Austin Anderson

An Operational Semantics and OCaml Implementation for a subset of Bash

Part II Computer Science

Gonville and Caius College

May 15, 2008

Proforma

Name:	**Austin Anderson**
College:	**Gonville and Caius College**
Project Title:	**An Operational Semantics and OCaml Implementation for a subset of Bash**
Examination:	**Computer Science Tripos, Part II**
Word Count:	**11873**
Project Originator:	Dr Peter Sewell
Supervisor:	Dr Tom Ridge

Original aims of the project

"... I propose to make an operational semantics for a shell with Bash like syntax ... [and] an implementation in ML of the defined operational semantics. "
The following sections of the Bash specification were suggested to be implemented:

Core functionality

- Looping constructs
- Conditional constructs
- Grouping commands
- Shell parameter expansion
- Arithmetic expansions

- Bash conditional expressions
- Shell arithmetic

Extensions expected to be done

- Pipelines
- Brace expansion
- Tilde expansion
- Redirections
- Aliases
- Arrays

(the above are from the project specification)

For both of the above, only the standard and appropriate parts of each section were intended to be defined and implemented.

Work completed

The core functionality has an operational semantics and an implementation. The extensions done were changed from the above to:

- Shell functions
- Redirections
- Selected shell builtin commands

These had an operational semantics defined and an implementation.

Special difficulties

None

Declaration

I, Austin Anderson, of Gonville and Caius College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. Signed

Date

Contents

List of Figures

Chapter 1

Introduction

Before graphical environments existed, command line shells were the main form of human computer interaction. Though graphical interfaces have become prevalent in many contexts shells are still highly used, particularly by power users and systems administrators. Bash is presently one of the most commonly used shells as it is the standard shell on many modern Linux distributions.

Unfortunately Bash was defined without the use of tools for rigorous language definition which, in combination with features and syntax which can be unintuitive, often results in confusion and an inability for the majority of users to employ more than a fraction of the full functionality and power of Bash. Hence this project attempts to represent the syntax and semantics in a formal manner. The following chapter presents the main features of the Bash language.

Bash is a versatile tool, but there are several situations in which it has particularly excelled and is often used. These situations include system administration, automating repetitive tasks for power users, providing quick access to applications and providing a convenient method of piping and specialising data. Figure 1.1 gives an example of a script which is used for automating a repetitive task to do with systems management. The script exhibits many of the features and functionality which give Bash its distinctiveness and expressiveness. More specifically the script takes the name of a file or directory as an parameter (the code `$1` indicates dereferencing the first parameter), checks to see if a file or directory exists with that name and if so appends the name to the end of a file which defines which files

should be backed up. Several of these features are explained below and illustrated using snippets of code from Figure 1.1 as well as other relevant example code.

Determining properties of files

The Bash language includes functionality to examine useful properties of files, such as if it exists or if it is a directory. An example of this from Figure 1.1 is the following:

```
if [ ! -f $SRC ]; then
   if [ ! -d $SRC ]; then
      echo "$SRC does not exists"
      exit 2
   fi
fi
```

This section of code provides error detection its behaviour is as follows. The code `$SRC` dereferences a variable named `SRC` and `[ ! -f $SRC ]` checks to see if the string (which is the dereferenced value of `SRC`) is the name of a normal file. If it isn't then `[ ! -d $SRC ]` checks to see if the string is the name of a valid directory. If it isn't then it prints a message to standard output (in this case the terminal) to indicate an error. File processing is a common action for systems administration and the ability to determine various properties of files is necessary to know how to process them.

Redirections

In Bash redirections are used to control where standard input, output and error come from/go to. This at first seems to be a slightly strange piece of functionality, as opposed to having file access functionality and printing functionality. However redirections are a very powerful construct. An example of a redirection from Figure 1.1 is the following:

```
echo "$SRC" >> $FILE
echo "$SRC added to $FILE"
```

In this example values have already been assigned to the variable `SRC` and `FILE`. In the first line the code `>> $FILE` is a redirection and indicates that Bash should take anything which is sent to standard output (from the line on which the redirection

```
#!/bin/bash
# mybackupadd - Add file to ~/.mybackup file, then backup and email all
# file as tar.gz to your email a/c.
# Copyright (C) 2004 nixCraft project
# Email   : http://cyberciti.biz/fb/
# -------------------------------------------------------------------------
# This program is free software; you can redistribute it and/or
# modify it under the terms of the GNU General Public License
# as published by the Free Software Foundation; either version 2
# of the License, or (at your option) any later version.
# -------------------------------------------------------------------------

FILE=~/.mybackup
MYH=~
CWD=‘pwd‘
SRC=$1

if [ "$SRC" == "" ]; then
  echo "Must supply dir or file name"
  exit 1
fi
# if list $FILE does not exist
[ ! -f $FILE ] && touch $FILE || :

# make sure that file or dir exists to backup
if [ ! -f $SRC ]; then
   if [ ! -d $SRC ]; then
      echo "$SRC does not exists"
      exit 2
   fi
fi
# make sure we don’t do add duplicate stuff
cat $FILE | grep -w $SRC > /dev/null
if [ "$?" == "0" ]; then
   echo "$SRC exists in $FILE"
   exit 3
fi
# okay now add that to backup list
echo "$SRC" >> $FILE
echo "$SRC added to $FILE"
```

Figure 1.1: A script which automates adding a file or directory to a list of files and directories to be backed up

is) and instead append it to the end of file name given by dereferencing `$FILE`. The second line does not contain any redirections, and hence when it outputs the string `echo "$SRC added to $FILE"` to standard output it is printed to the terminal.

From this example redirections seem reasonably straightforward. Redirections are often only used in a basic manner such as this, but this is due to the obfuscation of more complex parts of the syntax rather than due to the simple cases being all that the redirections can express. For example, the code:

```
echo "echo hello world" >ft
f () { read x ; ${x} ; }
f <ft >>ft
```

is more obfuscated in terms of its behaviour. The first line is some set up code which creates a file `ft` and puts into it the text "echo hello world". The second line defines a function which reads a string value from standard input, assigns that value to the variable `x`, and then attempts to execute the string. The third line sets up redirections so that, whilst f is being called, standard input comes from file `ft` and standard output appends to the end of file `ft`. The result of this code is the output of the text "hello world" to the end of file `ft`, i.e. its contents are the text:

```
echo hello world
hello world
```

The power of redirections is in their expressiveness and their succinctness - in a few character strokes one can define from where data for an expression should be read, to where the results go, both of these can be either from/to the terminal or a file and it can be done without having to open or deal with file descriptors.

Piping

In a similar way to redirections pipelines attach the standard inputs and outputs of a series of expressions together in order to process some information in a streamlined manner. An example from Figure 1.1 is the following:

```
cat $FILE | grep -w $SRC
```

Here `cat $FILE` outputs the contents of the file name given by dereferencing `$FILE`, and sends the contents to `grep -w $SRC` which further processes it (the details of how `grep` processes the contents are not important). The piping facility is of particular use in the automation of routine operations.

Evaluation of text is different depending on the context

The way that Bash evaluates strings differently depending on the context is a very complex and intricate aspect of Bash which will be discussed in more detail later in the project. An example of this is the following:

```
x=${pwd}{$(pwd),"pwd"}
```

This results in `x` having the value "foo{/home/austin/tmp,pwd}" (when run in directory */home/austin/tmp* and when variable `pwd` has value "foo"). Firstly `${pwd}` dereferences the variable `pwd`. Secondly `$(pwd)` calls the function which returns the present working directory. Finally `"pwd"` is the string "pwd" and is not evaluated in any way. This shows how the same string means different things depending on the context (e.g. for the first the important context is the syntax `${ ... }`).

1.1 Motivation

The principal aim of the project is to investigate the syntax and semantics of Bash and to try to document and reproduce them in a grammar, an operational semantics and an OCaml implementation. There are two main motivations to do this. The first is to create a piece of reference material so that instead of having to trawl through the Bash manual and have to follow all of the cross references one could (with a suitable understanding of the notation of operational semantics) look in the operational semantics and see how a particular aspect of Bash worked that one did not understand. The second motivation is to have a logical representation of the Bash semantics about which one could reason, and possibly attempt to prove properties of the language. This formally specified version of Bash is herein referred to as Bash_{light}.

To the author's knowledge there has been little other work in the area of

defining semantics for scripting languages. An OCaml implementation of a subset of the Scsh specification has been attempted[2], but this was only attempting to re-implement the specification in OCaml and not to clarify the specification in concrete operational semantics.

Chapter 2

Preparation

There are a great breadth of features which could be investigated in the Bash scripting language. In order to make the problem tractable a subset of these features had to be selected to be investigated and implemented. The following chapter details which features were chosen and why as well as other information to do with the project preparation.

2.1 Requirements analysis

The project was divided into two parts, a core part dealing primarily with common language features, and the extensions which dealt with more interesting Bash-specific constructs. The reason for this divide is as follows. Many Bash features were identified in the project proposal (Appendix C), but these features are complex and moreover they interact with each other in extremely non-intuitive ways. It was clear that a full understanding of these complex features would only arise after significant experience with Bash. Thus, specification and implementation of the Bash-specific features were mostly delayed until more significant experience with Bash had been acquired, whilst the common language features were examined for the core of the project.

2.1.1 Common language features

Looping constructs (*Bash manual section 3.2.4.1*) are an expressive and essential part of most languages. In order to be able to express the computable functions it

is necessary to include some form of looping or recursion in the control flow. Bash includes `while`, `until` and several different `for` loops - for example:

```
for name [in words ...]; do commands; done
for (( expr1 ; expr2 ; expr3 )) ; do commands ; done
```

are both acceptable forms of the `for` command. The first assigns each word value to the variable `name` in turn and then executes commands, whilst the second behaves roughly as a normal `for` loop.

Conditional constructs (*Bash manual section 3.2.4.2*) are required in most programming languages. Two major conditional constructs in Bash are the following:

```
if test-commands; then
  consequent-commands;
 [elif more-test-commands; then
  more-consequents;]
 [else alternate-consequents;]
fi

case word in [ [(] pattern [| pattern]...) command-list ;;]... esac
```

These can be used as the traditional conditional statement and for pattern matching respectively. Bash also uses a specific sub-grammar for expressing and evaluating certain expressions with a conditional (i.e. Boolean) result (*Bash manual section 6.4*).

Shell parameter expansion (*Bash manual section 3.5.3*) is the Bash implementation of variables. These encompass both function parameters and user defined variables, though these are distinct constructs. An example of an assignment in Bash is `x = 3`. An example of a dereference is `${y}`.

Shell arithmetic (*Bash manual section 6.5*) is a specific sub-grammar and semantics for arithmetic. This construct determines how arithmetic is evaluated. Also it is interesting to note that in Bash arithmetic cannot be included as a general expression - it can only be used in certain parts of certain expressions. For example it is used in the following:

```
for (( expr1 ; expr2 ; expr3 )) ; do commands ; done
 $(( expression ))
```

In the first example each of `expr1`, `expr2` and `expr3` are evaluated as arithmetic but `commands` is not. In the second example `expression` is evaluated as arithmetic and it demonstrates how arithmetic can be embedded in a normal Bash expression - this construct is called Arithmetic expansion (*Bash manual section 3.5.3*) and effectively evaluates the arithmetic of `expression` and returns the result.

All of the standard language features were to be included in the core of the project.

2.1.2 Bash-specific features

Grouping commands (*Bash manual section 3.2.4.3*) are used to group expressions together in a way which is used to control scoping. Bash is a dynamically scoped language, but control is given to the programmer as to how exactly the dynamic scoping works and this is done through grouping commands. The Bash implementation of the traditional example for dynamic scoping is:

```
x=0
f () { echo ${x}; }
g () { x=1; }
g ; f
```

This results in the value `1` being printed to the terminal. However in Bash one also can do:

```
x=0
f () { echo ${x}; }
g () ( x=1; )
g ; f
```

(note the difference in bracketing syntax for the function `g`) which results in `0` being printed to the terminal - the scoping is still dynamic but the non-intuitiveness which is normal to dynamic scoping can be removed, since the effects of deeper scopes can be ignored if desired. Scoping is almost completely controlled by grouping commands, except for the way function parameters are handled. The level of

complexity which grouping commands entailed was not realised at first and hence they were included in the core requirements of the project rather than in the extensions.

Redirections (*Bash manual section 3.6*) are used for controlling where standard input, output and error are sent to, and also provides file descriptors and a facility for file input/output. Their importance was discussed in Chapter 1. These were done as an extension to the project.

Pipes (*Bash manual section 3.2.2*) are used for processing data in a way which doesn't require intermediary storage. Their importance was discussed in Chapter 1. These were left to an extension.

Builtin functions (*Bash manual section 4*) are used to provide common functionality to scripts and to control the Bash environment. Common examples of such functions are `cd` and `echo`, but there are several more unusual constructs such as `help` which displays help information about builtin functions. These constructs provide similar functionality to a series of libraries which come bundled with a language, though they are actually part of the Bash language. There are many different builtin functions and implementing all of them would be impractical, especially as many of them are quite idiosyncratic. Hence builtin functions were examined as an extension since they needed to be studied in more detail.

2.1.3 OCaml implementation

As well as defining the operational semantics formally using Ott, the semantics were also implemented as an OCaml program. The implementation interprets Bash expressions and prints the result of execution. The main purpose of the implementation was to serve as an evaluation tool for the semantics: as is described later, test programs to exercise various Bash features were ran on both Bash and Bash_{light}, and the results were compared. The implementation was particularly useful to investigate expressiveness, usability and similarity of Bash_{light} to Bash. The implementation took a similar level of effort to design and to realise as the grammar and operational semantics of the language did.

2.2 Tools and preparatory learning

Several concepts and languages were required for this project. The majority of the theoretical constructs which were required had already been covered in Part IB of the Computer Science Tripos. The new languages required were learned from tutorials, examples and aid from supervisors. A summary of the relevant and non-trivial concepts are included here. Also, in order to reduce the incidental work related to the project standard tools were used where possible.

Formal grammar

The grammar was defined as a context free grammar. A context free grammar is a grammar where every production is of the form

$$V \rightarrow W$$

where V is a single non-terminal and W is a string of terminals and non-terminals (possibly empty).

Operational semantics syntax

The operational semantics of a language is used to, in a mathematical manner, define how the execution of such a language should behave as a sequence of steps.

In a typical operational semantics there are two important constructs. The first is a reduction relation, which is generally is of the form

$$\text{environment} \times \text{expression} \rightarrow \text{environment} \times \text{expression}$$

where an environment represents the system state[1], and an expression is an expression in the language. The reduction relation indicates a valid reduction step from one valid expression to another. The second construct is rules, which specify

[1]For example system state includes variable bindings.

when a given reduction step is valid. Examples of rules are:

$$\frac{}{env \,/\, \{\, f \,;\} \longrightarrow env \,/\, \mathbf{g2}\, f} \quad \text{OP\_EXP\_GROUPING\_2\_1}$$

This rule is an axiom, as it has no preconditions.

$$\frac{env \,/\, f \longrightarrow env' \,/\, f'}{env \,/\, \mathbf{g2}\, f \longrightarrow env' \,/\, \mathbf{g2}\, f'} \quad \text{OP\_EXP\_GROUPING\_2\_2}$$

This rule is dependent: if the reduction above the line is valid then the one below the line is also valid.

Ott

Ott is an open source tool built in-house in the University of Cambridge Computer Lab[3]. It is used to define the grammar and operational semantics of a language or calculus, and can be used to generate Coq, HOL and Isabelle versions of the definition for automated theorem proving. This tool was used to write and generate Tex for the grammar and operational semantics.

The Ott syntax was not familiar to the author. A reasonable degree of effort was spent in attempting to use this tool (which is not yet fully developed) to express the desired grammars, and particular difficulty was encountered in expressing bracketing correctly.

OCaml

OCaml was a new language to the author, but not completely so as the basic concepts had been covered in the Part IA ML course. This language was particularly appropriate for the rapid development of the implementation of the operational semantics because of its pattern matching over arbitrary user expressions and its Unix interface module.

The OCamllex and OCamlyacc[2] syntaxes were not familiar to the author. These were used to lex and parse the input code (which was text) into the relevant

[2]OCaml implementations of the widely used Lex and Yacc tools.

abstract syntax tree.

No IDE was used in the development of the code, the code was simply compiled and linked from the Bash command line using the OCaml compiler on Linux.

Other tools

Subversion version 1.4.4 was used for version control. The Subversion project was hosted on a remote redundant machine and hence when updates were committed to the project it also provided backup.

LaTeX was used for writing the dissertation, and a standard template made available through the Computer Lab was utilised. The LaTeX syntax was not familiar to the author.

2.3 Testing

Though there are facilities in Ott to generate relevant input for computer aided theorem provers from the formal specification, it was determined to be outside the scope of the project to perform mathematical proofs on the language. In particular it was not possible to do a rigorous comparison between the language in its operational semantics and the OCaml implementation. However it was still important to attempt to, informally, illustrate the equivalence between the operational semantics and the implementation, and between Bash_{light} and Bash.

2.3.1 Unit testing

The implementation could be broken down into three main units: the lexer, the parser and interpreter. Each of these was tested individually, as follows:

Lexer

The lexer was tested to ensure it delineated the string inputs which categorise keyword or key character sequences into the appropriate tokens. For each required token

this was done by passing the relevant string for each token to the lexer and ensuring that the token returned was the correct one.

Parser

The parser was tested to ensure it returned the correct abstract syntax tree by passing it a string and comparing the abstract syntax tree it returned to a hand written tree which was human checked to be correct. This was done for all right-hand-sides in the grammar. For example the following code tested the parser with regards to the `case` statement:

```
let s1 =
  create_node (create_case (create_node (create_str "hello"))
               (create_ocaseexp_leaf
                (create_caseexp_cons (create_node(create_str "world"))
                (create_caseexp_leaf (create_node (create_str "hello")))
                )
                (create_node (create_int 0))
               )
              ) in
  (test_parser "case hello in ( world | hello ) 0 ;; esac\n" s1 13) false;
```

Effectively the function `test_parser` takes the string "case hello in (world | hello) 0 ;; esac\n" as one of its arguments, runs the parser on it and compares the resulting syntax tree with a hand coded one (`s1` above).

Interpreter

The interpreter applies an implementation of the reduction rules to abstract syntax trees, and it was tested to ensure it applied the rules consistently with the operational semantics. To do this, for each for each reduction rule in the operational semantics the following was done. A custom abstract syntax tree was created which represented the left hand side of the postcondition (result) reduction relation of a rule. A custom abstract syntax tree was created which represented the right hand side of the same relation (the result of a reduction step). The relevant reduction step was run on the first custom abstract syntax tree. The result of the reduction step was compared to the second custom abstract syntax tree and checked for equivalence. An example of this is the following:

```
let env1 = const_interactive_start_env in
  let env2 = env1 in
    let aexp1 = create_aexp_plus (create_aexp_int 2)(create_aexp_int 2) in
      let aexp2 = create_aexp_int 4 in
        let e1 = create_aritheval aexp1 in
          let e2 = create_aritheval aexp2 in
            let (env3, e3) = evaluate_step env1 e1 in
              (test_rule_env e2 e3 env2 env3 36 true);
```

This code tests the rule

$$\frac{env \,/\, aexp \;\longrightarrow\; env' \,/\, aexp'}{env \,/\, \$((\, aexp \,)) \;\longrightarrow\; env' \,/\, \$((\, aexp' \,))} \quad \text{OP\_EXP\_ARITH\_1}$$

The rule denotes that if an arithmetic expression inside an arithmetic expansion can reduce then the whole expression can reduce. The code above is testing to see if `$(( 2 + 2 ))` reduces to `$(( 4 ))`, given an environment `env1`. The environment should not be changed by the reduction of the arithmetic expression. As a whole this tests whether the implemented reduction relation (in the interpreter) matches the reduction relation in the operational semantics, for the rule OP_EXP_ARITH_1.

Testing each rule in this way combined with the unoptimised implementation permitted the desired argument to be offered[3] of the equivalence between the operational semantics and the implementation which was described above (for more detail see Section 3.4).

2.3.2 Integration testing

In order to test the system in its entirety various complete Bash_{light} programs were run. These were small chunks of code which had a verifiable result, either in terms of the value they returned (which could be verified programatically) or of an effect on the file system/terminal (which would have to be verified manually by the author). The tests exercised all sections of the program: the lexer, the parser and the interpreter. The following tests were performed:

[3] In Section 4.1.

```
exec 3 > ft ;
echo "a" 1 >& 3 ;
echo "2" 1 >& 3 ;
echo "b" 1 >& 3 ;
exec 3 <&- ;
exec 3 < ft ;
for (( y = 0 ; y < 3 ; y++ )) do
 read x 0 <& 3 ;
 case ${x} in ( a | e | i | o | u ) echo #( ${x} "- vowel" )# ;;
              ( 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 ) echo #( ${x} "- digit" )# ;;
              ( * ) echo #( ${x} "- consonant" )# ;; esac
done ;
exec 3 <&-
```

Figure 2.1: Bash$_{light}$ script which does file input, output and case matching

Mathematical functions

These functions were tested in order to evaluate certain properties of the language. Firstly, they were to examine recursive calls. This tests firstly the capability to define functions recursively and also the scoping of recursive calls. Other properties which they tested include function calls and arithmetic. The mathematical functions that were tested were the Factorial function, McCarthy's 91 function and Ackermann's function.

Code which changes the state of the system

These scripts tested the "impure" parts of the system. Important functionality resides in the file input/output functionality and redirections and in the system state (such as the present working directory). Hence the following parts of the system were deemed important to test: outputting to the terminal, creating a file descriptor (a redirection) and using it to output to a file, creating a file descriptor (a redirection) and using it to read input from a file, copying file descriptors, changing the present working directory and outputting the present working directory.

```
cd "/home/austin/tmp" ;
exec 3 > ft ;
echo "It was the best of times" 1 >& 3 ;
echo "it was the worst of times" 1 >& 3 ;
echo "it was the age of wisdom" 1 >& 3 ;
echo "it was the age of foolishness" 1 >& 3 ;
exec 3 <&- ;
exec 3 < ft ;
cd "/home/austin/tmp/subfolder" ;
read a 0 <& 3 ;
read b 0 <& 3 ;
read c 0 <& 3 ;
read d 0 <& 3 ;
echo #( ${a} ${b} ${c} ${d} )# > ft ;
pwd >> ft
```

Figure 2.2: Bash$_{light}$ script which does redirections and file input/output

The first script which was used to test impure parts of the system is shown in Figure 2.1. This script tests outputting to the terminal, creating file descriptors to read and write from the file and copying file descriptors. The value which should be printed to the terminal by this script is as follows:

a - vowel
2 - digit
b - consonant

The second script tested outputting to the terminal, creating file descriptors to read and write from the file, copying file descriptors, changing the present working directory and outputting the present working directory. This script is shown in Figure 2.2, and should result in the file */home/austin/tmp/subfolder/ft* containing the text

It was the best of times it was the worst of times it was the age of wisdom it was the age of foolishness
/home/austin/tmp/subfolder

Chapter 3

Implementation

In this chapter a detailed description of Bash$_{light}$ is given. Conceptually there are two main parts to the implementation of the project: the formal specification of Bash$_{light}$ and the OCaml code that implements the formal specification. The formal specification includes the grammar which describes the syntax of the language, the environment in which the Bash$_{light}$ expressions are reduced and the reduction rules which define the operational semantics. The OCaml implementation includes a parser to generate the abstract expressions defined by the formal grammar from some textual input, an interpreter which applies the reduction rules in the manner determined by the operational semantics and a test harness. The implementation can execute Bash$_{light}$ expressions and hence (with minor modifications to syntax) can execute Bash expressions.

3.1 Bash$_{light}$ syntax and grammar

The Bash manual does not give a formal semantics for Bash but at least gives an informal prose description. The manual defines even less concretely the grammar of Bash[1], and though it describes the syntax the actual grammar which is used is

[1]The only notable exception to this is that Bash does have a concept of "compound" and "simple" commands (*Bash manual section 3.2.4*) where compound commands consist of looping constructs, conditional constructs and grouping commands, and simple commands are basically everything else. These different expression types are used in a few syntactic definitions but are largely un-useful.

implicit[2]. This section describes the Bash$_{light}$ grammar and discusses some of the more complex features of the Bash syntax, and the design decision choices.

3.1.1 Grammar overview

The Ott grammar for Bash$_{light}$ expressions and sequences of expressions is given in Figure 3.1 (the full grammar can be seen in Appendix B). This includes the main syntactic constructs used by Bash$_{light}$. An informal description of non-obvious productions is given below.

Sequences are implemented by the `f` non-terminal. In Bash sequences were part of the "lists of commands" construct (*Bash manual section 3.2.3*). Sequences of `expr` are not included in `expr` because if they were then some expressions could have multiple parses[3].

The `elist` expression is used to indicate a list of arguments. There are several circumstances in which this might be used. When calling a function the `|( elist )|` syntax is used, and the first value (if it is a string) is taken as the function name to be called. The `elist` expression is also used to give a set of values over which a variable in a `for` loop should range during iterations of the loop in the `for str in elist do f done` expression. The final use for it is within the `#( elist )#` syntax, which indicates a list of values which should be concatenated together into a single string.

The `for str do f done` expression is equivalent to `for str in $@ do f done`, where `$@` expands to an `elist` which contains all the function parameters, in the closest function parameter scope.

---

[2]An example of this is the definition of the `until` loop:

until test-commands; do consequent-commands; done

This does show the syntax of the `until` loop, but the definition is not really suitable for a formal grammar as it doesn't specify which expressions can be used in the `test-commands` or `consequent-commands` positions.

[3]e.g. e_1 `;` e_2 `;` e_3 can be parsed as e_1 `; (`e_2 `;` e_3`)` or `(`e_1 `;` e_2`);` e_3.

$$
\begin{array}{lll}
f & ::= & \\
 & | \quad e; f & \\
 & | \quad e & \\
 & | \quad e_1 \ ; \ .. \ e_n \ ; \ f & \mathsf{M} \\
 & & \\
expr,\ e & ::= & \\
 & | \quad constant & \\
 & | \quad |(\ elist\ )| & \\
 & | \quad \textbf{until}\ f_1\ \textbf{do}\ f_2\ \textbf{done} & \\
 & | \quad \textbf{while}\ f_1\ \textbf{do}\ f_2\ \textbf{done} & \\
 & | \quad \textbf{for}\ str\ \textbf{do}\ f\ \textbf{done} & \\
 & | \quad \textbf{for}\ str\ \textbf{in}\ elist\ \textbf{do}\ f\ \textbf{done} & \\
 & | \quad \textbf{for}\ ((\ e\ ;\ aexp_1\ ;\ aexp_2\ ))\ \textbf{do}\ f'\ \textbf{done} & \\
 & | \quad \textbf{if}\ f_1\ \textbf{then}\ f_2\ \textbf{else}\ f_3\ \textbf{fi} & \\
 & | \quad \textbf{case}\ f\ \textbf{in}\ oce\ \textbf{esac} & \\
 & | \quad [[\ condexp\ ]] & \\
 & | \quad (\ f\ ;\ ) & \\
 & | \quad \{\ f\ ;\ \} & \\
 & | \quad \textbf{g1}\ f & \mathsf{M} \\
 & | \quad \textbf{g2}\ f & \mathsf{M} \\
 & | \quad \$\{\ str\ \} & \\
 & | \quad \$\{\ intlit\ \} & \\
 & | \quad \$((\ aexp\ )) & \\
 & | \quad str\ =\ e & \\
 & | \quad \#(\ elist\ )\# & \\
 & | \quad (e) & \mathsf{M} \\
 & | \quad \text{“}str\text{”} & \mathsf{M} \\
 & | \quad \textbf{function}\ str\ e & \\
 & | \quad str\ ()\ e & \\
 & | \quad \textbf{function}\ str\ ()\ e & \\
 & | \quad \textbf{fun}\ e & \mathsf{M} \\
 & | \quad rdexp_1\ \textbf{cd}\ rdexp_2\ e\ rdexp_3 & \\
 & | \quad rdexp_1\ \textbf{pwd}\ rdexp_2 & \\
 & | \quad rdexp_1\ \textbf{echo}\ rdexp_2\ e\ rdexp_3 & \\
 & | \quad \textbf{exec}\ rdexp & \\
 & | \quad rdexp_1\ \textbf{read}\ rdexp_2\ str\ rdexp_3 &
\end{array}
$$

Figure 3.1: Bash_{light} syntax and grammar

Two different grouping commands (which were discussed in Section 2.1.1) are included in Bash_{light}. These are the constructs `( f ;)` and `{ f ;}`.

Conditional expressions are surrounded by the `[[..]]` syntax, and this was reflected in Bash_{light}.

Functions can be defined using several different syntaxes. These are all included in Bash_{light} and comprise of the syntaxes `function str e`, `str () e` and `function str () e`. These all denote defining a function with name `str` and body `e`.

Some Bash builtin functions are included in Bash_{light}, notably `cd`, `pwd`, `echo`, `exec` and `read`. The `rdexp` expressions which are included in the definitions of the builtin functions are places where redirections can be included.

Certain entries in the grammar are marked with an `M` which indicates that they are metaproductions and not a part of the actual grammar. Metaproductions were generally used to define trivial syntactic sugar.

3.1.2 Suitability of a formal grammar

In Bash, to the best of the author's understanding, all code is kept as a string and there is no real notion of a formal grammar. It appears that code is not parsed or interpreted until the point it is about to be used. When the code is used various rewrites (textual substitutions) are performed, interleaved with applications of reduction rules. Which rewrites are applied when depends on the context.

Representing these string-orientated actions would be complex and possibly equivalent to writing a grammar and operational semantics which could represent regular expressions. Hence the majority of the string-orientated features (most notably the rewrites) were ommited. This made it possible to represent Bash in a formal grammar.

There was, however, one string focused feature which was implemented, and the string orientated nature needed to be modeled in this case. The feature was how Bash treated dereferencing in situations where whether the value dereferenced

was a number or a string was important. If the dereferenced value is of the wrong 'type', e.g.

```
x=hello;
$(( ${x} + 1 ))
```

then a predefined value is substituted for the value with the incorrect 'type' (in the above example `${x}` would dereference to `0`). An example of a rule which does this dynamic rewriting is the following:

$$\frac{}{env\,\{\,\mathbf{S}\,\mathbf{ninc}\,str\;\mapsto\;c\,\}\,/\,\$\{\,str\,\}\;\longrightarrow\;env\,\{\,\mathbf{S}\,\mathbf{ninc}\,str\;\mapsto\;c\,\}\,/\,\mathbf{blank\_string}}\;\text{OP\_EXP\_DEREF\_2}$$

The rule denotes the following: if a variable named `str` is dereferenced, and there is not any such variable in the store (`S` represents the store and `ninc` denotes "not includes") then the dereferenced value is just a blank string.

There were two main options as to how this feature could be implemented.

Typing: In some languages typing is used to ensure that code such as `x = "hello"; x = x + 1` is considered malformed. In Bash store locations do not appear to keep type information - everything is kept as a string. The use of typing would enable the system to reject the above code as malformed. However that does not reflect Bash's behaviour where values are dynamically changed if they are not of the correct 'type'.

Dynamic value rewriting It appears that Bash dynamically changes values which are dereferenced in order that operations do not occur on values of the wrong 'type'. Despite appearances this is not dynamic typing as it is (to the best of the author's understanding) simply examining the string to see if it contains only certain characters (e.g. numerals) and if so then using it as the required 'type' and if not then substituting a default value of the correct 'type'. A possible modelling method was to internally keep track of some primitive types and to perform the dynamic substitution of values in the operational semantics.

The option chosen was dynamic value rewriting, as it was closer to the semantics of Bash. The primitive types that were included were integers, strings and

Booleans.

It might be possible to represent the string focus in a formal grammar and operational semantics, but it would be particularly complex (for more details see Section ??).

3.1.3 Grammatical ambiguity

The description which the Bash manual gives of the syntax leaves ambiguity in certain cases as to which grammatical structure a given string should be parsed. One case where this was relevant to Bash_{light} is with the builtin functions `cd` and `echo` and their parameters.

According to the Bash manual[1] *"The ... redirection operators may precede or appear anywhere within a simple command or may follow a command"*. The ambiguity is in distinguishing the redirections from other parameters to the function[4] for `cd` and `echo`[5]. This ambiguity can be seen in the following code:

```
echo hello 1>&3 world 1>&4 says 1>&5 me
```

The above is a single command, `echo`, with seven 'parameters'. The code results in printing *"hello world says me"* to the place represented by file descriptor five. Due to the fact that the redirections can be interspersed with function parameters it is difficult to differentiate between the two.

In Bash the decision about whether a parameter is a redirection or not appears to be done using string based recognition. Writing a grammar which could have reflected this would have been very hard, perhaps equivalent to writing a grammar and semantics which could represent regular expressions. Hence in the Bash_{light} grammar this is dealt with by restricting where redirections can be present with a builtin function, for example:

[4]For `cd` the other parameter is the name of the directory to which to change. For `echo` the parameters are strings to be printed to standard output.

[5]In Bash_{light} redirections were only permitted with builtin functions, but only the builtin functions `cd` and `echo` had parameters besides redirections. Hence these were the only expressions for which there could be ambiguity in distinguishing redirections from non-redirection parameters

$$
\begin{array}{lll}
env & ::= & \\
 & | & s : fun\_s : funslist : term : fsys : ioer : envvars \\
 & | & env \{ envchanges \}
\end{array}
$$

Figure 3.2: Bash_{light} environment

$$
\begin{array}{lll}
expr,\ e & ::= & \\
 & | & rdexp_1 \ \textbf{echo} \ rdexp_2 \ e \ rdexp_3
\end{array}
$$

3.2 Environment

The environment is the formal representation of the system state. In many languages the only state required is a store, but Bash requires a more complex state. Figure 3.2 shows the representation in the grammar of the system state (where the different parts of the environment are separated by colons). The first production shows the various parts of the environment, which are described below. The second production is used in order to abbreviate the first, in that one can simply give an environment name and then specify features of it which are important, rather than all of it. This production is effectively a record syntax.

Variables are represented by the `s` non-terminal. The standard way to implement state is a partial function from variable names to values. Rather than a single function, the `s` non-terminal is a list of such partial functions, which permits scoping to be easily represented and implemented.

Function parameters (the arguments which are passed to a function) are represented in the `fun_s` non-terminal. Similarly to the `s` non-terminal the `fun_s` non-terminal is represented as a list of partial functions (from parameter names to values).

Functions can be defined dynamically by the user in Bash, and hence there must be some system state mapping function names to expressions which are the body of the function. Similarly to the `s` non-terminal the `funslist` non-terminal is represented as a list of partial functions (from function names to `expr` expressions).

The terminal is represented by the `term` non-terminal symbol and is effectively just a list of strings which have been printed to the terminal.

The file system is represented by the `fsys` non-terminal. This is a coarse, pure representation of the file system as a partial function from strings (which are path and file name) to strings (which are file contents). The file system functionality is given mainly by meta operations[6]. This representation is discussed further below.

Redirections are represented by the `ioer` non-terminal (standing for `Input Output ERror`). Similarly to the `s` non-terminal the `ioer` non-terminal is represented as a list of partial functions (from file descriptors to either a file or the terminal).

Environment variables are the final piece of state that need to be stored, and are represented by the `envvars` non-terminal and is a list of environment variables. The operation of Bash is controlled by many environment variables (*Bash manual section 5*). The only of these which was implemented was the present working directory.

The environment representation can be seen in use in the OP_EXP_grouping_1_3 rule:

$$[\,mem\,]\;,\; s : fun\_s : [funs], funslist : term : fsys : [\,ioersub\,]\;,\; ioer : envvarssub\;,\; envvars \,/\, \mathbf{g1}\; c$$
$$\longrightarrow\; s : fun\_s : funslist : term : fsys : ioer : envvars \,/\, c$$

This rule deals with the case where the expression inside a ‘grouping command one’ has been reduced to a constant. The semantics of grouping command one denote that any changes to the environment made by the reduction of an expression within the grouping command should not persist after evaluation of the grouping command. This rule states that, given the expression with the ‘grouping command one’ is a constant (i.e. evaluation within the grouping command has been completed) then the head of the list must be removed for each part of the environment which is modifiable by the user[7]. Since The state was duplicated when evaluation of the

[6] For more detail on meta operations see Appendix A.3.

[7] Note that the comma used in the `s`, `funslist`, `ioer` and `envvars` non-terminals works as a cons operator.

grouping command started then this has the effect of state reverting to how it was before evaluation of the grouping command.

For more details on the syntax of the environment representation, particularly with respect to the abbreviations used, see Appendix A.2.

Initial configuration

At the beginning of execution a default environmental configuration is used. Most of the above non-terminals are effectively empty lists (denoting that there are no variables assigned, no redirections defined, etc.). However several parts of state can have useful values in the initial configuration:

There can be function parameters defined in the initial configuration if the shell is being run in batch mode[8]. These initial parameters are the values which are passed to the script when it is being called.

Redirections always have an initial configuration to define standard input, output and error as:

$0 \longmapsto$ terminal, in

which is stdin

$1 \longmapsto$ terminal, out

which is stdout

$2 \longmapsto$ terminal, err

which is stderr

The final part of the initial configuration is the environmental variables. This includes a present working directory, which is the directory from which Bash_{light} was invoked.

[8]For more information on batch mode, as opposed to interactive mode, see Section 3.4.2

File system and terminal representation

Being a scripting language and often used for system administration Bash requires many different impure file access operations. The impure operations required for Bash_{light} were file access (reading and writing) and also printing to the terminal.

These operations could be represented monadically. One primary use of Monads (as in functional languages such as Haskell) is to provide a pure description of what input/output is required. This permits the programmer to program in a pure manner until some given point where all impure operations are done at the same time and flushed to the file system. This would require a detailed representation of the file system, the terminal, and of any required actions.

Alternatively the file system and terminal could be represented by modelling. A coarse abstract model of the file system and the terminal could be designed, and the details of impure operations could be described in an informal manner using meta operations. This was the chosen option since designing an accurate monadic representation of the file system would be complex and was not one of the main aims of the project.

3.3 Operational semantics

From a formal specification perspective the main interest resides in defining how Bash behaves. The main features from a behavioural point of view are included below. The full operational semantics (and the definitive description of all behaviour) is included in Appendix B.

3.3.1 Scoping

Recall from Section 2.1.2 that the functions defined by the expressions `g() { x=1 ;}` and `g() ( x=1 ;)` have different semantics: after invoking the first the value of `x` is changed to `1`, whilst after invoking the second the value of `x` remains unchanged. This exemplifies the unusual and expressive nature of the scoping semantics. In Bash scoping is controlled in two ways.

$$\frac{}{[\,mem\,]\;,\; s : fun\_s : [funs], funslist : term : fsys : [\,ioersub\,]\;,\; ioer : envvarssub\;,\; envvars \,/\, (\,f\,;\,) \longrightarrow [\,mem\,]\;,\; [\,mem\,]\;,\; s : fun\_s : [funs], [funs], funslist : term : fsys : [\,ioersub\,]\;,\; [\,ioersub\,]\;,\; ioer : envvarssub\;,\; envvarssub\;,\; envvars} \quad \text{OP\_EXP\_GROUPING\_1\_1}$$

$$\frac{}{[\,mem\,]\;,\; s : fun\_s : [funs], funslist : term : fsys : [\,ioersub\,]\;,\; ioer : envvarssub\;,\; envvars \,/\, \mathbf{g1}\; c \longrightarrow s : fun\_s : funslist : term : fsys : ioer : envvars \,/\, c} \quad \text{OP\_EXP\_GROUPING\_1\_3}$$

Figure 3.3: Reduction rules for grouping command one

$$\frac{}{env \,/\, \{\,f\,;\,\} \longrightarrow env \,/\, \mathbf{g2}\, f} \quad \text{OP\_EXP\_GROUPING\_2\_1}$$

$$\frac{env \,/\, f \longrightarrow env' \,/\, f'}{env \,/\, \mathbf{g2}\, f \longrightarrow env' \,/\, \mathbf{g2}\, f'} \quad \text{OP\_EXP\_GROUPING\_2\_2}$$

$$\frac{}{env \,/\, \mathbf{g2}\; c \longrightarrow env \,/\, c} \quad \text{OP\_EXP\_GROUPING\_2\_3}$$

Figure 3.4: Reduction rules for grouping command two

Grouping commands

In Bash grouping commands provide the main scoping functionality. Bash is a dynamically scoped language, but the example in Section 2.1.2 (with respect to the different definitions of g) demonstrates how changes to the environment can persist or not depending on which grouping command is used. This effectively means that the user can control whether they are using dynamic or lexical scoping.

The first grouping command is defined using the `(...;)` syntax. The semantics of this grouping command is that any changes to the environment which occur as a result of a reduction of an expression within the `(...;)` syntax should not persist after finishing evaluating the grouping command.

Figure 3.3 shows some of the rules pertaining to the reduction of the first grouping command construct. This shows that when a 'grouping command one' starts to be evaluated the user-modifiable state is duplicated (OP_EXP_GROUPING_1_1), and when a 'grouping command one' is exited the (possibly modified) duplications are removed (OP_EXP_GROUPING_1_3). This has the effect that expressions within the

grouping command have access to the same environment as expression containing the grouping command, but any changes made by these expression are made to a local copy.

The second grouping command is defined using the `{...;}` syntax. The semantics of this grouping command is that any changes to the environment as a result of a within `{...;}` should. In effect this grouping command acts in a transparent manner with regards to scope.

Figure 3.4 shows some of the rules pertaining to the reduction of the second grouping command. Note that no copies of state are made (OP_EXP_GROUPING_2_1) and that any modifications to the state which occur during the reduction of the `f` expression within the grouping command persist (OP_EXP_GROUPING_2_3).

One important feature to note is how grouping commands affect function invocation. Functions are defined using the production:

$$\begin{array}{lll} \mathit{expr},\ e & ::= & \\ & \mid & \textbf{function}\ \mathit{str}\ ()\ e \end{array}$$

or some variation therein. The body of the function is an expression. Most users when writing a function will write it so that the function body is within some form of brackets[9], e.g. `g () { x = 1 ;}`. This is actually using a grouping command, and which brackets are used determines the grouping command used. Hence many users will be making decisions which affect function (and subprocess) invocation without realising it.

Function calls

$$\frac{}{\mathit{env}\,\{\,\mathbf{F\_S}\,=\,\mathit{fun\_s}\,,\,\mathbf{F\ inc\ funs},\mathit{str}\,\mapsto\,e\,\}\,/\,|(\ \mathit{str}\ c_1\ ..\ c_n\ )|\ \longrightarrow\ \mathit{env}\,\{\,\mathbf{F\_S}\,=\ \ [\ \mathit{int}_1\,\mapsto\,c_1\ ,\ ..\ \mathit{int}_n\,\mapsto\,c_n\ ,\ ]\ ,\ \mathit{fun\_s}\,\}\,/\,\mathbf{fun}\ e}\ \text{OP\_EXP\_ELIST\_2}$$

[9] Whilst a user could write `g () x = 1` it is unlikely for them to do so given programming conventions

The other way in which scope is affected is function calls. When a function is called with a series of arguments these arguments are used as dereference only parameters, within the body of the function. These values are only available within the main function body - if the body makes further function calls the values are not available within those functions' bodies. However if the body makes subprocess invocations (i.e. if it includes grouping commands) then the function parameters are still available.

This property of function calls is demonstrated in the rule above, which denotes that when a call is made to function `str`, and the state which holds the function bindings (**F**) includes a function named `str` then the body of the function is evaluated. Note also that the values which are given after the function names are bound to function parameters (the state which holds function parameter bindings is **F_S**).

These two different forms of scoping allow a large degree of user freedom and control of system state scoping.

3.3.2 Recursive and higher order functions

The facilities for defining and calling functions have already been touched upon, but no mention has been made of recursive functions. During evaluation, if Bash encounters a string that might be a function, the environment is examined to determine whether a function with that name exists, and if so the string in replaced with the body of the function.

This allows functions to be defined recursively without any extra facilities in the language - the string name of a function can be included in the function body itself. An example of this is given in the following code:

```
fact () { if $(( ${1} != 0 )) then
            y = $(( ${1} - 1 )) ;
            x = $(( x * ${1} )) ;
            |( fact ${y} )|
```

```
else 0
fi ;}
```

Moreover, higher order functions can also be handled without modification to the language - to pass function A as a parameter to function B all that is required is to pass the name of function A as a string parameter to function B. That parameter can then be dereferenced and used in a position where a function could be called, e.g.

```
g () { echo "hello world" ;}
apply () { ${1} ;}
|( apply g )|
```

would result in "hello world" being printed to the terminal.

The functionality for recursive and higher order functions is a part of Bash but is not standard usage.

3.3.3 Redirections

One useful and oft used (in its basic form) group of constructs in Bash are redirections. These permit an expressive means to change where standard input, output and error come from/go to, and hence control input and output from files and the terminal. Unfortunately the syntax is rather non-intuitive, and hence many users of Bash will not have progressed passed the basic cases such as:

```
echo "hello world" >fname
```

or

```
exec 3>fname
...
echo "hello world" 1>&3
```

The redirections functionality is much more powerful than this. It effectively allows one to redirect any of standard input, output or error in an arbitrary manner, at any point in an expression. These redirections can be from/to either a specified file, the terminal or a file descriptor which has already been specified. These redirections

Syntax	Informal semantics
`int1 > str`	Create a file descriptor `int1` for output, for file `str` (truncating it to zero length)
`int1 >> str`	Create a file descriptor `int1` for appending output, for file `str`
`int1 < str`	Create a file descriptor `int1` for input (reading from), for file `str`
`> str`	Set standard output to go to file `str` (truncating it to zero length)
`>> str`	Set standard output to append to file `str`
`< str`	Set standard input to come from file `str`
`&> str`	Set standard output and standard error to go to file `str` (truncating it to zero length)
`int1 >& int2`	Copy output file descriptor `int2` into file descriptor `int1`
`int1 <& int2`	Copy input file descriptor `int2` into file descriptor `int1`
`int1 <> str`	Create a file descriptor `int1` for random access, for file `str`
`int1 <&-`	Close file descriptor `int1`

Figure 3.5: Redirections: their syntax and their semantics

can either persist throughout the code or only apply to the expression to which they are attached[10]. Figure 3.5 gives an informal version of the operational semantics for the redirections which were implemented in Bash_{light}. One redirection is singled out and described in more detail as follows.

Opening a file for output on descriptor `int` is represented by the syntax

```
int > str
```

The relevant reduction rule regarding this production is:

OP_RDEXP_FD_OUT_TO_

$$env\,\{\,\mathbf{I} = [\,ioersub\,],\ ioer\,\}\,/\,int\ >\ str\ rdexp$$
$$\longrightarrow\ env\,\{\,\mathbf{FS\ inc\ fsval(str)} \mapsto \mathbf{blank\_string}\,,\ \mathbf{I} = [\,ioersub\,,\,\{\,int \mapsto \mathbf{fsval}(\,str\,)\,,\,\mathbf{out}\,\}\,],\ ioer\,\}\,/\,rdexp$$

This rule describes the effect of this redirection. The string `str` denotes a file name, either as a relative or an absolute address. The meta operation `fsval` is used to give the absolute address of a file given either a relative or absolute address as it's input. The initial environment simply defines that there is some state about redirections

[10] A redirection only has effect for the expression to which it is attached. This leads to redirections being non-persistent unless the expression to which it is attached is an `exec`.

(but that the contents of that definition are unimportant). After the reduction has taken place several changes will have been made to the environment. Firstly the file `str` is truncated to zero length (denoted by the syntax **FS inc fsval(str) $\mapsto$ blank string**). Secondly the redirection { `int` $\mapsto$ `fsval(str), out` } is added to the environment. This altogether has the effect that one can start writing to the blank file `str` on file descriptor `int`.

An interesting feature of redirections is their temporal nature. The persistance of redirections has already been addressed, but another interesting temporal effect is that redirections all appear to be evaluated before the evaluation of the expression to which they are attached. This is true even if they are defined in different places in an expression, and is illustrated by the code:

```
echo hello >file1 world >file2 says >file3 me
```

The redirections `>file1`, `>file2` and `>file3` redirect standard output to go to `file1`, `file2` and `file3` respectively. After this code has been executed these three files will have been created. The first two will be of zero length, and the third will contain the text "hello world says me". This demonstrates the property that all the redirections are evaluated before the main expression is evaluated. Also the redirections seem to be evaluated in the order they appear in the expression, from left to right.

Redirections are an archetypal Bash construct, and as shown provide a powerful tool for controlling standard input, output and error. The semantics of redirections is not conceptually difficult, but the syntax provides little indication of the semantics of the individual constructs and hence this operational semantics should increase clarity of understanding with respect to redirections.

3.4 OCaml implementation

The OCaml implementation consisted of a working shell and a test harness. The shell was made up of a parser which generated Bash$_{light}$ expressions given some code, and an interpreter which applied the Bash$_{light}$ reduction rules to the expressions. The shell was useful during the development of the formal semantics as it provided a way

to assess the 'real life' implications of the rules. The test harness was used primarily to ensure that the code correctly implemented the formal specification, but also that Bash_{light} and Bash code gave the same results in a series of tests. The following section discusses various aspects of the implementation.

3.4.1 Transcriptive implementation

The structure of the code is almost transcribed from the grammar and operational semantics, in that it lexes and parses the input string to accept only those as defined in the grammar and then applies reduction steps one at a time to the abstract syntax tree, in the order specified by the operational semantics. This implementation style may seem simplistic, but there were several advantages to its use which are described below. An example of the transcriptive nature of the code is shown below by three rules and the OCaml code which implements them.

$$\frac{env \,/\, condexp_1 \longrightarrow env' \,/\, condexp_1'}{env \,/\, condexp_1 \,\&\&\, condexp_2 \longrightarrow env' \,/\, condexp_1' \,\&\&\, condexp_2} \quad \text{OP\_CONDEXP\_AND\_1}$$

$$\frac{}{env \,/\, \mathbf{true} \,\&\&\, condexp_2 \longrightarrow env \,/\, condexp_2} \quad \text{OP\_CONDEXP\_AND\_2}$$

$$\frac{}{env \,/\, \mathbf{false} \,\&\&\, condexp_2 \longrightarrow env \,/\, \mathbf{false}} \quad \text{OP\_CONDEXP\_AND\_3}$$

```
evaluate_condexp_step env ce1 =
  match ce1 with
      ...
      | And (Val(true), ce3) -> (env, ce3)
      | And (Val(false), _) -> (env, Val(false))
      | And (ce2, ce3) -> let (env1, ce4) = evaluate_condexp_step env ce2 in
                          (env1, And(ce4,ce3))
```

This implements the same functionality, including the small step operational semantics in the `evaluate_condexp_step` function call. This is done by the single reduction step on the first sub conditional expression, as required in the rule OP_CONDEXP_AND_1. Also, the passing through of changes to the environment required for the rule OP_CONDEXP_AND_1 can be seen in the third matching expression.

Since designing the code for optimum execution was not a requirement, optimising it for human understandability was possible. This was particularly useful in order to enable the casual reader to compare the operational semantics to the implementation and informally satisfy themselves that the two do the same thing. This was similarly useful for when the implementation was being used as an evaluative tool, in that one could easily understand what the implementation was doing and how any unexpected or incorrect behaviour linked directly to the operational semantics.

3.4.2 Interactive and batch modes

The user can use the OCaml implementation in either interactive or batch mode. There are differences in semantics between interactive and batch modes, but these are controlled by the default values of certain environmental variables[11]. However, since in the early stages of the project it was decided to not recreate a large proportion of the environment variables. Hence there are no differences between the operational semantics of the interactive and batch modes in Bash_{light}.

3.4.3 Test harness

The primary aim of the test harness was to ensure that the code correctly implemented the formal specification. This was achieved by testing the interpreter in the manner described in Section 2.3. The test harness also verified the results of Bash code verses the results of Bash_{light} code, for various functions such as McCarthy's 91 function. The test harness implementation was significant, and was longer in terms of lines of code than the implementation of the shell itself. The testing is discussed in more detail in Section 4.1.

3.5 Summary

In the above chapter the formal semantics of Bash, including the syntax, formal grammar, and operational semantics were presented. A particular focus was given to two of the most interesting Bash features: redirections and scoping. In recapitulation,

[11] Hence one could make an interactive shell behave as a batch one by redefining certain environmental variables.

redirections provide an expressive functionality for standard input, output and error manipulation, and the scoping facilities in Bash are expressive and permit the user to employ lexical or dynamic scoping, or a mixture of the two.

Chapter 4

Evaluation

"The principal aim of the project is to investigate the semantics of Bash and to try to document and reproduce them in operational semantics and an OCaml implementation" Chapter 1

Bash is a very complex language, and this is reflected in the grammar and rules in the previous chapter. Several methods were used to evaluate the system, in order to investigate different features. Various tests were performed to informally show equivalence between the implementation and the formal specification, and between the behaviour of Bash and the Bash_{light} implementation. Also the Bash_{light} shell was evaluated with respects to its usefulness as a shell. Additionally some assessment of the tools used is presented.

4.1 Testing

Testing was the main technique which was used to evaluate the system, with an emphasis on two specific properties.

The first property was that the code correctly implemented the formal specification. This assurance permitted an argument of the equivalence between the formal specification and the OCaml implementation to be given. This testing was defined in Section 2.3. A test case was written for every token in the lexer, every production in the parser and every reduction step in the interpreter. The argument

is as follows. The testing shows that for a given rule in the formal specification the interpreter correctly reduces a certain number of hand coded expressions. This is taken to informally show that the interpreter correctly implements the reduction rules (it is informal as not all possible expressions which could be reduced are tested, and no logical proof of equivalence is provided). Given that this is done for all reduction rules, the interpreter is taken to correctly implement all of the reduction rules and hence the formal specification, when testing of the parser and lexer are included.

The second property was that evaluation of Bash_{light} and Bash code gave the same results in a series of cases. The factorial function was run for input values 0-15. McCarthy's 91 function was run for input values 0-105. Bash and Bash_{light} implementations of the functions shown in Figures 2.1 and 2.2 were both run, and it was manually checked to ensure that they gave the same result. This gave evidence of equivalence between Bash and Bash_{light}.

4.2 Syntax and semantics

Given the rather unclear nature of the definition of Bash (the perhaps one true definition being the old one for compilers of "it is defined by what it outputs") the similarity between Bash and Bash_{light} syntax and semantics is a more subjective evaluative criterion.

The changes of the syntax for Bash_{light} from Bash are detailed in Appendix A.1.1. Most of these changes were made due to limitations in the compiler and lexer tools that were being used, and were contained to the removing of a couple key characters from a few productions, and adding them to another. These could easily be detailed in a change log, and though it would be tedious to rewrite shell scripts to implement the change it is not a large one for those using the shell interactively.

From a design point of view the most major of the changes to the syntax was not implementing the string oriented nature of Bash. This string oriented nature included how Bash takes string values and treats them in different ways depending on several different factors, including bracketing, quoting and the

expression in which it is. This was not implemented due to the complexity involved.

4.3 Expressiveness of the language

Another subjective criterion is the 'expressiveness' of the language. This has to do with what types of programs Bash_{light} is able to express, and whether those types of programs would be useful to the people who use Bash. Figures 4.1 and 4.2 are taken from the set of scripts which are used on a Linux machine, and typify the type of short, repetitive process which Bash scripts traditionally implement.

Figure 4.1 demonstrates how the majority of constructs which are required by the script are present in the language. However, it would not however be possible to completely rewrite this script[1] in Bash_{light} - there are many features which are required by this script which are not expressible in Bash_{light}, including *cat*, *awk*, *ln* as well as various string execution features which were discussed in Section 4.2.

In examining these and other Bash scripts not presented here it appears that the primary features lacking in the language are the ability to use various external programs and pipes. The extension of pipes to the language would not require an excessive amount of work as most of the groundwork has been laid in the dealing with standard input and output for redirections. In order to use the external programs some way to interface with existing implementations would need to be developed. Similarly to pipes, the groundwork for this interface has already been laid by redirections.

Hence only a few extensions would need to be implemented in order to make Bash_{light} useable as a shell.

4.4 Reproduction and description of Bash

One of the principal aims of the project was to represent the operation of Bash in a formal manner, in order to clarify it. Some features were of particular interest, and they are presented here.

[1] Or the script in Figure 4.2.

```
#!/bin/bash
set -e

if test -e /boot/memtest86+.bin ; then
  echo "Found memtest86+ image: /boot/memtest86+.bin" >&2
  cat << EOF
menuentry "Memory test (memtest86+)" {
linux ${GRUB_DRIVE_BOOT}/memtest86+.bin
}
EOF
fi
```

Figure 4.1: /etc/grub.d/20_memtest86+

Scoping

The way that the scoping works in Bash is twofold. Firstly scoping comes from grouping commands and secondly it comes from function calls. However, the fact that variable assignments inside the body of a function will persist unless surrounded by `( ... ;)` syntax was something which was unintuitive for those familiar with more mainstream languages, including the author and supervisor. This freedom in control of the scoping permits the programmer much more flexibility in how variable assignments propagate. This feature, if used extensively, could seriously degrade the readability of any code. However since Bash scripts are often written by a single administrator to be run repeatedly and changed infrequently, it could be useful.

Redirections

The syntax of redirections is dense and the documentation sparse on how they should behave, and hence a reasonable degree of difficulty was encountered in attempting to write an operational semantics for them. Hopefully this clarification will be of use to novice and intermediate Bash programmers.

```
#!/bin/bash

. /lib/lsb/init-functions

if [ ! -d /sys/bus/pnp/devices ]; then exit; fi

cd /sys/bus/pnp/devices

case $1 in
    start|restart|reload|force-reload)
    log_begin_msg "Doing Wacom setup..."
    for x in *; do
PORT=unknown;
for y in `cat $x/id`; do
    case "$y" in
WACf008*|WACf007|WACf006*|WACf005*|WACf004*)
PORT=/dev/`echo $x/tty:* | awk -F: '{print $3}'`
ln -sf $PORT /dev/input/wacom
;;
    esac
done
    done
    log_end_msg 0;
    ;;
    stop)
    exit 0;
    ;;
esac
```

Figure 4.2: /etc/rc3.d/S10xserver-xorg-input-wacom

```
x=1 ;
fact () { if $(( ${1} != 0 )) then
            y = $(( ${1} - 1 )) ;
            x = $(( x * ${1} )) ;
            |( fact ${y} )|
          else 0
          fi ;} ;
|( fact ${1} )| ;
echo ${x}
```

Figure 4.3: The factorial function in Bash$_{light}$

```
x=1
fact () { if [ ${1} != 0 ]; then
            y=$((${1}-1))
            x=$((x*${1}))
            fact ${y}
          fi; }
fact ${1} ;
echo ${x}
```

Figure 4.4: The factorial function in Bash

4.5 Examples

As a showcase of the language some examples are presented here, for the most-part with their comparative Bash code.

Factorial function

Figures 4.3 and 4.4 show the code for the factorial function in Bash$_{light}$ and Bash respectively. The code is essentially the same between the two. One difference to note is that Bash$_{light}$ uses a slightly different condition evaluation.

```
r1 = 0 ;
mccarthy ()  { if $(( ${1} > 100 )) then
                 r1 = $(( ${1} - 10 ))
               else t1 = $(( ${1} + 11 )) ;
                    |( mccarthy ${t1} )| ;
                    |( mccarthy ${r1} )|
               fi ;} ;
|( mccarthy ${1} )| ;
echo ${r1}
```

Figure 4.5: McCarthy's 91 function in Bash_{light}

```
r1=0
mccarthy () { cmp=$((${1}>100))
              if [ ${cmp} != 0 ]; then
                r1=$((${1}-10))
              else t1=$((${1}+11))
                   mccarthy ${t1}
                   mccarthy ${r1}
              fi; }
mccarthy ${i}
echo ${r1}
```

Figure 4.6: McCarthy's 91 function in Bash

McCarthy's 91 function

Figures 4.5 and 4.6 show the code for McCarthy's 91 function in Bash_{light} and Bash respectively. An interesting difference to note is the requirement of semicolons at the end of lines in Figure 4.5 but not in Figure 4.6. This is due to the fact that in Bash one can replace semicolons with newlines.

Ackermann's function

Figures 4.7 and 4.8 show the code for Ackermann's function in Bash_{light} and Bash respectively. One difference to note is the use of the `|( ... )|` notation, to indicate lists of values which are grouped together. Another difference is the

```
r1 = 0 ;
ackermann ()  { if $(( ${1} == 0 )) then
                  r1 = $(( ${2} + 1 ))
                else ( if $(( ${1} > 0 && ${2} == 0 )) then
                         t1 = $(( ${1} - 1 )) ;
                         |( ackermann ${t1} 1 )|
                       else t1 = $(( ${2} - 1 )) ;
                            |( ackermann ${1} ${t1} )| ;
                            t2 = $(( ${1} - 1 )) ;
                            |( ackermann ${t2} ${r1} )|
                       fi)
                fi ;} ;
|( ackermann ${1} ${2} )| ;
echo ${r1}
```

Figure 4.7: Ackermann's function in Bash_{light}

requirement of a temporary variable in the Bash code since arbitrary expressions are not permitted as the first argument to the `if` construct.

File input, output and case matching

The script which was previously given in Figure 2.1 demonstrates several interesting features of Bash_{light}. Firstly it exhibits the use of redirections to perform input to and output from files. Standard input and output are redirected to file descriptors which are declared using the `exhibit` syntax. The value read is then processed using a case statement do matching.

It is evident that the changes to the syntax are minimal. It would easily be possible to write a pretty printer which transcribed between Bash code and Bash_{light}, for a specific subset of the Bash language (or vice-versa).

```
r1=0
ackermann ()  { if [ ${1} == 0 ]; then
                  r1=$((${2}+1))
                else cmp1=$((${1}>0&&${2}==0))
                     if [ ${cmp1} != 0 ]; then
                       t1=$((${1}-1))
                       ackermann ${t1} 1
                     else t1=$((${2}-1))
                          ackermann ${1} ${t1}
                          t2=$((${1}-1))
                          ackermann ${t2} ${r1}
                     fi
                fi; }
ackermann ${1} ${2}
echo ${r1}
```

Figure 4.8: Ackermann's function in Bash

4.6 Tools

This section reviews the tools which were specified in Chapter 2, and issues that were encountered in their use.

4.6.1 Ott

Ott was a very useful tool for defining operational semantics, and presenting them in an aesthetic manner. However the fact that it is a developing tool showed through in several ways:

Bracketing

Ott insisted that its input was completely unambiguous and would not work were it not (unlike OCamlyacc which would make some reasonable assumptions and issue a warning). This lead to a proliferation of bracketing using random characters in the Ott source, and consequently a degradation in the readability of the Ott source code.

String parsing

In the Ott source literal integer values were permitted (and utilised in this project, e.g. for 0 as a default return value for certain constructs). However similar facilities were not available for literal string values, which lead to more convoluted definitions (for example the treatment of '`*`' in `case` statements).

Side conditions

It was not possible to express side conditions to rules in a textual manner (alternate rules, grammars and reduction steps had to be developed) and this lead to the increased use of meta operations which had to be explained elsewhere, which consequently reduced readability.

General expressiveness

Certain properties, like return values could not be properly expressed in the Ott language, and hence even though they were implementable in OCaml they were left out of the language. This resulted in reduced backwards compatibility and perhaps the most unintuitive aspect of the new variant.

4.6.2 OCamllex and OCamlyacc

These tools were very effective, apart from a minor issue to do with white space (which is discussed in Appendix A.1.1).

4.7 Summary

Bash_{light} accurately emulates Bash in many areas, as was demonstrated by the testing. To make Bash_{light} feature complete would require further specification and implementation work, specifically pipelines and additional builtin functions. There are no evident technical reasons why this additional effort would not be straightforward now that the Bash_{light} prototype has been successfully implemented.

Chapter 5

Conclusion

This project presented the formal semantics for Bash_{light}, a subset of Bash, and an executable implementation of the semantics in OCaml. The implementation can be used to run many non-trivial Bash_{light} programs. The semantics and the implementation were extensively tested to ensure they corresponded to Bash.

Some additional work is required to produce a usable replacement for Bash. One feature that would need to be supported is pipes. This should not be too difficult, since the existing framework can already deal with redirections. A representative selection of builtin functions have also been included, such as `echo`, and adding the remaining builtins should be straightforward.

This is the first time a shell scripting language has been given a formal semantics. The main goal, to explicate the workings of Bash, has been achieved, and difficult parts of Bash, such as scoping and redirections, are now described formally, clarifying their behaviour. In detail, all the core goals, and several of the extensions originally described in the project proposal, have been completed.

Bibliography

[1] Bash reference manual, http://www.gnu.org/software/bash/manual/bashref.html.

[2] Cash shell, http://pauillac.inria.fr/cash/latest/doc/cash.html.

[3] Sewell et al. Ott: Effective tool support for the working semanticist, http://www.cl.cam.ac.uk/ pes20/ott/paper.pdf.

Appendix A

Additional information

This appendix includes additional explanations of technical aspects to do with the Bash_{light} syntax and semantics which are included for completeness and to give a more thorough understanding of the language.

A.1 Changes from Bash

Various changes were made to the syntax and semantics of Bash. The main changes are included here for completeness.

A.1.1 Changes to the syntax

In order to eliminate ambiguity various additions were made to the Bash_{light} syntax. Two similar constructs are those for lists of values and for values which are to be concatenated together:

$$
\begin{array}{lll}
expr,\ e & ::= & \\
 & | & |(\ elist\ )| \\
 & | & \#(\ elist\ )\#
\end{array}
$$

In each case an additional bracketing was introduced to make unambiguous which construct was being used. Another case where an addition was made to the syntax was grouping commands. The first grouping command is

```
( f )
```

However this is ambiguous with regards to bracketing. The other of the grouping commands is

```
{ f; }
```

This is ambiguous with respect to the definition of `f`. A simple solution to these issues was to modify the two syntaxes to be:

(f ;)
{ f ;}

i.e. to use tokens `;)` and `;}` which are distinct from `; )` and `; }` (note the space).

There was one example of part of the Bash syntax being removed. In Bash more semicolons were utilised, e.g.

if test-commands; then consequent-commands; [elif more-test-commands; then more-consequents;] [else alternate-consequents;] fi

These semicolons were however interchangeable with new lines, i.e. the following would be acceptable:

if test-commands
then consequent-commands
else alternate-consequents
fi

This would have been problematic to implement with OCamllex and OCamlyacc. The semicolon was not included in the `if` of `for` syntax to indicate that this interchangeability was no longer usable.

Various other minor changes were made to the syntax. In order to permit a slightly more expressive and intuitive use of the conditional constructs arbitrary expressions were permitted in the first argument of the `if` construct, rather that simply conditional expressions. Also, in Bash there are no particular requirements to do with white space, but due to the constraints of the lexing tool used it was necessary, in the OCaml implementation, to place white space (space or tab) between keywords (which include bracketing terminals) and expressions. An example of this is that while:

```
$((1+1))
```

is valid in Bash, in Bash$_{light}$ it must be written as

```
$(( 1 + 1 ))
```

A.1.2 Changes to the semantics

Various changes were made to the operational semantics. Firstly the standard error channel stderr was implemented, but no error messages for user errors were included. This was as the error messages were not detailed in the Bash specification[1]. In the OCaml implementation exceptions were implemented to indicate when such errors occurred, so given a suitable description of error messages it would be a trivial extension to add these error messages.

Secondly, due to complications to do with some of the tools used, it was not possible to implement most of the return values to indicate the return status of an expression, so they should not be used as an experienced used of Bash would expect.

A.2 Environment

$$
\begin{array}{lll}
env & ::= & \\
 & | & s : fun\_s : funslist : term : fsys : ioer : envvars \\
 & | & env \, \{ \, envchanges \, \}
\end{array}
$$

The Bash$_{light}$ environment is reasonably straightforward, however its representation in the Ott grammar is not. A more detailed explanation of the various non-terminals and productions is included here.

Memory

$$
\begin{array}{lll}
s & ::= & \\
 & | & \textbf{empty}
\end{array}
$$

$$\mid \quad [\, mem_1 \,], \ldots [\, mem_n \,], \ [\, mem' \,] \ , \ s$$

$$\begin{array}{lll} mem & ::= & \\ & \mid & \textbf{empty} \\ & \mid & mem \ , \ str_1 \mapsto c_1 \ , \ldots \ str_n \mapsto c_n \ , \ str' \mapsto c' \end{array}$$

The `s` non-terminal represents the memory. It is a list of `mem` non-terminals, each of which represents a partial function from variable names to constants. In both of these cases the comma acts as a cons operator. Unusually for a list, the rest of the (possibly empty) list is included on the left hand side of the construction for `mem`, but is on the normal right hand side for `s`.

Function parameters

$$\begin{array}{lll} fun\_s & ::= & \\ & \mid & \textbf{empty} \\ & \mid & [\, funmem_1 \,], \ldots [\, funmem_n \,], \ [\, funmem' \,] \ , \ fun\_s \end{array}$$

$$\begin{array}{lll} funmem & ::= & \\ & \mid & \textbf{empty} \\ & \mid & funmem \ , \ int_1 \mapsto c_1 \ , \ldots \ int_n \mapsto c_n \ , \ int' \mapsto c' \\ & \mid & int_1 \mapsto c_1 \ , \ldots \ int_n \mapsto c_n \ , \end{array}$$

The `fun_s` non-terminal represents the function parameters. It is similar to the memory in its being a list of partial functions, its use of the comma and its unusual placing of the remainder of the list for the partial function (`funmem`) but not for the list (`fun_s`).

Function definitions

$$\begin{array}{lll} funslist & ::= & \\ & \mid & \textbf{empty} \\ & \mid & [\, funs_1 \ , \ str_1 \mapsto e_1 \,], \ldots [\, funs_n \ , \ str_n \mapsto e_n \,], \ [\, funs' \ , \ str' \mapsto e' \,], \ funslist \\ & \mid & [funs], funslist \end{array}$$

$$\begin{array}{lll} funs & ::= & \end{array}$$

| | **empty**
| | **funs**, $str \mapsto e$
| | $str_1 \mapsto e_1$, .. $str_n \mapsto e_n$, $str' \mapsto e'$

The `funslist` non-terminal represents function definitions. It is similar to the memory in its being a list of partial functions, its use of the comma and its unusual placing of the remainder of the list for the partial function (`funs`) but not for the list (`funslist`).

Terminal

term ::=
| **empty**
| *term* **value** (*c*)

The terminal is simply a collection of strings, which are ordered such that the one which was printed first is on the left. The syntax **value(c)** is included to denote a function to convert arbitrary constants to string values so they can be printed to the terminal.

File system

fsys ::=
| **empty**
| **fsys**, *fname* $\mapsto$ *file*

fname ::=
| **fsval**(*str*) M

file ::=
| *str*
| **blank_string** M
| **write** (*str* , *c*) M
| **read** (str_1 , str_2) M

The file system representation is very coarse, and much of its functionality is defined using meta operations which are described in Appendix A.3. However it is effectively

a partial function from strings to strings, where the key string is the file name and the result string is the file contents. In a similar manner to before the comma is used is a cons operator and the remaining (possibly empty) list is included on the left hand side.

Redirections

ioer ::=
- | **empty**
- | [*ioersub*] , *ioer*

ioersub ::=
- | **empty**
- | *ioersub* , { *integer_literal* ↦ *fd* , *fdtype* }

fd ::=
- | *fname*
- | **terminal**

fdtype ::=
- | **in**
- | **out**
- | **app**
- | **err**
- | **ran**

The `ioer` non-terminal represents redirections. It is similar to the memory in its being a list of partial functions, its use of the comma and its unusual placing of the remainder of the list for the partial function (`ioersub`) but not for the list (`ioer`). The non-terminal `ioersub` indicates that a file descriptor `integer_literal` maps to some output `fd`, and has type `fdtype`. The non-terminal `fd` is used to show where a file descriptor is pointing: either a file or the terminal.

Environmental variables

$$
\begin{array}{lll}
\mathit{envvars} & ::= & \\
 & \mid & \mathit{envvarssub}\,,\ \mathit{envvars} \\
 & & \\
\mathit{envvarssub} & ::= & \\
 & \mid & \mathbf{pwd} \mapsto \mathit{str} \\
 & \mid & \mathbf{pwd} \mapsto \mathbf{val}\,(\,\mathit{str}_1\,,\ \mathit{str}_2\,) \quad \mathsf{M}
\end{array}
$$

The `envvars` non-terminal represents environmental variables. Since the only environmental variable which is implemented is the present working directory the `envvars` non-terminal is effectively a list of present working directories.

Abbreviation syntax

$$
\begin{array}{lll}
\mathit{env} & ::= & \\
 & \mid & s\ :\ \mathit{fun\_s}\ :\ \mathit{funslist}\ :\ \mathit{term}\ :\ \mathit{fsys}\ :\ \mathit{ioer}\ :\ \mathit{envvars} \\
 & \mid & \mathit{env}\,\{\,\mathit{envchanges}\,\} \\
 & & \\
\mathit{envchanges} & ::= & \\
 & \mid & \\
 & \mid & \mathit{subenvchn}\,\mathbf{inc}\,\mathit{changes} \\
 & \mid & \mathit{subenvchn}\,\mathbf{inc}\,\mathit{changes}\,,\ \mathit{envchanges} \\
 & \mid & \mathit{subenvchn}\ =\ \mathit{envval} \\
 & \mid & \mathit{subenvchn}\ =\ \mathit{envval}\,,\ \mathit{envchanges} \\
 & \mid & \mathit{subenvchn}\,\mathbf{ninc}\,\mathit{changes} \\
 & \mid & \mathit{subenvchn}\,\mathbf{ninc}\,\mathit{changes}\,,\ \mathit{envchanges} \\
 & & \\
\mathit{subenvchn} & ::= & \\
 & \mid & \mathbf{S} \\
 & \mid & \mathbf{F\_S} \\
 & \mid & \mathbf{F} \\
 & \mid & \mathbf{T} \\
 & \mid & \mathbf{FS} \\
 & \mid & \mathbf{I} \\
 & \mid & \mathbf{E}
\end{array}
$$

$$
\begin{array}{lll}
\textit{changes} & ::= & \\
 & | & \textit{str} \mapsto \textit{c} \\
 & | & [\,\textit{mem}\,,\ \textit{str} \mapsto \textit{c}\,] \\
 & | & \textit{int} \mapsto \textit{c} \\
 & | & \textbf{pwd} \mapsto \textit{str} \\
 & | & \textbf{funs}, \textit{str} \mapsto \textit{e} \\
 & | & \textbf{fsval}(\textbf{str}) \mapsto \textbf{blank\_string} \\
 & | & \textit{fname}_1 \mapsto \textit{str}_1 \\
 & | & \textit{fname}_1 \mapsto \textbf{write}\,(\textbf{str1}, \textbf{c}) \\
 & | & \textit{fname}_1 \mapsto \textbf{read}(\textbf{str1}, \textbf{str2})
\end{array}
$$

$$
\begin{array}{lll}
\textit{envval} & ::= & \\
 & | & \textit{s} \\
 & | & \textit{fun\_s} \\
 & | & \textit{funslist} \\
 & | & \textit{term} \\
 & | & \textit{fsys} \\
 & | & \textit{ioer} \\
 & | & \textit{envvars}
\end{array}
$$

In order to reduce the length of the rules an abbreviation syntax was introduced, so that an entire environment need not be written out. Using the abbreviation syntax it is only necessary to name the environment and the specify specific aspects of it. This pertains to the second production of the `env` non-terminal.

The specific aspects of the environment are specified using the `envchanges` non-terminal. This is effectively a list of facts about the environment, either completely defining one of parts of the state mentioned previously (the productions which include `=`) or some specific feature is said to be included (**inc**) or not included (**ninc**) in one of the parts of state. The names of the parts of state are abbreviates with **S** representing memory, **F_S** representing function parameters, **F** representing function definitions, **T** representing the terminal, **FS** representing the file system, **I** representing redirections and **E** representing environmental variables.

A.3 Meta operations

Meta operations were used to perform some operation or represent some concept that was not strictly within the scope of the language (and hence the operational semantics or grammar). The meta operations used are as follows:

blank_string

This is a part of the constant expression. It is used to represent the blank string, as it was not possible to represent concrete strings within the Ott tool.

g1

This is a part of the main expression. It used to represent an `f` expression being evaluated which had originally been inside the first grouping command. When this operation is evaluated with a constant it returns and any changes to the environment effected during its evaluation are ignored.

g2

This is a part of the main expression. It is used to represent an `f` expression being evaluated which had originally been inside the second grouping command. When this operation is evaluated with a constant it returns and any changes to the environment effected during its evaluation persist.

(e)

This is a part of the main expression. It is used to permit bracketing.

“str”

This is a part of the main expression. It is used in the parser to indicate that the characters between the “ ... ” were a string, and could include spaces, etc.

fun

This is a part of the main expression. This is used to represent the body of a function being evaluated. When this operation is evaluated with a constant it returns and the parameters to the functions are dropped (as explained in Section 3.3.1).

val(aexp)

This is a part of the aexp expression, and is used to represent arithmetic values.

constval(constant)

This is a part of the aexp expression. As mentioned in Section 3.1.2, in cases where a value of the wrong 'type' is used, a default value of the correct 'type' is inserted instead. This meta operation acts according to the following function:

constval(x) = y if x = Integer(y)
0 otherwise

(condexp)

This is a part of the condexp expression. It is used to permit bracketing.

blank_string

This is a part of the file expression. It is used to indicate an empty file.

write(str,c)

This is a part of the file expression. The operation of a file descriptor in a file is that there is a file pointer, at which operations are performed at the same time as the pointer advancing through the file. Most notably, when writing n characters to a file the n characters after the file pointer would be overwritten, and after the write the file pointer would be pointing to just after where the n characters had been written. Hence this meta operation would write the string value of constant c to the relevant place in *str*, depending on an implicit file pointer.

read(str$_1$, str$_2$)

This is a part of the file expression. The operation of the file descriptors in a file is that there is a file pointer at which operations are performed at the same time as the pointer advancing through the file, i.e. when a read of a line occurred the file pointer moved to the next line. Hence this meta operation would read a line from `str`$_1$ (which represents the contents of a file) and change the environment so that the variable called `str`$_2$ would be bound to the value of the line read. In order to move the pointer back to the start of the file it would be necessary to close the file descriptor, re-open it and move through the file to the correct place.

fsval(str)

This is a part of the fname expression. This meta operation examines `str` to see if its format is an absolute address, in which case it uses str to access the file, or if it is a relative address it uses the present working directory to generate the correct file address.

pwd $\longmapsto$ val(str$_1$, str$_2$)

This is a part of the envvarssub expression where `str`$_1$ is taken to be the present working directory and `str`$_2$ is the directory to which the user is attempting to change the present working directory. It is important to note that `str`$_2$ could be a relative or an absolute address. This meta operation examines `str`$_2$ to see if its format is an absolute address, in which case it uses `str`$_2$, or if it is a relative address it also uses `str`$_1$ (the present working directory) to generate the correct directory. This is similar to *fsval(*`str`*)*.

Appendix B

Operational semantics

value_name, *x*
ident
integer_literal, *intlit*, *int*
string_literal, *str*
index, *j*, *k*, *n*, *m*

bool, *b*	::=		
	|	**true**	
	|	**false**	
constant, *c*	::=		
	|	*integer_literal*	
	|	*string_literal*	
	|	*b*	
	|	**blank_string**	M
f	::=		
	|	$e; f$	
	|	*e*	
	|	e_1 ; .. e_n ; f	M
expr, *e*	::=		
	|	*constant*	
	|	|(*elist*)|	
	|	**until** f_1 **do** f_2 **done**	
	|	**while** f_1 **do** f_2 **done**	
	|	**for** *str* **do** f **done**	
	|	**for** *str* **in** *elist* **do** f **done**	
	|	**for** ((e ; $aexp_1$; $aexp_2$)) **do** f' **done**	
	|	**if** f_1 **then** f_2 **else** f_3 **fi**	
	|	**case** f **in** *oce* **esac**	
	|	[[*condexp*]]	
	|	(f ;)	
	|	{ f ; }	
	|	**g1** f	M
	|	**g2** f	M
	|	${ *str* }	

	| \${ *intlit* }	
	| \$((*aexp*))	
	| *str* = *e*	
	| #(*elist*)#	
	| (*e*)	M
	| "*str*"	M
	| **function** *str e*	
	| *str* () *e*	
	| **function** *str* () *e*	
	| **fun** *e*	M
	| $rdexp_1$ **cd** $rdexp_2$ *e* $rdexp_3$	
	| $rdexp_1$ **pwd** $rdexp_2$	
	| $rdexp_1$ **echo** $rdexp_2$ *e* $rdexp_3$	
	| **exec** *rdexp*	
	| $rdexp_1$ **read** $rdexp_2$ *str* $rdexp_3$	

arithexp, *aexp*	::=	
	| $aexp_1$ *bop* $aexp_2$	
	| *preuop aexp*	
	| *vpreuop str*	
	| *vpreuop* **constval** (*constant*)	M
	| *str postuop*	
	| **constval** (*constant*) *postuop*	M
	| *integer_literal*	
	| *str*	
	| \${ *str* }	
	| \${ *intlit* }	
	| ((*aexp*))	
	| **val** (*aexp*)	M
	| **constval** (*constant*)	M

bop	::=	
	| *ibop*	
	| *lbop*	
	| *cbop*	

| | *bbop*

preuop ::=
| | *ipreuop*
| | *lpreuop*
| | *bpreuop*

postuop ::=
| | *ipostuop*

ibop ::=
| | +
| | −
| | /
| | %
| | *
| | **

cbop ::=
| | <
| | >
| | <=
| | >=
| | ==
| | ! =

lbop ::=
| | &&
| | ||

bbop ::=
| | &
| | |
| | ˆ

	|	$<<$
	|	$>>$
vpreuop	::=	
	|	$--$
	|	$++$
ipreuop	::=	
	|	$-$
	|	$+$
lpreuop	::=	
	|	!
bpreuop	::=	
	|	$\sim$
ipostuop	::=	
	|	$--$
	|	$++$
exp_list, *elist*	::=	
	|	*selist* $selist_1$ .. $selist_n$
	|	$selist_1$ .. $selist_n$
	|	*selist elist*
sub_exp_list, *selist*	::=	
	|	*c*
	|	\${ *str* }
	|	\${ *intlit* }
	|	\$@
caseexp, *ce*	::=	
	|	*f* | *ce*

	|	f
	|	$subce_1 \mid .. \; subce_m \mid subce \mid subce'_1 \mid .. \; subce'_n \mid subce'$
	|	$subce_1 \mid .. \; subce_m \mid * \mid subce'_1 \mid .. \; subce'_n \mid subce'$
subcaseexp, *subce*	::=	
	|	c
	|	**strval**(c) M
ocaseexp, *oce*	::=	
	|	(*ce*) f ; ;
	|	(*ce*) f ; ; *oce*
condexp	::=	
	|	$-\mathbf{a}\, f$
	|	(*condexp*) M
	|	! *condexp*
	|	$condexp_1$ && $condexp_2$
	|	$condexp_1$ || $condexp_2$
	|	b
rdexp	::=	
	|	
	|	*integer_literal* > *str rdexp*
	|	*integer_literal* >> *str rdexp*
	|	*integer_literal* < *str rdexp*
	|	> *str rdexp*
	|	>> *str rdexp*
	|	< *str rdexp*
	|	& > *str rdexp*
	|	$integer\_literal_1$ > & $integer\_literal_2$ *rdexp*
	|	$integer\_literal_1$ < & $integer\_literal_2$ *rdexp*
	|	*integer_literal* <> *str rdexp*
	|	$integer\_literal_1$ < &− *rdexp*
input_output, *io*	::=	

$$
\begin{array}{lll}
 & | & \textbf{empty} \\
 & | & \textbf{write}\ c \\
 & | & \textbf{write\_n}\ c \\
 & | & \textbf{read}\ str \\
 & & \\
env & ::= & \\
 & | & s : fun\_s : funslist : term : fsys : ioer : envvars \\
 & | & env\ \{\ envchanges\ \} \\
 & & \\
s & ::= & \\
 & | & \textbf{empty} \\
 & | & [\ mem_1\ ],\ ..\ [\ mem_n\ ],\ [\ mem'\ ]\ ,\ s \\
 & & \\
mem & ::= & \\
 & | & \textbf{empty} \\
 & | & mem\ ,\ str_1 \mapsto c_1\ ,\ ..\ str_n \mapsto c_n\ ,\ str' \mapsto c' \\
 & & \\
fun\_s & ::= & \\
 & | & \textbf{empty} \\
 & | & [\ funmem_1\ ],\ ..\ [\ funmem_n\ ],\ [\ funmem'\ ]\ ,\ fun\_s \\
 & & \\
funmem & ::= & \\
 & | & \textbf{empty} \\
 & | & funmem\ ,\ int_1 \mapsto c_1\ ,\ ..\ int_n \mapsto c_n\ ,\ int' \mapsto c' \\
 & | & int_1 \mapsto c_1\ ,\ ..\ int_n \mapsto c_n\ , \\
 & & \\
funslist & ::= & \\
 & | & \textbf{empty} \\
 & | & [\ funs_1\ ,\ str_1 \mapsto e_1\ ],\ ..\ [\ funs_n\ ,\ str_n \mapsto e_n\ ],\ [\ funs'\ ,\ str' \mapsto e'\ ],\ funslist \\
 & | & [funs], funslist \\
 & & \\
funs & ::= & \\
 & | & \textbf{empty} \\
 & | & \textbf{funs}, str \mapsto e
\end{array}
$$

	|	$str_1 \mapsto e_1 \, , \, .. \, str_n \mapsto e_n \, , \, str' \mapsto e'$	
term	::=	
	|	**empty**
	|	*term* **value** (*c*)
fsys	::=	
	|	**empty**
	|	**fsys**, *fname* $\mapsto$ *file*
fname	::=	
	|	**fsval**(*str*) M
file	::=	
	|	*str*
	|	**blank_string** M
	|	**write** (*str* , *c*) M
	|	**read** (str_1 , str_2) M
ioer	::=	
	|	**empty**
	|	[*ioersub*] , *ioer*
ioersub	::=	
	|	**empty**
	|	*ioersub* , { *integer_literal* $\mapsto$ *fd* , *fdtype* }
fd	::=	
	|	*fname*
	|	**terminal**
fdtype	::=	
	|	**in**
	|	**out**

	|	**app**
	|	**err**
	|	**ran**
envvars	::=	
	|	*envvarssub* , *envvars*
envvarssub	::=	
	|	**pwd** $\mapsto$ *str*
	|	**pwd** $\mapsto$ **val** (str_1 , str_2) M
envchanges	::=	
	|	
	|	*subenvchn* **inc** *changes*
	|	*subenvchn* **inc** *changes* , *envchanges*
	|	*subenvchn* = *envval*
	|	*subenvchn* = *envval* , *envchanges*
	|	*subenvchn* **ninc** *changes*
	|	*subenvchn* **ninc** *changes* , *envchanges*
subenvchn	::=	
	|	**S**
	|	**F_S**
	|	**F**
	|	**T**
	|	**FS**
	|	**I**
	|	**E**
changes	::=	
	|	*str* $\mapsto$ *c*
	|	[*mem* , *str* $\mapsto$ *c*]
	|	*int* $\mapsto$ *c*
	|	**pwd** $\mapsto$ *str*
	|	**funs**, *str* $\mapsto$ *e*

$$
\begin{array}{lll}
 & | & \mathbf{fsval}(\mathbf{str}) \mapsto \mathbf{blank\_string} \\
 & | & \mathit{fname}_1 \mapsto \mathit{str}_1 \\
 & | & \mathit{fname}_1 \mapsto \mathbf{write}\,(\mathbf{str1}, \mathbf{c}) \\
 & | & \mathit{fname}_1 \mapsto \mathbf{read}(\mathbf{str1}, \mathbf{str2})
\end{array}
$$

$$
\begin{array}{lll}
\mathit{envval} & ::= & \\
 & | & s \\
 & | & \mathit{fun\_s} \\
 & | & \mathit{funslist} \\
 & | & \mathit{term} \\
 & | & \mathit{fsys} \\
 & | & \mathit{ioer} \\
 & | & \mathit{envvars}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{formula} & ::= & \\
 & | & \mathit{judgement} \\
 & | & \mathit{formula}_1 \quad .. \quad \mathit{formula}_n
\end{array}
$$

$$
\begin{array}{lll}
\mathit{terminals} & ::= & \\
 & | & \mathit{bop} \\
 & | & \mathit{preuop} \\
 & | & \mathit{postuop} \\
 & | & \{ \\
 & | & \} \\
 & | & \rightarrow \\
 & | & \mapsto \\
 & | & \vdash \\
 & | & \longrightarrow \\
 & | &
\end{array}
$$

$$
\begin{array}{lll}
\mathit{Jop} & ::= & \\
 & | & \mathit{env}_1 \,/\, f_1 \longrightarrow \mathit{env}_2 \,/\, f_2 \\
 & | & \mathit{env} \,/\, e_1 \longrightarrow \mathit{env}_2 \,/\, e_2 \\
 & | & \mathit{env}_1 \,/\, \mathit{aexp}_1 \longrightarrow \mathit{env}_2 \,/\, \mathit{aexp}_2 \\
 & | & \mathit{env}_1 \,/\, \mathit{elist}_1 \longrightarrow \mathit{env}_2 \,/\, \mathit{elist}_2
\end{array}
$$

	|	$env_1 \,/\, ce_1 \longrightarrow env_2 \,/\, ce_2$
	|	$env_1 \,/\, oce_1 \longrightarrow env_2 \,/\, oce_2$
	|	$env_1 \,/\, condexp_1 \longrightarrow env_2 \,/\, condexp_2$
	|	$env_1 \,/\, rdexp_1 \longrightarrow env_2 \,/\, rdexp_2$
	|	$env_1 \,/\, io_1 \longrightarrow\!\!\mathbf{io}\ env_2 \,/\, io_2$
judgement	::=	
	|	*Jop*
user_syntax	::=	
	|	*value_name*
	|	*ident*
	|	*integer_literal*
	|	*string_literal*
	|	*index*
	|	*bool*
	|	*constant*
	|	*f*
	|	*expr*
	|	*arithexp*
	|	*bop*
	|	*preuop*
	|	*postuop*
	|	*ibop*
	|	*cbop*
	|	*lbop*
	|	*bbop*
	|	*vpreuop*
	|	*ipreuop*
	|	*lpreuop*
	|	*bpreuop*
	|	*ipostuop*
	|	*exp_list*
	|	*sub_exp_list*
	|	*caseexp*

| *subcaseexp*
| *ocaseexp*
| *condexp*
| *rdexp*
| *input_output*
| *env*
| *s*
| *mem*
| *fun_s*
| *funmem*
| *funslist*
| *funs*
| *term*
| *fsys*
| *fname*
| *file*
| *ioer*
| *ioersub*
| *fd*
| *fdtype*
| *envvars*
| *envvarssub*
| *envchanges*
| *subenvchn*
| *changes*
| *envval*
| *formula*
| *terminals*

$\boxed{env_1 \,/\, f_1 \longrightarrow env_2 \,/\, f_2}$

$$\frac{env \,/\, e_1 \longrightarrow env' \,/\, e_1'}{env \,/\, e_1; f \longrightarrow env' \,/\, e_1'; f} \quad \text{OP\_F\_SEQ\_1}$$

$$\frac{}{env \,/\, intlit; f \longrightarrow env \,/\, f} \quad \text{OP\_F\_SEQ\_2}$$

$$\frac{}{env \,/\, b; f \longrightarrow env \,/\, f} \quad \text{OP\_F\_SEQ\_3}$$

$$\frac{}{env\,\{\,\mathbf{F\ inc\ funs}, str \mapsto e\,\} \,/\, str; f \longrightarrow env \,/\, \mathbf{fun}\ e; f} \quad \text{OP\_F\_SEQ\_4}$$

$$\frac{}{env\,\{\,\mathbf{F\,inc\,funs}, str \mapsto e\,\}\,/\,str';f \longrightarrow env\,/\,f}\ \text{OP\_F\_SEQ\_5}$$

$$\frac{env\,/\,e_1 \longrightarrow env'\,/\,e_1'}{env\,/\,e_1 \longrightarrow env'\,/\,e_1'}\ \text{OP\_F\_NODE\_1}$$

$\boxed{env\,/\,e_1 \longrightarrow env_2\,/\,e_2}$

$$\frac{env\,/\,elist \longrightarrow env'\,/\,elist'}{env\,/\,|(\ elist\ )| \longrightarrow env'\,/\,|(\ elist'\ )|}\ \text{OP\_EXP\_ELIST\_1}$$

$$\frac{}{\begin{array}{l}env\,\{\,\mathbf{F\_S} = fun\_s\ ,\ \mathbf{F\,inc\,funs}, str \mapsto e\,\}\,/\,|(\ str\ c_1\ ..\ c_n\ )| \\ \longrightarrow env\,\{\,\mathbf{F\_S} = \ [\ int_1 \mapsto c_1\ ,\ ..\ int_n \mapsto c_n\ ,\ ]\ ,\ fun\_s\,\}\,/\,\mathbf{fun}\ e\end{array}}\ \text{OP\_EXP\_ELIST\_2}$$

$$\frac{}{env\,\{\,\mathbf{F\,ninc\,funs}, str \mapsto e\,\}\,/\,|(\ str\ c_1\ ..\ c_n\ )| \longrightarrow env\,/\,str}\ \text{OP\_EXP\_ELIST\_3}$$

$$\frac{}{env\,/\,|(\ int_1\ c_1\ ..\ c_n\ )| \longrightarrow env\,/\,int_1}\ \text{OP\_EXP\_ELIST\_4}$$

$$\frac{}{env\,/\,|(\ b_1\ c_1\ ..\ c_n\ )| \longrightarrow env\,/\,b_1}\ \text{OP\_EXP\_ELIST\_5}$$

$$\frac{}{env\,/\,\mathbf{until}\,f_1\,\mathbf{do}\,f_2\,\mathbf{done} \longrightarrow env\,/\,\mathbf{if}\,f_1\,\mathbf{then}\,0\,\mathbf{else}\,\{\,\{\,f_2\,;\,\};\mathbf{until}\,f_1\,\mathbf{do}\,f_2\,\mathbf{done}\,;\,\}\,\mathbf{fi}}\ \text{OP\_EXP\_UNTIL}$$

$$\frac{}{env\,/\,\mathbf{while}\,f_1\,\mathbf{do}\,f_2\,\mathbf{done} \longrightarrow env\,/\,\mathbf{if}\,f_1\,\mathbf{then}\,\{\,\{\,f_2\,;\,\};\mathbf{while}\,f_1\,\mathbf{do}\,f_2\,\mathbf{done}\,;\,\}\,\mathbf{else}\,0\,\mathbf{fi}}\ \text{OP\_EXP\_WHILE}$$

$$\frac{}{env\,/\,\mathbf{for}\,str\,\mathbf{do}\,f_1\,\mathbf{done} \longrightarrow env'\,/\,\mathbf{for}\,str\,\mathbf{in}\,\$@\ \ \mathbf{do}\,f_1\,\mathbf{done}}\ \text{OP\_EXP\_FOR\_1}$$

$$\frac{env\,/\,elist \longrightarrow env'\,/\,elist'}{env\,/\,\mathbf{for}\,str\,\mathbf{in}\,elist\,\mathbf{do}\,f_2\,\mathbf{done} \longrightarrow env'\,/\,\mathbf{for}\,str\,\mathbf{in}\,elist'\,\mathbf{do}\,f_2\,\mathbf{done}}\ \text{OP\_EXP\_FOR\_2\_1}$$

$$\frac{}{env\,/\,\mathbf{for}\,str\,\mathbf{in}\,c\ \ \mathbf{do}\,f\,\mathbf{done} \longrightarrow env\,/\,\{\,str\ =\ c;f\,;\}}\ \text{OP\_EXP\_FOR\_2\_2}$$

$$\frac{}{env\,/\,\mathbf{for}\,str\,\mathbf{in}\,c\,elist\,\mathbf{do}\,f\,\mathbf{done} \longrightarrow env\,/\,\{\ \{\,str\ =\ c;f\,;\}\ ;\ \{\,\mathbf{for}\,str\,\mathbf{in}\,elist\,\mathbf{do}\,f\,\mathbf{done}\,;\,\}\ ;\}}\ \text{OP\_EXP\_FOR\_2\_3}$$

$$\frac{env\,/\,e \longrightarrow env'\,/\,e'}{env\,/\,\mathbf{for}\,((\ e\ ;\ aexp_1\ ;\ aexp_2\ ))\,\mathbf{do}\,f\,\mathbf{done} \longrightarrow env'\,/\,\mathbf{for}\,((\ e'\ ;\ aexp_1\ ;\ aexp_2\ ))\,\mathbf{do}\,f\,\mathbf{done}}\ \text{OP\_EXP\_FOR\_3\_1}$$

$$\frac{}{\begin{array}{l}env\,/\,\mathbf{for}\,((\ integer\_literal\ ;\ aexp_1\ ;\ aexp_2\ ))\,\mathbf{do}\,f\,\mathbf{done} \\ \longrightarrow env\,/\,\mathbf{if}\,\$((\ aexp_1\ ))\,\mathbf{then}\,\{\,\{\,\{\,f\,;\};\$((\ aexp_2\ ))\,;\}; \\ \mathbf{for}\,((\ integer\_literal\ ;\ aexp_1\ ;\ aexp_2\ ))\,\mathbf{do}\,f\,\mathbf{done}\,;\,\}\,\mathbf{else}\,0\,\mathbf{fi}\end{array}}\ \text{OP\_EXP\_FOR\_3\_2}$$

$$\frac{env \,/\, f_1 \;\longrightarrow\; env' \,/\, f_1'}{env \,/\, \mathbf{if}\, f_1\, \mathbf{then}\, f_2\, \mathbf{else}\, f_3\, \mathbf{fi} \;\longrightarrow\; env' \,/\, \mathbf{if}\, f_1'\, \mathbf{then}\, f_2\, \mathbf{else}\, f_3\, \mathbf{fi}} \quad \text{OP\_EXP\_IF\_1}$$

$$\frac{}{env \,/\, \mathbf{if}\, 0\, \mathbf{then}\, f_2\, \mathbf{else}\, f_3\, \mathbf{fi} \;\longrightarrow\; env \,/\, \{\, f_2\, ;\}} \quad \text{OP\_EXP\_IF\_2}$$

$$\frac{}{env \,/\, \mathbf{if}\, 1\, \mathbf{then}\, f_2\, \mathbf{else}\, f_3\, \mathbf{fi} \;\longrightarrow\; env \,/\, \{\, f_3\, ;\}} \quad \text{OP\_EXP\_IF\_3}$$

$$\frac{env \,/\, f \;\longrightarrow\; env' \,/\, f'}{env \,/\, \mathbf{case}\, f\, \mathbf{in}\, oce\, \mathbf{esac} \;\longrightarrow\; env' \,/\, \mathbf{case}\, f'\, \mathbf{in}\, oce\, \mathbf{esac}} \quad \text{OP\_EXP\_CASE\_1}$$

$$\frac{env \,/\, oce \;\longrightarrow\; env' \,/\, oce'}{env \,/\, \mathbf{case}\, str\, \mathbf{in}\, oce\, \mathbf{esac} \;\longrightarrow\; env' \,/\, \mathbf{case}\, str\, \mathbf{in}\, oce'\, \mathbf{esac}} \quad \text{OP\_EXP\_CASE\_2}$$

$$\frac{}{env \,/\, \mathbf{case}\, str\, \mathbf{in}\, (\, str_1 \mid .. \, str_j \mid str \mid str_1' \mid .. \, str_n' \mid str''\,)\, f\, ;\, ;\, oce\, \mathbf{esac} \;\longrightarrow\; env \,/\, \{\, f\, ;\}} \quad \text{OP\_EXP\_CASE\_3}$$

$$\frac{}{\begin{array}{l} env \,/\, \mathbf{case}\, str\, \mathbf{in}\, (\, str_1 \mid .. \, str_j \mid * \mid str_1' \mid .. \, str_n' \mid str''\,)\, f\, ;\, ;\, oce\, \mathbf{esac} \\ \longrightarrow\; env \,/\, \{\, \{\, f\, ;\}; \mathbf{case}\, str\, \mathbf{in}\, oce\, \mathbf{esac}\, ;\} \end{array}} \quad \text{OP\_EXP\_CASE\_4}$$

$$\frac{}{env \,/\, \mathbf{case}\, str\, \mathbf{in}\, (\, str_1 \mid .. \, str_j \mid str' \mid str_1' \mid .. \, str_n' \mid str''\,)\, f\, ;\, ;\, oce\, \mathbf{esac} \;\longrightarrow\; env \,/\, \mathbf{case}\, str\, \mathbf{in}\, oce\, \mathbf{esac}} \quad \text{OP\_EXP\_CASE\_5}$$

$$\frac{}{env \,/\, \mathbf{case}\, str\, \mathbf{in}\, (\, str_1 \mid .. \, str_j \mid * \mid str_1' \mid .. \, str_n' \mid str'\,)\, f\, ;\, ;\, \mathbf{esac} \;\longrightarrow\; env \,/\, \{\, f\, ;\}} \quad \text{OP\_EXP\_CASE\_6}$$

$$\frac{}{env \,/\, \mathbf{case}\, str\, \mathbf{in}\, (\, str_1 \mid .. \, str_j \mid str \mid str_1' \mid .. \, str_n' \mid str'\,)\, f\, ;\, ;\, oce\, \mathbf{esac} \;\longrightarrow\; env \,/\, \{\, f\, ;\}} \quad \text{OP\_EXP\_CASE\_7}$$

$$\frac{}{env \,/\, \mathbf{case}\, str\, \mathbf{in}\, (\, str_1 \mid .. \, str_j \mid str' \mid str_1' \mid .. \, str_n' \mid str''\,)\, f\, ;\, ;\, \mathbf{esac} \;\longrightarrow\; env \,/\, 0} \quad \text{OP\_EXP\_CASE\_8}$$

$$\frac{env \,/\, condexp \;\longrightarrow\; env' \,/\, condexp'}{env \,/\, [[\, condexp\,]] \;\longrightarrow\; env' \,/\, [[\, condexp'\,]]} \quad \text{OP\_EXP\_COND\_1}$$

$$\frac{}{env \,/\, [[\, \mathbf{false}\,]] \;\longrightarrow\; env \,/\, 1} \quad \text{OP\_EXP\_COND\_2}$$

$$\frac{}{env \,/\, [[\, \mathbf{true}\,]] \;\longrightarrow\; env \,/\, 0} \quad \text{OP\_EXP\_COND\_3}$$

$$\frac{}{\begin{array}{l} [\, mem\,]\,,\, s\, :\, fun\_s\, :\, [funs], funslist\, :\, term\, :\, fsys\, :\, [\, ioersub\,]\,,\, ioer\, :\, envvarssub\,,\, envvars \,/\, (\, f\, ;\,) \\ \longrightarrow\; [\, mem\,]\,,\, [\, mem\,]\,,\, s\, :\, fun\_s\, :\, [funs], [funs], funslist\, :\, term\, :\, fsys\, :\, [\, ioersub\,]\,,\, [\, ioersub\,]\,,\, ioer\, :\, envvarssub\,,\, envvarssub\,,\, envvars \,/\, \mathbf{g1}\, f \end{array}} \quad \text{OP\_EXP\_GROUPING\_1\_1}$$

$$\frac{env \,/\, f \;\longrightarrow\; env' \,/\, f'}{env \,/\, \mathbf{g1}\, f \;\longrightarrow\; env' \,/\, \mathbf{g1}\, f'} \quad \text{OP\_EXP\_GROUPING\_1\_2}$$

$$\frac{}{[\,mem\,]\,,\ s\ :\ fun\_s\ :\ [funs], funslist\ :\ term\ :\ fsys\ :\ [\,ioersub\,]\,,\ ioer\ :\ envvarssub\,,\ envvars\ /\ \mathbf{g1}\ c\ \longrightarrow\ s\ :\ fun\_s\ :\ funslist\ :\ term\ :\ fsys\ :\ ioer\ :\ envvars\ /\ c}\ \text{OP\_EXP\_GROUPING\_1\_3}$$

$$\frac{}{env\ /\ \{\,f\,;\,\}\ \longrightarrow\ env\ /\ \mathbf{g2}\,f}\ \text{OP\_EXP\_GROUPING\_2\_1}$$

$$\frac{env\ /\ f\ \longrightarrow\ env'\ /\ f'}{env\ /\ \mathbf{g2}\,f\ \longrightarrow\ env'\ /\ \mathbf{g2}\,f'}\ \text{OP\_EXP\_GROUPING\_2\_2}$$

$$\frac{}{env\ /\ \mathbf{g2}\ c\ \longrightarrow\ env\ /\ c}\ \text{OP\_EXP\_GROUPING\_2\_3}$$

$$\frac{}{env\,\{\,\mathbf{S\ inc}\ str\ \mapsto\ c\,\}\,/\,\$\{\,str\,\}\ \longrightarrow\ env\,\{\,\mathbf{S\ inc}\ str\ \mapsto\ c\,\}\,/\,c}\ \text{OP\_EXP\_DEREF\_1}$$

$$\frac{}{env\,\{\,\mathbf{S\ ninc}\ str\ \mapsto\ c\,\}\,/\,\$\{\,str\,\}\ \longrightarrow\ env\,\{\,\mathbf{S\ ninc}\ str\ \mapsto\ c\,\}\,/\,\mathbf{blank\_string}}\ \text{OP\_EXP\_DEREF\_2}$$

$$\frac{}{env\,\{\,\mathbf{F\_S\ inc}\ int_1\ \mapsto\ c_1\,\}\,/\,\$\{\,int_1\,\}\ \longrightarrow\ env\,\{\,\mathbf{F\_S\ inc}\ int_1\ \mapsto\ c_1\,\}\,/\,c_1}\ \text{OP\_EXP\_DEREF\_PARAM\_1}$$

$$\frac{}{env\,\{\,\mathbf{F\_S\ ninc}\ int_1\ \mapsto\ c_1\,\}\,/\,\$\{\,int_1\,\}\ \longrightarrow\ env\,\{\,\mathbf{F\_S\ ninc}\ int_1\ \mapsto\ c_1\,\}\,/\,\mathbf{blank\_string}}\ \text{OP\_EXP\_DEREF\_PARAM\_2}$$

$$\frac{env\ /\ aexp\ \longrightarrow\ env'\ /\ aexp'}{env\ /\ \$((\,aexp\,))\ \longrightarrow\ env'\ /\ \$((\,aexp'\,))}\ \text{OP\_EXP\_ARITH\_1}$$

$$\frac{}{env\ /\ \$((\,integer\_literal\,))\ \longrightarrow\ env\ /\ integer\_literal}\ \text{OP\_EXP\_ARITH\_2}$$

$$\frac{env\ /\ e\ \longrightarrow\ env'\ /\ e'}{env\ /\ str\ =\ e\ \longrightarrow\ env'\ /\ str\ =\ e'}\ \text{OP\_EXP\_ASSIGN\_1}$$

$$\frac{}{env\,\{\,\mathbf{S\ inc}\,[\,mem\,,\ str\ \mapsto\ c'\,]\,\}\,/\,str\ =\ c\ \longrightarrow\ env\,\{\,\mathbf{S\ inc}\,[\,mem\,,\ str\ \mapsto\ c\,]\,\}\,/\,0}\ \text{OP\_EXP\_ASSIGN\_2}$$

$$\frac{}{env\,\{\,\mathbf{S\ ninc}\ str\ \mapsto\ c'\,,\ \mathbf{S}\ =\ \ [\,mem\,]\,,\ s\,\}\,/\,str\ =\ c\ \longrightarrow\ env\,\{\,\mathbf{S}\ =\ \ [\,mem\,,\ \ str\ \mapsto\ c\,]\,,\ s\,\}\,/\,c}\ \text{OP\_EXP\_ASSIGN\_3}$$

$$\frac{env\ /\ elist\ \longrightarrow\ env'\ /\ elist'}{env\ /\ \#(\,elist\,)\#\ \longrightarrow\ env'\ /\ \#(\,elist'\,)\#}\ \text{OP\_EXP\_CONCAT\_1}$$

$$\frac{}{env\ /\ \#(\ <\ c_1\,..\,c_n\ >\ )\#\ \longrightarrow\ env\ /\ str}\ \text{OP\_EXP\_CONCAT\_2}$$

$$\frac{}{env\ /\ (e)\ \longrightarrow\ env\ /\ e}\ \text{OP\_EXP\_BRACKETING\_1}$$

$$\frac{}{env \,/\, \text{“}str'' \longrightarrow env \,/\, str} \quad \text{OP\_EXP\_QUOTING}$$

$$\frac{}{env\,\{\,\mathbf{F} = [funs], funslist\,\}\,/\,\mathbf{function}\;str\;e \longrightarrow env\,\{\,\mathbf{F} = [\mathbf{funs}, str \mapsto e], funslist\,\}\,/\,0} \quad \text{OP\_EXP\_FUN\_1}$$

$$\frac{}{env\,\{\,\mathbf{F} = [funs], funslist\,\}\,/\,str\,()\;e \longrightarrow env\,\{\,\mathbf{F} = [\mathbf{funs}, str \mapsto e], funslist\,\}\,/\,0} \quad \text{OP\_EXP\_FUN\_2}$$

$$\frac{}{env\,\{\,\mathbf{F} = [funs], funslist\,\}\,/\,\mathbf{function}\;str\,()\;e \longrightarrow env\,\{\,\mathbf{F} = [\mathbf{funs}, str \mapsto e], funslist\,\}\,/\,0} \quad \text{OP\_EXP\_FUN\_3}$$

$$\frac{env \,/\, e \longrightarrow env' \,/\, e'}{env \,/\, \mathbf{fun}\;e \longrightarrow env' \,/\, \mathbf{fun}\;e'} \quad \text{OP\_EXP\_FUN\_EVAL\_1}$$

$$\frac{}{env\,\{\,\mathbf{F\_S} = \;\; [\,funmem_1\,]\;,\; fun\_s_1\,\}\,/\,\mathbf{fun}\;c \longrightarrow env\,\{\,\mathbf{F} = fun\_s_1\,\}\,/\,c} \quad \text{OP\_EXP\_FUN\_EVAL\_2}$$

$$\frac{env \,/\, e \longrightarrow env' \,/\, e'}{env \,/\, rdexp_1\;\mathbf{cd}\;rdexp_2\;e\;rdexp_3 \longrightarrow env' \,/\, rdexp_1\;\mathbf{cd}\;rdexp_2\;e'\;rdexp_3} \quad \text{OP\_EXP\_CD\_1}$$

$$\frac{\begin{array}{l} env\,\{\,\mathbf{I} = ioer\,,\,\mathbf{E} = \mathbf{pwd} \mapsto str_1\,,\,envvars\,\}\,/\,rdexp_1 \longrightarrow env\,\{\,\mathbf{I} = ioer'\,\}\,/ \\ env\,\{\,\mathbf{I} = ioer'\,\}\,/\,rdexp_2 \longrightarrow env\,\{\,\mathbf{I} = ioer''\,\}\,/ \\ env\,\{\,\mathbf{I} = ioer''\,\}\,/\,rdexp_3 \longrightarrow env\,\{\,\mathbf{I} = ioer'''\,,\,\mathbf{E} = \mathbf{pwd} \mapsto str_1\,,\,envvars\,\}\,/ \end{array}}{\begin{array}{l} env\,\{\,\mathbf{E} = envvars\,\}\,/\,rdexp_1\;\mathbf{cd}\;rdexp_2\;str_2\;rdexp_3 \\ \longrightarrow env\,\{\,\mathbf{E} = \mathbf{pwd} \mapsto \mathbf{val}\,(\,str_1\,,\,str_2\,)\,,\,envvars\,,\,\mathbf{I} = ioer\,\}\,/\,str_2 \end{array}} \quad \text{OP\_EXP\_CD\_2}$$

$$\frac{\begin{array}{l} env\,\{\,\mathbf{I} = ioer\,,\,\mathbf{E} = \mathbf{pwd} \mapsto str\,,\,envvars\,\}\,/\,rdexp_1 \longrightarrow env\,\{\,\mathbf{I} = ioer'\,\}\,/ \\ env\,\{\,\mathbf{I} = ioer'\,\}\,/\,rdexp_2 \longrightarrow env\,\{\,\mathbf{I} = ioer''\,\}\,/ \\ env\,\{\,\mathbf{I} = ioer''\,,\,\mathbf{E} = \mathbf{pwd} \mapsto str\,,\,envvars\,\}\,/\,\mathbf{write}\;str_1 \longrightarrow \mathbf{io}\;env\,\{\,\mathbf{T} = term'\,,\,\mathbf{F} = fsys'\,\}\,/\,\mathbf{empty} \end{array}}{env\,\{\,\mathbf{I} = ioer\,\}\,/\,rdexp_1\;\mathbf{pwd}\;rdexp_2 \longrightarrow env\,\{\,\mathbf{T} = term'\,,\,\mathbf{F} = fsys'\,,\,\mathbf{I} = ioer\,\}\,/\,str} \quad \text{OP\_EXP\_PWD}$$

$$\frac{env \,/\, e \longrightarrow env' \,/\, e'}{env \,/\, rdexp_1\;\mathbf{echo}\;rdexp_2\;e\;rdexp_3 \longrightarrow env' \,/\, rdexp_1\;\mathbf{echo}\;rdexp_2\;e'\;rdexp_3} \quad \text{OP\_EXP\_ECHO\_1}$$

$$\frac{\begin{array}{l} env\,\{\,\mathbf{I} = ioer\,\}\,/\,rdexp_1 \longrightarrow env\,\{\,\mathbf{I} = ioer'\,\}\,/ \\ env\,\{\,\mathbf{I} = ioer'\,\}\,/\,rdexp_2 \longrightarrow env\,\{\,\mathbf{I} = ioer''\,\}\,/ \\ env\,\{\,\mathbf{I} = ioer''\,\}\,/\,rdexp_3 \longrightarrow env\,\{\,\mathbf{I} = ioer'''\,\}\,/ \\ env\,\{\,\mathbf{T} = term\,,\,\mathbf{F} = fsys\,,\,\mathbf{I} = ioer'''\,\}\,/\,\mathbf{write}\;c \longrightarrow \mathbf{io}\;env\,\{\,\mathbf{T} = term'\,,\,\mathbf{F} = fsys'\,\}\,/\,\mathbf{empty} \end{array}}{env \,/\, rdexp_1\;\mathbf{echo}\;rdexp_2\;c\;rdexp_3 \longrightarrow env\,\{\,\mathbf{T} = term'\,,\,\mathbf{F} = fsys'\,,\,\mathbf{I} = ioer\,\}\,/\,0} \quad \text{OP\_EXP\_ECHO\_2}$$

$$\frac{\begin{array}{l} env\,\{\,\mathbf{I} = ioer\,\}\,/\,rdexp_1 \longrightarrow env\,\{\,\mathbf{I} = ioer'\,\}\,/ \\ env\,\{\,\mathbf{I} = ioer'\,\}\,/\,rdexp_2 \longrightarrow env\,\{\,\mathbf{I} = ioer''\,\}\,/ \\ env\,\{\,\mathbf{I} = ioer''\,\}\,/\,rdexp_3 \longrightarrow env\,\{\,\mathbf{I} = ioer'''\,\}\,/ \\ env\,\{\,\mathbf{I} = ioer'''\,\}\,/\,\mathbf{read}\;str \longrightarrow \mathbf{io}\;env\,\{\,\mathbf{S} = s'\,,\,\mathbf{I} = ioer'''\,\}\,/\,\mathbf{empty} \end{array}}{env \,/\, rdexp_1\;\mathbf{read}\;rdexp_2\;str\;rdexp_3 \longrightarrow env\,\{\,\mathbf{S} = s\,,\,\mathbf{I} = ioer\,\}\,/\,0} \quad \text{OP\_EXP\_READ\_1}$$

$$\frac{env\,\{\,\mathbf{I}\,=\,ioer\,\}\,/\,rdexp_1\;\longrightarrow\;env\,\{\,\mathbf{I}\,=\,ioer'\,\}\,/}{env\,\{\,\mathbf{I}\,=\,ioer\,\}\,/\,\mathbf{exec}\,rdexp_1\;\longrightarrow\;env\,\{\,\mathbf{I}\,=\,ioer'\,\}\,/\,0}\;\text{OP\_EXP\_EXEC}$$

$\boxed{env_1\,/\,aexp_1\;\longrightarrow\;env_2\,/\,aexp_2}$

$$\frac{env\,/\,aexp_1\;\longrightarrow\;env'\,/\,aexp_1'}{env\,/\,aexp_1\;bop\;aexp_2\;\longrightarrow\;env'\,/\,aexp_1'\;bop\;aexp_2}\;\text{OP\_ARITHEXP\_BOP\_1}$$

$$\frac{env\,/\,aexp_2\;\longrightarrow\;env'\,/\,aexp_2'}{env\,/\,integer\_literal\;bop\;aexp_2\;\longrightarrow\;env'\,/\,integer\_literal\;bop\;aexp_2'}\;\text{OP\_ARITHEXP\_BOP\_2}$$

$$\frac{}{env\,/\,int_1\;bop\;int_2\;\longrightarrow\;env\,/\,\mathbf{val}\,(\,int_1\;bop\;int_2\,)}\;\text{OP\_ARITHEXP\_BOP\_3}$$

$$\frac{env\,/\,aexp_1\;\longrightarrow\;env'\,/\,aexp_1'}{env\,/\,preuop\;aexp_1\;\longrightarrow\;env'\,/\,preuop\;aexp_1'}\;\text{OP\_ARITHEXP\_UOP\_1\_1}$$

$$\frac{}{env\,/\,preuop\;int_1\;\longrightarrow\;env\,/\,\mathbf{val}\,(\,preuop\;int_1\,)}\;\text{OP\_ARITHEXP\_UOP\_1\_2}$$

$$\frac{}{env\,\{\,\mathbf{S}\;\mathbf{inc}\;str\;\mapsto\;c\,\}\,/\,vpreuop\;str\;\longrightarrow\;env\,/\,\mathbf{val}\,(\,vpreuop\;\mathbf{constval}\,(\,c\,)\,)}\;\text{OP\_ARITHEXP\_UOP\_2}$$

$$\frac{}{env\,\{\,\mathbf{S}\;\mathbf{inc}\;str\;\mapsto\;c\,\}\,/\,str\;postuop\;\longrightarrow\;env\,/\,\mathbf{val}\,(\,\mathbf{constval}\,(\,c\,)\;postuop\,)}\;\text{OP\_ARITHEXP\_UOP\_3}$$

$$\frac{}{env\,\{\,\mathbf{S}\;\mathbf{inc}\;str\;\mapsto\;c\,\}\,/\,str\;\longrightarrow\;env\,/\,\mathbf{constval}\,(\,c\,)}\;\text{OP\_ARITHEXP\_DEREF\_1\_1}$$

$$\frac{}{env\,\{\,\mathbf{S}\;\mathbf{ninc}\;str\;\mapsto\;c\,\}\,/\,str\;\longrightarrow\;env\,/\,\mathbf{constval}\,(\,0\,)}\;\text{OP\_ARITHEXP\_DEREF\_1\_2}$$

$$\frac{}{env\,\{\,\mathbf{S}\;\mathbf{inc}\;str\;\mapsto\;c\,\}\,/\,\$\{\,str\,\}\;\longrightarrow\;env\,/\,\mathbf{constval}\,(\,c\,)}\;\text{OP\_ARITHEXP\_DEREF\_2\_1}$$

$$\frac{}{env\,\{\,\mathbf{S}\;\mathbf{ninc}\;str\;\mapsto\;c\,\}\,/\,\$\{\,str\,\}\;\longrightarrow\;env\,/\,\mathbf{constval}\,(\,0\,)}\;\text{OP\_ARITHEXP\_DEREF\_2\_2}$$

$$\frac{}{env\,\{\,\mathbf{F\_S}\,=\;\;[\,funmem\,,\;\;int_1\;\mapsto\;c\,]\,,\,fun\_s\,\}\,/\,\$\{\,int_1\,\}\;\longrightarrow\;env\,/\,\mathbf{constval}\,(\,c\,)}\;\text{OP\_ARITHEXP\_DEREF\_PARAM\_1}$$

$$\frac{}{env\,\{\,\mathbf{F\_S}\,=\;\;[\,funmem\,]\,,\,fun\_s\,\}\,/\,\$\{\,int_1\,\}\;\longrightarrow\;env\,/\,\mathbf{constval}\,(\,0\,)}\;\text{OP\_ARITHEXP\_DEREF\_PARAM\_2}$$

$$\frac{env\,/\,aexp\;\longrightarrow\;env'\,/\,aexp'}{env\,/\,((\,aexp\,))\;\longrightarrow\;env'\,/\,((\,aexp'\,))}\;\text{OP\_ARITHEXP\_COND\_1}$$

$$\frac{}{env\,/\,((\,0\,))\;\longrightarrow\;env\,/\,1}\;\text{OP\_ARITHEXP\_COND\_2}$$

$$\frac{}{env \,/\, ((\, int \,)) \;\longrightarrow\; env \,/\, 0} \quad \text{OP\_ARITHEXP\_COND\_3}$$

$\boxed{env_1 \,/\, elist_1 \;\longrightarrow\; env_2 \,/\, elist_2}$

$$\frac{}{env\,\{\,\mathbf{S\,inc}\; str \;\mapsto\; c\,\}\,/\,\$\{\,str\,\} \;\longrightarrow\; env \,/\, c} \quad \text{OP\_ELIST\_DEREF\_1}$$

$$\frac{}{env\,\{\,\mathbf{S\,ninc}\; str \;\mapsto\; c\,\}\,/\,\$\{\,str\,\} \;\longrightarrow\; env \,/\, \mathbf{blank\_string}} \quad \text{OP\_ELIST\_DEREF\_2}$$

$$\frac{}{env\,\{\,\mathbf{F\_S} = \;\; [\,funmem\,,\;\; int_1 \;\mapsto\; c_1\,]\;,\; fun\_s\,\}\,/\,\$\{\,int_1\,\} \;\longrightarrow\; env \,/\, c_1} \quad \text{OP\_ELIST\_DEREF\_PARAM\_1}$$

$$\frac{}{env\,\{\,\mathbf{F\_S} = \;\; [\,funmem'\,]\;,\; fun\_s\,\}\,/\,\$\{\,int_1\,\} \;\longrightarrow\; env \,/\, \mathbf{blank\_string}} \quad \text{OP\_ELIST\_DEREF\_PARAM\_2}$$

$$\frac{env \,/\, elist \;\longrightarrow\; env' \,/\, elist'}{env \,/\, c\; elist \;\longrightarrow\; env' \,/\, c\; elist'} \quad \text{OP\_ELIST\_EVAL}$$

$$\frac{}{env\,\{\,\mathbf{F\_S} = \;\; [\;int_1 \;\mapsto\; c_1\;,\; ..\; int_n \;\mapsto\; c_n\;,\;]\;,\; fun\_s\,\}\,/\,\$@ \;\longrightarrow\; env \,/\, <\; c_1\; ..\; c_n\; >} \quad \text{OP\_ELIST\_DEREF\_ALL\_PARAMS}$$

$\boxed{env_1 \,/\, ce_1 \;\longrightarrow\; env_2 \,/\, ce_2}$

$$\frac{env \,/\, f \;\longrightarrow\; env' \,/\, f'}{env \,/\, f \,|\, ce \;\longrightarrow\; env' \,/\, f' \,|\, ce} \quad \text{OP\_CASEEXP\_CASEEXP\_1}$$

$$\frac{env \,/\, ce \;\longrightarrow\; env' \,/\, ce'}{env \,/\, c \,|\, ce \;\longrightarrow\; env' \,/\, c \,|\, ce'} \quad \text{OP\_CASEEXP\_CASEEXP\_2}$$

$$\frac{env \,/\, f \;\longrightarrow\; env' \,/\, f'}{env \,/\, f \;\longrightarrow\; env' \,/\, f'} \quad \text{OP\_CASEEXP\_CASEEXP\_3}$$

$\boxed{env_1 \,/\, oce_1 \;\longrightarrow\; env_2 \,/\, oce_2}$

$$\frac{env \,/\, ce \;\longrightarrow\; env' \,/\, ce'}{env \,/\, (\; ce\;)\, f\;;; \;\longrightarrow\; env' \,/\, (\; ce'\;)\, f\;;;} \quad \text{OP\_OCASEEXP\_OCE\_1}$$

$$\frac{}{\begin{array}{l} env \,/\, (\;\; c \,|\, c'_1 \,|\, ..\, c'_n \,|\, c''\;)\, f\;;;\; oce \\ \longrightarrow\; env \,/\, (\;\; \mathbf{strval}(\, c\,) \,|\, \mathbf{strval}(\, c'_1\,) \,|\, ..\, \mathbf{strval}(\, c'_n\,) \,|\, \mathbf{strval}(\, c''\,)\;)\, f\;;;\; oce \end{array}} \quad \text{OP\_OCASEEXP\_OCE\_2}$$

$$\frac{env \,/\, oce \;\longrightarrow\; env' \,/\, oce'}{env \,/\, (\;\; str \,|\, str'_1 \,|\, ..\, str'_n \,|\, str''\;)\, f\;;;\; oce \;\longrightarrow\; env' \,/\, (\;\; str \,|\, str'_1 \,|\, ..\, str'_n \,|\, str''\;)\, f\;;;\; oce'} \quad \text{OP\_OCASEEXP\_OCE\_3}$$

$\boxed{env_1 \,/\, condexp_1 \longrightarrow env_2 \,/\, condexp_2}$

$$\frac{env \,/\, f \longrightarrow env' \,/\, f'}{env \,/\, -\mathbf{a}\, f \longrightarrow env' \,/\, -\mathbf{a}\, f'}\ \text{OP\_CONDEXP\_FILE\_EXISTS\_1}$$

$$\frac{}{env \,/\, -\mathbf{a}\, str \longrightarrow env \,/\, b}\ \text{OP\_CONDEXP\_FILE\_EXISTS\_2}$$

$$\frac{env \,/\, condexp \longrightarrow env' \,/\, condexp'}{env \,/\, !\, condexp \longrightarrow env' \,/\, !\, condexp'}\ \text{OP\_CONDEXP\_NOT\_1}$$

$$\frac{}{env \,/\, !\, \mathbf{false} \longrightarrow env \,/\, \mathbf{true}}\ \text{OP\_CONDEXP\_NOT\_2}$$

$$\frac{}{env \,/\, !\, \mathbf{true} \longrightarrow env \,/\, \mathbf{false}}\ \text{OP\_CONDEXP\_NOT\_3}$$

$$\frac{env \,/\, condexp_1 \longrightarrow env' \,/\, condexp_1'}{env \,/\, condexp_1 \,\&\&\, condexp_2 \longrightarrow env' \,/\, condexp_1' \,\&\&\, condexp_2}\ \text{OP\_CONDEXP\_AND\_1}$$

$$\frac{}{env \,/\, \mathbf{true} \,\&\&\, condexp_2 \longrightarrow env \,/\, condexp_2}\ \text{OP\_CONDEXP\_AND\_2}$$

$$\frac{}{env \,/\, \mathbf{false} \,\&\&\, condexp_2 \longrightarrow env \,/\, \mathbf{false}}\ \text{OP\_CONDEXP\_AND\_3}$$

$$\frac{env \,/\, condexp_1 \longrightarrow env' \,/\, condexp_1'}{env \,/\, condexp_1 \,||\, condexp_2 \longrightarrow env' \,/\, condexp_1' \,||\, condexp_2}\ \text{OP\_CONDEXP\_OR\_1}$$

$$\frac{}{env \,/\, \mathbf{false} \,||\, condexp_2 \longrightarrow env \,/\, condexp_2}\ \text{OP\_CONDEXP\_OR\_2}$$

$$\frac{}{env \,/\, \mathbf{true} \,||\, condexp_2 \longrightarrow env \,/\, \mathbf{true}}\ \text{OP\_CONDEXP\_OR\_3}$$

$\boxed{env_1 \,/\, rdexp_1 \longrightarrow env_2 \,/\, rdexp_2}$

$$\frac{}{\begin{array}{l} env\,\{\,\mathbf{I} = [\,ioersub\,],\ ioer\,\} \,/\, int\ >\ str\ rdexp \\ \longrightarrow env\,\{\,\mathbf{FS\ inc\ fsval(str)} \mapsto \mathbf{blank\_string}\,,\ \mathbf{I} = [\,ioersub\,,\,\{\,int \mapsto \mathbf{fsval}(\,str\,)\,,\,\mathbf{out}\,\}\,],\ ioer\,\} \,/\, rdexp \end{array}}\ \text{OP\_RDEXP\_FD\_OUT\_TO\_FILE}$$

$$\frac{}{\begin{array}{l} env\,\{\,\mathbf{I} = [\,ioersub\,],\ ioer\,\} \,/\, int >> str\ rdexp \\ \longrightarrow env\,\{\,\mathbf{I} = [\,ioersub\,,\,\{\,int \mapsto \mathbf{fsval}(\,str\,)\,,\,\mathbf{app}\,\}\,],\ ioer\,\} \,/\, rdexp \end{array}}\ \text{OP\_RDEXP\_REDIRECT\_FD\_APP\_TO\_FILE}$$

$$\frac{}{env\,\{\,\mathbf{I} = [\,ioersub\,]\,,\,ioer\,\}\,/\,int\,<\,str\,rdexp \longrightarrow env\,\{\,\mathbf{I} = [\,ioersub\,,\,\{\,int \mapsto \mathbf{fsval}(\,str\,)\,,\,\mathbf{in}\,\}\,]\,,\,ioer\,\}\,/\,rdexp}\ \text{OP\_RDEXP\_REDIRECT\_FD\_IN\_TO\_FILE}$$

$$\frac{}{env\,\{\,\mathbf{I} = [\,ioersub\,]\,,\,ioer\,\}\,/\,>\,str\,rdexp \longrightarrow env\,\{\,\mathbf{FS\ inc\ fsval(str)} \mapsto \mathbf{blank\_string}\,,\,\mathbf{I} = [\,ioersub\,,\,\{\,1 \mapsto \mathbf{fsval}(\,str\,)\,,\,\mathbf{out}\,\}\,]\,,\,ioer\,\}\,/\,rdexp}\ \text{OP\_RDEXP\_REDIRECT\_STD\_OUT\_TO\_FILE}$$

$$\frac{}{env\,\{\,\mathbf{I} = [\,ioersub\,]\,,\,ioer\,\}\,/\,>>\,str\,rdexp \longrightarrow env\,\{\,\mathbf{I} = [\,ioersub\,,\,\{\,1 \mapsto \mathbf{fsval}(\,str\,)\,,\,\mathbf{app}\,\}\,]\,,\,ioer\,\}\,/\,rdexp}\ \text{OP\_RDEXP\_REDIRECT\_STD\_OUT\_TO\_FILE\_APP}$$

$$\frac{}{env\,\{\,\mathbf{I} = [\,ioersub\,]\,,\,ioer\,\}\,/\,<\,str\,rdexp \longrightarrow env\,\{\,\mathbf{I} = [\,ioersub\,,\,\{\,0 \mapsto \mathbf{fsval}(\,str\,)\,,\,\mathbf{in}\,\}\,]\,,\,ioer\,\}\,/\,rdexp}\ \text{OP\_RDEXP\_REDIRECT\_STD\_IN\_TO\_FILE}$$

$$\frac{}{env\,/\,\&\,>\,str\,rdexp \longrightarrow env\,\{\,\mathbf{I} = [\,ioersub\,,\,\{\,1 \mapsto \mathbf{fsval}(\,str\,)\,,\,\mathbf{out}\,\}\,,\,\{\,2 \mapsto \mathbf{fsval}(\,str\,)\,,\,\mathbf{err}\,\}\,]\,,\,ioer\,\}\,/\,rdexp}\ \text{OP\_RDEXP\_REDIRECT\_STD\_OUT\_ERR\_TO\_FILE}$$

$$\frac{}{env\,\{\,\mathbf{I} = [\,ioersub\,,\,\{\,int_2 \mapsto fd\,,\,fdtype\,\}\,]\,,\,ioer\,\}\,/\,int_1\,>\,\&\,int_2\,rdexp \longrightarrow env\,\{\,\mathbf{I} = [\,ioersub\,,\,\{\,int_1 \mapsto fd\,,\,fdtype\,\}\,,\,\{\,int_2 \mapsto fd\,,\,fdtype\,\}\,]\,,\,ioer\,\}\,/\,rdexp}\ \text{OP\_RDEXP\_REDIRECT\_FD\_OUT\_TO\_FD\_OUT}$$

$$\frac{}{env\,\{\,\mathbf{I} = [\,ioersub\,,\,\{\,int_2 \mapsto fd\,,\,fdtype\,\}\,]\,,\,ioer\,\}\,/\,int_1\,<\,\&\,int_2\,rdexp \longrightarrow env\,\{\,\mathbf{I} = [\,ioersub\,,\,\{\,int_1 \mapsto fd\,,\,fdtype\,\}\,,\,\{\,int_2 \mapsto fd\,,\,fdtype\,\}\,]\,,\,ioer\,\}\,/\,rdexp}\ \text{OP\_RDEXP\_REDIRECT\_FD\_IN\_TO\_FD\_IN}$$

$$\frac{}{env\,\{\,\mathbf{I} = [\,ioersub\,]\,,\,ioer\,\}\,/\,int<>\,str\,rdexp \longrightarrow env\,\{\,\mathbf{I} = [\,ioersub\,,\,\{\,int \mapsto \mathbf{fsval}(\,str\,)\,,\,\mathbf{ran}\,\}\,]\,,\,ioer\,\}\,/\,rdexp}\ \text{OP\_RDEXP\_REDIRECT\_IN\_OUT\_TO\_FILE}$$

$$\frac{}{env\,\{\,\mathbf{I} = [\,ioersub\,,\,\{\,int_2 \mapsto fd\,,\,fdtype\,\}\,]\,,\,ioer\,\}\,/\,int_1\,<\,\&-\,rdexp \longrightarrow env\,\{\,\mathbf{I} = [\,ioersub\,]\,,\,ioer\,\}\,/\,rdexp}\ \text{OP\_RDEXP\_CLOSE\_RD}$$

$\boxed{env_1\,/\,io_1 \longrightarrow\!\mathbf{io}\ env_2\,/\,io_2}$

$$\frac{}{env\,\{\,\mathbf{I} = [\,ioersub\,,\,\{\,int \mapsto \mathbf{terminal}\,,\,\mathbf{out}\,\}\,]\,,\,ioer\,,\,\mathbf{T} = term\,\}\,/\,\mathbf{write}\,c \longrightarrow\!\mathbf{io}\ env\,\{\,\mathbf{T} = term\,\mathbf{value}\,(\,c\,)\,\}\,/\,\mathbf{empty}}\ \text{OP\_IO\_WRITE\_TERM\_1}$$

$$\frac{}{env\,\{\,\mathbf{T} = term\,,\,\mathbf{I} = [\,ioersub\,,\,\{\,int \mapsto \mathbf{terminal}\,,\,\mathbf{ran}\,\}\,]\,,\,ioer\,\}\,/\,\mathbf{write}\,c \longrightarrow\!\mathbf{io}\ env\,\{\,\mathbf{T} = term\,\mathbf{value}\,(\,c\,)\,\}\,/\,\mathbf{empty}}\ \text{OP\_IO\_WRITE\_TERM\_2}$$

$$\frac{}{env\,\{\,\mathbf{FS\ inc}\,fname_1 \mapsto str_1\,,\,\mathbf{I} = [\,ioersub\,,\,\{\,int \mapsto fname_1\,,\,\mathbf{out}\,\}\,]\,,\,ioer\,\}\,/\,\mathbf{write}\,c \longrightarrow\!\mathbf{io}\ env\,\{\,\mathbf{FS\ inc}\,fname_1 \mapsto \mathbf{write}\,(\mathbf{str1}, \mathbf{c})\,\}\,/\,\mathbf{empty}}\ \text{OP\_IO\_WRITE\_FILE\_1}$$

$$\frac{}{env\,\{\,\mathbf{FS\ inc}\,fname_1 \mapsto str_1\,,\,\mathbf{I} = [\,ioersub\,,\,\{\,int \mapsto fname_1\,,\,\mathbf{ran}\,\}\,]\,,\,ioer\,\}\,/\,\mathbf{write}\,c \longrightarrow \mathbf{io}\,env\,\{\,\mathbf{FS\ inc}\,fname_1 \mapsto \mathbf{write}\,(\mathbf{str1}, \mathbf{c})\,\}\,/\,\mathbf{empty}}\ \text{OP\_IO\_WRITE\_FILE\_2}$$

$$\frac{}{env\,\{\,\mathbf{FS\ inc}\,fname_1 \mapsto str_1\,,\,\mathbf{I} = [\,ioersub\,,\,\{\,int \mapsto fname_1\,,\,\mathbf{app}\,\}\,]\,,\,ioer\,\}\,/\,\mathbf{write}\,c \longrightarrow \mathbf{io}\,env\,\{\,\mathbf{FS\ inc}\,fname_1 \mapsto \mathbf{write}\,(\mathbf{str1}, \mathbf{c})\,\}\,/\,\mathbf{empty}}\ \text{OP\_IO\_WRITE\_FILE\_3}$$

$$\frac{}{env\,\{\,\mathbf{FS\ inc}\,fname_1 \mapsto str_1\,,\,\mathbf{I} = [\,ioersub\,,\,\{\,int \mapsto fname_1\,,\,\mathbf{in}\,\}\,]\,,\,ioer\,\}\,/\,\mathbf{read}\,str \longrightarrow \mathbf{io}\,env\,\{\,\mathbf{S\ inc}\,str \mapsto str_2\,,\,\mathbf{FS\ inc}\,fname_1 \mapsto \mathbf{read}(\mathbf{str1}, \mathbf{str2})\,\}\,/\,\mathbf{empty}}\ \text{OP\_IO\_READ\_1\_1}$$

$$\frac{}{env\,\{\,\mathbf{S} = [\,mem\,]\,,\,s\,,\,\mathbf{FS\ inc}\,fname_1 \mapsto str_1\,,\,\mathbf{I} = [\,ioersub\,,\,\{\,int \mapsto fname_1\,,\,\mathbf{in}\,\}\,]\,,\,ioer\,\}\,/\,\mathbf{read}\,str \longrightarrow \mathbf{io}\,env\,\{\,\mathbf{FS\ inc}\,fname_1 \mapsto \mathbf{read}(\mathbf{str1}, \mathbf{str2})\,,\,\mathbf{S} = [\,mem\,,\ str \mapsto str_2\,]\,,\,s\,\}\,/\,\mathbf{empty}}\ \text{OP\_IO\_READ\_1\_2}$$

$$\frac{}{env\,\{\,\mathbf{S\ inc}\,str \mapsto str_3\,,\,\mathbf{FS\ inc}\,fname_1 \mapsto str_1\,,\,\mathbf{I} = [\,ioersub\,,\,\{\,int \mapsto fname_1\,,\,\mathbf{ran}\,\}\,]\,,\,ioer\,\}\,/\,\mathbf{read}\,str \longrightarrow \mathbf{io}\,env\,\{\,\mathbf{S\ inc}\,str \mapsto str_2\,,\,\mathbf{FS\ inc}\,fname_1 \mapsto \mathbf{read}(\mathbf{str1}, \mathbf{str2})\,\}\,/\,\mathbf{empty}}\ \text{OP\_IO\_READ\_2\_1}$$

$$\frac{}{env\,\{\,\mathbf{S} = [\,mem\,]\,,\,s\,,\,\mathbf{FS\ inc}\,fname_1 \mapsto str_1\,,\,\mathbf{I} = [\,ioersub\,,\,\{\,int \mapsto fname_1\,,\,\mathbf{ran}\,\}\,]\,,\,ioer\,\}\,/\,\mathbf{read}\,str \longrightarrow \mathbf{io}\,env\,\{\,\mathbf{S} = [\,mem\,,\ str \mapsto str_2\,]\,,\,s\,,\,\mathbf{FS\ inc}\,fname_1 \mapsto \mathbf{read}(\mathbf{str1}, \mathbf{str2})\,\}\,/\,\mathbf{empty}}\ \text{OP\_IO\_READ\_2\_2}$$

Appendix C

Project proposal

Austin Anderson
Gonville and Caius College
ama42

Part II Computer Science Project Proposal

Language definition and clean implementation of a subset of Bash

Project Originator: Dr Peter Sewell
Resources Required: See the attached Project Resource Form

Project Supervisor: Tom Ridge
Signature

Director of Studies: Peter Robinson
Signature

Overseers: Neil Dodgson and Tim Griffin
Signature

Proposal in brief

Design an operational semantics for and implement in ML a scripting language with Bash like syntax and behaviour.

Introduction

Shells are used for many everyday administrative tasks, both in the interactive and scripted form. The definition of how many of the shells work was done several decades ago before significant advances in the development of rigorous techniques for programming language design and definition. Therefore I propose to make an operational semantics for a shell with Bash like syntax (this will hopefully enable current users of Bash to use this tool without much retraining). This will allow a concrete definition as to what the scripting language should do (instead of a prose one with many corner cases), which could be used to prove theorems (such as determinacy) about the language.

I also propose to do an implementation in ML of the defined operational semantics.

Resources required

The only resource required is access to the SRCF in order to perform backups and a version control system. This is accessible using standard protocols.

Starting point

An ML implementation of the Scsh shell exists in a project named Cash. This may be used for reference in terms of implementation of some common features between the shells, but it is not expected that there will be a huge overlap in terms of the implementation code and code reuse.

The Bash specification will be used as a basis for much of the definition of the syntax/implementation. The Bash specification can be found at http://www.gnu.org/software/bash/manual/bashref.html

OTT is a tool for defining language specifications in ways which can easily be output in multiple formats, including those appropriate for machine verification. Information on OTT can be found at http://www.cl.cam.ac.uk/ pes20/ott

Project Substance

The Bash specification is quite large and complicated, despite the fact that it is one of the shorter shell definitions. Hence it is not feasible to attempt to implement or even define a semantics for much of its functionality. To that end, possible work areas in terms of sections of the Bash specification to be implemented have been divided up as follows (where the bracketed numbers indicate relevant sections of the Bash specification):

Core functionality

- Looping constructs (3.2.4)
- Conditional constructs (3.2.5)
- Grouping commands (3.2.6)
- Shell parameter expansion (3.5.3)
- Arithmetic expansions (3.5.5)
- Bash conditional expressions (6.4)
- Shell arithmetic (6.5)

Extensions expected to be done

- Pipelines (3.2.2)
- Brace expansion (3.5.1)
- Tilde expansion (3.5.2)
- Redirections (3.6)
- Aliases (6.6)

- Arrays (6.7)

Extra extensions (to be done if everything else goes very quickly)

- Lists of commands (3.2.3)
- Command substitutions (3.5.4)
- Process substitution (3.5.6)
- Word splitting (3.5.7)
- Filename expansion (3.5.8)
- Quoting (3.1.2)
- Shell functions (3.3)
- Shell Parameters (3.4)
- Shell Scripts (3.8)
- Bourne shell builtins (4.1)
- Some of the Bash builtins (4.2)
- Some clean way to get functionality similar to the Shell Variables (5)
- The directory stack (6.8)
- Job Control (7)

The syntax and semantics will be designed to be Bash-like, however one of the main reasons to do a clean definition and implementation is to get rid of some of the more obscure corner cases and effects, not to find new ways of replicating them, and hence the definition will differ (perhaps significantly) from the original Bash specification.

A type system would be useful in terms of a clean language definition, but it is a sufficiently complex sub-problem that it is categorised as an extra extension.

The most important part of the project is the language design, and the implementation is primarily an agent to allow easier evaluation as to how effective a design it is, and to see some of the practicabilities of programming language design.

Criteria

Minimal functionality:

in a worst case I would submit the core language definition and as much of the language definition of the extensions as possible.

Core functionality:

this will consist of the core language definition and core implementation, as defined in the Project Substance section.

Implementation:

The language definition will be in OTT. The implementation will be in ML, and the language will be interpreted. The lexing of the language will be done using libraries, probably Ocamllex.

Evaluation:

The language definition will be evaluated against the Bash definition. This will both be done by hand in terms of comparing the syntaxes. Differences between the Bash specification and the language semantics which seem significant will be explicitly emphasised. If they are appropriate HCI techniques will be used to evaluate the ease of migration between Bash and the new shell. The implementation will be evaluated programmatically by creating test scripts and viewing the results running then in Bash and running them in my shell (for scripts whose syntax is valid in both). If there is time I will make use of the tools suggested by Peter Sewell for verification of the implementation against the specification (this is an extension in terms of evaluation as it requires learning a new tool).

Plan of work

1. Oct 22-Nov 2:
    (a) Background reading
        i. Revise operational semantics

ii. Read ahead in Types
iii. Read the source implementation of Cash

2. Nov 5-Nov 16:
   (a) Core goals language definition
   (b) Research the lexing tools available and do some examples
3. Nov 19-Nov 30:
   (a) Core goals implementation
      i. Lexing framework
      ii. Arithmetic expansions (3.5.5)
      iii. Bash conditional expressions (6.4)
      iv. Shell arithmetic (6.5)
4. Dec 3- Dec 14:
   (a) Core goals implementation
      i. Looping constructs (3.2.4)
      ii. Conditional constructs (3.2.5)
      iii. Grouping commands (3.2.6)
      iv. Shell parameter expansion (3.5.3)
   (b) Evaluation
5. Jan 7-Jan 18:
   (a) Extensions language definition
   (b) Extensions implementation
      i. Arrays (6.7)
      ii. Brace expansion (3.5.1)
      iii. Tilde expansion (3.5.2)
6. Jan 21-Feb 1:
   (a) Extensions implementation

i. Pipelines (3.2.2)
ii. Redirections (3.6)

7. Feb 4-Feb 15:
   (a) Extensions implementation
      i. Lists of commands (3.2.3)
      ii. Aliases (6.6)
8. Feb 18-Feb 29:
   (a) Write up (core)
9. Mar 3-Mar 14:
   (a) Write up (extensions expected to be done)
10. Mar 24-Apr 4:
   (a) Write up (any additional extensions done)
   (b) Submit first complete draft (April 4th)

Deliverables for each work package

1. Oct 22-Nov 2:
   (a) Notes on any design decisions made
2. Nov 5-Nov 16:
   (a) OTT definition of the core language syntax and operational semantics
   (b) lexing software installed and some basic example languages are being lexed
3. Nov 19-Nov 30:
   (a) lexing framework implemented
   (b) ML implementation of the goals mentioned
4. Dec 3- Dec 14:

   (a) ML implementation of the goals mentioned
   (b) Evaluate the work done so far versus the criteria and the deliverables

5. Jan 7-Jan 18:
   (a) Progress report produced using the evaluation work done in the previous deliverable
   (b) OTT definition of the language syntax and operational semantics of the extensions that are expected to be done
   (c) ML implementation of the goals mentioned

6. Jan 21-Feb 1:
   (a) ML implementation of the goals mentioned

7. Feb 4-Feb 15:
   (a) ML implementation of the goals mentioned

8. Feb 18-Feb 29:
   (a) Draft write up of the core sections of the project

9. Mar 3-Mar 14:
   (a) Draft write up of the extensions which were expected to be done

10. Mar 24-Apr 4:
    (a) Draft write up of any additional extensions
    (b) Complete draft submitted