not be used in the grammar

-- sequence may be empty.  Only one sequence per name is required

-- no actions may be provided (these are generated automatically)

-- begin with %start to specify the goal symbol

%start goal_symbol

name :  sequence of non-terminals or tokens | second sequence | ... ;

name :  sequence of non-terminals or tokens | second sequence | ... ;

...

Once AdaGOOP has been run, you can create the lexer and parser by running (for our example):

```
scaflex ada95.l
gnatchop -w ada95.a
gnatchop -w ada95_io.a
gnatchop -w ada95_dfa.a
scayacc ada95.y
gnatchop -w ada95.a
```

```
gnatchop -w test_goto.a
sed -e 1d ada95_tokens.a > ada95_modified_tokens.a
gnatchop –w ada95_modified_tokens.a
gnatchop -w ada95_shift_reduce.a
```

Note gnatchop is needed only if your compilation environment (such as GNAT) requires package specifications and bodies to appear in separate files. sed is used only to delete the first line of ada95_tokens.a. We needed to add with context clauses to this file, but they appeared after the package declaration. To resolve this, we add the context clauses, another package declaration, and within a batch script use sed to delete the first package declaration.

## 4 CONCLUSIONS AND FUTURE WORK

AdaGOOP allows a programmer to quickly create a parser that generates an object-oriented parse tree. As a result, it is quite useful for generating simple language translators. Currently, we are using AdaGOOP to generate a translator from a subset of Ada 95 to NQC [Bau99]. This will allow us to teach a section of our introductory course using Ada to program the Lego Mindstorms robots.

From this work, we have discovered several ideas for improvements to AdaGOOP. First, since it is quite common to do an in-order traversal of the parse tree, it would be useful to automatically generate code that performs this traversal. Second, it would be helpful to be able to specify methods for each different object type, rather than simply methods that are common to all grammar rules. Third, comments are not included in the parse tree (since they do not appear in the grammar). The tool would be useful for creating a reformatter if the comments were somehow encoded. Fourth, the generated source code can become quite large. Another possibility to explore is breaking up large grammars into multiple Ada 95 source files. We would welcome additional contributions to this project (either grammars or improvements to the tool).

## REFERENCES

[Arc93] Arcadia Project. “Aflex and Ayacc.” http://www.ics.uci.edu/~arcadia/Aflex-Ayacc/aflex-ayacc.html

[Bau99] Baum, David. 1999. “Not Quite C Home Page.” http://www.enteract.com/~dbaum/nqc/

[Con97] Conn, Richard. 1997. “The Source Code Analysis Tool Construction Project.” *Proceedings of Tri-Ada ’97*, 141-148. See also http://wuarchive.wustl.edu/languages/ada /userdocs/html/cardcat/scatcdsk.html

[FSF99] Free Software Foundation. 1999. “Bison-GNU Project-Free Software Foundation.” http://www.gnu.org/software/bison/bison.html

[Joh75] Johnson, S.C. 1975. “Yacc—yet another compiler compiler.” C.S. Technical Report #32. Murray Hill, NJ: Bell Telephone Laboratories.

[LS75] Lesk, M.E., and Schmidt, E. 1975. “Lex—a lexical analyzer generator.” In *Unix Programmer’s Manual 2*. Murray Hill, NJ: AT&T Bell Laboratories.

[MF81] Mauney, Jon, and Fischer, Charles N. 1981. “An improvement to immediate error detection in Strong LL(1) parsers.” *Information Processing Letters* 12(5):211-12.

[Met99] Metamata, Inc. 1999. “The JavaCC Story.” http://www.metamata.com/JavaCC/story.html

[Pax90] Paxson, Vern. 1990. “Flex users manual.” Ithaca, NY: Cornell University.

[SH97] Sward, R.E., and Hartrum, T.C. 1997. “Extracting objects from legacy imperative code.” In “*Proceedings of the 1997 International Conference on Automated Software Engineering.”*

## A SAMPLE AdaGOOP SPECIFICATION

```
token_macros
DIGIT           [0-9]
INTEGER         ({DIGIT}*)
EXPONENT        ([eE](\+?|-){INTEGER})
DECIMAL_LITERAL  {INTEGER}\.{DIGIT}{INTEGER}{EXPONENT}?

tokens
-- Reserved Words
OPEN      [oO][pP][eE][nN]
PROCEDURE [pP][rR][oO][cC][eE][dD][uU][rR][eE]
WRITE     [wW][rR][iI][tT][eE]
--Other
```

**NUMBER  ({INTEGER}|{DECIMAL_LITERAL})**

**global_methods**

**Print**

**grammar**

**-- testing**

**%start A**

**A : A PROCEDURE B | B;**

**B : B OPEN C | C;**

**C : WRITE NUMBER;**