**JJ : Build System**

## SYNOPSIS

```
   jj  [-m] build_script_file
            build_script_file
 or
   jj -v
```

**jj** is a **make** replacement. It uses a postfix script language for defining the build commands. Unlike **make**, it contains no default actions. However, it does enable the user to write new functions which can be used for building the same kind of targets using the same commands.

## SCRIPT SYNTAX

The script files have a very simple syntax. Here are the various language elements:
- Strings
- Quoted Strings
- Function Calls
- Variables

Strings are written without quoting. A string starts with a printable character other than the special characters and ends when a whitespace character is seen. Therefore, you can not have spaces within your strings.

Quoted strings are in fact lists of strings, separated by whitespace. Since whitespace is used as a separator for list items, you can not have spaces in list items either. This rule has some exceptions, but it will hold true in most cases. The quote characters are **{** and **}**. Here is an example of a quoted string:

```
{a.c b.c d.c}
```

Function calls are strings starting with the # character. When you call a function, the preceding strings are passed as arguments to the function. If there are less arguments than required, missing arguments will be assigned to the empty string. If there are more arguments than required, they are discarded.

Variables are similar to the shell variables in syntax. When you want to access a variable, you shall refer to is as **$VAR** where *VAR* is the name of the variable. When you want to set a variable, you should simply refer to it as **VAR**. Although any sequence of printable characters (except for special characters) is acceptable as a variable name, you should probably restrict yourself to alphanumeric characters. This program is a work in progress and later versions may not support that.

## COMMANDS

In order to build files, you need to describe the building commands using the **#command** function. This function takes a single argument, which describes the various fields of the command object. Here is an example command:

```
{ triggers {a.c b.h c.h}
  outputs {a.o}
  action  { gcc -o a.o a.c -Wall }
  message { Compiling a.c }
} #command
```

The **triggers** field contains a list of files which cause the output to be re-generated if any of them is newer than an output file. The **outputs** field contains the list of files generated by this command. Note that there are no phony targets as in **make**. All members of **triggers** and **outputs** shall be real files residing on disk.

The **action** field is a string, in which whitespace is kept intact. This string is given to the shell using the **system** function of libc. Therefore, you can have environment variable references within this action string. It's also possible to refer to the fields of the command struct as variables in this action string. For instance,

```
{ triggers {a.c b.h c.h}
  outputs {a.o}
  action { gcc -o $outputs a.c -Wall }
  message { Compiling a.c }
} #command
```

Could be used instead of the previous command. Note that the value of the action string is computed right before the shell command is executed. Therefore, you could do modifications to the command using the **#modify** function and the field-as-variable references in the action field would still work.

The **message** field is parsed similarly to the **action** field; the spaces are kept intact. When you use the **-m** command line switch, **jj** prints these message strings instead of the shell commands being executed.

The above fields are the only pre-defined fields of the **command** struct. You can add other fields

to the struct as well. Compiler options, library options and etc are good candidates to be put in fields of command structs. Note that the user defined fields have string list semantics instead of the single-string semantics like the **action** and **message** fields.

Only the **outputs** and **action** fields are mandatory for a command struct. You can have commands which have no **message** field. In this case **jj** will always print the **action** string. You can also have commands without any triggers. In that case, the command will run every time the script is executed.

### Modifying Already Defined Commands

It's also possible to modify already existing commands using the **#modify** function. This function takes two arguments:
•*TARGETS*, which is a list of target files.
•*BODY*, which describes the modifications to the commands generating the given targets.

For instance, for the above command, we could do:

```
a.o  { +triggers g.h } #modify
```

This would add *g.h* to the list of triggers generating *a.o*. You can also give the field name without a plus sign:

```
a.o  { triggers {a.c g.h} } #modify
```

This will replace the value of the **triggers** field.

### VARIABLES

Variables are written similarly to shell variables. You shall use the **#set** function to assign a simple string to a variable. Other functions take the name of a result variable as their first argument and set the value of that variable accordingly. Here is a simple assignment:

```
  X HELLO #set
```

From now on, **$X** will refer to the string **HELLO**. You can also use **#set** to implement indirection similar to pointers:

```
$X xyzzy #set
```

will set the variable **$HELLO** to the string **xyzzy**.

There are two flavors of variables, local and global variables. When you assign to a variable that doesn't exist, the variable is created in the global scope unless the variable has been created using the **#local** function.

New scopes are created when the code enters a user defined function. For instance, if you have:

```
pretty_print A B C {
 $A #print $B #print $C #print
 pretty #print
} #function
```

**$A**, **$B** and **$C** are all new variables bound to the arguments and have no relationship to global variables with the same names.

If you set a value to a new variable in this function

```
pretty_print A B C {
   $A #print $B #print $C #print
   pretty #print
  X HELLO #set
} #function
```

Then, **$X** will be created as a global variable. However, if you had:

```
pretty_print A B C {
   $A #print $B #print $C #print
   pretty #print
   X #local
   X HELLO #set
} #function
```

Then, **$X** will go out of scope when the function returns.

When you have an undefined variable, its value is simply the name of the variable. For instance, if you have

```
{ outputs {a.o}
  triggers {a.c}
  action { gcc $CFLAGS -o a.o -c a.c }
} #command
```

and *$CFLAGS* is not defined, it's passed unmodified to the shell. This way, you can make use of environment variables within your actions.

## USER DEFINED FUNCTIONS

The syntax for a new user defined function is:

```
function_name ARGUMENT-LIST {
    statements
} #function
```

The function name is a simple variable name without the preceding # character. The *ARGUMENT-LIST* is either a single variable name or a list of variable names quoted using the **{}** pair. Here is a simple function:

```
compile {SOURCE HEADERS OBJECT} {
  { triggers { $SOURCE $HEADERS }
    outputs {$OBJECT}
    action { gcc -o $OBJECT
                 -c $SOURCE }
    } #command
} #function
```

We could call this function as follows:

```
a.c {parser.h env.h} a.o  #compile
```

You can have multiple statements in the body of the function. You can also create local variables using the **#local** function as explained in the VARIABLES section.

## BUILT-IN FUNCTIONS

•*VARNAME VALUE* #**set**
•*VARNAME* #**local**
•*FUNCNAME ARGS BODY* #**function**
•*VARNAME STR1 STR2* #**strcat**

This function concatenates two string lists and stores the result in the variable called *VARNAME*. The actual operation is a cross-product. If we have

```
STR1 {a1 a2 .. aN} #set
STR2 {b1 b2 .. bM} #set
STRX $STR1 $STR2 #strcat
```

the value of **STRX** would be

```
{a1b1 a1b2 .. a1bM
 a2b1 a2b2 .. a2bM ... aNb1
 aNb2 .. aNbM}
```

If *STR1* is a single string, this operation is equivalent to prefixing all in *STR2* with *STR1*. If *STR2* is a single string, then this operation appends *STR2* to all in *STR1*. If both are single strings, then this operation simply concatenates them together. If you want to join lists instead of making a cross product, you can use the **#set** function.

```
RESULT {$X $Y} #set
```

•*VARNAME LIST BODY* #**foreach**

This function iterates over the elements of the *LIST* argument. At each step, the variable *VARNAME* is set to the next element of the list and statements in the *BODY* argument is executed. When the loop terminates, value of the loop variable is the last element of the list.

•*FILENAME* #**include**

This function runs another script file given by *FILENAME* and continues from the current file after that. The *FILENAME* argument can also be a list of files, which are processed in the given order.

•*VARNAME LIST NEWSUFFIX* #**replace_suffix**

Replaces the suffixes of the files in the given list with the new suffix and stores the result in variable *VARNAME*. The suffix of a file name is the part that follows the last dot character. If a file has no suffix, the new suffix is simply appended to the file name.

•*VARNAME LIST INDEX* #nth

Stores the *INDEX*th element of *LIST* in variable *VARNAME*. Indices start from zero. If the index is out of range, the result variable is set to empty string.

•*FIELDS* #**command**
•*TARGETS FIELDS* #**modify**
•*VARNAME COMMAND-AND-ARGS* #**prgout**

Runs the program given in *COMMAND-AND-ARGS* and stores the output of the program (i.e. what it prints to **stdout** in the given variable. The second argument is a list of strings. Each string is an element of the **argv** vector given to **execvp**. Since the program is executed directly instead of going through a shell, environment variables and etc will not work correctly. If you want shell effects, you can try running the shell (**/bin/sh**) with appropriate options (**-c** etc).

•*STRING* #**print**

This function simply prints its sole argument to **stdout**. If there are variables in the string, they are expanded before being printed. Note that this function is for debugging only, since it also prints a prompt and an end-marker for the string. If you want to get meaningful output at stdout, try using a command which runs the **echo** program.

•*STRING* #**comment**

This function simply ignores its argument and does nothing.

## BUILDING IN ANOTHER DIRECTORY

If you want to build in a directory other than your source directory, you can make use of the **$HERE** variable. A method of doing so is given here, you can maybe develop others.

The **$HERE** variable points to the directory in which the current script being executed resides. In other words, it's the directory name of the script file. You can prefix your source files with this directory in order to get file names independent of where you run the **jj** program. The variable always includes a trailing slash so it's easy to do so.

For instance let's have a definition somewhere in your included scripts:

```
compile {SOURCE HEADERS} {
  {SRC HDR OBJECT} #local
  SRC $HERE $SOURCE #strcat
  HDR $HERE $HEADERS #strcat
  OBJECT $SOURCE .o #replace_suffix
  { triggers { $SRC $HDR }
    outputs { $OBJECT }
    action { gcc -c $SRC
             -o $OBJECT -Wall -g }
  } #command
} #function
```

Given this function, and two scripts, one in *project/src*:

```
a.c {a.h parser.h} #compile
```

and one in *project/build*:

```
{ ../jj-functions
  ../src/jjcool} #include
```

We can build in the *build* directory easily. Since the call to **#compiler** will be from *project/src/jjcool*, **$HERE** will have the value of *project/src/*. The prefixed file names will be absolute names and the compiler can work with that. One shortcoming is that, local C include directives do not work with this approach but it can be solved easily by adding another option to the compiler command line:

```
action { gcc -c $SRC -o $OBJECT
            -Wall -g -I $HERE}
```

## BUGS AND LIMITATIONS

Builtin functions do not check their arguments. If you get them wrong, you might end up with a very unhelpful segfault.

Since there is no error checking, there is very little error reporting. When **jj** reports errors, it doesn't tell you on which line you have made an error. This is a serious limitation but I intend to fix it soon, when I have more time to work on it.

## COMMAND LINE SWITCHES

The **-v** command line switch causes **jj** to print its version number and exit.