Comput Lang Vol 11, No 2, pp 85–104, 1986
Printed in Great Britain

0096-0551 86 $3 00 + 0 00
Pergamon Journals Ltd

INTERPRETERS FROM FUNCTIONS AND GRAMMARS

GREG MICHAELSON
Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, U K.

(*Received* 23 *September* 1985)

Abstract—The addition of context free grammar rules to a functional language simplifies the construction of interpreters from denotational semantic language definitions. Functional abstraction over grammar rules enables the specification and processing of context sensitive language syntax aspects in a functional style.

Denotational semantics Language design Language implementation Functional language Grammar rules Interpreters Context sensitive processing

1. INTRODUCTION

1.1 *Denotational semantics*

Denotational semantics [1] is a formalism for defining programming languages. A denotational semantic definition consists of semantic domains, abstract syntax and semantic functions. The semantic domains define the universe of objects in which the language is given meaning, the abstract syntax defines meaningful syntactic constructs and the meaning of each abstract syntax construct is defined by a semantic function over the semantic domains. Abstract syntax is distinguished from concrete syntax which describes the physical representation of programs and contains details which are not significant from a semantic point of view [2].

Approaches based on denotational semantics are used to provide succinct and theoretically rigorous programming language standards. These then inform language implementations and program manipulation. Such approaches are also used for programming language design [3] but there are few software tools which are appropriate for experimenting with denotational semantic based designs.

1.2 *Compiler generators*

Compiler generators [4], which are used to automate the construction of production compilers, are based on very different methodologies to denotational semantics and place a far greater emphasis on concrete syntax. A typical compiler generator, like YACC [5], inputs a formalised definition of the context free [6] concrete syntax of a language along with semantic actions written in an imperative language. These implement context sensitive concrete syntax checking and the generation of the target language.

When compiler generators are used with denotational semantic designs a basic problem is the lack of correspondence between the functional meta-language used to define semantic equations and the imperative semantic action language. Translating semantic equations into semantic actions loses clarity and is a potential source of errors. When the design changes so must the translation for the compiler generator.

One alternative is to use an existing compiler generator with a new semantic action language which has a well defined correspondence with the functional meta-language [7]. Another is to develop a system which is oriented specifically for use with denotational semantics.

1.3 *SIS*

SIS [8] is a language implementation system which enables direct experimentation with denotational semantic language designs. It takes in a full denotational semantic definition and uses it to generate what is in effect the code for an abstract machine. The code is then interpreted.

The strict adherence to denotational semantics, however, with the formal separation of concrete syntax, abstract syntax and semantics, appears to restrict the ease with which SIS may be used.

In particular, the denotational semantic methodology treats concrete syntax considerations as of secondary importance with context sensitive syntax aspects often bring ignored or pushed into the semantics. Concrete syntax is a major part of a language design as it determines the physical appearance of programs and influences the ease with which they are understood and manipulated. SIS provides grammar rules for specifying a context free concrete syntax and associating it with the abstract syntax but there is no provision for context sensitivity other than incorporation in semantic equations.

SIS is essentially a batch system and definitions are frozen by compiler-compilation. Thus, there is no facility for moving easily from definition to test without using the host filing system and associated tools, and re-compiling the whole definition for each change. There is, of course, no reason why SIS should not be implemented within an interactive environment but the need for separate but related sub-definitions may restrict flexibility and integration.

1.4 *Navel*

Navel*, which is described in the rest of this paper, was conceived of as an interpreter-interpreter language which would enable the interactive testing and modification of language designs. It is a functional language, heavily influenced by the original 'let... in...' SASL [9], and has grammar rules fully integrated as functional objects. Navel interpreters are written in a denotational semantic style using functional semantic equations but, unlike SIS, denotational semantics is not directly implemented. The main differences are the loss of the traditional distinction between the syntactic and semantic meta-languages, the lack of need for abstract syntax to mediate concrete syntax and semantics, and the absence of semantic domain specifications. These differences lose much of the rigour of SIS but simplify interpretive implementation from a formal definition. Context sensitive aspects of concrete syntax may be specified in a functional style through higher order grammar rules.

Subsequent sections describe the functional language, the construction of an interpreter for a small language from a denotational semantic definition, rule generalisation through functional abstraction and the use of higher order rules for checking assignment before use, declarations and type consistency.

2. NAVEL

2.1 *Introduction*

Navel [10] is a weakly typed functional language. It is implemented within an interactive environment which provides file access to the host system and local editing facilities.

The main features of the language are described through examples. Each example may pre-suppose preceding examples. In examples, the prompt 'ok' heralds the input and execution of an expression or command ending with a ';'.

2.2 *Objects and definitions*

The Navel objects are integers, characters which coerce to integers, booleans, lists, fields, functions and rules. Global definition commands are used to establish name/object associations which are retained for use by subsequent definitions and expressions:

```
ok
def Value_Added_Tax = 15;
ok
Value_Added_Tax;
15
ok
def pie_beans_and_chips = 135;
ok
pie_beans_and_chips*Value_Added_Tax/100;
20
```

*Not A Very Exciting Language.

2.3 *Lists and strings*

The concatenation operator ':' is used to construct lists. The empty list is '( )'. The operators 'hd' and 'tl' are used to extract the head and tail from lists:

```
ok
def squares = 1:4:9:16:25:( );
ok
hd tl squares;
4
```

List elements may also be accessed through indexing.

The empty list is normally omitted from the end of lists:

```
ok
squares;
1:4:9:16:25:
```

Strings are lists of characters. They are usually denoted by sequences of characters within '"'s:

```
ok
def orange = 'o':'r':'a':'n':'g':'e':;
ok
orange;
"orange"
```

Lists are lazy evaluated [11], i.e. function calls in lists are not evaluated until they are selected. Lazy infinite lists may be constructed using recursive functions.

2.4 *Fields*

A field is an object tagged with a name and is denoted by the name followed by the object within '[' and ']'. The object is selected from a field with the operator ' ^ ':

```
ok
def plant = [tree "larch"];
ok
plant ^ tree;
"larch"
```

The field selection operator is overloaded to enable field object selection from lists:

```
ok
def plants = [tree "larch"]:[bush "privet"]:[tree "fir"]:;
ok
plants ^ bush;
"privet"
```

When there are several fields in a list with the same name, the index operator '@' is used to identify the requisite instance:

```
ok
plants ^ tree @ 2;
"fir"
```

2.5 *Functions*

Functions are based on lambda calculus [1]. They may be defined with a single formal parameter:

```
ok
def cube = lam x.x*x*x;
ok
cube 3;
27
```

or a list of formal parameters:

```
ok
def sum_squares = lam x:y.x*x + y*y;
ok
sum_squares 3:4;
25
```

For function definitions, the 'lam' may be dropped and the formal parameters moved to the left of the ' = 's:

```
ok
def volume length:depth;height = length*depth*height;
```

Function calls are evaluated in applicative order.

Formal parameter lists may be arbitrarily complicated and used for structure match selection with actual parameter lists:

```
ok
def tl_hd (hh:ht):t = ht;
ok
tl_hd ("bangers":"mash"):("fish":"chips"):;
"mash"
```

2.6 *Control expressions*

Navel provides 'if' and 'case' control expressions which enable the construction of recursive functions:

```
ok
def power x:n =
  if n = 0
  then 1
  else x*(power x:n-1);
ok
def fib n =
  case n of
  0-> 1,
  1-> 1,
  (fib n-1) + (fib n-2);
```

The 'case' expression ends with a default.

2.7 *Local definitions*

Local definitions are used to introduce temporary name/object associations within expressions:

```
ok
def sum_squares x:y =
  let sq n = n*n
  in (sq x) + (sq y);
```

They are evaluated in applicative order and are directly equivalent to function calls.

Local definitions may be mutually recursive. For global definitions, as in POP2 [12], a forward reference to an undefined name in a function definition results in the creation of a dummy definition for the name which may be fully defined later.

2.8 *Partial application*

Functions may return functions through partial application:

```
ok
def add x = lam y.x + y;
```

```
ok
def increment = add 1;
ok
increment 99;
100
```

Partial application is used in subsequent sections to construct update functions which, in effect, extend functions. For example, an array may be modelled as a function from addresses to the values held at those addresses. Initially, the array is empty and returns "fail" for any address:

```
ok
def array_0 addr = "fail";
```

When a value is assigned to an array address, a new function is constructed which returns that value if passed the corresponding address or calls the old function to find the value:

```
ok
def new_array old_array:address:value =
  lam addr. (if addr = address
             then value
             else old_array addr);
```

Now, for example, '33' may be assigned to address '3':

```
ok
def array_1 = new_array array_0:3:33;
```

and '55' may be assigned to address '5':

```
ok
def array_2 = new_array array_1:5:55;
ok
array_2 3;
33
ok
array_2 6;
"fail"
```

2.9 *Input and output*

Output is effected with the 'write' expression which returns the value of its argument and also sends it to the standard output. Characters and strings are output without quotes:

```
ok
def writeln l =
  let line = write l
  in
    let nl = write '\n'
    in ();
ok
writeln "festive greetings!";
festive greetings!
```

The 'readln' object returns the next line from the standard input as a string:

```
ok
readln;
testing testing 1 2 3
"testing testing 1 2 3"
```

Amusing effects result from its use in function calls in lazy evaluated lists!

Other facilities include character input, lazy file input and file output.

2.10 *Grammar rules*

Navel grammar rules are based on context free productions. The notation used is yet another variant of B.N.F. [13]. As in B.N.F., productions with the same left hand side name are combined into a single rule with a number of options. Unlike B.N.F., right hand sides of productions are not necessarily bound to particular left hand side non-terminals but are objects in their own right.

For Navel rules, the terminal symbols are strings and the non-terminal symbols are names associated with strings or rules.

A rule consists of one or more options separated by '|'s within '{' and '}'. At simplest, an option is a sequence of one or more strings and names. For example, the B.N.F. for binary numbers:

```
⟨binary⟩ ::= ⟨digit⟩ ⟨binary⟩ | ⟨digit⟩
⟨digit⟩ ::= 0 | 1
```

might be rendered as:

```
ok
def binary = {digit binary | digit};
ok
def digit = {"0" | "1"};
```

2.11 *Parsing with rules*

A rule is used like a function to parse a string to return a list containing the list representation of the parse tree, or the empty list if the parse fails, and the rest of the string. Parsing is left to right.

For a rule, each option is tried in turn until one succeeds. If an option begins with a string and the argument string starts with that string then the rest of the option is applied to the rest of the argument string and the resulting parse tree list starts with the string starting the option. Otherwise the option fails:

```
ok
{"1"} "101";
("1":):"01"
ok
{"1" "0"} "101";
("1":"0":):"1"
```

Similarly, if an option begins with a name then its associated object is found and used and a field consisting of the sub-tree tagged with the name forms the start of the parse tree list:

```
ok
{digit} "10";
([digit "1":]:):"0"
ok
{binary} "10";
([binary [digit "1":]:[binary [digit "0":]:]:]:):
ok
{binary} "0123";
([binary [digit "0":]:[binary [digit "1":]:]:]:):"23"
```

No attempt is made to automatically factor rules or to remove left recursion because, as is discussed in Section 4, rules may be abstracted over and may contain any expressions that return strings or rules. Thus, it is not possible in general to check statically a rule for left recursion or options with common initial symbol sequences.

With some rules the parser may fail to recognise valid strings because there is no backtracking within options. Option order may be important. If two options have a common initial symbol sequence and the shorter option precedes the longer then the shorter may lead to successful recognition of a sub-string:

```
ok
def binary = {digit | digit binary}:
ok
{binary} "111";
([binary [digit "1":]:]:):"11"
```

2.12 *Finding a rule that produced a tree*

The 'rule' expression is similar to a 'case' expression and is used to identify a rule (not necessarily unique) that produced a parse tree. For example, to interpret binary numbers:

```
ok
def mdigit d =
  rule d of
  {"0"}-> 0,
  {"1"} -> 1,
  "fail";
ok
def mbinary v b =
  rule b of
  {digit binary}-> mbinary 2*v + (mdigit b^ digit) b^ binary,
  {digit}-> 2*v + (mdigit b^ digit),
  "fail";
ok
def m string =
  let tree:rest = binary string
  in
    if tree = () | rest< >()
    then "syntax error"
    else mbinary 0 tree;
ok
m "101";
5
```

2.13 *Simplifying rules*

Rules may contain sub-rules. A successful sub-rule option is treated as if it were in-line.
The empty rule '{ }' matches anything and returns the empty list '( )'. This may be used to factor rules:

```
ok
def binary = {digit {binary | { }}};
```

The Navel rule 'number' matches a sequence of decimal digits and the rule 'word' matches a sequence of letters:

```
ok
word "praxis";
"praxis":
ok
{number} "2001";
([number "2001"]:):
```

The rule 'fail' always fails. This is used in subsequent sections to initialise rules which are to be extended.
Rules may be used for the generation of strings but this is not discussed here.

2.14 *Implementation*

The present Navel implementation is written in C and runs under UNIX*.

Navel objects are represented by linked lists of tagged two-field cells held in a heap. All name/object associations are made and accessed via a stack. Programs are compiled to parse trees which contain stack frame relative addresses for names. These are found by symbolic evaluation at compile time.

At run time, when a function is encountered the positions of objects for free variables are known from the compile time symbolic evaluation and the objects are picked up directly to make a closure. When a function is entered, the actual parameter and free variable objects are pushed onto the stack. When the object associated with a name is required it is accessed directly through the stack.

No text is held for programs. A pretty printer is used to reconstruct text from parse trees for function editing and saving global name/object associations as definitions in the hostess filing system.

Preliminary timing tests suggest that for worst case insertion sort and for Ackermann's function, Navel runs at more than twice the speed of the 1979 SASL [14], the same speed as interpreted FRANZ LISP [15] and half the speed of CProlog [16].

3. FROM DENOTATIONAL SEMANTICS TO A NAVEL INTERPRETER

3.1 *Introduction*

The following sections describe a denotational semantics based on that for TINY [2] and discuss the construction of a Navel interpreter from it.

TINY is an imperative language with assignment, integer arithmetic and I/O. Here, a subset of TINY is used without conditional and iterative commands to simplify the presentation.

For example, to output double the value of an input:

```
a: = read
b: = a + a
output b
```

3.2 *Denotational semantics*

3.2.1 *Semantic domains and abstract syntax.* The semantic domains for TINY are:

```
State = Memory*Input*Output
Memory = Ide->[Num + {unbound}]
Input = Num*
Output = Num*
Ide = identifiers
Num = numbers
```

A memory is a function mapping names onto values and returns 'unbound' for a name without an associated value.

Inputs and outputs are lists of numbers.

The abstract syntax is:

```
E ::= 0 | 1 | read | I | E1 + E2
```

where 'E' ranges over the domain 'Exp' of expressions, and

```
C ::= I: = E | output E | C1 C2
```

where 'C' ranges over the domain 'Com' of commands.

3.2.2 *Semantic notation and auxiliary definitions.* In the semantic notation, a list is denoted as a sequence of elements separated by ','s within '(' and ')'. Alternatively, '.' is used as the concatenation operator. 'hd' and 'tl' select the head and tail of lists respectively.

*UNIX is a trademark of Bell Laboratories.

The conditional expression—'a->b, c'—means that if 'a' is true then 'b' is returned; otherwise 'c' is returned. This may also be used to match a list of values with a list of variables and bind the variables, as, for example, in the second definition for function sequencing below.

The operator '*' is used for function sequencing. There are cases for sequencing a monadic function with monadic and dyadic functions:

```
a)        f: D1->[D2 + {error}]  g: D2->[D3 + {error}]

          f * g: D1->[D3 + {error}]
          f * g = lambda x.(f x = error)->error,g(f x)

b)        f: D1->[[D2 * D3] + {error}]  g: D2->D3->[D4 + {error}]

          f * g: D1->[D4 + {error}]
          f * g = lambda x.(f x = error)->error,(f x = (d2,d3))->g d2 d3
```

It also traps errors.

For function updating, 'f[rv/lv]' means that a new function is constructed which when called returns 'rv' for an argument with value 'lv'; otherwise 'f' is called with the argument:

```
f[rv/lv] l = l = lv->rv,f l
```

This is used to extend the memory function with a name and value from an assignment.

The following auxiliary function is used in semantic definitions:

```
result: Num->State ->[[Num * State] + {error}]

result = lambda v s.(v,s)
```

3.2.3 *Semantic functions.* The semantic function for expressions is:

```
E: Exp->State->[[Num * State] + {error}]

E[0] = result 0
E[1] = result 1
E[read] = lambda (m,i,o).null i->error, (hd i,(m,tl i,o))
E[I] = lambda (m,i,o).m I = unbound->error, (m I,(m,i,o))
E[E1 + E2] = E[E1]*lambda v1.E[E2]*lambda v2.result (v1 + v2)
```

Note that for an identifier 'I' in an expression, the memory function 'm' returns 'unbound' if the identifier has not been assigned to. Thus the context sensitive syntax requirement that names be assigned before use has been dealt with in the semantics. In Section 5, a syntactic specification for this using higher order grammar rules is discussed.

The semantic function for commands is:

```
C: Com->State ->[State + {error}]

C[I: = E] = E[E] * lambda v (m,i,o).(m[v/I),i,o)
C[output E] = E[E} * lambda v (m,i,o).(m,i,v.o)
C[C1 C2] = C[C1] * C[C2]
```

3.3 *Navel interpreter*

3.3.1 *Concrete syntax.* To construct a Navel interpreter from this definition, it is first necessary to contrive a concrete syntax for the language. Left recursion is removed through the introduction of auxiliary rules:

```
ok
def base = {"0" | "1" | "read" | word};
ok
def exp = {base "+" exp | base};
ok
def comm = {word ":=" exp | "output" exp};
ok
```

```
def comms = {comm comms | comm};
ok
{comms} "a: = 0  b: = a";
([comms
  [comm
    [word "a"]:": = ":[exp [base "0":]:]:
  ]:
  [comms
    [comm
      [word "b"]:": = ":[exp [base [word "a"]:]:]:
    ]:
   ]:
  ]:
):
```

3.3.2 *Semantic notation and auxiliary definitions in Navel.* Many aspects of the semantic notation have direct Navel equivalents but function sequencing must be modelled explicitly:

```
ok
def stara f g x = if (f x) = error
                  then error
                  else g (f x);
ok
def starb f g x = if (f x) = error
                  then error
                  else
                    let d2:d3 = f x
                    in g d2 d3;
```

Function updating is modelled by the functional:

```
ok
def u f rv lv =
  lam arg. (if arg = lv
            then rv
            else f arg);
```

'unbound' and 'error' are modelled as strings:

```
ok
def unbound = "unbound";
ok
def error = "error";
```

The auxiliary function 'result' translates to:

```
ok
def result v s = v:s;
```

3.3.3 *Semantic functions.* The semantic functions in Navel are:

```
ok
def mbase b =
  rule b of
  {"0"}->result 0,
  {"1"}->result 1,
  {"read"}->lam m:i:o.(if i = ()
                        then error
                        else (hd i):(m:(tl i):o)),
    {word}->lam m:i:o.(if (m b ^ word) = unbound
                       then error
                       else (m b ^ word):(m:i:o)),
    "error-base":b;
  ok
  def mexp e =
    rule e of
    {base}->mbase e ^ base,
    {base "+" exp}->starb (mbase e ^ base)
                          lam v1.(starb (mexp e ^ exp)
                                        lam v2.(result v1 + v2)),
    "error-exp":e
  ok
  def mcomm c =
    rule c of
    {word ":=" exp}->starb (mexp c ^ exp)
                           lam v m:i:o.(u m v c ^ word):i:o,
    {"output" exp}->starb (mexp c ^ exp) lam v m:i:o.m:i:(v:o),
    "error-comm":c;
  ok
  def mcomms c =
    rule c of
    {comm}->mcomm c ^ comm,
    {comm comms}->stara (mcomm c ^ comm) (mcomms c ^ comms),
    "error-comms":c;
```

3.3.4 *Interpreter*. To run the semantic functions as an interpreter, a function is constructed which takes the string for a TINY program with the input list, parses the program and calls the 'mcomms' semantic function with the tree, an initial memory which returns 'unbound' for all names, the input and an empty output list:

```
ok
def interpret text input =
  let tree:rest = comms text
  in
    if tree = () | rest<>()
    then "syntax error":rest
    else
      let s = mcomms tree (lam name.unbound):input:
      in
        if s = error
        then error
        else
          let m:i:o = s
          in o;
ok
interpret "a:= read output a + a" 1:;
2:
```

3.3.5 *Interactive testing*. For interactive testing, 'interpret' may be called repeatedly by a recursive function that prompts for and inputs the program text and input lists:

```
ok
def prompt_read prompt =
  let p = write prompt
  in readln;
```

```
ok
def run n =
  let text = prompt_read "program>"
  in
    let data = convert (prompt_read "data>")
    in
      let output = writeln (interpret text data)
      in run n;
ok
run ();
program> output read + read
data> 1 0
1:
program> ...
```

The function 'convert' should transform a string into a list of numbers. One way is to parse the string and strip out the fields from the tree:

```
ok
def convert string =
  let numb = {"0" | "1"}
  and numbs = {numb numbs | numb}
  and value n =
    if n = "0":
    then 0
    else 1
  and strip n =
    rule n of
    {numb}-> (value n^numb):,
    {numb numbs}-> (value n^numb):(strip n^numbs),
    "input error":n
  in strip (hd (numbs string));
```

Thus, rules may also be used to define and process the inputs for a programming language. Full checks that the inputs are correct are not included here.

3.4 *Denotational semantics and Navel*

A denotational semantics consists of three separate sections with distinct description languages. In Navel, there are no semantic domain specifications and syntax and semantics are described in a unitary language.

The domain specifications are used to describe the functionality of the semantic equations. This would correspond to specifying explicitly the types of formal parameters and results of functions in a strongly typed language and would enable checks for type consistency at compile time. Navel is weakly typed with run-time type checking. It is thus less rigorous than denotational semantics and has a relatively inefficient implementation.

In denotational semantics, abstract syntax mediates concrete syntax and semantic equations. In Navel, there is no necessity to introduce abstract syntax. In principle, its absence will make the interpreter less general: the semantic functions must be changed if the concrete syntax is changed whereas for a denotational semantics it would only be necessary to change the concrete/abstract syntax association specification. In practise, however, it is not clear that language design and development proceed cleanly with changes in one area isolated from other areas. The omission of abstract syntax may lead to extra semantic functions, as in this example, but reduces the number of different stages in a design.

There is one denotational semantic equation for each abstract syntax construct with sub-equations for each sub-construct. Sub-constructs and sub-equations are associated through implicit pattern matching on the sub-construct abstracted implicitly from a sub-tree. Sub-trees are then selected implicitly by mention of the associated sub-construct name.

Part of the motivation for Navel's development was to investigate the association of syntax and semantics in language definitions. This is highlighted through explicit operations on tree representations for rule identification and sub-tree selection. Here, Navel interpreters are more akin to operational semantics [17].

4. GENERALISING RULES

Rules have the same civil rights as other Navel objects. In particular they may be abstracted over which enables the specification of rule schemas. For example, consider the following rules for sequences of numbers and words:

```
ok
def numbers = {number {"," numbers |{ }}};
ok
def words = {word {":" words | { }}};
```

These have a common structure which may be expressed by:

```
ok
def sequence one separator many = {one {separator many |{ }}};
```

and they may now be defined by:

```
ok
def numbers = sequence number "," {numbers};
ok
def words = sequence word ":" {words};
```

When rules are formed in this way, the parse tree list fields will contain the names from the schema:

```
ok
numbers "11,22";
([one [number "11"]]:
 [separator ",":]:
 [many [numbers [one [number "22"]]:]:]:):
```

Unlike LISP[18] literals are not objects and so here there is no literal replacement. As with functions, the values of the arguments are used.

Navel allows rules to contain bracketed expressions returning rules or strings. These are then treated as if they were in-line. Bracketing a name results in its value being used without a field being constructed for the resulting sub-tree. This may be used to make rule schemas which give appropriate names to fields:

```
ok
def sequence one separator many = {(one) {(separator) (many) | { }}};
ok
def numbers = sequence number "," {numbers};
ok
numbers "11,22";
([number "11"]:
 ",":
 [numbers [number "22"]:]:):
```

Such abstraction over rules enables the equivalent of consistent substitution in two-level grammars [19]. Two-level grammars are based on the generation of rules from rule schemas through what is in effect the macro expansion of rule names. This contrasts with Navel where names are fixed and rules are generated from schemas through parameter substitution.

For example:

```
ok
def match n symb =
  if n = 1
  then {(symb)}
  else {(symb) (match n-1 symb)};
```

will generate a rule to recognise a specified length sequence for a given symbol. This may be used to recognise:

$a^n b^n c^n$

for any given 'n';

```
ok
def anbncn a b c n = {(match n a) (match n b) (match n c)};
ok
def xnynzn =
  let x = "x"
  and y = "y"
  and z = "z"
  in anbncn {x} {y} {z};
ok
xnynzn 2 "xxyyzz";
([x "x"]:[x "x"]:[y "y"]:[y "y"]:[z "z"]:[z "z"]:):
```

Recognising:

$a^n b^n c^n$

for an unknown 'n' is more tricky. It might appear that:

```
ok
def anbncnstar a b c n =
                {(anbncn a b c n) | (anbncnstar a b c n+1)};
```

would do the trick. This is, alas, only a partial solution. If the string being checked is not of the form:

$a^n b^n c^n$

then there is no value of 'n' for which the first rule option can succeed and the rule will recurse indefinitely.

Such context sensitive checking requires the passing of information about the parse from one part of a rule to another. Context free rules have no mechanism for information passing other than that an earlier rule part must have succeeded for a later rule part to be reached. This may be overcome within a functional framework by making the application of each rule part explicit and then generating and passing information between them through intermediate processing. In effect, information from the start of the parse is used to select or construct appropriate rules from the rest of the parse.

For example, to return to:

$a^n b^n c^n$

where 'n' is unknown until the sequence has been examined, processing the 'a's may be used to guide the construction of a new rule to recognise the corresponding number of 'b's and 'c's.

For each 'a', a new 'b' is added to the rule for 'b's and a new 'c' is added to the rule for 'c's. At the same time, the tree for the 'a's is accumulated. When all the 'a's have been found, a rule is built to recognise the same number of 'b's and 'c's. The trees for the 'a's and for the 'b's and 'c's are joined to form the final tree.

```
ok
def an a b c atree bn cn text =
  let t1:s1 = a text
  in
    if t1〈 〉()
    then an a b c (hd t1):atree {(b) (bn)} {(c) (cn)} s1
    else
      let t2:s2 = {(bn) (cn)} s1
      in (append atree t2):s2;
ok
def anbncn a b c = an a b c () {} {};
```

These approaches are used in the next two sections for parsing context sensitive aspects of programming languages.

5. CHECKING VARIABLE ASSIGNMENT BEFORE USE

5.1 *Assignment checking*

Considering the language TINY from section 4 with the additional requirement that an identifier in an expression must have appeared on the left had side of a preceding assignment. This may be specified in Navel in a manner similar to dynamic syntax [20, 21] by constructing rules from assignments for use in checking subsequent commands.

Initially, a base may only be a digit or a 'read':

```
ok
def base = {"0" | "1" | "read"};
ok
def exp = {base "+" exp | base};
ok
def comm = {word ":=" exp | "output" exp};
```

and there are no assigned variables:

```
ok
def assigned = fail;
```

The rule for assigned variables is extended by the function:

```
ok
def new id old = {(id) | (old)};
```

which when passed an identifier and the previous rule for assigned variables returns a rule which tries to recognise that identifier before using the previous assigned variable rule.

When an assignment has been recognised, the identifier is checked to see if it is recognised by the rule for assigned identifiers. If it is then it has been assigned already and the current rules for assigned identifiers and commands are returned. Otherwise, the identifier is added to the rule for assigned identifiers which is then used to build a new rule for recognising commands:

```
ok
def check id:assigned:comm =
  let t:r = assigned id
  in
    if t〈 〉()
    then assigned:comm
    else
      let newassigned = new id assigned
      in newassigned:(makecomm newassigned);
```

```
ok
def makecomm assigned =
  let base = {assigned | "0" | "1" | "read"}
  and exp = {base "+" exp | base}
  and comm = {word ":=" exp | "output" exp}
  in {comm};
```

5.2 *Command sequence and program checking*

Command sequences are recognized by a rule that picks up the first command. If it is the last command in the sequence then the tree for it is returned. Otherwise, if it is an assignment then new command and assigned identifier rules are constructed. The commands rule is called recursively to recognise the rest of the sequence and a final tree is constructed and returned:

```
ok
def comms assigned:comm text =
  let t1:r1 = comm text
  in
    if t1 = ()
    then ():text
    else
      if r1 = ()
      then ([comms t1]:):
      else
        let t2:r2 =
          rule t1 ^ comm of
          {word ":=" exp}-> comms (check (id t1):assigned:comm) r1,
          comms assigned:comm r1
        in
        if t2 = ()
        then ():text
        else ([comms (hd t1):t2]:):r2;
ok
def id c = c ^ comm ^ word;
```

The rule for a program is constructed from the rule for commands with the initial assigned identifier and command rules:

```
ok
def program = comms assigned:{comm};
ok
program "a:=0 b:=a";
([comms
  [comm
    [word "a"]:":=":[exp [base "0":]:]:
  ]:
  [comms
    [comm
      [word "b"]:":=":[exp [base [assigned "a":]:]:]:
    ]:
  ]:
 ]:
):
```

The interpreter can now be simplified with the removal of checks for errors from unassigned identifiers.

5.3 *Checking in syntax and semantics*

In the TINY semantics in Section 4, the update function:

```
def u f rv lv =
  lam arg.(if arg = lv
           then rv
           else f arg);
```

was used to extend the memory function. Here:

```
def new id old = {(id) | (old)};
```

is used in the same way to extend rules. 'lv' and 'f' in the update function correspond to 'id' and 'old' in the rule extender. The return of 'rv' for 'lv' in the update function corresponds to the return of a tree for the successful recognition of 'id'. There is no equivalent for 'arg' because rules have implicit formal parameters. Here it may be made explicit by:

```
def new id old =
  lam arg.({(id) | (old)} arg);
```

The use of the update and rule extender functions are further analogous. In TINY, assignments both introduce and update name/value associations For a name in an expression, the memory function is used to find the corresponding value. If the required name had not been introduced by a previous assignment then the memory function would return 'unbound'. Assignment before use is a syntactic requirement and is irrelevant for semantics [2] but the use of a memory function implicitly encompasses the checking of this requirement.

6. DECLARATION AND TYPE CHECKING

6.1 *Syntax for declared and typed variables*

The following rules describe partially the syntax of a language fragment with declarations. assignment and integer arithmetic:

```
ok
def program = {declarations statements};
ok
def declarations = {declaration {declarations | { }}};
ok
def declaration = {{"long" | "short"} word};
ok
def statements = {statement {statements | { }}};
ok
def statement = {word ": = " expression};
ok
def expression = {base {" + " base | { }}};
ok
def base = {word | number | "(" expression ")"};
```

It is required that no identifier may appear in more than one declaration and that an identifier must appear in a declaration before it appears in a statement. It is intended that the language should enable 'long' and 'short' precision arithmetic, that variables be typed according to the precision of the values they may hold and that 'long' values may not be coerced to 'short'. Thus if the identifier on the left of a statement appears in a declaration preceded by "short" then all identifiers in the expression on the right of the statement must also appear in declarations preceded by "short".

6.2 *Denotational semantics with declarations*

For a denotational semantic definition of an imperative language with declarations it is sometimes convenient to model name/value associations in two stages, mediated by addresses, through an environment function which records name/address associations and a store function which records address/value associations. When a declaration is encountered, a new name/address association is added to the environment. When a name is encountered in a command, the environment is used to find the associated address. If the name is in an expression then the address is used to find the corresponding value from the store. If the name is that of a variable which is being assigned to then a new address/value association is added to the store.

As with the memory function for TINY, environment use may encompass name checking. If a name in a command has not appeared in a preceding declaration then the environment will not have an address for it. Similarly, if a name in a declaration has appeared in a preceding declaration then the environment will already have an associated address for it. The environment function may also be extended to enable the recording and checking of type information for names.

6.3 *Declaration checking*

Name checking may be transferred to the syntax with the construction of appropriate rules from declarations. For the above language, separate rules will be used to check whether or not identifiers have been declared as "short" or "long". Initially, nothing has been declared and so these rules should not match anything:

```
ok
def init_short = fail;
ok
def init_long = fail;
```

As before, rules are extended with the function:

```
ok
def new old name = {(name) | (old)};
```

When a declaration is found, the rules are used to check the identifier for re-declaration and the appropriate rule is then extended with the identifier:

```
ok
def decls string:long:short =
  let tree:rest = {{"short" | "long"} word} string
  in
    if tree = ()
    then long:short:string
    else
      if hd(long tree ^ word)⟨ ⟩() | hd(short tree ^ word)⟨ ⟩()
      then long:short:string
      else
        rule tree of
        {"long" word}-⟩ decls rest:(new long tree ^ word):short,
        {"short" word}-⟩ decls rest:long:(new short tree ^ word),
        "never going to reach here...";
```

6.4 *Assignment and program checking*

Separate rules are required to recognise assignment to variables declared as 'long' and 'short'. They are built from a schema which defines local rules for recognising expressions:

```
ok
def makeassign nametype basetype =
  let base = {(basetype) | number | "(" expression ")"}
  and expression = {base {" + " base | { }}}
  in {(nametype) ": = " expression};
```

Programs are parsed by a function which uses 'decls' to parse the declarations and construct the rules for 'long's and 'short's. They are then used with the assignment schema to construct the assignment rules which are incorporated into local rules for recognising statements:

```
ok
def program string =
  let long:short:rest = decls string:init_long:init_short
  in
    let lassign = makeassign {long} {long | short}
    and sassign = makeassign {short} {short}
    and statement = {lassign | sassign}
    and statements = {statement {statements | { }}}
    in statements rest;
ok
program "short a long b short c c: = a + a b: = c + b";
([statement
  [sassign
    [short "c":]:
    ":=":
    [exp [base [short "a":]:]:"+":[base [short "a":]:]:]:
  ]:
]:
[statements
  [statement
    [lassign
      [long "b":]:
      ":=":
      [exp [base [short "c":]:]:"+":[base [long "b":]:]:]:
    ]:
  ]:
  ]:
):
```

To simplify the example there are no diagnostics and no tree is built for declarations: these could easily be added.

7. CONCLUSIONS

The approaches described here could be used with any language with general pattern recognition facilities. For example, string scanning in Icon [22] is ideal for context free parsing and may be combined naturally with list construction for tree production. Prolog [23] provides context free grammar rules as syntactic sugaring of the underlying pattern recognition aspects of unification. Navel's novelty lies in the integration of grammar rules as first class objects within a functional language.

Work continues on the use of higher order grammar rules in interpretive language implementations, for example for handling forward references and to associate abstract and concrete syntax. Another area of interest is the more general use of grammar rules for the direct construction of data structures from data specifications.

8. SUMMARY

This paper discusses Navel, a functional language which is intended for use as an interpreter-interpreter to simplify the construction of interactive tests intepreters from formal language definitions.

Denotational semantics provides a rigorous and versatile methodology for defining programming languages. However, current production compiler generators are not suitable for direct experimentation with such language definitions because their semantic definition facilities are based on imperative languages which do not correspond closely to denotational semantics semantic equations. The compiler generator SIS is designed for use with denotational semantic definitions but is based on the separation of lexical, syntactic and semantic formalisms which may restrict its flexibility.

Pure functional languages are very similar to the semantic equations of denotational semantics but lack facilities for syntactic specification. Navel is a functional language with fully integrated grammar rules, implemented in an interactive environment. Its use enables the unitary expression of the syntactic as well as the semantic aspects of a denotational language definition. Although there is some loss of rigour, interpreter construction is simplified greatly. Here, Navel is illustrated through examples and used to construct an interpreter from the denotational semantic definition of a simple assignment language.

Denotational semantics does not address the context sensitive aspects of programming language syntax. Navel provides a means to handle context sensitivity through functional abstraction over grammar rules to construct rule schemas, which are analogous to higher order functions. Here, this approach is presented through discussion of the classic '$a^nb^nc^n$' problem and is used to process assignment and declaration before used, and simple type matching.

Acknowledgements—This work was started in the Department of Computing Science, University of Glasgow and is continued in the Department of Computer Science, Heriot-Watt University. The author wishes to thank Bill Findlay (Glasgow), David Watt (Glasgow), Paul Chisholm (Heriot-Watt) and Stuart Anderson (Heriot-Watt) for much fruitful disputation, Zdrav Podolski (Glasgow) and Chris Miller (Heriot-Watt) for making UNIX intelligible and David Watt, Paul Chisholm, Stuart Anderson and Howard Williams (Heriot-Watt) for chewing through previous drafts of this paper.

REFERENCES

1. Stoy J *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, Mass. (1977).
2. Gordon M. *The Denotational Description of Programming Languages*. Springer, New York (1979).
3. Tennent R. Language design methods based on semantic principles. *Acta Informatica* **8**, 97–112 (1977).
4. Gries D. *Compiler Construction for Digital Computers*. Wiley, New York (1971).
5. Johnson S. Yacc: yet another compiler compiler No 32, Computing Science Technical Report, Bell Laboratories, Murray Hill, NJ 07974 (1975).
6. Hopcroft J. and Ullman J. *Formal Languages and their Relation to Automata*. Addison–Wesley, Reading, Mass. (1969).
7. Sethi R. Control flow aspects of semantic-directed compiling. *ACM Trans. Programming Lang. Systems* 5(4), 554–595 (1983).
8. Mosses P. *SIS—Semantics Implementation System Reference Manual and User Guide*. DAIMI MD-30, Computer Science Department, Aarhus University, Denmark (1979).
9. Turner D. *SASL Language Manual*. Department of Computational Science, University of St Andrews, St Andrews, Scotland (1975).
10. Michaelson G. *User Guide to NAVEL*. Department of Computer Science, Heriot-Watt University, Endinburgh, Scotland (1984)
11. Friedman D. and Wise D. CONS should not evaluate its arguments. In *Automata, Languages and Programming* (Edited by Milner R. and Michaelson S.), pp. 257–284. Endinburgh University Press, Edinburgh, Scotland (1976).
12. Burstall R., Collins J. and Popplestone R. *Programming in POP2*. Edinburgh University Press, Edinburgh, Scotland (1977).
13. Naur P. (Ed.), Revised report on the algorithmic language ALGOL 60. *Commun. ACM* 6(1), 1–17 (1963)
14. Turner D. *SASL Language Manual*. CS/79/3, Department of Computational Science, University of St Andrews, St Andrews, Scotland (1979).
15. Foderaro J. and Sklowar K. *The FRANZ LISP Manual*. University of California at Berkeley (1983).
16. Pereira F. *CProlog User's Manual Version 1.1* EdCAAD, Department of Architecture, University of Edinburgh, Edinburgh, Scotland (1982).
17. Wegner P. The Vienna definition language. *ACM Comput. Surv.* 4(1), 5–63 (1972).
18. McCarthy J. *The LISP 1.5 Programmers' Manual*. MIT Press, Cambridge, Mass. (1969).
19. Cleaveland J. and Uzgalis R. *Grammars for Programming Languages, 4*. Elsevier Computer Science Library (Programming Languages Series), Amsterdam (1977).
20. Hanford K. and Jones C. Dynamic syntax: A concept for the definition of the syntax of programming languages. TR 12.090, IBM Tech. Rep. (1971).
21. Ginsburg S. and Rounds E. Dynamic syntax specification using grammar forms. *IEEE Trans. Software Engng* SE-4(1), 44–55 (1978).
22. Griswold R. E. and Griswold M. T. *The Icon Programming Language*. Prentice-Hall, Englewood Cliffs, N.J. (1983).
23. Clocksin W. F. and Mellish C. S. *Programming in Prolog*. Springer, Berlin (1981).

About the Author—GREGORY JOHN MICHAELSON has a B A. in Computer Science from the University of Essex (1973) and an M.Sc. in Computational Science from the University of St Andrews (1982). He has taught Computer Studies at Napier College, Edinburgh, Computing Science at the University of Glasgow, and is at present a Lecturer in Computer Science at the Heriot-Watt University, Edinburgh. His current research is into the use of formal definitions in programming language implementations.