Interpreter prototypes from language definition style specifications

by GREG MICHAELSON

Abstract: Language definition techniques may be applied to the specification of a wide range of problems. Language implementation techniques may then be used to construct interpreter prototypes. The use of an interpreter-interpreter language for the implementation of interpreter prototypes circumvents some of the disadvantages of contemporary programming languages and software tools.

Keywords: prototyping, programming languages, interpreters.

A prototype[1] forms a bridge between a specification and an implementation. It enables a static specification to be animated to illuminate its dynamic implications. It also helps clarify how abstract constructs might be finally realized in the implementation without making any concrete commitments to implementation details.

A prototype should reflect the functionality of the specification but this may be compromized by differences between the specification and prototyping formalisms. Prototyping may require the construction of explicit representations for some or all of the specification constructs and it may be difficult to maintain the functional correspondence between them. In particular, in developing the prototype, changes may be made which are not reflected fully in specification changes through carelessness or lack of time. Thus, prototyping may cease to be specification directed and the prototype may drift from the specification. The continuity from specification to implementation through prototyping may be lost and if a solution is finally implemented then the specification and the prototype may constitute incompatible implementation standards.

It is also easy to underestimate the effort involved in prototyping, particularly if very different specification and prototyping formalisms are used. Once a prototype works it may be tempting to adopt it as an intermediate or production implementation without further development. Thus, inappropriate or inefficient prototype representations for specification constructs, which were chosen to ease prototyping or for experimentation, become ossified.

The problems may be circumvented if appropriate tools are available to simplify the construction of prototypes from specifications. If the tool corresponds closely to the specification formalism then prototyping is simplified because specification constructs have direct prototype representations.

This paper presents an approach to specification based on language definition formalisms. The use of the Not a Very Exciting Language (NAVEL) interpreter-interpreter language to implement interpreter prototypes for language definition style specifications is then discussed.

Interpreters and interpretation

Interpreters are usually associated with language implementation but may be applied to a wide range of problems. If language definition techniques are used for problem specification then an interpreter may be used for solution implementation.

A language is defined formally[2] by specifying the symbols, the smallest meaningful linguistic units, the syntax, the well-formed symbol sequences, and the semantics which are what the well-formed symbol sequences mean. Symbols are specified as lists of literals or as finite state rules over literals. Syntax is specified through context-free grammar rules over

Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, UK

 0950-5849/88/010023-09$3.00

symbols. The semantics associate a function with each syntax construct. It is often useful to distinguish concrete syntax, the syntax of representation, from abstract syntax, the syntax of structure, which is used to associate syntax and semantics with structurally irrelevant details discarded from the concrete syntax. In the examples here, abstract syntax will not be used as the concrete syntax is simple.

A language definition may be used to implement a language processor for programs, specific symbol sequences, through three conceptual stages:

- lexical analysis – recognizing symbols from a concrete representation of a program, often character sequences,
- syntax analysis – recognizing well-formed symbol sequences and constructing a structural representation,
- interpretation – carrying out semantic actions corresponding to the structural representation.

Compilation is a special case of interpretation where the semantic actions generate a symbol sequence in another language corresponding to the structural representation. A language definition may be used to compile into the semantic action language by treating each action as a text generation macro rather than an executable function.

General purpose approaches based on formal language definition techniques include the VDM specification language[3]. The use of a language definition formalism directly as a programming language[4] has also been proposed.

An example of defining data as a language

Consider finding the sum of a list of numbers. Here, the data might consist of a sequence of numbers separated by commas. The basic symbols are numbers and commas. Numbers are made up of digits. Thus, the lexicon might be specified in the first instance by simply listing the characters that will be used:

0 1 2 3 4 5 6 7 8 9 ,

A number symbol consists of a sequence of one or more digits and might be specified as:

digit = **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
number = *digit* | *number digit*

Numbers and digits have been defined by rules. On the left is the rule name in *italics*. On the right is a sequence of options separated by '|'s. Each option is a basic character, or the name of a rule which has a basic character option or a basic character followed by a rule name. This form of rule is known as a 'regular expression'.

A sequence of characters is checked for lexical validity and to identify symbols by trying to match rule options. Successful matches are replaced with the corresponding rules name. For example, to check the validity of a character sequence which is supposed to be a number:

987 =>
digit **87** =>
number **87** =>
number digit **7** =>
number **7** =>
number digit =>
number

The lexicon is used as the basis of the concrete syntax which describes well-formed symbol sequences. For example, a number sequence consists of a single number or a number followed by a comma followed by more numbers which are in turn a sequence:

sequence = *number* | *number* , *sequence*

Here again a rule has been used to define the structure of a sequence. For concrete syntax, a more powerful form of rule is used where options may consist of arbitrary sequences of rule names and symbols. These are known as 'context free' rules. The concrete syntax rules are used to check that lexically valid symbol sequences are well-formed and to identify their structures. For example, assuming that the lexical check has validated:

42 , 31 , 20

as:

number , *number* , *number*

then the concrete syntax rules might be applied as:

number , *number* , *number* =>
number , *number* , *sequence* =>
number , *sequence* =>
sequence

The meaning of well-formed symbol sequences is defined using a pure functional notation. This lacks any concept of time ordering in evaluation and gives specifications substantial implementation independence.

At simplest, function definitions have a left-hand side which names the function and formal parameters, and a right-hand side consisting of an expression which is to be evaluated. For example, the squaring function is:

```
square x = x * x
```

and the sum of squares function is:

```
sum_squares x y = square x + square y
```

A function may have a number of cases corresponding to different formal parameter values. For example, the 'power' function is:

```
power x 0 = 1
power x n = x * power x (n - 1)
```

Here there are cases for a zero exponent and a non-zero exponent. These cases are equivalent to the use of an explicit conditional expression:

```
power x n = if n = 0 then 1 else x * power x (n - 1)
```

For example:

```
power 2 3 =>
2 * power 2 2 =>
2 * 2 * power 2 1 =>
2 * 2 * 2 * power 2 0 =>
2 * 2 * 2 * 1 =>
8
```

Note that the functions are not typed explicitly. While this simplifies presentation and implementation it also decreases security and rigour.

This style of definition is extended to allow pattern matching on syntactic constructs and the selection of subconstructs. Syntactic construct formal parameters are bracketed with [and]. Subconstructs may then be referred to in the function expression. When the function is applied to a symbol sequence, if the symbol corresponds to the construct then the references to subconstructs will select the appropriate subsequences.

In general, the structure of the semantic functions corresponds to the structure of the syntax rules. For a rule consisting of a number of options, the function will have a number of cases with one for each option. For a recursive rule, the semantic function will be recursive. Functions may be required for individual meaningful symbols. If a symbol is defined by a lexical rule then its lexical structure may be relevant.

Here, for example, the values of individual numbers will be needed to process a comma separated number sequence. First of all, the values of individual digits are specified as a sequence of cases:

```
value [0] = 0
value [1] = 1
etc
value [9] = 9
```

So far, the definition has one case for each *digit*. When the function is applied to an actual parameter consisting of a single digit symbol, each case is tried in turn until there is a match and then the corresponding value is returned. A sequence of digits is defined recursively as a *number* followed by a digit. Thus, its value is found by multiplying the value of the number by 10 and adding in the value of the digit.

```
value [number digit] = 10 * value [number] + value [digit]
```

Here, if a symbol sequence matches the construct: *number digit* then the subsequencies corresponding to the number and digit are selected for further evaluation. For example, the value of:

```
987
is:
value [987] =>
10 * value [98] + value [7] =>
10 * (10 * value [9] + value [8]) + 7 =>
10 * (10 * 9 + 8) + 7 =>
10 * 98 + 7 =>
987
```

Now, for a whole sequence, if there is only one number in the sequence then the total is that number's value:

```
sum [number] = value [number]
```

Otherwise, the value of the first number is added to the sum of the rest of the sequence:

```
sum [number , sequence] = value [number] +
                          sum [sequence]
```

For example, the meaning of:

```
9,8,7
```

is:

```
sum [9,8,7] =>
value [9] + sum [8,7] =>
9 + value [8] + sum [7] =>
9 + 8 + value [7] =>
9 + 8 + 7 =>
24
```

Program and data

It is usual to distinguish an active program from the passive data it processes. Thus, a sequence of data items has no inherent meaning and any sequence may have different interpretations within the same program. For example, the comma separated number sequence above might not only mean:

find the sum

but also:

count the non-zero numbers

with a different semantic function:

```
count [number] = if value [number] <> 0 then 1 else 0
count [number , sequence] = if value [number] <> 0 then 1
 + count [sequence] else count [sequence]
```

For example, to count the non-zero numbers in:

3 , 0 , 6 , 9

involves:

count **[3,0,6,9]** =>
1 + count **[0,6,9]** =>
1 + count **[6,9]** =>
1 + 1 + count **[9]** =>
1 + 1 + 1 =>
3

This ability to give multiple meanings to the same syntactic constructs suggests that the data itself contains no indication as to its meaning. An alternative view is that a data sequence processed in different ways corresponds to several different languages within a program. For example, in the comma separated number sequence, the comma is an 'add' or a 'test and count' operator in different languages which, by coincidence, have the same syntax. To illustrate this further, consider a fast-food takeaway which offers a variety of deep-fried food. A manual system might be based on a menu consisting of a list of food items and prices, for example:

fish 85,
spare rib 50,
frankfurter 55,
pizza 42,
burger 48,
kebab 45,
french fries 30

and a cash register which totals prices entered on numeric keys to produce a bill. When a customer orders food, for example, *Two pizzas, one burger and three french fries please,* the operatives use the menu (or their memories of it) to key prices corresponding to food items into the cash register to find the total bill. An automated cash register might hold the menu in its memory. The operatives would then push keys corresponding to food items and the register would total the corresponding prices to print out the final amount. A food item might be preceded by a number for multiple purchases of the same item. The sequence of food item and numeric keys corresponding to a purchase is a simple language with statements like:

2 pizza burger 3 french fries total

The lexicon consists of symbols for cash register keys:

fish spare rib frankfurter pizza burger kebab french fries *etc* **total**

along with the digits and rules for numbers.

For the concrete syntax, a purchase is a sequence of names, each of which may be preceded by a number, ending with the symbol **total:**

purchase = **total** | *name purchase* | *number name purchase*
name =
fish | **spare rib** | **frankfurter** | **pizza** | **burger** | **kebab** | **french fries** | *etc*

For example:

pizza 2 french fries total

after lexical validation as:

pizza *number* **french fries total**

is checked as:

pizza *number* **french fries total** =>
pizza *number* **french fries** *purchase* =>
pizza *number name purchase* =>
pizza *purchase* =>
name purchase =>
purchase

The meaning of a purchase is its total cost. If there are no items then the final cost is 0:
mpurchase **[total]** menu = 0. For a purchase starting with a named item, the item's price is found from the menu and added to the rest of the purchase:

mpurchase [*name purchase*] menu = lookup [*name*] menu
+ mpurchase [*purchase*] menu

If the named item is preceded by a number then its price is multiplied by the value of the number:

mpurchase [*number name purchase*] menu =
value [*number*] * lookup [*name*] menu +
mpurchase [*purchase*] menu

The look up function 'lookup' will be discussed below. For example, the cost of **pizza 2 french fries total** is:

mpurchase **[pizza 2 french fries total]** *menu* =>
lookup **[pizza]** *menu* + mpurchase **[2 french fries total]** =>
42 + value **[2]** * lookup **[french fries]** *menu* +
mpurchase **[total]** *menu* =>
42 + 2 * 30 + mpurchase **[total]** *menu* =>
42 + 2 * 30 + 0 =>
102

with a menu *menu* corresponding to the example above. Here it appears that the sequence of purchases is the program with the menu as data. The menu does not change and the way it is processed depends on the structure and hence the semantics of the purchases. However, this approach can also be used to specify the menu look up function. A menu is another language consisting of a sequence of comma separated item name and price pairs. The lexicon is the food name symbols, comma: , digits and number rules as above.

For the concrete syntax, a menu is either a single name/price pair or a comma separated sequence of pairs:

menu = name number | name number , menu

The meaning of a menu involves finding prices corresponding to food items. If the menu is a single pair then for a given name, if it matches that in the pair the price is the corresponding value:

lookup [*name*] [*name number*] = value [*number*]

Otherwise, the name is not in the menu so the price is 0:

lookup [*name*] [*name1 number 1*] = 0

If the menu is a sequence of pairs for a given name, if it matches that for the first pair then the price is the corresponding value:

lookup [*name*] [*name number , menu*] = value [*number*]

Otherwise, the rest of the menu is searched:

lookup [*name*] [*name1 number 1 , menu*] = lookup [*name*] [*menu*]

For example, given the menu:

fish 85, spare rib 50, frankfurter 55

the cost of a **frankfurter** is:

lookup **[frankfurter]** [**fish 85, spare rib 50, frankfurter 55**]
=>
lookup **[frankfurter]** [**spare rib 50, frankfurter 55**] =>
lookup **[frankfurter]** [**frankfurter 55**] =>
value [**55**] =>
55

Now the data/program distinction is lost. How a name is processed depends on the structure and hence the semantics of the menu. For the interpreter 'lookup', the menu is the program and the food name is the data. This distinction is irrelevant; both program and data can be specified using langauge definition techniques. What is important is the interplay of interpreted symbol sequences in the specification as a whole.

Implementing interpreters from language definitions

Most implementations are written in programming languages. A standard structured imperative language like C[5] has little direct correspondence with the specification formalism so implementation requires the explicit construction of lexical, syntactic and semantic processors through translation and explication of the formal specification. The extended imperative language ICON[6] provides structured pattern recognition which simplifies the construction of the lexical and syntactic stages but which still does not correspond closely to the specification formalism. The language BCL[7] had a control structure which was similar to simplified ICON pattern matching with semantic actions nested within structure recognition. BCL was used to implement a variety of systems[8] but is no longer extant. Proposals have also been made to extend imperative languages like ALGOL60[9] and ALGOL68[10] with parsing and symbol sequence structure representation.

Functional languages like STANDARD ML[11] and MIRANDA[12] correspond closely to semantic formalisms. They may be used as the basis of language definition oriented prototypes but have no direct equivalents to the syntactic formalism. The logic programming language PROLOG[13] provides context-free grammar rules as syntactic sugaring for underlying pattern recognition. PROLOG has been used to prototype VDM specifications[14].

Software tools designed for language implementation may also be used to construct interpreters from specifications. The most common are compiler construction tools like YACC[15] which combine a lexical or syntactic notation with actions in a standard imperative language. These again require translation from the specification semantics unless a semantic action language closer to the formal semantic notation is used[16]. Systems like SIS[17] and PSG[18] are based directly on formal language definitions. They are large and relatively slow. The strict adherence to formality maintains rigour but may decrease flexibility as a number of independent stages in different notations are required.

The OBJ[19] algebraic specification technique uses a functional notation for equations in specifications. These correspond to the semantic formalism in language definitions. OBJ specifications may be executed as prototypes on OBJ implementations but there is no direct equivalent to the syntactic formalism in language definitions.

Prototypes with NAVEL interpreters

NAVEL[20] is a weekly-typed applicative order functional language with integrated grammar rules. It was developed as an experimental interpreter-interpreter language to investigate syntax/semantics linkage in language implementations. NAVEL is implemented in C within an interactive environment which provides elementary imperative I/O and access to the hostess system files and editor (currently Unix).

NAVEL is less rigorous than formal semantic-based systems but it is also simpler as there are no distinct

notations for the lexical, syntactic and semantic stages of a definition. For example, symbols are not listed separately but are implicit in the grammar rules for the concrete syntax. Concrete syntax and semantics are associated directly through concrete parse trees without abstract syntax.

Consider once more the fast food take away. The concrete syntax for purchases may be written as NAVEL grammar rules:

```
ok
def name = { "fish" | "spare rib" | "frankfurter" | "pizza" |
             "burger" | "kebab" | "french fries" };
ok
def purchase = { "total" | name purchase | number name
                 purchase };
```

These rules are used to parse strings to construct list representations of parse trees. The rule for 'name' contains a sequence of optional symbols represented as strings. If 'name' is applied like a function to a string argument starting with one of these options the argument will be split at the match point and a list with the option in the head and the rest of the argument in the tail is returned:

```
ok
name "kebab 2 french fries";
("kebab":):" 2 french fries"
```

The rule for 'purchase' has options which use 'name' and the predefined 'number' rule which matches digit sequences. When 'purchase' is successfully applied to a string then the substrings matched by the named rules within an option are tagged with the rule's names:

```
ok
purchase "pizza total";
([name "pizza":]:"total":):()
```

The rule for 'purchase' also uses itself to recognize item sequences:

```
ok
purchase "fish 2 frankfurter 3 french fries total";
([name "fish":]:
 [purchase [number "2"]:
           [name "frankfurter":]:
           [purchase [number "3"]:
                     [name "french fries":]:
                     [purchase "total":]:]:]:):()
```

Subtrees are extracted from parse trees using the selector ^ and the rule name tags. For example:

```
ok
def p = hd (purchase "fish 2 frankfurter 3 french fries
       total");
```

sets 'p' to the tree from the previous example. The first 'name' is selected by:

```
ok
p^name
"fish":
```

The rest of the purchase is selected by:

```
ok
p^purchase;
[purchase [number "2"]:
          [name "frankfurter":]:
          [purchase [number "3"]:
                    [name "french fries":]:
                    [purchase "total":]:]:]:
```

The second 'name' is selected by:

```
ok
p^purchase^name;
"frankfurter":
```

and so on. The semantic functions match the parse tree list representations to identify rules which might have built the trees:

```
ok
def mpurchase purchase menu =
 rule purchase of
 {"total"} -> 0,
 {name purchase} -> (lookup purchase^name menu) +
                    (mpurchase purchase^purchase
                    menu)'
 {number name purchase} -> (value purchase^number) *
                           (lookup purchase^name
                           menu) +
                           (mpurchase
                           purchase^purchase menu),
 ();
```

There is no semantic rule for a number corresponding directly to the function 'value' in section three because the system rule 'number' is used to match numbers and produce digit strings. Instead, an analogous function is used to convert a digit string into its value:

```
ok
def value numb =
 let convert v n =
  if n=""
  then v
  else convert 10*v+(hd n)-'0' (tl n)
 in convert 0 numb;
```

The value of each digit is found by substracting the code for '0' from its code. For example:

```
value "987" =>
convert 0 "987" =>
```

```
convert 10*0+'9'-'0' "87" =>
convert 9 "87" =>
convert 10*9+'8'-'0' "7" =>
convert 98 "7" =>
convert 10*98+'7'-'0' "" =>
convert 987 "" =>
987
```

The menu is implemented in the same way as a purchase:

```
ok
def menu = {name number "," menu | name number};
ok
def lookup name menu =
 rule menu of
 {name number "," menu} -> if name=menu^name
                           then value menu^number
                           else lookup name
                           menu^menu
 {name number} -> if name=menu^name
                  then value menu^number
                  else 0,
 ();
```

These functions are sown together to build an interpreter. This parses the menu and the purchase before evaluating the purchase:

```
ok
def REGISTER MENU PURCHASE =
 let mtree:rest = menu MENU
 in
  if mtree = () | rest <> ()
  then "mistake in menu":rest
  else
   let ptree:rest = purchase PURCHASE
   in
    if ptree = () | rest <> ()
    then "mistake in purchase":rest
    else mpurchase ptree mtree;
```

A cash register with a specific menu is built by partial application:

```
ok
def MENU = "fish 85,
            spare rib 50,
            frankfurter 55,
            pizza 42,
            burger 48,
            kebab 45,
            french fries 30";
ok
def TAKEAWAY = REGISTER MENU;
```

This may now be used to total purchases:

```
ok
TAKEAWAY "fish 2 french fries total";
145
ok
TAKEAWAY "3 kebab 3 french-frys total";
"mistake in purchase":"3 kebab 3 french-frys total"
```

Interactive prototype

The model above processes a whole purchase as a complete symbol sequence. A more realistic model would prompt for and input the individual subpurchases one by one, checking each and printing out the corresponding subtotal. The NAVEL program may be modified to use imperative I/O for interactive prototype testing.

First of all, each item must be input and checked:

```
ok
def get_item prompt =
 let p = write prompt
 in
  let itree:rest = {name | number name | "total"} readln
  in
   if itree = () | rest <> ()
   then get_item prompt
   else itree;
```

Here the prompt is output by the system function 'write', a subpurchase is input as a one line string by the system function 'readln' and parsed to check that it is valid. If it is not then another purchase is requested recursively. Otherwise the tree is returned. Next, the running total must be output and reset or the subpurchase value must be found from the menu, output and included in the running total:

```
ok
def process_item prompt total menu =
 let itree = get_item prompt
 in
  rule itree of
  {"total"} -> let t = writeln total
               in process_item prompt 0 menu,
  {name} -> let sub_t = writeln (lookup itree^name
                                 menu)
            in process_item prompt total+sub_t menu,
  {number name} -> let sub_t =
                   writeln ((value itree^number)*
                            (lookup itree^name menu))
                   in process_item prompt total+sub_t
                   menu,
  ();
```

Here, the system function 'writeln' is used to send its argument's value to the screen and to return that value for possible subsequent use. Finally, the whole menu must be checked before input commences:

```
ok
def REGISTER MENU =
 let mtree:rest = menu MENU
 in
  if mtree = () | rest <> ()
  then "mistake in menu":rest
  else process_item ">" 0 mtree;
```

For example:

```
ok
REGISTER MENU;
> 2 french fries
60
> 3 fish
255
> total
315
> . . .
```

This interactive prototype simplifies testing but is less close to the initial specification. Language-style definitions with pure functional notations are not well-suited to specifying interactive systems as they lack any concept of time. The effective use of imperative I/O in a functional framework depends on knowledge of the system's function evaluation strategy. Thus, the evaluation independence of the original functional specification is lost.

Conclusions

NAVEL is less rigorous than the language definition formalism and translation is required from the formalism to NAVEL. An implementation of the specification language would avoid these drawbacks. This would require an interface which was able to manipulate and interpret different typefaces to avoid the complications and inflexibility inherent in the representation of different notations in a unitary typeface. However, NAVEL's integration of syntax rules in a functional framework eases the construction of interpreter-based prototypes from language definition style specifications.

Acknowledgements

This work was started in the Department of Computing Science, University of Glasgow, UK and continues in the Department of Computer Science, Heriot-Watt University, UK. The author wishes to thank Paul Chisholm for applying the red pen to previous drafts of this paper.

References

1 **Ince, D C and Hekmatpour, S** 'Software prototyping – progress and prospects' *Inf. & Software Technol.* Vol 29 No 1 (January/February 1987) pp 8–14

2 **Schmidt, D A** *Denotational Semantics: a Methodology for Language Development* Allyn & Bacon, Inc, USA (1986)

3 **Jones, C B** *Systematic Software Development Using VDM* Prentice-Hall (1986)

4 **Schmidt, D A** 'Denotational semantics as a programming language' CSR-100-82, Dept of Computer Science, University of Edinburgh, UK (January 1982)

5 **Kernighan, B W and Ritchie, D M** *The C Programming Language* Prentice-Hall (1978)

6 **Griswold, R E and Griswold, M T** *The ICON Programming Language* Prentice-Hall (1983)

7 **Hendry, D F and Mohan, B** 'BCL1 Manual' ICSP 110, Institute of Computer Science, University of London, UK (June 1969)

8 **Housden, R J W** The definition and implementation of LSIX in BCL' *Compter J.* Vol 12 No 1 (February 1969) pp 15–23

9 **Maurer, H and Stucky, W** 'Ein Vorschlag fuer die Verwendung syntaxorientierter Methoden in hoeheren Programmiersprachen' *Angewandte Informatik* Vol 5 (1976) pp 189–195

10 **Linnemann, V** 'Context-free grammars and derivation trees as programming tools' CSRG-117, Computer Systems Research Group, University of Toronto, Canada (September 1980)

11 **Harper, R, MacQueen, D and Milner, R** 'STANDARD ML' ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK (March 1986)

12 **Turner, D** 'MIRANDA: a non-strict functional language with polymorphic types' *Functional Programming Languages and Computer Architecture* (Lecture Notes in Computer Science) Vol 201 Springer-Verlag, New York, USA (1985)

13 **Clocksin, W F and Mellish, C S** *Programming in PROLOG* Springer-Verlag (1981)

14 **Cottam, I** discussed in 'Workshop on software tools for formal methods' *FACS FACTS* Vol 8 No 2 (February 1986)

15 **Johnson, S** 'YACC: Yet Another Compiler Compiler' No 32, Computing Science Technical Report, Bell Laboratories, Murray Hill, NJ 07974, USA (1975)

16 **Sethi, R** 'Control flow aspects of semantic-directed compiling' *ACM Trans. Programming Languages and Syst.* Vol 5 No 4 (1983) pp 554–595

17 **Mosses, P** 'SIS – Semantics Implementation System: reference manual and user-guide' MAIMI MD-30, Computer Science Dept, Aarhus University, Denmark (August 1979)

18 **Bahlke, R and Snelting, G** 'The PSG – Programming System Generator' *ACM SIGPLAN Notices* Vol 20 No 7 (July 1985) pp 28–33

19 **Duce, D A and Fielding, E V C** 'Formal specification – a comparison of two techniques' *Computer J.* Vol 30 No 4 (August 1987) pp 316–327

20 **Michaelson, G** 'Interpreters from functions and grammers' *Computer Languages* Vol 11 No 2 (June 1986) pp 85–104 □