Contents

Small Awk

Status

THis is a frst rough draft for a new language. The symbol **TBD** indicates a known area of incompleteness "To Be Done".

Authors

Richard J Botting **rbotting@csusb.edu**

Purpose

Small Awk (not **smalltalk**!) is designed as sample language studied and worked on in Computer Science classes that study high level computer languages. It is a subset of the **awk** language of Aho, Weinberg and Kernighan. IT is a subset created by deleting a lot of useful convenient options and features for **awk**. Even so it is a languages that has control structures variables, input, and output. For simple, every day programs it is rather like a small and safe C or C++.

Small awk programs can be tested by running them thru any **awk** interpreter on a UNIX system. **awk** has also been ported to other platforms and can also be used to test smallawk programs. Suppose your smallawk program is in a file called

```
hello.awk
```

then the command

```
awk -f hello.awk
```

will test the program.

Examples

Like **awk**, smallawk is designed for processing data files. It works with simpler files than **awk** however. Smallawk assumes that it will read a single file and each line in the file is a list of "fields" of data separated by one or more spaces or tabs. Smallawk reads the inoput file and produces a single stream of output by processing the input. Notice that a smallawk program does not start until it is given some data, and doesn't stop until it gets the usual end-of-file. Here is a traditional simple program in smallawk:

```
END{ print "hello, world"; }
```

It reads the input and at the end of the inpoyut data it outputs the "hello, world message. The following program reads the input file and numbers the lines:

```
{ print NR, $0; }
```

The next program prints outs lines that contain the string "AWK":

```
/AWK/{ print ;}
```

The next program Assumes that each line has one number, and at the end of the file outputs the total of these nunmbers:

```
{sum = sum + $0;}

END{print sum;}
```

This one checks the input and only adds a line if it is a valid integer, with one or more decimal digits:

```
/^[0-9][0-9]*$/{sum = sum + $0;}

END{print sum;}
```

This reads in a file of names, student_ids, and scores and calculates the mean score. It assume input with data separated by spaces like this:

```
ShortName 9999 3.2

AnotherName 1234 17
```

The program is

```
{sum = sum + $3; count=count+1;}

END{print sum/count;}
```

Syntax

Lexemes

smallawk has the normal lexical scan separating variables, constants, strings, from some reserved words:

1. **reserved_words**::= "END" | "NF" | "NR" | "BEGIN" | "print" | "if" | "else".
2. **END**::lexeme, true after last line is read.
3. **NF**::lexeme, *Number of Fields*.
4. **NR**::lexeme, *Number of this line/record*.
5. **BEGIN**::lexeme, *true on first line/record only*.
6. **print**::lexeme, *starts a command to output the value of an expression*.
7. **if**::lexeme, *introduces a conditional statement*.
8. **else**::lexeme, *alternative branch of a conditional statements*.
9. **comma**::lexeme= ",".

Programs

A program is a sequence of pieces. Each piece has two parts: a pattern and an action. The pattern states when the action is to be applied. When smallawk is running it takes each line and tries each piece of the program in turn and carries out the actions with patterns that match the line.

10. **program**::= # **piece**.

```
BEGIN{product=1;}

{product = product * $0;}

END{print product;}
```

[**example.awk**]

11. **piece**::= **O**(**pattern**) "{" **action** "}". The pattern is used to recognise the lines of data that the action applies to.

```
/^[0-9][0-9]*$/{sum = sum + $0;}

END{print sum;}
```

Patterns

12. **disjunction**::= **conjunction** #("||" **conjunction**).

```
/Botting/||/Dick/
```

13. **conjunction**::= **possible_complement** #("&&" **possible_complement**).

```
/Botting/ && /Richard/
```

14. **possible_complement**::= "!" **elementary_pattern** | **elementary_pattern**.

```
!/Blotting/
```

15. **elementary_pattern**::= "END" | "BEGIN" | "/" **regular_expression** "/".

```
/Banana/
```

```
/[bB]anana/

/^\.As_is/

/^[0-9]*$/
```

16. **regular_expression**::= **left_most** | **rightmost** | **exact** | **contained**.
17. **leftmost**::= "^" **contained**.

```
^.As_is
```

18. **rightmost**::= **contained** "$".

```
two part:$
```

19. **exact**::= "^" **contained** "$".

```
^Dick.*Botting$
```

20. **contained**::= #possibly_repeated_set_of_chars.
21. **possibly_repeated_set_of_chars**::= **possible_set** "*" | **possible_set**.

```
pos*ibil*ity
```

The "*" means "zero or more of". The above matches "posssibiity" for example.

22. **possible_set**::=**normal_character** | **escaped_character** | **set_of_characters**.
23. **normal_character**::= letter | digit | symbol ~ **special_character**.
24. **escaped_character**::= "\" **special_character**.
25. **special_character**::= "[" | "]" | "\" | "." | "*".
26. **set_of_characters**::= "[" chars "]" | "[^" chars "]".
27. **chars**::= # character | character "-" character.
28. **character**::= normal_character | escaped_character.

Actions

An action is a series of at least one statement. These are executed one after another.

29. **action**::= **statement** #**statement**.

Statements

30. **statement**::= **assignment** | **print** | **selection** | **do_nothing**.
31. **assignment**::= **variable** "=" **expression** ";"

```
sum = sum + $0;
```

32. **print**::= "print" ";"| "print" **expression** ";".

```
print sum;
```

33. **selection**::= "if(" **condition** ")" **body** "else" **else**.

```
if ( sum > 0 ) print "greater"; else ;
```

34. **body**::=**action** | **do_nothing**.
35. **else**::=**action** | **do_nothing**.
36. Note: avoids an ambiguity (dangling else) by always requiring an else for every if. real *awk* does not have this restriction...
37. **condition**::=**expression**.
38. **do_nothing**::= ";".

Expressions

39. **operation**::= "+" | "-" | "*" | "/" | "%" | "&&" | "||" | "!".
40. **function**::= "sin" | "cos" | "log" | "exp" | "sqrt".
41. **expression**::= **concatenation**.
42. **concatenation**::= **arithmetic_expression** #**arithmetic_expression**.
43. **arithemetic_expression**::= simple_expression #(#**operation** simple_expression).
44. **simple_expression**::= function_call | **variable** | **constant** | parentesized_expression.
45. **parenthesized_expression**::= "(" **expressison** ")".
46. **function_call**::= **function** "(" expression #(**comma** expression).

TBD : Operations, variables, functions....

Variables

47. **variable**::= **whole_line** | **field** | **global_variable**.
48. **whole_line**::= **dollar** "0".

```
$0
```

49. **field**::= **dollar** expression.

```
$5
```

50. **global_variable**::=letter #(letter|digit).

```
sum
```

Constants

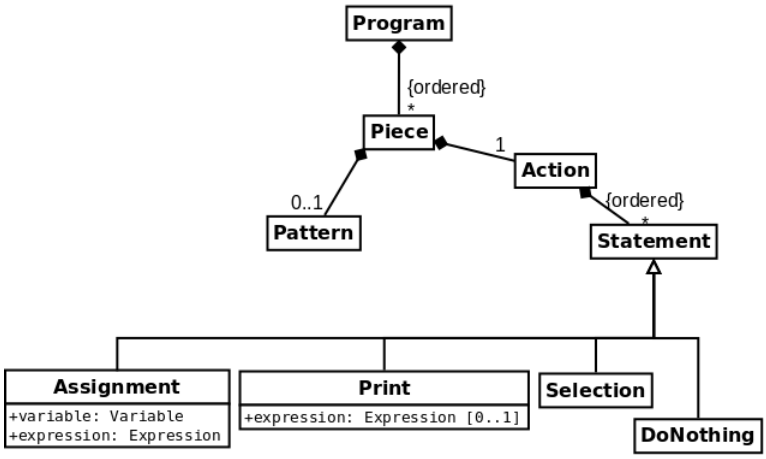
TBD

. . . . . (end of section **Syntax.**) **<<**Contents | End**>>**

Semantics

Here is a diagram:

Partial Semantic Structure of SmallAwk



TBD

An smallawk program describes a transformation of a text file into another text file. This is done one line at a time. Each line is read and matched against the patterns. Where they match the actions are executed. These lead to outputing lines of data as well.

Here are the informal operational semantics of a smallawk program *P*. *P* wll consist of a sequence of *n* pieces *p*[1]..*p*[*n*]. Each piece 'p'['i'] has two parts a pattern 'p'['i'].pattern and an action 'p'['i'].action. Here is a C++like description of what the program *P* does.

```
NR=1;
while( get next line until end of input )
{
        for(i = 1; i<=n; i++)
                if( line matches p[i].pattern )
                        apply p[i].action to line;
        NR++;
}
//after end of file
for(i = 1; i<=n; i++)
        if( p[i].pattern is "END" )
                apply p[i].action;
```

A line matches a pattern according to the rules **TBD** ... [**regular_expressions.html**]

Applying an action to the line starts by assigning the whole line to variable *$0*. Then each field in the line (separated by one or more spaces) is assigned to *$1* thru to ***NF*** where *NF* is set to the number of fields. An action is a sequence of one or more instructions and these are executed in turn. If an instruction is an assignment then the expression on the right hand side is evaluated and the resulting value is placed in the variable on the left hand side of the '=' sign. This may change the whole line or any field in the line if the variable is '$0' or '**i**' for some other i. If the action is a print command with an expression then the expression is evaluated and output plus a new line. If it is a print with no expression then the whole line (with any changes) is printed. If the instruction is a selection with condition *c* and body *b* and else part 'e', then the condition is evaluated and if it not zero the body *b* is executed. Other wise if *c* evaluates to zero then *e* is executed.

Expressions are evaluated in the usual way: constants beome their values, variables return their current value, and operations are applied in order of precedence to give a value.

See Also

1. **awk**::= See **http://cse.csusb.edu/dick/cs360/notes/awk.html**.
2. **smalltalk**::= See **http://cse.csusb.edu/dick/samples/smalltalk.html**

Glossary

3. **TBA**::="To Be Announced".
4. **TBD**::="To Be Done".
5. For X, **O**(X)::= (X |), Optional X.

. . . . . (end of section **Small Awk**) **<<**Contents | End**>>**

End