An Exercise in Denotational Semantics

Ken Slonneger
The University of Iowa
Iowa City, Iowa 52242
slonnegr@herky.cs.uiowa.edu

ABSTRACT

This paper describes an exercise used in a first-year graduate course, Programming Languages Foundations, which deals with formal methods of specifying the semantics of a programming language. By having the students translate semantic equations directly into Prolog clauses thereby constructing an interpreter, this exercise makes denotational semantics more tangible and practical. After a brief description of the course, the two parts of the exercise are presented and illustrated by an example.

INTRODUCTION

Although some undergraduates take Programming Languages Foundations, it is primarily intended for graduate students. The textbook for the course is **Formal Specification of Programming Languages: A Panoramic Primer** [Pagan81], and the main topics covered in the course are:

a) Syntactic Issues
b) Attribute Grammars
c) Operational Semantics
d) Denotational Semantics
e) Axiomatic Semantics
f) Algebraic Specifications

For many students, this is their first encounter with the formal methods of defining programming language semantics. Specifically, the complexity and succinctness of denotational specifications cause the most trouble to students in this course. They tend to be overwhelmed by the formalism and notational conciseness of denotational semantics, viewing it as a purely theoretical drill.

The purpose of the exercise described here is to study denotational semantics in a more practical and experimental framework. Other authors have suggested this sort of project, namely translating a denotational definition into an interpreter for the language [Pagan79], [Allison83]. These examples have been carried out in imperative languages such as Pascal. The interpreters include a scanner and a recursive descent parser, which create a pointer structure representing the source program and then use function subprograms to implement the denotational definitions.

Denotational interpreters written in an imperative language are more difficult to write, read, and modify than the example to be described in this exercise. Using Prolog for this purpose puts the programming at a higher level so that time is not wasted in debugging the code that builds and manipulates the pointer structures for the abstract syntax tree and in fitting the denotational functions into the limited configuration of Pascal functions.

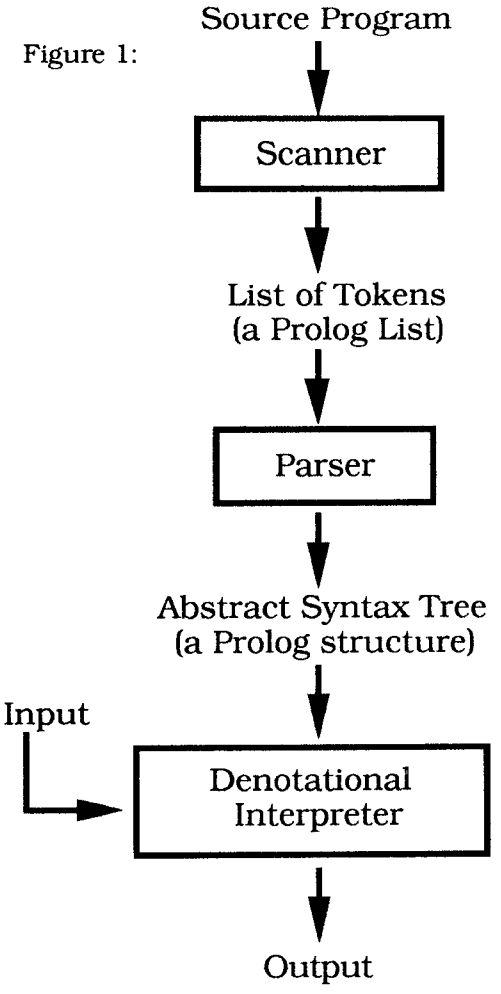
The goal of the exercise is to have the students produce a working interactive language processor based on a denotational definition of a programming language with a minimum number of distractions generated by the details of the implementation language.

The structure of the resulting interpreter program is shown in Figure 1.

PART ONE: Scanner and Parser

The first part of the exercise is to write a scanner and a parser for the Pam programming language presented in the Pagan text. Pam is a toy algorithmic language with integer arithmetic using a single global scope. Figure 2 contains a BNF specification of its syntax.

Figure 1:



After a brief introduction (or review) of Prolog, students are given a short problem to solve in Prolog, for example, defining a set of string processing predicates. This is sufficient preparation for the first part of the exercise, producing a scanner and a parser for Pam.

The program for reading English sentences found in Clocksin and Mellish [Clocksin84] is used as a model for the scanner. The scanner for Pam reads a text file containing a source program and produces a Prolog list of tokens. The scanner employs the native data types of Prolog to represent the tokens of Pam, which have the following forms:

1. Variables and reserved words are represented as Prolog literal atoms.

<program> ::= <series>

<series> ::= <statement> | <series> ; <statement>

<statement> ::= <variable> := <expr> |
read <variable list> | **write** <variable list> |
if <comparison> **then** <series> **fi** |
if <comparison> **then** <series> **else** <series> **fi** |
while <comparison> **do** <series> **end** |
to <expr> **do** <series> **end**

<variable list> ::= <variable> |
<variable list> , <variable>

<comparison> ::= <expr> <relation> <expr>

<expr> ::= <term> | <expr> <weak op> <term>

<term> ::= <element> | <term> <strong op> <element>

<element> ::= <constant> | <variable> | (<expr>)

<relation> ::= **<=** | **<** | **=** | **>** | **>=** | **<>**

<weak op> ::= **+** | **-** <strong op> ::= ***** | **/**

Figure 2: BNF for Pam

2. Numerals (nonnegative integers) are Prolog integers.

3. Single character (+, <, etc.) and double character (:=, <=, etc.) tokens are translated into Prolog literal atoms.

4. The end of the file is signaled by a Prolog atom "eof".

Pam has ten reserved words (**read**, **while**, etc.), which are constructed first as variables and then recognized as reserved words. Variables that are not reserved words (negation by failure) are passed on as tokens of Pam with the form "var(sum)". Numeral tokens are forwarded in the form "num(123)".

A sample Pam program is shown in Figure 3, and the resulting token list in Figure 4.

```
read x;
p := 1;
while p < 1000 do
    p := x*p;
    write x,p
end
```

Figure 3: Sample Pam Program

```
[ read, var(x), semicolon,
 var(p), assign, num(1), semicolon,
 while, var(p), less, num(1000), do,
 var(p), assign,
 var(x), times, var(p), semicolon,
 write, var(x), comma, var(p), end, eof ]
```

Figure 4: Token List for Sample Program

Students develop a parser for Pam as a logic grammar obtained by converting the BNF definition into grammar clauses with as little modification as necessary. The principle change is to remove the left recursion in the definitions of <series>, <variable list>, <expr>, and <term>. The standard approach is to write a BNF rule of the form

<expr> ::= <term> | <expr> <weak op> <term>

as a set of grammar rules of the form

```
expr(E) --> term(T), remexpr(T,E).

remexpr(T,E) -->  weakop(Op), term(T1),
                  remexpr(exp(Op,T,T1),E).
remexpr(E,E) -->  [].
```

As the tokens are parsed, an abstract syntax tree is constructed by means of the parameters to the grammar rules. For example, an expression such as

b - b / a * a

will produce the syntax tree

```
exp(minus,
    var(b),
    exp(times, exp(divides,var(b),var(a)),
               var(a)))
```

Since statement sequencing is associative, a series of statements in Pam can be converted into a list of statements. Also a <variable list> can be represented as a Prolog list of variables. The parse tree for the program in Figure 2 is shown in Figure 5.

```
[ read([var(x)]),
 assign(p, num(1)),
 while(exp(less, var(p), num(1000)),
       [assign(p, exp(times, var(x), var(p))),
        write([var(x), var(p)]) ]) ]
```

Figure 5: Abstract Syntax Tree for Sample Program

Because of the close correspondence between the BNF specification of the syntax of a programming language and its logic grammar, a parser is easily constructed by the students. They can see the usefulness and power of a BNF definition from this simple translation into a logic grammar.

PART TWO: The Denotational Definition

The second part of the exercise is to take the abstract syntax tree produced in part one and apply to it Prolog translations of the functions in a denotational definition of Pam. Such a definition consists of semantic functions that map the syntactic constructs of Pam into semantic domains. For the specification of these functions, the syntax of Pam is expressed as abstract productions rules relating the syntactic domains of Pam.

These syntactic domains are shown in Figure 6 and the abstract production rules are listed in Figure 7. Note that the recursion in the definition of a variable list has been altered to right recursion, which agrees more closely with the representation of variable lists as Prolog lists.

In the abstract syntax, comparisons have been included in the syntactic domain of expressions, assuming that only syntactically correct programs will have their denotational semantics elaborated.

Ψ	: Prog	programs
Ξ	: Var	variables
Λ	: Vars	lists of variables
Σ	: Stmt	statements
E	: Exp	expressions
Θ	: Opr	binary operations
N	: Num	integer constants

Figure 6: Syntactic Domains

$\Psi ::= \Sigma$

$\Sigma ::= \Sigma_1 ; \Sigma_2 \mid \Xi := E \mid$ **read** $\Lambda \mid$ **write** $\Lambda \mid$
if E **then** Σ **fi** | **if** E **then** Σ_1 **else** Σ_2 **fi** |
to E **do** Σ **end** | **while** E **do** Σ **end**

$\Lambda ::= \Xi \mid \Xi, \Lambda$

$E ::= \Xi \mid N \mid E_1 \Theta E_2$

$\Theta ::= + \mid - \mid * \mid / \mid = \mid <= \mid < \mid > \mid >= \mid <>$

Figure 7: Abstract Production Rules

The semantic domains used to define the meaning of a Pam program are shown in Figure 8. The semantics of Pam can be specified by three functions that map its syntactic structures into these domains (Figure 9).

The semantic equations which define the functions are listed in Figure 10.

τ	: T = {true,false}	
ν	: N = {...,-2,-1,0,1,2,...}	
ε	: Ev = Intg + Bool	Expressible values
δ	: Sv = Intg	Storable values
σ	: State = Var $\rightarrow$ Sv	

Figure 8: Semantic Domains

$\mathcal{M}$	:	Prog $\rightarrow$ State
$\mathcal{S}$	:	Stmt $\rightarrow$ State $\rightarrow$ State
$\mathcal{E}$	:	Exp $\rightarrow$ State $\rightarrow$ Ev

Figure 9: Semantic Functions

A Pam program, which is simply a list (series) of statements, is mapped to a final state by calling the function $\mathcal{M}$ that defines the meaning of a statement with an everywhere undefined state (store). In direct denotational semantics, a statement and a state directly produce a new state. Normally, the state in a denotational definition also contains an input list, serving as an input file, and an output list, initially empty, which may be modified by IO statements.

$\mathcal{M}[\![\Psi]\!] = \mathcal{S}[\![\Psi]\!](\lambda\Xi \,.\, \perp)$

$\mathcal{S}[\![\Xi := E]\!]\sigma = \sigma[\mathcal{E}[\![E]\!]\sigma \,/\, \Xi]$

$\mathcal{S}[\![$**if** E **then** Σ **fi**$]\!]\sigma =$ if $\mathcal{E}[\![E]\!]\sigma$ then $\mathcal{S}[\![\Sigma]\!]\sigma$ else σ

$\mathcal{S}[\![$**if** E **then** Σ_1 **else** Σ_2 **fi**$]\!]\sigma =$ if $\mathcal{E}[\![E]\!]\sigma$ then $\mathcal{S}[\![\Sigma_1]\!]\sigma$ else $\mathcal{S}[\![\Sigma_2]\!]\sigma$

$\mathcal{S}[\![$**while** E **do** Σ **end**$]\!]\sigma =$ if $\mathcal{E}[\![E]\!]\sigma$ then $\mathcal{S}[\![$**while** E **do** Σ **end**$]\!] \circ \mathcal{S}[\![\Sigma]\!]\sigma$ else σ

$\mathcal{S}[\![\Sigma_1 ; \Sigma_2]\!] = \mathcal{S}[\![\Sigma_2]\!] \circ \mathcal{S}[\![\Sigma_1]\!]$

$\mathcal{S}[\![$**to** E **do** Σ **end**$]\!]\sigma =$ if $\nu > 0$ then $\mathcal{S}[\![\Sigma]\!]^{\nu}\sigma$ else σ where $\nu = \mathcal{E}[\![E]\!]\sigma$

$\mathcal{S}[\![$**read** $\Xi]\!]\sigma = \sigma[$val$/\Xi]$ where val is the next value read

$\mathcal{S}[\![$**read** Ξ , $\Lambda]\!]\sigma = \mathcal{S}[\![$**read** $\Lambda]\!](\sigma[$val$/\Xi])$ where val is the next value read

$\mathcal{S}[\![$**write** $\Xi]\!]\sigma = \sigma$ where the value of $\sigma[\![\Xi]\!]$ is printed next

$\mathcal{S}[\![$**write** Ξ , $\Lambda]\!]\sigma = \mathcal{S}[\![$**write** $\Lambda]\!]\sigma$ where the value of $\sigma[\![\Xi]\!]$ is printed next

$\mathcal{E}[\![\Xi]\!]\sigma = \sigma[\![\Xi]\!]$ $\qquad$ $\mathcal{E}[\![N]\!]\sigma = N$ $\qquad$ $\mathcal{E}[\![(E)]\!] = \mathcal{E}[\![E]\!]$

$\mathcal{E}[\![E_1 \ \Theta \ E_2]\!]\sigma = \mathit{compute}(\Theta, \mathcal{E}[\![E_1]\!]\sigma, \mathcal{E}[\![E_2]\!]\sigma)$

Figure 10: Semantic Equations for Pam

Since the intention here is to produce an interpreter, input and output are treated as side effects of the semantic functions for statements, which explicitly modify only the store. The interpreter will be run in an interactive environment to encourage experimentation, so input and output are handled by Prolog procedures through the standard devices.

The semantic equations of denotational semantics can be translated into Prolog predicates in a straightforward manner letting each semantic function be expressed as a Prolog predicate. Observe that the curried functions of the definition become uncurried as predicates. The delayed binding of logical variables in Prolog creates the same effect as the partial parameterization of curried functions. See Figure 11 for the correspondence between the semantic functions of the denotational definiton and the Prolog predicates used to implement them.

Statement composition can be handled by simply treating the state transitions for a list of statements:

```
sS([Stmt | Stmts],State,NewState) :-
        sS(Stmt,State,TpState),
        sS(Stmts,TpState,NewState).

sS([],State,State).
```

A **while** statement requires evaluating the test expression and then determining whether to execute the body or to return the state unchanged:

```
sS(while(Test,Body),State,NewState) :-
    eE(Test,State,Ev),
    iterate(Ev,Test,Body,State,NewState).

iterate(true,Test,Body,State,NewState) :-
    sS(Body,State,TpState),
    sS(while(Test,Body),TpState,NewState).

iterate(false,_,_,State,State).
```

A Prolog representation of the state (or store) needs to be formulated for statements that directly alter the state. The state is considered to be a finite function, which is undefined wherever it is not explicitly specified. It can be represented as list-like structure, say "sto(a, 5, sto(b, 8, sto(c, 13, nil)))" to represent the bindings [5/a, 8/b, 13/c] where "nil" stands for the totally undefined store. A Prolog implementation can handle modifications and queries to the store by means of predicates "updateSto" and "applySto", respectively. An assignment statement requires an update of the store:

```
sS(assign(Var,Exp),State,NewState) :-
        eE(Exp,State,Ev),
        updateSto(State,Var,Ev,NewState).
```

Expressions are evaluated in the context of a store, which provides values for variables. The actual calculations are carried out by a Prolog predicate "compute". For example,

```
compute(divides,N,0,0) :-
        write('Division by zero'), nl, abort.

compute(divides,N,D,R) :- R is N//D.
```

A sample execution of the denotational interpreter with the source program in Figure 3 is shown in Figure 12. User input is displayed in boldface.

MECHANICS

The students are given a predicate definition to handle the file processing:

```
go :- write('>>> Interpreting Pam<<<'), nl,
    write('Enter name of source file: '),
    readstr(File), exists(File), see(File),
    scan(Tokens), seen, write(Tokens), nl,
    program(AST,Tokens,[eof]), write(AST),
    nl, mM(AST,State), nl,
    write('Final State:'), nl, printstate(State).
```

Semantic Functions	Prolog Predicates
$\mathcal{M}$: Prog → State	mM(Stmts,State)
$\mathcal{S}$: Stmt → State → State	sS(Stmt,State,NewState)
$\mathcal{E}$: Exp → State → Ev	eE(Exp,State,Evalue)

Figure 11

```
>>> Interpreting Pam <<<
Enter name of source file: powers
Scan successful
Parse successful
Input: 17
Output = 17
Output = 17
Output = 17
Output = 289
Output = 17
Output = 4913
Final State:
                x       17
                p       4913
yes
```

Figure 12

They are instructed to have their scanner produce a listing of the Pam program and to be able to print the token list, the abstract syntax tree, and the final state as well as the output of the program.

The predicate "scan" has one output parameter for its result. The predicate "program" takes the token list, "Tokens", and after a successful parsing by the logic grammar, which leaves only the token "eof", produces the abstract syntax tree in the variable "AST". The predicate "mM" takes that tree and constructs the final state in the variable "State" by calling the statement predicate "sS" with an initially undefined state:

```
mM(Stmts,State) :- sS(Stmts,nil,State).
```

CONCLUSIONS

The goal of this paper has been to describe an exercise that provides a concrete application of denotational semantics using Prolog. Students generally find the actual programming assignments surprisingly easy.

The interpreter can serve as a starting point for a prototyping tool for investigating extensions and alterations in the semantic constructs of a programming language. For example, although single-scope Pam has no need for environments, a block structured language with its associated environments can be interpreted in Prolog in a similar manner.

Sequencers, such as "goto" and "exit", complicate the denotational semantics by requiring continuations. Prolog can also express continuation semantics [Slonneger89] with some increased complexity in the denotational semantics and in the Prolog interpreter.

REFERENCES

[Allison83]
Lloyd Allison, "Programming Denotational Semantics", **The Computer Journal**, Volume 26, Number 2, pp.164-174.

[Allison86]
Lloyd Allison, **A Practical Introduction to Denotational Semantics**, Cambridge University Press.

[Clocksin84]
W.F.Clocksin and C.S.Mellish, **Programming in Prolog, Second Edition**, Springer -Verlag.

[Moss82]
Chris D. Moss, "How to define a language using Prolog", **Proc. 1982 ACM Symp. on Lisp and Functional Programming**, pp.67-73.

[Pagan79]
Frank Pagan, "Algol68 as a metalanguage for denotational semantics", **The Computer Journal**, Volume 22, Number 1, pp.63-66.

[Pagan81]
Frank Pagan, **Formal Specification of Programming Languages: A Panoramic Primer**, Prentice-Hall.

[Slonneger89]
Ken Slonneger, "Denotational Semantics in Prolog", Technical Report 89-02, Department of Computer Science, The University of Iowa.