Ayacc User's Manual

Version 1.1
Arcadia Document UCI-94-01
March 1994

Designed by
David Taback and Deepak Tolani

Enhanced by
Ronald J. Schmalz
Yidong Chen

Arcadia Environment Research Project
Department of Information and Computer Science
University of California, Irvine

Contents

List of Figures

1   Introduction

Ayacc provides Ada programmers with a tool for the automatic construction of parsers. The parsers are constructed from a high level description of a context free grammar. The input to **Ayacc** consists of a BNF style specification of a grammar accompanied by a set of Ada program statements to be executed as each rule is recognized. **Ayacc** generates a set of Ada program units that act as a parser for the specified grammar. These program units may be interfaced to additional user supplied routines to produce a functional program. **Ayacc** was inspired by the popular UNIX utility, **Yacc**, and it closely mimics the features and conventions of its C counterpart. The error recovery features are similar to those of eyacc.

2   Description

The following chapter is intended to serve as a tutorial and reference guide to **Ayacc**. No previous knowledge of **Yacc** is assumed although basic principles of compiler theory such as context-free grammars, lexical analysis, and parsing are taken for granted. For the sake of clarity, a consistent set of terminology is adopted in this chapter. The term **token** denotes a structure recognized by a lexical analyzer and **nonterminal** refers to a structure recognized by a parser. **Grammar symbols** collectively refers to nonterminals and tokens.

   Ayacc generates four Ada program units that may be compiled and interfaced to other Ada code provided by the user. To enable the compilation of **Ayacc** output, the user must provide a minimum of two routines: a lexical analyzer function and an error reporting procedure. These routines may be kept inside the specification file or provided as external modules.

   Ayacc generates a total of four files. Assuming that the original specification file is named *base.y*, the corresponding **Ayacc** output files would be: *base_tokens.ada*, *base_shift_reduce.ada*, *base_goto.ada*, and *base.ada*. If the *Error_Recovery* command line parameter is on then an additional file named *base_error_report.ada* is also generated. In addition, **Ayacc** also generates a temporary file, *base.accs*, which is automatically deleted upon completion and should not be of concern to the user. A brief description of these files follows:

base.ada

    The primary output of **Ayacc**. Procedure *YYParse* is generated into
    this file along with any Ada code provided in the declaration section at
    the end of the specification file.

base_tokens.ada

    The *tokens* package that provides the type and variable declarations needed by both *YYParse* and the user supplied lexical analyzer. The package is named *Base_Tokens*.

base_shift_reduce.ada **base**_goto.ada

    The parse tables used by *YYParse*. They are generated as separate packages rather than nested within *YYParse* to prevent them from being pushed onto the stack with each invocation of *YYParse*. Strictly speaking, the tables could be placed within a single package, however some Ada compilers may have problems compiling the large preinitialized arrays which comprise the tables. The parse table packages are named *Base_Goto* and *Base_Shift_Reduce* respectively.

3   Command Line Interface

When the **Ayacc** command is entered without arguments, the following specification is displayed on the user's terminal.

    −−Ayacc: An Ada Parser Generator.
    type Switch **is** (On, Off);

procedure Ayacc (	File	: **in** String;
	C_Lex	: **in** Switch := Off;
	Debug	: **in** Switch := Off;
	Summary	: **in** Switch := On;
	Verbose	: **in** Switch := Off;
	Error_Recovery	: **in** Switch := Off;
	Extension	: **in** String := ".a");

−− File	Specifies the Ayacc Input Source File.
−− C_Lex	Specifies the Generation of a 'C' Lex Interface.
−− Debug	Specifies the Production of Debugging Output
−−	By the Generated Parser.
−− Summary	Specifies the Printing of Statistics About the
−−	Generated Parser.
−− Verbose	Specifies the Production of a Human Readable
−−	Report of States in the Generated Parser.
−− Error_Recovery	Specifies the Generation of Automatic
−−	Error Recovery in the Generated Parser.
−− Extension	Specifies the File Extension to be Used for
−−	Generated Ada Files.

Figure 1. Ayacc Command Line Specification

3.1 Overview

The **Ayacc** command line interface is modeled after the syntax and semantics of Ada procedure calls. Both *Named* and *Positional* parameter associations and *Default Parameters* are supported. [1] Although the command line interface does follow the syntax and semantics of Ada, the strictness of these rules has been relaxed to improve the user interface. The nature of these relaxations is discussed in the following section.

3.2 Command Format

The command line interface has several relaxations to promote friendlier usage. A summary of these relaxations are listed in **Figure 2.**

---

1. Final Semicolon on the procedure call is optional.

2. Outermost Parentheses are optional.

3. Parentheses around aggregate parameters are optional when the aggregate consists of only one component.

4. Commas in the parameter list are optional.

5. Quotes around string literals are optional.

Figure 2. Syntactic Relaxations in the Command Line Interface

3.3 Invoking Ayacc

When **Ayacc** is invoked, the command line interface analyzes the command line for the correct number and types of parameters. If no errors are detected, the command line is echoed to the terminal in the form of an Ada procedure call with all parameters displayed as Named Associations. A typical invocation is shown in Figure 3. Once the command line is analyzed, the parameters are passed on to the tool for processing. **Note:** Some Operating Systems, for example *Unix*, may interpret the => prior to passing the argument to the tool. As a result, any OS special characters should be *escaped* on the command line to prevent interpretation.

---

[1]A complete discussion of this topic can be found in the Ada Language Reference Manual, §6.4-6.4.2.

ayacc parser.y debug => [2] on

```
Ayacc (  File     => "parser.y",
         C_Lex => Off,
         Debug => On,
         Summary => On,
         Verbose => Off,
         Error_Recovery => Off,
         Extension => ".a");
```

Figure 3. Echoed Command Line Following Invocation

3.4 Command Line Errors

1. **Invalid Named Association.**

2. **Invalid Parameter,** *Bad_Parameter* **is not a legal value for type** *Parameter_Type***.**

3. **Invalid Parameter Association,** *Bad_Formal* **is not a valid Formal Parameter.**

4. **Invalid Parameter Order, Positional arguments must precede Named.**

5. **Missing Positional Argument.**

6. **Unbalanced Parentheses.**

4 Input to the Tool

4.1 Command Line Options

Input specifies the Ayacc specification file which will be translated into an appropriate parser. The format of the file is described in the *Ayacc Specification File* section.

C_Lex specifies the generation of an Ada interface to a lexical analyzer written in C. When run with the *C_Lex* option, **Ayacc** will generate a file called *base.h* which contains a sequence of #define's for the tokens (analogous to the file created by **Yacc** when run with the -d option) and a package "Base_C_Lex" that converts integers returned by the C lexical analyzer into their corresponding Ada enumeration type. This feature is particularly suited

---
[2] In Unix, => should be replaced with = \ >.

to interfacing to lexical analyzers generated by the popular UNIX tool **Lex** or any lexical analyzer that adheres to the conventions expected by **Yacc**.

When using the *C_Lex* option, the user must supply a C function called *get_token* which returns an integer corresponding to a token recognized by the lexical analyzer. The values returned by the lexical analyzer must adhere to the following conventions: *Character literals have the same value as their ASCII representation*, and *All other tokens have the value are as defined in the base.h file. It is the user's responsibility to insure that the C lexical analyzer always returns an integer corresponding to a valid token or character literal.*

The package *Base_C_Lex* contains the Ada function *YYLex* which converts the integer returned by *get_token* into the token enumeration type expected by the parser. The user will have to make minor changes to *YYLex* if it is necessary for the lexical analyzer to set the value of YYLVal or to perform actions when specific tokens are recognized.

Debug specifies that **Ayacc** should generate a version of *YYParse* that prints the shift, reduce, error, and accept actions as they are executed by the parser. Figure 4 lists the messages produced by **Ayacc** in *Debug* mode.

---

1. **Accepting Grammar...**

2. **Can't Discard End_Of_Input, Quitting...**

3. **Error Recovery Clobbers** *Token***.**

4. **Error Recovery Popped Entire Stack, Aborting...**

5. **Examining State** *State***.**

6. **Looking for State with Error as Valid Shift.**

7. **Reduce by Rule** *Rule* **Goto State** *State***.**

8. **Shifted Error Token in State** *State***.**

9. **Shift** *State* **on Input Symbol** *Token***.**

Figure 4. Debug Option Informative Messages

The output may be used with the *Verbose* option output file to debug grammar specifications by identifying states where the parser behaves incorrectly. The debugging output is also useful for observing error recovery mechanisms taken by the parser.

Verbose specifies that **Ayacc** should generate a file called *base.verbose* which contains a readable form of the parser. This is very useful when the user wants to see the finite state machine associated with the parser. A detailed example of using the *Debug* and *Verbose* option can be found in Appendix 2.

4.2   Ayacc Specification File

Error_Recovery specifies that a parser which does automatic syntax error recovery is to be generated. The generated parser will create a file named *base.lis* when run which records the parsed lines of the input text and specifies where errors occurred. Further explanation of this option can be found in Appendix C.

Extension specifies the file extension to be used for generated Ada files. The default value is ".a".

An Ayacc specification consists of three parts: the *token declarations*, *grammar rules*, and an *optional user declarations*. A double percent *%%* delimits each of these sections. Ada style comments may appear anywhere in the specification file. A sample input file for a calculator grammar is shown in **Figure 5.** An optional fourth part is added if the *Error_Recovery* option is used. See Appendix C for more information.

−− Declarations Section

%token IDENTIFIER – Tokens which will be returned
%token NUMBER – by the lexical analyzer.

```
{

    −Declarations that will be
    −written to the tokens package.
    subtype YYSType is Integer;
}

%% ───────────────────────────────────── -
−− Rules section

−− Rules specifying the syntax of arithmetic expressions.
−− "expression", "term", and "factor" are the nonterminals
−− recognized by the parser.

expression : term
           | expression '+' term
           | expression '-' term
           ;

term : factor
     | term '*' factor
     | term '/' factor
     ;

factor : IDENTIFIER
       | NUMBER
       | '(' expression ')'
       ;
%% ─────────────────────────────────

−− User declarations
−− Empty in this case
```

Figure 5. Sample Calculator Grammar

4.2.1 Token Declarations Section

Ayacc requires tokens of the grammar to be explicitly declared in the token declarations section. A token declaration consists of a *%token* keyword followed by a list of identifiers that may optionally be separated by commas. All token names must follow Ada enumeration type naming conventions as the tokens are directly translated into an enumeration type. An example of a tokens declaration is shown in Figure 6.

```
%token identifier , number
%token if_statement while_loop -- comma is optional
%token ',' ''' -- literals are allowed
```

Figure 6. Legal Ayacc Token Declarations

Ayacc also allows the user to place declarations in the tokens package by enclosing a collection of Ada declarations in braces in the tokens declaration section.

4.2.2 Associating Ada Types with Grammar Symbols

Ayacc also provides a way to associate an Ada data type to nonterminals and tokens. The data type is defined by associating an Ada type declaration to the identifier *YYSType*. Once this type is defined, actions can access the values associated with the grammar symbols. This declaration must appear in the tokens declarations section or the Ayacc output will fail to compile. For example, a declaration of the form

%token a b d e
{

    subtype YYSType **is** Integer;

}
%%

allows the grammar symbols to have integer values that can be accessed and set by the user-defined actions.

Since the types declared in the Tokens Declaration section may require the visibility of types and operations defined in other packages, **Ayacc** provides a mechanism for specifying a *Context Clause* for the generated tokens package. The keywords defined for this purpose are **%with** and **%use**. Although these keywords are used with the same syntax as **%token** keyword, they may only be used prior to the Ada declarations section. An example of their usage is shown in Figure 7.

%token '=' '+' '-' '/' '*' NUMBER IDENTIFIER

%with Binary_Operator_Manager —— These MUST precede the Ada
%use Binary_Operator_Manager —— declarations section.

```
{
    type YYSType is
        record
            Operation : Binary_Operator_Manager.Operator_Expression_Type;
            Precedence : Binary_Operator_Manager.Precedence_Type;
        end record;
}
%%
—— The Tokens package generated by Ayacc is shown below:

with Binary_Operator_Manager;
use Binary_Operator_Manager;
package Test_Tokens is

    type YYSType is
        record
            Operation : Binary_Operator_Manager.Operator_Expression_Type;
            Precedence : Binary_Operator_Manager.Precedence_Type;
        end record;

    YYLVal, YYVal : YYSType;
    type Token is (End_Of_Input, Error,
                    '=', '+', '-', '/', '*',  Number, Identifier );

    Syntax_Error : exception;

end Test_Tokens;
```

Figure 7. Specifying a Context Clause for the Tokens Package

4.2.3 Rules Section

The rules define the grammar to be parsed. Each rule consists of a nonterminal symbol followed by a colon and a list of grammar symbols terminated by a semicolon. For example, a rule corresponding to a street address could be represented as:

Address : Street City ',' State Zip ;

Street, *City*, *State*, and *Zip* must be either nonterminal or token symbols that are defined elsewhere within the specification file. Characters enclosed in single quotes, such as the comma in the example above, are tokens that appear as character literals in the input. Unlike other tokens, character literals do not have to be explicitly declared in the declarations section. Unlike **Yacc**, **Ayacc** does not allow escape characters to be entered as literals.

For convenience, the vertical bar may be used to factor rules with identical left hand sides. When using the vertical bar notation, the semicolon is used only at the end of the last rule. For example,

```
A : B C D ;
A : E F;
A : G ;
```

can be abbreviated as

```
A : B C D
  | E F
  | G
  ;
```

Nonterminal names consist of a sequence of alphanumeric characters as well as periods and underscores. Ada reserved words may be used as nonterminal identifiers. Some examples are shown in Figure 8.

---

pragma
..parameter_list..
_system
-
.

Figure 8. Typical Nonterminals

Unlike token symbols, nonterminals are not explicitly declared; they are implicitly defined by appearing on the left hand side of a rule. However, one nonterminal, the start symbol, has such significance that a provision exists for explicitly declaring it. The start symbol is the most general structure described by the grammar and it may be declared in the declarations section by preceding it with the *%start* keyword. In the absence of a *%start* construct, **Ayacc** uses the left hand side of the first grammar rule as the start symbol.

10

Unlike **Yacc** identifiers, all token and nonterminal names are case insensitive. Thus, *ABC* and *aBc* denote the same grammar symbol in an Ayacc specification file.

4.2.4   Actions

It is often necessary for the parser to take some action when certain syntactic structures are recognized. For example, it may be necessary to generate code as an arithmetic expression is parsed or to update a symbol table as keywords appear in the input. **Ayacc** allows each grammar rule to have associated actions which are executed whenever the rule is recognized by the parser. An action consists of a sequence of Ada statements enclosed in braces and placed after the body of a rule. Some examples follow:

```
N : x y z
  { count := count + 1; } -- Counts the occurrences of N
  ;


A : B  C  D
  { Put_Line("hello"); }  -- Prints Hello whenever A is parsed
  ;
```

The user may need to provide declarations of types and variables used in actions. These declarations can be provided in separate packages used by *YYParse* or they may be provided within the user declarations section at the end of the specification file.

Ayacc uses a pseudo-variable notation to denote the values associated with nonterminal and token symbols. The left hand side of a rule may be set to a specific value by an assignment to the variable *$$*. For example, if YYSType is an integer, the action:

$$A : B \ C \ D \ \{ \ \$\$:= 1; \ \}$$

sets the value of A to 1. To use the values of symbols on the right hand side of the rule, the action may use the pseudo-variables 1..n, where n refers to the nth element of the right hand side of the rule. For example,

$$A : B \ '+' \ C \ \{ \ \$\$:= \$1 + \$3; \ \}$$

sets A to the sum of the values of B and C.

Sometimes it is necessary to execute actions before a rule is fully parsed. **Ayacc** permits actions to appear in the middle of a rule as well at the end. These *nested* actions are assumed to return a value accessible through the usual *$$* notation. A nested action may access values returned by symbols to its left. For example,

```
A : B
   { $$ := $1 + 1; } -- The reference to $$ refers to the value
                     -- of the the action not the value of A
   C
   { x := $2; } -- The reference to $2 is the value of the
                     -- previous action.  A reference to $$ here
                     -- would refer to the value of A.
   ;
```

has the effect of setting x to the value of B plus 1. Nested actions cause **Ayacc** to manufacture a new rule that matches the empty string. For example, the rule

```
   A : B { $$ := 1; } C ;
```

is treated as if it were written

```
   $act : { $$ := 1; }

   A : B $act C;
```

4.2.5   User Declarations

By default, **Ayacc** generates a parameterless procedure, *YYParse*, that must *with* the tokens package and will call the user supplied routines, *YYLex* and *YYError*. If the user desires, the procedure may be incorporated within a package by providing a package declaration in the last section of the specification file. The package declaration is identical to that of an Ada package declaration with the key marker, *##*, substituted where the body of YYParse is to be inserted. See Figure 9 for an example.

The user is responsible for providing the with and use clauses for the Tokens, Parse Table, and Text_IO packages used by the parser. An example of the user declarations section is shown in **Figure 9.** The filename associated with this specification is **example_parser.y**.

---

[3]Note: The '$' in *$act* is used to prevent collision with other nonterminals and is not permitted in a legal nonterminal name.

−−Token Declarations and Rules Section would be up here.
%%

package Example_Parser **is**
    procedure YYParse;
    Syntax_Error : **exception**;
end Example_Parser;

with Example_Parser_Tokens,
      Example_Parser_Shift_Reduce,
      Example_Parser_Goto,
      Text_IO;
use Example_Parser_Tokens,
      Example_Parser_Shift_Reduce,
      Example_Parser_Goto,
      Text_IO;

package body Example_Parser **is**

    function YYLex **return** Token **is**
    begin
     . . .
    end YYLex;

    procedure YYError(S : **in** string) **is**
    begin
     Put_Line(S);
     raise Syntax_Error;
    end YYError;

    −−Miscellaneous declarations and subprograms
    . . .

−−YYParse will be inserted here.

end Example_Parser;

Figure 9. Sample User Declarations Section

13

4.2.6 User Supplied Routines

The user must provide a lexical analyzer to read the input and to send the appropriate tokens along with their values to the parser generated by **Ayacc**. The lexical analyzer must be declared as an Ada function *YYLex* that returns an enumeration type value corresponding to a token in the grammar. The enumeration type is declared in the *tokens* package generated by **Ayacc** for use by the lexical analyzer.

For example, given the input

```
%token a b
{
    subtype YYSType is Integer;
}
%%
S : a ',' b;
%%
```

Ayacc will generate a file, *base_tokens.ada*, containing the following package declaration:

> **package** Base_Tokens **is**
>
>   **subtype** YYSType **is** Integer;
>
>   YYLVal,YYVal : YYSType;
>
>   **type** Token **is** (Error, End_of_Input, A, B, ',');
>
>   Syntax_Error : **exception**;

end Base_Tokens;

The user's corresponding lexical analyzer might look like:

```
with Text_IO, Tokens;
use Text_IO, Tokens;
function YYLex return Token is
    Char : Character;
begin

    if End_of_File then
        return End_of_Input;
    end if;

    loop
        Get(Char);
        case Char is
            when 'A' | 'a' =>
                YYLVal := 1;
                return a;
            when 'B' | 'b' =>
                YYLVal := 2;
                return b;
            when ',' =>
                YYLVal := 0;
                return ',';
            when others =>
                return Error;
        end case;
    end loop;
end YYLex;
```

The tokens *Error* and *End_of_Input* are special predefined tokens that should not be declared by the user. The *End_of_Input* token should be returned by the lexical analyzer after all the lexical input has been read. The *Error* token is used for error recovery and is discussed later. If tokens have values associated with them, the lexical analyzer may return these values by assigning them to the variable, *YYLVal*. In the example above, token 'A' will have a value of 1 and token 'B' will have a value of 2. *YYVal* may be used to access the current value associated with the last symbol recognized by the parser. For example, at the end of the parse *YYVal* contains the value associated with the start symbol. *Although the user can assign values to YYVal,* **it is not recommended since it will overwrite assignments made by previous actions**.

In addition to the lexical analyzer, the user must provide an error reporting procedure, *YYError*, that takes a string, corresponding to an error message, as an argument. *YYError* is automatically called by the parser when it detects a syntax error.

4.3   Advanced Topics

4.3.1   Ambiguity and Conflicts

A grammar is ambiguous if the parser can reach a configuration where it has a choice between a shift or one or more reduce actions or a choice among several reduce actions. There is never a shift/shift conflict. **Ayacc** detects and reports ambiguous grammars and provides two default rules for resolving ambiguity:

1. In a shift/reduce conflict, the shift is chosen.

2. In a reduce/reduce conflict, the reduce involving the earlier rule is chosen.

The verbose file reports ambiguities in the grammar and shows how they have been resolved. For example, consider the infamous *dangling else* grammar:

```
%token IF_TOKEN  COND THEN_TOKEN  ELSE_TOKEN  ID
%%

stat : ID
     | if_statement
     ;

if_statement  : IF_TOKEN COND THEN_TOKEN stat
              | IF_TOKEN COND THEN_TOKEN stat ELSE_TOKEN stat
          ;
%%
```

The grammar causes a shift/reduce conflict reported in state 8 of the verbose file

```
------------------
State  8

Kernel
( 3)  IF_STATEMENT : IF_TOKEN  COND THEN_TOKEN  STAT _
( 4)  IF_STATEMENT : IF_TOKEN  COND THEN_TOKEN  STAT _
  ELSE_TOKEN STAT

Closure
( 3)  IF_STATEMENT : IF_TOKEN  COND THEN_TOKEN  STAT _
( 4)  IF_STATEMENT : IF_TOKEN  COND THEN_TOKEN  STAT _
  ELSE_TOKEN  STAT


*** Conflict on input ELSE_TOKEN
Reduce  3 or Shift  9

  ELSE_TOKEN   shift  9
  default   reduce  4
------------------
```

The verbose entry states that if the parser sees an ELSE_TOKEN in state 8, it has a choice between shifting the token and entering state 9 or reducing by rule 3. In other words, an expression of the form:

```
        if cond1 then if cond2 then statement else statement
```

can be parsed as,

```
        if cond1 then statement else statement
```

or as,

```
        if cond1 then statement
```

The default action taken by **Ayacc** is to shift the ELSE_TOKEN; this has the effect of matching an *else* with the nearest *if* token.

4.3.2 Precedence and Associativity

Rewriting a grammar to eliminate ambiguities will often result in an unnatural looking grammar and a less efficient parser. For example, consider the original calculator grammar :

```
  expression : term
             | expression '+' term
             | expression '-' term
             ;

  term       : factor
             | term '*' factor
             | term '/' factor
             ;

  factor     : IDENTIFIER
             | NUMBER
             | '(' expression ')'
             ;
```

In the above grammar, the productions:

```
  expression : expression '+' term
             | expression '-' term
```

exist only to enforce the precedence of multiplicative operators over additive operators. For example, the input:

```
a * b + c
```

is parsed as,

```
Factor * b + c
Term * b + c
Term * Factor + c
Term + c
Expression + c
Expression + Factor
Expression + Term
Expression
```

Similarly,

```
a + b * c
```

is parsed as,

```
a + b * c
Factor + b * c
Term + b * c
Expression + b * c
Expression + Term * c
Expression + Term * Factor
Expression + Term
Expression
```

Ideally we would prefer to represent the grammar using the more natural but ambiguous specification:

```
expression : expression '+' expression
           | expression '-' expression
           | expression '*' expression
           | expression '/' expression
           | '(' expression ')'
           | IDENTIFIER
           | NUMBER
           ;
```

Note that the grammar above also contains fewer productions and may result in a faster parser. However, the grammar is ambiguous because inputs of the form

```
a op1 b op2 c
```

can be parsed as,

```
(a op1 b) op2 c
```

or as,

```
a op1 (b op2 c).
```

Moreover, the default resolution scheme used by **Ayacc** will result in an incorrect parser.

Fortunately, **Ayacc** provides a scheme to allow the user to assign precedence and associativity to tokens and productions when the default disambiguating rules are inadequate. The notion of precedence and associativity is particularly useful for grammars involving arithmetic expressions as in the example above. For example, the shift-reduce conflicts shown below:

```
( 1)  EXPRESSION : EXPRESSION _ '+' EXPRESSION
( 3)  EXPRESSION : EXPRESSION '*' EXPRESSION _

*** Conflict on input '+'
Reduce  3 or Shift  6

( 1)  EXPRESSION : EXPRESSION '+' EXPRESSION _
( 3)  EXPRESSION : EXPRESSION _ '*' EXPRESSION

*** Conflict on input '*'
Reduce  1 or Shift  8
```

can be resolved by giving priority to the action involving the token with the highest precedence. Since '*' has higher precedence than '+', the first conflict will be resolved in favor of a reduce and the second in favor of a shift.

In addition, shift-reduce conflicts also exist for

```
( 1)   EXPRESSION : EXPRESSION _ '+' EXPRESSION
( 1)   EXPRESSION : EXPRESSION '+' EXPRESSION _

*** Conflict on input '+'
Reduce  1 or Shift  6

( 3)   EXPRESSION : EXPRESSION _ '*' EXPRESSION
( 3)   EXPRESSION : EXPRESSION '*' EXPRESSION _

*** Conflict on input '*'
Reduce  3 or Shift  8
```

In these cases, precedence is of no help since the conflicts involve the same tokens. However, conflicts involving tokens with the same precedence may be resolved using associativity rules. Left associative operators imply a reduce. Conversely, right associative operators imply a shift. In the above example, both '*' and '+' are left associative, and therefore the conflicts should be resolved in favor of the reduce action.

Precedence and associativity is assigned to tokens in the declarations section using the keywords, *%left*, *%right*, and *%nonassoc*, followed by a list of tokens. *%nonassoc* denotes nonassociative tokens such as the Ada relational operators. Precedence declarations are listed in order of increasing precedence with tokens on the same line having identical precedence and associativity. For example, an Ayacc specification of an arithmetic expression grammar might look like:

```
%token number    -- No prec/assoc

%right '='
%left  '+' '-'
%left  '*' '/'
%left  DUMMY     -- This token is not used by
                 -- the lexical analyzer


exp : exp '=' exp
    | exp '+' exp
    | exp '-' exp
    | exp '*' exp
    | exp '/' exp
    | '-' exp   %prec dummy  -- changes the default precedence of
                             -- this rule to that of token dummy;
    | NUMBER
    ;
%%
```

The precedence and associativity rules used by **Ayacc** to resolve conflicts are summarized below :

1. A grammar rule inherits the precedence and associativity of the last token or literal in its body.

2. If either the grammar rule or the token has no precedence and associativity, **Ayacc** uses its default scheme for resolving conflicts. Reduce/reduce conflicts are always resolved according to the rule that appears first in the specification file.

3. If there is a shift/reduce conflict and the rule and token have precedence associated with them, the conflict is resolved in favor of the rule/token with the highest precedence. If the precedences are equal, left associativity implies a reduce, right associativity implies a shift, and nonassociativity implies an error.

4. The precedence of a grammar rule may be explicitly set by the keyword *%prec* and a trailing token that specifies that the rule inherits the precedence of the token. The example above uses a dummy token to give unary minus the highest precedence for arithmetic operators.

4.3.3   Error Recovery

By default, the Ayacc generated parser calls *YYError* and aborts as soon as a syntax error is encountered. Although this behavior is adequate for some applications, it is often useful for the parser to continue parsing so that further syntax errors can be detected. To

accomplish this, **Ayacc** provides a simple method to indicate where error recovery should be attempted. The parser makes no attempt to repair incorrect input. Instead, it attempts to reduce the phrase containing the syntax error to a user specified nonterminal. After the reduction, parsing resumes as usual.

4.3.4 Error Productions

Certain user-specified nonterminals form the basis of error recovery. These nonterminals are specified by adding rules to the grammar of the form

$A : \alpha \; error \; \beta$

Where α and β are possibly empty strings of terminals and nonterminals. The token *Error* is predefined and should not be used for any other purposes. As with other rules, an action may be associated with any error production.

4.3.5 The Error Recovery Algorithm

When a syntax error is detected, the parser calls procedure *YYError* with the message *Syntax Error* and then attempts to recover. The error recovery process can be broken down into three steps.

1. The parser pops the stack zero or more times, until it finds the top most state where a shift on *error* is legal. This state will be associated with some item of the form

   $A : \alpha \; \_ \; error \; \beta$

   If the stack contains no such state, the parser raises the exception *Syntax_Error* after popping the entire stack.

2. Next, the parser executes a shift on *Error* pushing the state associated with the item

   $A : \alpha \; error \; \_ \; \beta$

   onto the stack.

3. The parser then attempts to resume parsing with the current lookahead token set to the token that caused the error. All new tokens that would cause an error are quietly discarded until one token is shifted. If the parser discards the *End_of_Input* token it will raise a *Syntax_Error* exception; otherwise, parsing resumes normally after the first token is shifted.

   If a new syntax error is detected before three valid shifts, error recovery is reinitiated without reporting a syntax error. This prevents an avalanche of error messages caused by incomplete error recovery.

4.3.6   An Example of Error Recovery

A rule designed to recover from syntax errors in Ada statements might have the form:

statement : error ;

This identifies *statement* as a location where errors are expected and will cause the parser to attempt to skip over statements containing syntax errors.

Now suppose that the parser was parsing the following statement:

$$i := 5 + +j - f(1) + 1;$$

When the parser detects the syntax error, it will have already pushed a sequence of states on top of the stack corresponding to the input up to the second plus sign. These states would be popped one at a time until a state that had an action shift on *Error* was encountered. Since a statement was being parsed, this state would be associated with the item:

statement : _ error

The parser will now perform a shift on input *Error* and enter a state associated with the item:

statement : error _

The remaining input would be:

+ j - f(1) + 1;

Now the parser would attempt to resume parsing, discarding any tokens that would cause an error, until a shift action is executed. Since the parser will reduce the *Error* token to a statement, the only token that would allow a shift to occur would be one that could follow a statement. The plus sign would be discarded because it could not follow a statement, but because identifier j could follow a statement, the parser would shift the identifier j and resume parsing as if j was at the start of a new statement. The minus sign would cause a new error since no statement can begin with:

j -

Although error recovery will occur again, a new syntax error would not be reported because three tokens have not been successfully shifted. After popping the stack and shifting the *Error* token, the parser would again enter the state associated with:

statement : error _

and the remaining input would be:

- f(1) + 1;

The parser would discard tokens again until it found one that could follow a statement. Parsing would resume with the identifier f, since f could be the start of a new statement. The parser would shift f, the left parenthesis, the integer, and the right parenthesis. When it reads the plus sign it would think it encountered a new syntax error. This time it would report a syntax error since three tokens have been successfully shifted. After the states are popped and the *Error* token has been shifted the parser would continue discarding tokens up to and including the semicolon. If the input following the semicolon is a legal statement, parsing would resume normally.

4.3.7 More Control over Error Recovery

In the previous example we saw how the error recovery scheme might cause the parser to incorrectly resume parsing while it was still in the phrase that caused the error. It is possible to exert more control over error recovery by placing tokens after the *Error* token. For example, if a syntax error was detected while parsing a statement, the production:

statement : error ';'

would cause the parser to discard tokens until a semicolon is read because after simulating the shift on the *Error* token, the parser would enter a state associated with the item:

statement : error _ ';'

Since the only legal action is shift on semicolon, all tokens would be discarded until a semicolon was encountered. This would prevent the false starts of the previous example.

Another way of obtaining more control over error recovery is to place tokens before the Error token. Consider the following rules taken from an Ada grammar.

```
loop_statement : ..iteration_scheme..
    LOOP_TOKEN
    sequence_of_statements
    END_TOKEN
    ';'
;

..iteration_scheme.. :   -- empty
    | FOR_TOKEN     loop_param_spec
    | WHILE_TOKEN   condition
    | FOR_TOKEN     error
    | WHILE_TOKEN   error
    ;
```

Here, *..iteration_scheme..* would not be reduced if an error was detected unless either a FOR_TOKEN or a WHILE_TOKEN was seen on the input. Given the following production:

..iteration_scheme.. : error ;

it is possible that *error* could be reduced to *..iteration_scheme..* even though a syntax error was detected when the parser was in a state where it could expect a string derived from *..iteration_scheme...* If this happened, error recovery would discard tokens until a *LOOP_TOKEN* (the only token that can follow an *..iteration_scheme..*), was encountered. This is clearly unacceptable if the next string was really a statement.

Sometimes it is useful for the parser to report errors before correctly shifting three tokens. The procedure *YYErrOK* will force the parser to believe it has fully recovered from any syntax error causing it to report errors in the following vthree tokens. For example, an interactive application might have the rules

```
      lists : lists list END_OF_LINE
                  {
                      -- print the value of $1
                  }
          | list END_OF_LINE
                  {
                      -- print the value of $1
                  }
          | error END_OF_LINE
                  {
                      YYErrOK;
                      Put_Line("Reenter previous line");
                  }
          ;
```

The call to YYErrOK will tell the parser it has correctly shifted three tokens causing the next syntax error to be reported. If the call to YYErrOK was not in the action, a syntax error in the next three tokens would not be reported.

The user could provide an action in an error production that decides what tokens to discard. Here, the old lookahead token must be cleared. To accomplish this, the parser provides the procedure *YYClearIn* which will force the parser to read the next token.

4.3.8   Automatic Error Recovery

See Appendix C for information on how to generate parsers that will attempt to automatically recover from syntax errors.

5   Error Messages

This section describes the error messages which may be displayed by
Ayacc . The error messages are divided into three categories: *Internal, Fatal, and Non Fatal.* The error message text is presented in **Bold** type with variable items in *Italics*.

5.1   Internal Error Messages

The following error message will always be displayed when an error is detected within the tool, and may be preceded by additional descriptive messages.

1. **Unexpected Error, Terminating...**

5.2   Fatal Error Messages

The following error messages are produced when a fatal error condition is detected which can be resolved by the tool user. Where appropriate, the error message will be followed by the associated file specification, and the context and column location of the error.

1. **Can't Open** *Source_Filename***.**

2. **Too Many Parameters.**

An excessive number of parameters were specified on the command line.

5.3   Non Fatal Error Messages

The following error messages display conditions which may be of interest to the tool user. However, the displayed condition will not cause the tool to terminate execution. Where appropriate, the error message will be followed by the associated file specification, and the context and column location of the error.

1. **Attempt to Define Terminal as Start_Symbol.**

2. **Attempt to Redefine Precedence.**

3. **Context Clause Specifications May Not Appear After Ada Declarations.**

4. **Expecting a Colon after the Lefthand Side of the Rule.**

5. **Expecting Identifier.**

6. **Expecting Next Section.**

7. **Expecting Package Name.**

8. **Expecting a Semicolon.**

9. **Expecting a Terminal after %prec.**

10. **Expecting Token Declaration.**

11. **Illegal Context Clause Specification.**

12. **Illegal Filename.**

13. **Illegal Symbol Following $.**

14. **Illegal Symbol as Token.**

15. **Illegal Token.**

16. **Illegal Token Following %prec.**

17. **Illegal use of $*Integer*.**

18. *Integer* **Shift/Reduce Conflicts.** In a shift/reduce conflict, the shift is chosen.

19. *Integer* **Reduce/Reduce Conflicts.** In a reduce/reduce conflict, the reduce involving the earlier rule is chosen.

20. **Nonterminal** *Symbol Name* **Does Not Appear on the Left Hand Side of Any Rule.**

21. **Nonterminal** *Symbol Name* **Does Not Derive a Terminal String.**

22. **%prec Cannot be Preceded by an Action.**

23. **Syntax Error detected in** *File Spec.*

24. **The Start Symbol has been Defined Already.**

25. **Terminals Cannot be on the Lefthand Side of a Rule.**

26. **Terminal Following %prec has no Precedence.**

27. **Unexpected End of File before First '%%'.** The **Ayacc** input specification should consist of three (four if *Error_Recovery* is used) parts delimited by **%%**.

28. **Unexpected Symbol.**

29. **Use Verbose Option.**

6   Known Deficiencies

Ayacc has no known deficiencies.

7 Release Notes

Ayacc was designed and developed by David Taback and Deepak Tolani at UC Irvine in support of the *Arcadia Research Project.*

Enhancements made by Ronald J. Schmalz are summarized below.

1. Addition of *%with* and *%use* directives which permits automatic insertion of a context clause information on the generated token package.

2. Unique unit name generation; the units created by **Ayacc** are now created as a function of the input file specification. This allows the user to have multiple **Ayacc** generated parsers within the same library.

Yidong Chen of the Arcadia Project at the University of Massachusetts in Amherst added the advanced automatic error recovery described in appendix C.

A A Detailed Example

Below is a full Ayacc specification for an integer desk calculator and a **Lex** specification of the corresponding lexical analyzer. The Ayacc specification file is named *calculator.y*. The calculator has 26 variables labeled 'A' through 'Z' and supports most of the Ada integer arithmetic operators. The example illustrates most of the advanced features of **Ayacc** including precedence, associativity, error recovery, and interfacing to **Lex**.

−− The Ayacc specification file −−

```
%token '(' ')' NUMBER IDENTIFIER NEW_LINE

%right '='
%left '+' '-'
%left '*' '/'
%right DUMMY
%nonassoc EXP

{

type key_type is (Cval, Ival, Empty);

type YYSType (Key : Key_Type := Empty) is
   record
      case Key is
         when Cval =>
            Register : Character;
         when Ival =>
            Value : Integer;
         when Empty =>
            null;
      end case;
   end record;
}

%%

statements : statements statement
           |
           ;

statement : expr NEW_LINE
              { Put_Line(Integer'Image($1.value)); }
```

```
             | error NEW_LINE
                 { Put_Line("Try again");
                 YYErrOK;
                 }
             statement
             ;

expr     : IDENTIFIER '=' expr
             { registers($1.register) := $3.value;
             $$ := (key => ival, value => $3.value); }
         | expr '+' expr
             { $$ := (key => ival, value => $1.value + $3.value); }
         | expr '-' expr
             { $$ := (key => ival, value => $1.value - $3.value); }
         | expr '*' expr
             { $$ := (key => ival, value => $1.value * $3.value); }
         | expr '/' expr
             { $$ := (key => ival, value => $1.value / $3.value); }
         | expr EXP expr
             { $$ := (key => ival,
             value => Integer(float($1.value) ** $3.value)); }
         | '-' expr %prec DUMMY
             { $$ := (key => ival, value => - $2.value ); }
         | '(' expr ')'
             { $$ := (key => ival, value => $2.value); }
         | NUMBER
             { $$ := (key => ival, value => $1.value) ; }
         | IDENTIFIER
             { $$ := (key => ival, value => registers($1.register)); }
         ;

%%

package Calculator is
    procedure YYParse;
end Calculator;

with Calculator_Tokens,
        Calculator_Shift_Reduce,
        Calculator_Goto,
        Text_IO;
use Calculator_Tokens,
        Calculator_Shift_Reduce,
```

```ada
        Calculator_Goto,
        Text_IO;

package body Calculator is

    Registers : array('A'..'Z') of Integer;

    procedure YYError(Text : in String) is
    begin
        Put_Line(Text);
    end;
##

end Calculator;
```

```
                        /* The Lex specification file */

%{
#include "calculator.h"
static char reg;
static int  value;
%}

%%
[A-Z]        { reg = yytext[0]; return IDENTIFIER; }
[a-z]        { reg = yytext[0] - 'a' + 'A'; return IDENTIFIER; }
[0-9]+       { value = atoi(yytext); return NUMBER; }
"**"          { return EXP; }
[()+/*=-]    { return yytext[0]; }
\n           { return NEW_LINE; }
[\t ]+       {}
%%

yywrap()
{
    return 1;
}

get_token()
{
    return yylex();
}

get_register()
{
    return reg;
}

get_value()
{
    return value;
}
```

−−A modified version of the C Lex package −−

with Calculator Tokens; **use** Calculator Tokens;
package Calculator C Lex **is**
    function YYLex **return** Token;
end Calculator C Lex;

package body Calculator C Lex **is**

    function GET TOKEN **return** Integer;

    pragma INTERFACE(C, GET TOKEN);

    −−These four declarations have been added

    function get register **return** Character;
    function get value **return** Integer;
    pragma interface(c, get register);
    pragma interface(c, get value);

    type Table **is array**(0..255) **of** token;

    Literals : **constant** Table := Table'(0 => END OF INPUT,
                                        40 => '(',
                                        41 => ')',
                                        61 => '=',
                                        43 => '+',
                                        45 => '-',
                                        42 => '*',
                                        47 => '/',
                                        others => ERROR);
−−Continued on next page ...

34

```
function YYLex return TOKEN is
    X : Integer;
begin
    X := GET_TOKEN;
    if X > 255 then

    -- This case statement has been added to assign appropriate
    -- values to YYLVal.
    case TOKEN'VAL(X-256) is
        when number =>
            YYLVal := (key => ival, value => get_value);
        when identifier =>
            YYLVal := (key => cval, register => get_register);
        when others => null;
        end case;

        return TOKEN'VAL(X-256);
    else
        return LITERALS(X);
    end if;
end YYLex;
end Calculator_C_Lex;
```

B   Using the Verbose and Debug Options

We will introduce the concept of an **item** to describe the output file created by the *Verbose* option. An item consists of a rule with an underscore at some position on the right side. The underscore denotes the amount of input seen by the parser. For example, an item of the form

    X : A B _ C

shows that the parser is examining input corresponding to the rule *X : A B C*. The underscore states that the parser has has already seen *A B* and is expecting input that will match *C*.

Items provide a means of finitely representing the possible legal inputs to the parser. Each state contains a set of items corresponding to configurations indistinguishable to the parser. For technical reasons, the set of items associated with a state fall into two categories termed the kernel and closure. Users familiar with LR parsers may be interested in both the kernel and closure items, however the typical user need only be concerned with items in the closure.

The verbose file contains a list of the states along with their corresponding item sets and parse actions. A sample verbose file for the simple grammar,

```
%token id
%%
E : E '+' T
  | T
  ;

T : T '*' id
  | id
  ;
```

is shown below.

```
------------------
State  0

Kernel
( 0)  $accept : _ E END_OF_INPUT

Closure
( 0)  $accept : _ E END_OF_INPUT
( 1)  E : _ E '+' T
( 2)  E : _ T
( 3)  T : _ T '*' ID
( 4)  T : _ ID


        T    goto   2
        E    goto   1
        ID   shift  3
        default    error
------------------

State  1

Kernel
( 0)  $accept : E _ END_OF_INPUT
( 1)  E : E _ '+' T

Closure
( 0)  $accept : E _ END_OF_INPUT
( 1)  E : E _ '+' T


        END_OF_INPUT    accept
        '+'      shift  5
        default    error
------------------
```

```
State  2

Kernel
( 2)  E : T _
( 3)  T : T _ '*' ID

Closure
( 2)  E : T _
( 3)  T : T _ '*' ID


       '*'    shift  6
      default    reduce  2
------------------

State  3

Kernel
( 4)  T : ID _

Closure
( 4)  T : ID _


      default    reduce  4
------------------

State  4

Kernel
( 0)  $accept : E END_OF_INPUT _

Closure
( 0)  $accept : E END_OF_INPUT _


      default    error
------------------
```

```
State  5

Kernel
( 1)  E : E '+' _ T

Closure
( 1)  E : E '+' _ T
( 3)  T : _ T '*' ID
( 4)  T : _ ID


      T    goto   7
      ID   shift  3
      default     error
------------------

State  6

Kernel
( 3)  T : T '*' _ ID

Closure
( 3)  T : T '*' _ ID


  ID  shift  8
  default     error
------------------
```

39

```
State  7

Kernel
( 1)  E : E '+' T _
( 3)  T : T _ '*' ID

Closure
( 1)  E : E '+' T _
( 3)  T : T _ '*' ID


      '*'     shift   6
      default          reduce  1
------------------

State  8

Kernel
( 3)  T : T '*' ID _

Closure
( 3)  T : T '*' ID _


      default       reduce  3
```

Using the verbose file it is possible to trace how the parser will process a string of tokens. For example the string 'ID * ID + ID' would be treated as follows:

```
State Stack       Input
   0         ID * ID + ID END_OF_INPUT
```

The verbose entry for state 0 shows that on a lookahead token of ID the action is a shift and the new state is 3. Thus, the new configuration of the parser becomes

```
State Stack     Input
   3
   0           * ID + ID END_OF_INPUT
```

For state 3, there is no explicit action associated with the lookahead token and therefore the default action is consulted. The default action associated with state 3 is a reduction

by rule 4) T : ID. Recall that a reduction consists of two steps. First state 3 is popped leaving state 0 on top of the stack. Next the parser determines a new state by consulting the current top of the stack and the right hand side of the rule. This new state is signified by a *goto* entry in the current state. The verbose entry for state 0 and symbol T is a goto to state 2 producing the new configuration:

```
State Stack     Input
    2
    0              * ID + ID END_OF_INPUT
```

The remaining configurations in the parse are shown below:

```
Shift 6

State Stack     Input
    6
    2
    0               ID + ID END_OF_INPUT

Shift 8

State Stack     Input
    8
    6
    2
    0               + ID END_OF_INPUT

Default Reduce 3) T : T '*' ID
Pop 8 6 2
Goto state 2

State Stack     Input
    2
    0               + ID END_OF_INPUT

Default Reduce 2) E : T
Pop 2
Goto state 1

State Stack     Input
    1
    0               + ID END_OF_INPUT
```

```
Shift 5

State Stack    Input
    5
    1
    0            ID END_OF_INPUT

Shift 3

State Stack    Input
    3
    5
    1
    0            END_OF_INPUT

Default reduce 4) T : ID
Pop  3
Goto 7

State Stack    Input
    7
    5
    1
    0            END_OF_INPUT

Default reduce 1) E : E +  T
Pop  7 3 5
Goto 1

State Stack    Input
    1
    0            END_OF_INPUT

Accept END_OF_INPUT
Parse completes successfully
```

C   Automatic Error Recovery

If the *Error_Recovery* command line parameter is set to *On* then **Ayacc** will generate an extension for automated syntax error correction. Note that the lexical analyzer must contain additional functions which give the line number and column for each token. This can be done by giving the *-E* option to *Aflex*. **Ayacc** generates an additional file named *base*_error_report.a and the user may specify another optional section in the user specification file. Here is a description of that section.

C.1   User Error-correction Messages section

This is a section for the user to supply routines if he/she wishes to control the reporting of correction messages. Ayacc-extension supports automatic correction of some syntactic errors, as explained later. Along with each correction, a message is printed. In addition to the default message printed, the user is able to report a message of his/her own. Here is the spec of the message-reporting procedure which is generated:

```
procedure Report_Continuable_Error(Line_Number : in Natural;
                       Offset  : in Natural;
                       Finish  : in Natural;
                       Message : in String;
                       Error   : in Boolean);
```

Line_Number is the line at which the error occurred; Offset is the index into the line of the start of the error; Finish is the index into the line of the end of the error; Message is a string describing the error. Error is true for all genuine syntax errors, when it is false it indicates a syntax warning.

This section resembles the Token and Stack Element section syntactically in that its entries begin with the '%' character. There are the options available in this section:

1. %with ....;
   Generates a line 'With ....;' in the error report file (package body)

2. %use ....;
   Generates a line 'Use ....;' in the error report file (package body)

3. %initialize_error_report
   Following this line there should be the body of a no-argument procedure which will be called once at the beginning of yyparse. It can be used to initialize any data structures used by the user's error report.

4. %terminate_error_report

Following this line there should be the body of a no-argument procedure which will be called once at the end of yyparse. It can be used to close any data ports utilized by the user's error reporting mechanism.

5. %report_error

Following this line there should be the body of the procedure Report_Continuable_Error described above (i.e. with those arguments).

The %with and %use lines, if present, should precede the others. An example is shown below.

```
-- The other declarations would go up here.
%%

%with text_io;

%initialize_error_report
begin
  text_io.put_line("Initializing Error Report...");
end;

%terminate_error_report
begin
  text_io.put_line("Finishing Error Report...");
end;

%report_error
begin
  text_io.put_line("Error at line" & natural'image(line_number)
                  & ": " & message);
end;
```

C.2   Change in running the Parser

If *Error_Recovery* is set to *On*, the generated parser has more power in error recovery. Here is the additional power:

C.2.1   Output

A run of YYParse will produce a listing file which records the parsed lines of the input text and indicates where errors occurred. If the specification file is named *base*.y, then the listing file will be named *base*.lis. The rest of the output of the program is dependent on

the action routines; as arbitrary code these are free to perform output operations.

C.2.2 Error Recovery

When the Ayacc generated parser encounters a syntax error, it tries to correct it. To correct the error it will try either to insert a legal token before the error, to change the error to a legal token, or to delete the error from the token stream. When a correction can be made, a message is printed describing the correction. Even when it is possible to correct an error, the correction will only be syntactic. For example, say the user is parsing a Pascal program and the input is missing a semicolon at the end of a statement. The parser may be able to detect that and insert the semicolon. In all likelihood, the semicolon has no semantic value, and the grammar rule in which it appears would not reference it in its semantic action. However, consider a case where the parser decides to insert an identifier token in order to correct a syntax error. It is very likely that an action routine for the grammar rule using the identifier would reference it semantically, to find out what characters are in the string making up the identifier. However, the parser has no way of knowing which identifier would be best to insert at the point of insertion, so it cannot provide this information. The action routine would therefore probably make a mistake, since it relies on the semantic information being present. If executing an action routine following a syntax error raises an exception, YYParse handles the exception and stops performing the code in action routines for the remainder of the parse. Even before action routines are stopped, any actions following a syntax error should not be trusted. To emphasize the abortive nature of a run of YYParse with syntax error, the parser raises the exception Syntax_Error at the end of an input with syntax errors, even if all have been corrected. Here is a sample listing file from a two line calculator program.

```
----------------------------------------------------------------------


1    1 * 2 * 3  * 2 3
Error                 ^
token deleted

2    4 + 28 / 14 + 2

Ayacc.YYParse : 1 syntax error found.


----------------------------------------------------------------------
```

In the listing file, non-blank lines of text are listed with their line number at the left. The last token of the first line is a syntactic error; evidently an infix notation was specified in this grammar. Errors are indicated by a line that begins "Error" and then contains a

caret character "^" underneath the start of the erroneous token. YYParse can continue its parse if it deletes this token from the token stream.

C.2.3   User Error Messages

As previously mentioned, in addition to the default messages produced during error recovery, the user, in the last section of the specification file, can provide routines for reporting error messages in his/her own way. Also, the user is given an interface to those routines which he/she can use even when there is no syntactic error as defined by the grammar rules. This is useful in cases where there is input which parses properly but which "really" represents a syntax error in the input file. For this the following package is provided:

```
package user_defined_errors is
      procedure parser_error(Message : in String);
      procedure parser_warning(Message : in String);
end user_defined_errors;
```

This package is automatically visible to code in the user's action routines. Calling user_defined_errors.parser_error will increase the count of total syntax errors and call the procedure report_continuable_error, from the "User Error-correction Messages" section. The Message argument to parser_error becomes the Message argument to report_continuable_error, and the Error argument to report_continuable_error is given the value True. The rest of the arguments to report_continuable_error are taken from context. User_defined_errors.parser_warning is similar, except that the Error argument in the generated call to report_continuable_error is False. It also increments a counter of syntax warnings rather than syntax errors. There is an exception Syntax_Warning like Syntax_Error mentioned above which will be raised if during the parse there is no syntax errors but there are warnings. Note that the procedures from package user_defined_errors can be called even if the procedure report_continuable_error is not defined by the user; in that case there will be no reporting of the Message, but the incrementing of the proper counter will still take place.

D Differences between Yacc and Ayacc

Ayacc was modeled after **Yacc** and adheres to most of the conventions and features of its C analogue. Most of the differences between the two programs are minor, but some differences will make it difficult to convert Yacc specifications into **Ayacc** counterparts. Some of the most important differences are listed below:

1. Ayacc identifiers are case insensitive.

2. **Ayacc** does not provide a feature analogous to the %union and %type constructs of **Yacc**. At some sacrifice of convenience, similar functionality may be obtained by declaring YYSType as a variant record.

3. **Ayacc** requires the user to define YYSType.

4. There are no default actions in **Ayacc** .

5. **Ayacc** does not support the old and discouraged features of **Yacc**. In particular, %binary and %term are not allowed, actions cannot be specified using the ={ delimiter and all rules must end in a semicolon. In addition, **Ayacc** uses braces rather than, %{ and %}, to denote declarations that should be written to the tokens package.

6. In **Ayacc** the tokens are an enumeration type rather than integers.

7. **Ayacc** does not permit escape characters to be entered as literals.

8. **Yacc** generates a file containing the parser and parse tables and another file containing the macro definitions of the tokens declaration. **Ayacc** generates four separate files corresponding to the parser, the two parse tables, and the tokens package. In addition, **Ayacc** can generate the parser as a procedure or as a package depending on the user's specification file.

E   Ayacc Specification File Guidelines

The key to preparing efficient and readable Ayacc specification files is to make each part of the specification distinguishable from the rest. This appendix is provided to give the new **Ayacc** user suggestions on how to prepare specification files which are:

1. Efficient.

2. Easy to read.

3. Easy to modify.

Specification File Format

1. Group grammar rules with a common left hand side together, and line up the right hand sides with the ':', '|', and ';'. This enhances readability and makes adding new rules or actions easier.

2. Use upper case for *Tokens* returned by the lexical analyzer and lower case for *Non-Terminals* of the grammar. This makes the distinction between terminals and non-terminals very clear.

3. Place grammar rules and their corresponding actions on separate lines. This enhances readability and maintainability.

The rule fragment shown in **Figure 10** defines the rules/actions for Ada identifier lists and is intended to exemplify the guidelines listed above.

```
identifier_list  :  IDENTIFIER
                    {
                      $$ := (Tag  => Identifier_List,
                             List => Empty_List);
                      Insert (Item => $1,
                              Into => $$.List);
                    }
                 |  identifier_list ',' IDENTIFIER
                    {
                      $$ := $1;
                      Insert (Item => $3,
                              Into => $$.List);
                    }
                 ;
```

Figure 10. Identifier List Grammar and Actions

F   How the Parser Works

The parser generated by **Ayacc** belongs to the class of parsers technically known as LALR(1). Although the use of **Ayacc** does not require extensive knowledge of LALR grammars, an intuitive understanding of how the parser works will help the user to resolve ambiguities in the grammar specification and to write more efficient parsers.

The parser generated by **Ayacc** can be seen as a finite state machine with a stack of states. The current state is always on the top of the stack; initially the stack contains state 0. At any given moment during the parse, the parser uses its current state and the value of the next lookahead token (obtained by calling *YYLex*) to determine its next action. Based on the current state and lookahead token, the parser performs one of four actions:

ACCEPT

   The parser accepts the grammar, completing the parse.

ERROR

   The parser detects a syntax error and calls *YYError*. If no error recovery
   is specified the parser aborts.

SHIFT

   The parser uses the current state and the lookahead token to choose a new
   state that is pushed onto the stack. The lookahead token is advanced
   to the next token in the input.

REDUCE

   Reduce actions occur when the parser recognizes the right hand side of a
   rule. In a reduce action, the parser pops a state for every symbol on the
   right hand side of the rule. The parser then uses the current state
   uncovered by the succession of pops and the nonterminal on the left hand
   side of the rule to choose a new state that is pushed onto the stack. The
   lookahead token is unaffected by a reduce action.

G   Porting Ayacc to Other Systems

Installing Ayacc

Ayacc was developed using the Verdix Ada compiler (version v04.06) running under UNIX 4.2 BSD and enhanced using the Dec Ada Compiler (version 1.2-15) running under VAX/VMS 4.3. If you are using a different system, you may have to make a minor change to the **Ayacc** source.

Current **Ayacc** development is done using the SunAda (version 1.1i) on Sun workstations. Ports done by users for many other compilers are included in the distribution.

Reading arguments from the command line

The Verdix compiler uses a library package *U_Env* to provide C-like facilities for reading arguments from the command line. If your machine uses a different mechanism for passing parameters to the program, you will have to modify the subunit *Command_Line_Interface.Read_Command_Line* from the Command_Line_Interface package.