NAME

a2p - Awk to Perl translator

SYNOPSIS

a2p [*options*] [*filename*]

DESCRIPTION

A2p takes an awk script specified on the command line (or from standard input) and produces a comparable *perl* script on the standard output.

OPTIONS

Options include:

-D<number>

sets debugging flags.

-F<character>

tells a2p that this awk script is always invoked with this **-F** switch.

-n<fieldlist>

specifies the names of the input fields if input does not have to be split into an array. If you were translating an awk script that processes the password file, you might say:

```
a2p -7 -nlogin.password.uid.gid.gcos.shell.home
```

Any delimiter can be used to separate the field names.

-<number>

causes a2p to assume that input will always have that many fields.

-o

tells a2p to use old awk behavior. The only current differences are:

- Old awk always has a line loop, even if there are no line actions, whereas new awk does not.
- In old awk, sprintf is extremely greedy about its arguments. For example, given the statement

  ```
  print sprintf(some_args), extra_args;
  ```

  old awk considers *extra_args* to be arguments to `sprintf`; new awk considers them arguments to `print`.

"Considerations"

A2p cannot do as good a job translating as a human would, but it usually does pretty well. There are some areas where you may want to examine the perl script produced and tweak it some. Here are some of them, in no particular order.

There is an awk idiom of putting int() around a string expression to force numeric interpretation, even though the argument is always integer anyway. This is generally unneeded in perl, but a2p can't tell if the argument is always going to be integer, so it leaves it in. You may wish to remove it.

Perl differentiates numeric comparison from string comparison. Awk has one operator for both that decides at run time which comparison to do. A2p does not try to do a complete job of awk emulation at this point. Instead it guesses which one you want. It's almost always right, but it can be spoofed. All such guesses are marked with the comment "#???". You should go through and check them. You might want to run at least once with the **-w** switch to perl, which will warn you if you use == where you

should have used eq.

Perl does not attempt to emulate the behavior of awk in which nonexistent array elements spring into existence simply by being referenced. If somehow you are relying on this mechanism to create null entries for a subsequent for...in, they won't be there in perl.

If a2p makes a split line that assigns to a list of variables that looks like (Fld1, Fld2, Fld3...) you may want to rerun a2p using the **-n** option mentioned above. This will let you name the fields throughout the script. If it splits to an array instead, the script is probably referring to the number of fields somewhere.

The exit statement in awk doesn't necessarily exit; it goes to the END block if there is one. Awk scripts that do contortions within the END block to bypass the block under such circumstances can be simplified by removing the conditional in the END block and just exiting directly from the perl script.

Perl has two kinds of array, numerically-indexed and associative. Perl associative arrays are called "hashes". Awk arrays are usually translated to hashes, but if you happen to know that the index is always going to be numeric you could change the {...} to [...]. Iteration over a hash is done using the keys() function, but iteration over an array is NOT. You might need to modify any loop that iterates over such an array.

Awk starts by assuming OFMT has the value %.6g. Perl starts by assuming its equivalent, $#, to have the value %.20g. You'll want to set $# explicitly if you use the default value of OFMT.

Near the top of the line loop will be the split operation that is implicit in the awk script. There are times when you can move this down past some conditionals that test the entire record so that the split is not done as often.

For aesthetic reasons you may wish to change index variables from being 1-based (awk style) to 0-based (Perl style). Be sure to change all operations the variable is involved in to match.

Cute comments that say "# Here is a workaround because awk is dumb" are passed through unmodified.

Awk scripts are often embedded in a shell script that pipes stuff into and out of awk. Often the shell script wrapper can be incorporated into the perl script, since perl can start up pipes into and out of itself, and can do other things that awk can't do by itself.

Scripts that refer to the special variables RSTART and RLENGTH can often be simplified by referring to the variables $`, $& and $', as long as they are within the scope of the pattern match that sets them.

The produced perl script may have subroutines defined to deal with awk's semantics regarding getline and print. Since a2p usually picks correctness over efficiency. it is almost always possible to rewrite such code to be more efficient by discarding the semantic sugar.

For efficiency, you may wish to remove the keyword from any return statement that is the last statement executed in a subroutine. A2p catches the most common case, but doesn't analyze embedded blocks for subtler cases.

ARGV[0] translates to $ARGV0, but ARGV[n] translates to $ARGV[$n-1]. A loop that tries to iterate over ARGV[0] won't find it.

ENVIRONMENT

A2p uses no environment variables.

AUTHOR

Larry Wall <*larry@wall.org*>

FILES

SEE ALSO

```
perl The perl compiler/interpreter

s2p sed to perl translator
```

DIAGNOSTICS

BUGS

It would be possible to emulate awk's behavior in selecting string versus numeric operations at run time by inspection of the operands, but it would be gross and inefficient. Besides, a2p almost always guesses right.

Storage for the awk syntax tree is currently static, and can run out.