ParsesraP: Using one grammar to specify both input and output

Steven J. Beaty
Cray Computer Corporation
1110 Bayfield Drive Colorado Springs, Colorado 80906
Vox: (719) 540-4129 FAX: (719) 540-4028
beaty@craycos.com
beaty@longs.lance.colostate.edu
http://www.craycos.com/~beaty

Abstract

In the past, there have been many different descriptive languages used to specify the syntax of information read into programs. These types of descriptions have not been used to produce the output of programs. Output has used ad hoc methods. This paper presents a method to use a single specification to produce a consumer of input, and a producer of output.

1 The Idea

There are many different types of specification languages that have been developed to detail the input to a program. Yacc [Joh79], PCCTS [PDC] are examples of languages that specify the exact form source information can be in and that construct parsers for that information. Interestingly, these specifications have not been used to produce the output from programs that supply input to the generated parser. There is a wealth of information present in the input specification that can be reused in the production of acceptable output. If a file is produced such that it obeys the rules specified by the input language, it is guaranteed to be readable by any program using that specification. ParsesraP's core idea is to use a single language specification for both the creation and consumption of data. Historically, output data have be produced ad hoc; i.e., using the output operators built into the programming language. The ad hoc approach does nothing to help the programmer create acceptable data nor does it ensure the output is readable by another program. What ParsesraP's method of producing output does is, in essence, do an "anti-parse" to produce valid output files. The palindrome ParsesraP denotes that one specification is used for both parsing, and its opposite.

With both the input and output in the same description, the data structures filled by input actions can be easily mirrored in the output actions. Filters that perform some operation on a file, leaving it in the same syntax, become trivial to specify as both the input and output are correctly described in a single ParsesraP file. It is also easy to use the input half of one ParsesraP description for the input to a program and the output half from a different description. A good example of when this would be useful is when designing an assembler that accepts code from multiple front ends. If the input format of the assembler is described in a ParsesraP form, the various front ends could use this description to ensure valid assembler output.

The interchangeability benefits of using a single specification file are great. When an interface to any program is defined using ParsesraP, an empty ParsesraP description (i.e., one without any actions) could be distributed to those wanting to interface with that program. Anyone using that description would be guaranteed to produce legal input source. If the original programmer provided a ParsesraP description for the output of a program, anyone could also parse the output easily. Public interfaces to programs then become very easy to interface with, a bonus to both those who write and those who use the programs.

2 The Details

To be able to describe both input and output in one file, the specification should contain all the language elements in one place. Historically, the parsing process has been broken into two parts (which usually have two mostly independent

specifications): the lexical analysis phase and the parsing phase. As a combined input and output method must know all the language elements, including the "syntactic sugar", this information should be available in the specification. Consider these grammar rules for a simple addition operation:

```
expr : number "+" number

number : '[0-9]+'
```

There are two non-terminals in the first rule, both named `number` and one terminal `"+"`. The generation of a parser to parse this language is straightforward [ASU86]. To produce output accepted by this language, we need not specify the terminal as there is no choice of what to produce. To produce the two `number`s, we need to check to see if the output strings are valid by comparing them against the regular expression (r.e.) specified (`[0-9]+`.) Therefore, on output, lexical analysis is done by r.e. checking and syntactical checking can be done by performing an "anti-parse" trace. This is very similar to input parsing in that a current state is maintained as the output is produced, instead of as input is consumed. Next states are known a priori and are checked during the production of output.

There are currently two methods to generate the output from a ParsesraP description. The first is simply to let the anti-parse decide, whenever possible, what to produce. For terminals, there is no decision to be made. For non-terminals, the default is currently to call the first pattern that appears in the non-terminal rule set.

If this is not the desired action, the user can gain direct control over the production of output. ParsesraP produces functions for each rule/action, and the user can call these directly. There are two reasons a user might wish to do this:

- the default action is not the desired one, or
- the user wishes to build the output from a number of different points in the program.

In the second case, ParsesraP diverges from traditional parser generators, which usually have only one entry point (e.g., `yyparse` for yacc.) This allows more flexibility in the production and consumption of data, as either can occur anywhere in a program, and all the data does not have to be produced or consumed at one point. This is similar to traditional ad hoc methods of I/O, but adds the important aspect of correctness checking.

The tables used to produce the input parser can be reused to check the transitions from one output state to another. This works as follows: place the start state on the anti-parse trace. As each output routine is called, check to see if the routine's state is derivable from the current state on the anti-parse trace. If it is, produce the output and put this routine's state in the trace. If it is not, raise an error because the output created would be illegal, and place an error state on the anti-parse trace.

3 The Prototype

A prototype has been developed to study the usefulness and design aspects of a ParsesraP implementation. The prototype was developed using two Gnu [Fie90, Woe94] tools: flex [Pax88] for lexical analysis and bison [DS88] for parsing. The language in the actions is C [KR88]. Please note that

- this prototype's implementation should not be taken to be the only way to implement ParsesraP,
- nor is C the only language that should be considered for the actions,
- nor is this language the only way to express the ParsesraP idea.

The prototype language (in its own language [1]) this implementation of ParsesraP accepts is as follows:

```
/*
** the overall structure is a set of rules
*/
rules   : rule { rule } <> <>

/*
** each rule has a type that it may return, an identifier, a ":", and
** some number of patterns and related actions to perform when a
** pattern is matched.
*/
rule    : [ "(" type ")" ] ruleid ":" patterns actions
          { "|" patterns actions } <> <>

/*
** patterns are multiple pattern's
*/
patterns: pattern { pattern } <> <>

/*
** a pattern is a symbol, or a collection of pattern's or an optional
** pattern
*/
pattern : symbol <> <>
        | "{" pattern "}" <> <>
        | "[" pattern "]" <> <>

/*
** there are two actions: one for input and one for output.
*/
actions : '"<"[^>]*">"' '"<"[^>]*">"' <> <>

/*
** a symbol is either a literal string or a terminal
*/
symbol  : '\"[^"]*\"' <> <>
        | ruleid <> <>

ruleid: '[A-Za-z][A-Za-z0-9]*' <> <>

type: '[A-Za-z][A-Za-z0-9_*]*' <> <>
```

The `{}`'s denote zero or more repetitions of any symbol within. The `[]`'s denote an optional symbol. The `''`'s designate r.e., and the `""`'s denote fixed strings. Each rule in the grammar has some number of patterns that are matched on input and produced on output. Each rule also has two actions, each in `<>`'s, the first for input and the second for output.

For input, the ParsesraP prototype produces a flex file of all the `symbol`s, giving each a unique name. This name is only used to communicate with the parser. For output, if a `symbol` is a simple text string, it copies it to the produced C function, as its appearance on output is exactly known. If it is a r.e., it can produce multiple strings so the user must specify which of the possible strings is intended. Therefore, a r.e. verifier is included in the C function, so that when the user calls the output action routine a check of the string is made.

[1] The implementation was written in its own language to gain experience and to give some assurance it was flexible enough for real-world problems.

The ParsesraP prototype also produces a bison file with the input actions copied over for the parsing operation. For the output production phase, ParsesraP creates a C file with a number of functions, one for each rule/pattern pair in the description. The name of the rule and number of the pattern is encoded in the function names. For example, if a rule looks like:

```
keyword : "FOR"    <> <>
        | "WHILE"  <> <>
        | "IF"     <> <>
```

one can access the output action for the `WHILE` pattern by calling a routine named `keyword2()`. In this example, none of the patterns require output actions as the form is known exactly. On the other hand, consider:

```
for     : "FOR" 'i|j' "=" '[0-9]' <> <>
```

Here, the `FOR` and the `=` are known entities, but the other two terminals are ambiguous, and must be specified by either the output action or as parameters to the `for1()` routine.

The prototype maintains correct output by automatically inserting the proper calls to the output routines within the grammar. When an output member is a string, it is produced. When it is another rule, the generated code calls the proper routine with an argument the programmer specifies in the output action. When it is a R.E., the code checks the validity of the string, and produces it if it is valid.

4 The Example

To get a better idea of the power of the ParsesraP concept, a small example from the electronic computer-aided design field will be given. The example uses a simplified EDIF [Gro87] "`acload`" construct. The `acload` construct specifies the loading a part of an electronic circuit has with a minimum/nominal/maximum value. The construct can have two different forms: one where values are represented using whole numbers and another where an exponential form is used. A ParsesraP prototype file will be shown that can read and write both forms.

```
/*
** this pattern returns a pointer to an integer array on input, and the
** created routine acload1 takes a pointer to an integer for output.
*/
(int *) acload : "(" opt_white_space "acload" white_space minomax
opt_white_space ")"
<
        /*
        ** on input, print a properly formatted version of the input
        ** which is contained in the 'minomax' term.
        */
        puts (acload1 ($5));

        /*
        ** and return the value
        */
        $$ = $5;
>
<
        /*
        ** on output, just set the minomax value to what was passed in and
        ** let ParsesraP print it out correctly.
        */
        $5 = $$;
>
```

The input action calls a routine that is created by ParsesraP, called `acload1()`, creating a simple text filter by having the overall input function call the overall output function. The input action also returns the array of integers to the calling function. The output action simply specifies which argument (`$5`) receives the value that is passed in (`$$`).

```
(int *) minomax : "(" opt_white_space "minomax" white_space number
white_space number white_space number opt_white_space ")"
<
        /*
        ** we need to set aside some dynamic space to store the result
        ** of this input action so we can return a pointer to it.
        */
        $$ = malloc (3 * sizeof (int));

        /*
        ** now that we've got the space, put the values in it.
        */
        $$[0] = $5;
        $$[1] = $7;
        $$[2] = $9;
>
<
        /*
        ** on output, we just need to assign the 'number' values from
        ** the integers that are passed in.
        */
        $5 = $$[0];
        $7 = $$[1];
        $9 = $$[2];
>
```

On input, the parser creates a three dimensional integer array, sets the three values, and passes the array back. On output, the values passed in (in `$$`) are assigned to the proper arguments, and the lower level routine properly formats them.

```
/*
** this rule has two patterns, both of which return an integer.  the
** actions are empty as there is nothing to do; the default actions work
** well.
*/
(int) number : integer <> <>
        | e <> <>
```

The number action does nothing on input, which implicitly performs the assignment `$$ = $1`, as in yacc. On output, the opposite default action is produced, namely `$1 = $$`, leading to the correct action (i.e., automatically calling `integer1()` to perform the formatting.)

```
(int) integer : '[0-9][0-9]*'
<
        /*
        ** on input, simply call the atoi() builtin function to convert
        ** the characters to an integer.
        */
        $$ = atoi ($1);
>
```

```
<
        /*
        ** on output, get some space and use sprintf() to format the
        ** number passed in.
        */
        $1 = malloc (10);
        sprintf ($1, "%d", $$);
>
```

When a string of characters representing an integer is scanned, it is converted to an integer and returned. When an integer needs to be produced, the number is passed in and converted to characters. The user sets `$1` to the desired characters and they are automatically tested by the ParsesraP-generated function to make sure they correspond to the input R.E.

```
(int) e : "(" opt_white_space "e" white_space integer white_space integer
        opt_white_space ")"
<
       /*
       ** call the exp10() builtin to get the number raised to the
       ** correct power.  cast the result to an integer so the return
       ** type is correct.
       */
       $$ = $5 * (int) exp10((double) $7);
>
<
        /*
        ** on output, we need to do the opposite; that is, take the
        ** single integer passed in and break it into an integer and an
        ** exponent.
        */
        $7 = log10 ((double) $$);
        $5 = $$ / exp10 ((double) $7);
>
```

On input of a number with an exponent, the base number is raised to the proper value of ten and returned. On output, the number is split in two, and then both are automatically passed to `integer1()` for formatting.

```
/*
** on input, just each the white space; on output, write out nothing.
*/
(char *) opt_white_space : [ white_space ] <> < $1 = ""; >

/*
** on input, just each the white space.  on output, check to see if
** something was passed in.  if it was, print that, otherwise, print a
** single space.
*/
(char *) white_space : "[ \t\n]" <> < if (!$$) $1 = " "; else $1 = $$; >
```

For optional white space, no white space is specified by setting `$1` to the empty string. This sets `$$` of `white_space1()` to the empty string. `white_space1()` checks to see if `$$` has already been set: if it has it prints it, if it has not it produces a single space.

To use the code generated by ParsesraP, a main program must be specified. The first example will use the grammar

as a filter, simply by calling the `yyparse()` routine:

```
main ()
{
        /*
        ** call the main parsing routine.
        */
        return yyparse ();
}
```

With an input of:

```
(acload (minomax (e 5 1) (e 6 2) (e 7 3)))
```

this program generates the following output:

```
(acload (minomax 50 600 7000))
```

This simple text filter just changes from exponential form to an expanded integer form. Note the lack of programming involved in such a filter; all the input and output specifications are contained within the grammar.

To use the output half of a rule, a call must be made to it with the appropriate arguments. Here is an example that calls the highest-level routine:

```
#include "acload.h"

void main ()
{
/*
** set up a three-dimensional array of data
*/
        int a[3] = { 1, 2, 3 };

        /*
        ** call the overall formatting routine -- the first
        ** and only one generated by the acload rule
        */
        puts (acload1 (a));
}
```

The output of this short program is simply:

```
(acload (minomax 1 2 3))
```

5 Conclusions

ParsesraP is a framework for both the construction and consumption of data. It extends the parser-generator concept to cover the production of data. This is a great help to programmers who produce data that is accepted by any type of formal grammar, by freeing them of the need to produce all the known elements of the language, and by checking the variable parts for correctness.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers—Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[DS88] Charles Donnelly and Richard M. Stallman. “BISON—the YACC-compatible parser generator”. Technical report, Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, 1988. Bison was largely written by Robert Corbett, and made yacc-compatible by Richard Stallman. Electronic mail: `rms@prep.ai.mit.edu`. Software also available via ANONYMOUS FTP to `prep.ai.mit.edu`. See also [Pax88].

[Fie90] D. Fiedler. “Free software: What is it, how do you get it, and how good is it?”. *Byte*, 15(6):97, June 1990.

[Gro87] EDIF User Group. *EDIF Version 2 0 0*, June 1987.

[Joh79] Steven C. Johnson. “Yacc: Yet another compiler compiler”. In *UNIX Programmer’s Manual*, volume 2, pages 353–387. Holt, Reinhart, and Winston, New York, NY, USA, 1979. AT&T Bell Laboratories Technical Report July 31, 1978.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1988.

[Pax88] Vern Paxson. “flex—fast lexical analyzer generator”. Free Software Foundation 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, 1988. Electronic mail: `vern@lbl-csam.arpa` or `vern@lbl-rtsg.arpa`. Software also available via ANONYMOUS FTP to `lbl-csam.arpa`, `lbl-rtsg.arpa`, or `prep.ai.mit.edu`. See also [DS88].

[PDC] T. J. Parr, H. G. Dietz, and W. E. Cohen. *PCCTS Reference Manual*. School of Electrical Engineering, Purdue University, West Lafayette, IN, 47907.

[Woe94] Jack J. Woehr. “What’s gnu?”. *Embedded systems programming*, 7(1):70, January 1994.