The PCAT Programming Language Reference Manual

Andrew Tolmach and Jingke Li
Dept. of Computer Science
Portland State University

September 27, 1995
(revised December 30, 1998)

1 Introduction

The **PCAT** language (**P**ascal **C**lone with an **AT**titude) is a small imperative programming language with nested functions, record values with implicit pointers, arrays, integer and real variables, and a few simple structured control constructs.

This manual gives an informal definition for the language. Fragments of syntax are specified in BNF as needed; the complete grammar is attached as an appendix.

2 Lexical Issues

PCAT's character set is the standard ASCII set. PCAT is case sensitive; upper and lower-case letters are *not* considered equivalent.

White space (blank, tab or end-of-line) serve to separate tokens; otherwise they are ignored. Whitespace is needed between two adjacent keywords or identifiers, or between a keyword or identifier and a number. However, no whitespace is required between a number and a keyword, since this causes no ambiguity. Delimiters and operators don't need whitespace to separate them from their neighbors on either side. White space may not appear in any token except a string (see below).

Comments are enclosed in the pair `(*` and `*)`; they cannot be nested. Any character is legal in a comment. Of course, the first occurrence of the sequence of characters `*)` will terminate the comment. Comments may appear anywhere a token may appear; they are self-delimiting; i.e. they do not need to be separated from their surroundings by whitespace.

2.1 Tokens

Tokens consist of keywords, literal constants, identifiers, operators, and delimiters.

The following are reserved *keywords*. They must be written in upper case.

```
AND      ARRAY    BEGIN    BY       DIV       DO       ELSE
ELSIF    END      EXIT     FOR      IF        IS       LOOP
MOD      NOT      OF       OR       PROCEDURE PROGRAM  READ
RECORD   RETURN   THEN     TO       TYPE      VAR      WHILE
WRITE
```

Literal constants are either integer, real, or string. *Integers* contain only digits; they must be in the range 0 to $2^{31}-1$. *Reals* consist of one or more digits, followed by a decimal point, followed by zero or more digits. There is no specific range constraint on reals, but the literal as a whole is limited to 255 characters in length. Note that no numeric literal can be negative, since there is no provision for a minus sign. *Strings* begin and end with a double quote (`"`) and contain any sequence of printable ASCII characters (i.e., having decimal character codes in the range 32 – 126) except double quote. Note in particular that strings may not contain tabs or newlines. String literals are limited to 255 characters in length, not including the delimiting double quotes.

Identifiers are strings of letters and digits starting with a letter, excluding the reserved keywords. Identifiers are limited to 255 characters in length.

The following are the *operators*:

```
:=  +  -  *  /  <  <=  >  >=  =  <>
```

and the *delimiters*:

```
:  ;  ,  .  (  )  [  ]  {  }  [<  >]
```

3 Programs

A program is the unit of compilation for PCAT. Programs have the following syntax:

```
program          -> PROGRAM IS body ';'
body             -> {declaration} BEGIN {statement} END
```

A program is executed by executing its statement sequence and then terminating.

Each file read by the compiler must consist of exactly one program. There is no facility for linking multiple programs or for separate compilation of parts of a program.

4 Declarations

All identifiers occurring in a program must be introduced by a declaration, except for a small set of predefined identifiers: `REAL`, `INTEGER`, `BOOLEAN`, `TRUE`, `FALSE` (see Section 5.1), and `NIL` (see Section 5.3).

Declarations serve to specify whether the identifier represents a type, a variable, or a procedure (all of which live in a single **name space**) or a record component name (which live in separate name spaces; see Section 5.3).

```
declaration      -> VAR {var-decl}
                 -> TYPE {type-decl}
                 -> PROCEDURE {procedure-decl}
```

Declarations may be **global** to the program or **local** to a particular procedure. The **scope** of a declaration extends roughly from the point of declaration to the end of the enclosing module (for local declarations) or the end of the program (for global declarations). The detailed scope rules differ for each kind of declaration (see Sections 5,7, and 8). A local declaration of an identifier **hides** any outer declarations and makes them inaccessible in the inner scope. No identifier may be declared twice in the same procedure or at global level.

5 Types

PCAT is a strongly-typed language; every expression has a unique type, and types must match at assignments, calls, etc. (except that an integer can be used where a real is expected; see Section 5.1.)

Types are referred to by **type names**. The built-in **basic types** (see Section 5.1) have predefined names; new types are created by **type declarations** in which the **type constructors** `ARRAY` or `RECORD` are applied to existing types.

```
declaration      -> TYPE {type-decl}
type-decl        -> typename IS type ';'
typename         -> ID
type             -> ARRAY OF typename
                 -> RECORD component {component} END
component        -> ID ':' typename ';'
```

PCAT essentially uses a **name equivalence** model for types; each type declaration produces a new, unique type, incompatible with all the others.

5.1 Built-in Types

There are three **built-in** basic types: `INTEGER`, `REAL`, and `BOOLEAN`; these can be redefined by inner declarations (though this is unwise). Integer literal constants all have type `INTEGER`, real literal constants all have type `REAL`, and the built-in values `TRUE` and `FALSE` have type `BOOLEAN`.

`INTEGER` and `REAL` collectively form the **numeric** types. An `INTEGER` value will always be implicitly **coerced** to a `REAL` value if necessary. The boolean type has no relation to the numeric types, and a boolean value cannot be converted to or from a numeric value.

5.2 Array Types

An array is a structure consisting of zero or more elements of the same **element type**. The elements of an array can be accessed by **dereferencing** using an **index**, which ranges from 0 to the length of the array minus 1. The length of an array is not fixed by its type, but is determined when the array is created at runtime. It is a checked runtime error to dereference outside the bounds of an array.

5.3 Record Types

A record type is a structure consisting of a fixed number of **components** of (possibly) different types. The record type declaration specifies the name and type of each component. Component names are used to initialize and dereference components; the components for each record type form a separate namespace, so different record types may reuse the same component names.

The special built-in value `NIL` belongs to every record type. It is a checked runtime error to dereference a component from the nil record.

5.4 Constructed Type Values

Arrays and records are always manipulated by value, so a value of array or record type is "really" a pointer to a heap object containing the array or record, though this pointer cannot be directly manipulated by the programmer. Thus, a record type that appears to contain other record types as components actually contains pointers to these types. In particular, a record type may contain (a pointer to) itself as a component, i.e., it may be recursive.

To permit mutually recursive types, the set of type declarations following a single `TYPE` keyword is taken to be a recursive set; the scope of all the declarations in the set begins at the **first** declaration. Note the utility of the `NIL` record for building values of recursive types.

Records and arrays have unlimited lifetimes; the heap object containing a record or array exists from the moment when its defining expression is evaluated (see Sections 10.6 and 10.7) until the end of the program. In principle, a garbage collector could be used to remove heap objects when no more pointers to them exist, but this is invisible to the **PCAT** programmer.

6 Constants

There are three **built-in constant** values: `TRUE` and `FALSE` of type `BOOLEAN`, and `NIL`, which belongs to every record type. There is no provision for user-defined constants.

7 Variables

Variables are declared thus:

```
declaration      -> VAR {var-decl}
var-decl         -> ID { ',' ID } [ ':' typename ] ':=' expression ';'
```

Every value must have an initial value, given by `expression`. The type designator can be omitted whenever the type can be deduced from the initial value, i.e., except when the initial value is `NIL`.

The scope of each variable declaration begins just before next declaration; it does *not* include the initializing expression, so declarations are never recursive.

8 Procedures

Procedures are declared thus:

```
declaration      -> PROCEDURE {procedure-decl}
procedure-decl   -> ID formal-params [':' typename] IS body ';'
formal-params    -> '(' fp-section {';' fp-section } ')'
                 -> '(' ')'
fp-section       -> ID {',' ID} ':' typename
body             -> {declaration} BEGIN {statement} END
```

Procedures encompass both **proper procedures**, which are activated by the execution of a procedure call statement and do not return a value, and **function procedures**, which are activated by the evaluation of a procedure call expression and return a value which becomes the value of the call expression. Proper procedure declarations are distinguished by the lack of a return type (see also Section 11.10).

A procedure may have zero or more **formal parameters**, whose names and types are specified in the procedure declaration, and whose actual values are specified when the procedure is activated. The scope of formal parameters is the body of the procedure (including its local declarations). Parameters are always passed by value.

There is an implicit `RETURN` statement at the bottom of every procedure body.

Each set of procedures declared following a single `PROCEDURE` keyword is treated as (potentially) mutually recursive; that is, the scope of each procedure name begins at the point of declaration of the first procedure in the set, and includes the bodies of all the procedures in the set as well as the body of the enclosing procedure (or, for top-level procedures, the whole program).

9 L-values

An **l-value** is a location whose value can be either read or assigned. Variables, procedure parameters, record components, and array elements are all l-values.

```
lvalue           -> ID
                 -> lvalue '[' expression ']'
                 -> lvalue '.' ID
```

The square brackets notation (`[]`) denotes array element dereferencing; the expression within the brackets must evaluate to an integer expression within the bounds of the array.

The dot notation (`.`) denotes record component dereferencing; the identifier after the dot must be a component name within the record.

10 Expressions

10.1 Simple expressions

```
expression       -> number
                 -> lvalue
                 -> '(' expression ')'
number           -> INTEGER | REAL
```

A number expression evaluates to the literal value specified. Note that reals are distinguished from integers by lexical criteria (see Section 2). An l-value expression evaluates to the current contents of the specified location. Parentheses can be used to alter precedence in the usual way.

10.2 Arithmetic operators

```
expression      -> unary-op expression
                -> expression binary-op expression
unary-op        -> '+' | '-'
binary-op       -> '+' | '-' |  '*' | '/' | DIV | MOD
```

Operators `+,-,*` require integer or real arguments. If both arguments are integers, an integer operation is performed and the integer result is returned; otherwise, any integer arguments are coerced to reals, a real operation is performed, and the real result is returned. Operator `/` requires integer or real arguments, coerces any integer arguments to reals, performs a real division, and always returns a real result. Operators `DIV` (integer quotient) and `MOD` (integer remainder) take integer arguments and return an integer result. All the binary operators evaluate their left argument first.

10.3 Logical operators

```
expression      -> unary-op expression
                -> expression binary-op expression
unary-op        -> NOT
binary-op       -> OR | AND
```

These operators require boolean operands and return a boolean result. `OR` and `AND` are "short-circuit" operators; they do not evaluate the right-hand operand if the result is determined by the left-hand one.

10.4 Relational operators

```
expression      -> expression binary-op expression
binary-op       -> '>' | '<' | '=' | '>=' | '<=' | '<>'
```

These operators all return a boolean result. These operators all work on numeric arguments; if both arguments are integer, an integer comparison is made; otherwise, any integer argument is coerced to real and a real comparison is made. Operators `=` and `<>` also work on pairs of boolean arguments, or pairs of record or array arguments of the same type; for the latter, they test "pointer" equality (that is, whether two records or arrays are the same instance, not whether they have the same contents). These operators all evaluate their left argument first.

10.5 Procedure call

```
expression      -> ID actual-params
actual-params   -> '(' expression {',' expression} ')'
                -> '(' ')'
```

This expression is evaluated by evaluating the argument expressions left-to-right to obtain actual parameter values, and then executing the function procedure specified by `ID` with its formal parameters bound to the actual parameter values. The procedure returns by executing an explicit `RETURN` statement (with an expression for the value to be returned). The returned value becomes the value of the procedure call expression.

10.6 Record construction

```
expression      -> typename record-inits
record-inits    -> '{' ID ':=' expression { ';' ID ':=' expression} '}'
```

If *typename* is a record type name, then *typename* $\{id_1{=}exp_1, id_2{=}exp_2, \ldots\}$ evaluates each expression left-to-right, and then creates a new record instance of type `typename` with named components initialized to the resulting values. The names and types of the component initializers must match those of the named type, though they need not be in the same order.

10.7 Array construction

```
expression      -> typename array-inits
array-inits     -> '[<' array-init { ',' array-init } '>]'
array-init      -> [ expression 'OF' ] expression
```

If `typename` is an array type name, then *typename* `[<`$expr_1^n$ `OF` $expr_1^v$`,` $expr_2^n$ `OF` $expr_2^v$`,...>]` evaluates each pair of expressions in left-to-right order to yield a list of pairs of integer counts n_i and initial values v_i, and then creates a new array instance of type `typename` whose contents consist of n_1 copies of v_1, followed by n_2 copies of v_2, etc. If any of the counts is 1, it may be omitted. For example, the specification `[<1,2 OF 3,3 OF 2,4>]` yields an array of length 7 with contents `1,3,3,2,2,2,4`. If any of the n_i is less than 1, no copies of the corresponding v_i are included. The types of the v_i must match the element type of the named array type.

10.8 Precedence and associativity

Procedure call and parenthesization have the highest (most binding) precedence; followed by the unary operators; followed by `*`, `/`, `MOD`, `DIV`, and `AND`; followed by `+`, `-`, and `OR`.

Relational expressions cannot be embedded into other expressions unless parenthesized. For example, `a < b OR e > f` and `a < b = c` are illegal expressions, whereas `(a < b) OR (e > f)` and `(a < b) = c` are legal (presuming `c` has type `BOOLEAN`).

Within precedence classes, the binary operators are all left-associative.

11 Statements

11.1 Assignment

```
statement       -> lvalue ':=' expression ';'
```

The l-value is evaluated to a location; then the expression is evaluated and its value is stored in the location.

Assigning a record or array value actually assigns a pointer to the record or array.

11.2 Procedure Call

```
statement       -> ID actual-params
actual-params   -> '(' expression {',' expression} ')'
                -> '(' ')'
```

This statement is executed by evaluating the argument expressions left-to-right to obtain actual parameter values, and then executing the proper procedure specified by `ID` with its formal parameters bound to the actual parameter values. The procedure returns when its final statement or an explicit `RETURN` statement (with no expression) is executed.

11.3 Read

```
statement       -> READ '(' lvalue {',' lvalue} ')' ';'
```

This statement is executed by evaluating the l-values to locations in left-to-right order, and then reading numeric literals from standard input, evaluating them, and assigning the resulting values into the locations. The l-values must have type integer or real, and their types guide the evaluation of the corresponding literals. Input literals are delimited by whitespace, and the last one must be followed by a carriage return.

11.4 Write

```
statement       -> WRITE write-params ';'
write-params    -> '(' write-expr {',' write-expr } ')'
                -> '(' ')'
write-expr      -> STRING
                -> expression
```

Executing this statement evaluates the specified expressions (which must be simple integers, reals, booleans, or string literals) in left-to-right order, and then writes the resulting values to standard output (with no separation between values), followed by a new line.

11.5 If-then-else

```
statement       -> IF expression THEN {statement}
                   {ELSIF expression THEN {statement}}
                   [ELSE {statement}] END ';'
```

This statement specifies the conditional execution of guarded statements. The expression preceding a statement sequence, which must evaluate to a boolean, is called its **guard**. The guards are evaluated in left-to-right order, until one evaluates to `TRUE`, after which its associated statement sequence is executed. If no guard is satisfied, the statement sequence following the `ELSE` (if any) is executed.

11.6 While

```
statement       -> WHILE expression DO {statement} END ';'
```

The statement sequence is repeatedly executed as long as the expression evaluates to `TRUE`, or until the execution of an `EXIT` statement within the sequence (but not inside any nested `WHILE`, `LOOP`, or `FOR`).

11.7 Loop

```
statement       -> LOOP {statement} END ';'
```

The statement sequence is repeatedly executed. The only way to terminate the iteration is by executing an `EXIT` statement within the sequence but not inside any nested `WHILE`, `LOOP`, or `FOR`.

11.8 For

```
statement       -> FOR ID ':=' expression TO expression [ BY expression ]
                   DO {statement} END ';'
```

Executing the statement `FOR` *id* `:=` exp_1 `TO` exp_2 `BY` exp_3 `DO` *stmts* is equivalent to the following steps: (i) evaluate expressions exp_1, exp_2, and exp_3 in that order to values v_1, v_2, v_3 (which must be integers); (ii) if the value of *id* is less than or equal to v_2, execute *stmts*; otherwise terminate the loop. (iii) set $id := id + v_3$ and repeat step (ii).

If the `BY` clause is omitted, v_3 is taken to be 1.

`ID` is an ordinary integer variable; it must be declared in the scope containing the `FOR` statement, and it can be inspected or set above, within, or below the loop body.

If an `EXIT` statement is executed within the body of the loop (but not within the body of any nested `WHILE`, `LOOP` or `FOR` statement), the loop is prematurely terminated, and control passes to the statement following the `FOR`.

11.9 Exit

```
statement       -> EXIT ';'
```

Executing `EXIT` causes control to pass immediately to the next statement following the nearest enclosing `WHILE`, `LOOP` or `FOR` statement. If there is no such enclosing statement, the `EXIT` is illegal.

11.10 Return

```
statement       -> RETURN [expression] ';'
```

Executing `RETURN` terminates execution of the current procedure and returns control to the calling context. There can be multiple `RETURN`s within one procedure body, and there is an implicit `RETURN` at the bottom of every procedure. A `RETURN` from a function procedure must specify a return value expression of the return type; a `RETURN` from a proper procedure must not. The main program body must not include a `RETURN`.

12 Complete Concrete Syntax

```
program         -> PROGRAM IS body ';'
body            -> {declaration} BEGIN {statement} END
declaration     -> VAR {var-decl}
                -> TYPE {type-decl}
                -> PROCEDURE {procedure-decl}
var-decl        -> ID { ',' ID } [ ':' typename ] ':=' expression ';'
type-decl       -> ID IS type ';'
procedure-decl  -> ID formal-params [':' typename] IS body ';'
typename        -> ID
type            -> ARRAY OF typename
                -> RECORD component {component} END
component       -> ID ':' typename ';'
formal-params   -> '(' fp-section {';' fp-section } ')'
                -> '(' ')'
fp-section      -> ID {',' ID} ':' typename
statement       -> lvalue ':=' expression ';'
                -> ID actual-params ';'
                -> READ '(' lvalue {',' lvalue} ')' ';'
                -> WRITE write-params ';'
                -> IF expression THEN {statement}
                   {ELSIF expression THEN {statement}}
                   [ELSE {statement}] END ';'
                -> WHILE expression DO {statement} END ';'
                -> LOOP {statement} END ';'
                -> FOR ID ':=' expression TO expression [ BY expression ]
                   DO {statement} END ';'
                -> EXIT ';'
                -> RETURN [expression] ';'
write-params    -> '(' write-expr {',' write-expr } ')'
                -> '(' ')'
write-expr      -> STRING
                -> expression
expression      -> number
                -> lvalue
                -> '(' expression ')'
                -> unary-op expression
                -> expression binary-op expression
                -> ID actual-params
                -> ID record-inits
                -> ID array-inits
lvalue          -> ID
                -> lvalue '[' expression ']'
                -> lvalue '.' ID
actual-params   -> '(' expression {',' expression} ')'
                -> '(' ')'
record-inits    -> '{' ID ':=' expression { ';' ID ':=' expression} '}'
array-inits     -> '[<' array-init { ',' array-init } '>]'
array-init      -> [ expression OF ] expression
number          -> INTEGER | REAL
unary-op        -> '+' | '-' | NOT
binary-op       -> '+' | '-' |  '*' | '/' | DIV | MOD | OR | AND
                -> '>' | '<' | '=' | '>=' | '<=' | '<>'
```

13 Abstract Syntax

Here is the official communication format for the abstract syntax for **PCAT**.

```
program         -> body
body            -> '(' BodyDef line '(' {declarations} ')'  statement ')'
declarations    -> '(' VarDecs '(' {var-dec} ')' ')'
                -> '(' TypeDecs '(' {type-dec} ')' ')'
                -> '(' ProcDecs '(' {proc-dec} ')' ')'
var-dec         -> '(' VarDec line ID type expression ')'
type-dec        -> '(' TypeDec line ID type ')'
proc-dec        -> '(' ProcDec line ID '(' { formal-param } ')' type body ')'
type            -> '(' NamedTyp line ID ')'
                -> '(' ArrayTyp line type ')'
                -> '(' RecordTyp line '(' { component } ')' ')'
                -> '(' NoTyp ')'
component       -> '(' Comp line ID type ')'
formal-param    -> '(' Param line ID type ')'
statement       -> '(' AssignSt line lvalue expression ')'
                -> '(' CallSt line ID '(' { expression } ')' ')'
                -> '(' ReadSt line '(' { lvalue } ')' ')'
                -> '(' WriteSt line '(' { expression } ')' ')'
                -> '(' IfSt line expression statement statement ')'
                -> '(' WhileSt line expression statement ')'
                -> '(' LoopSt line statement ')'
                -> '(' ForSt line ID expression expression expression statement ')'
                -> '(' ExitSt line ')'
                -> '(' RetSt line [ expression ] ')'
                -> '(' SeqSt '(' { statement } ')' ')'
expression      -> '(' BinOpExp line binop expression expression ')'
                -> '(' UnOpExp line unop expression ')'
                -> '(' LvalExp lvalue ')'
                -> '(' CallExp line ID '(' { expression } ')' ')'
                -> '(' RecordExp line ID '(' { record-init } ')' ')'
                -> '(' ArrayExp line ID '(' { array-init } ')' ')'
                -> '(' IntConst line INTEGER ')'
                -> '(' RealConst line STRING ')'
                -> '(' StringConst line STRING ')'
record-init     -> '(' RecordInit ID expression ')'
array-init      -> '(' ArrayInit expression expression ')'
lvalue          -> '(' Var line ID ')'
                -> '(' ArrayDeref line lvalue expression ')'
                -> '(' RecordDeref line lvalue ID ')'
binop           -> GT | LT | EQ | GE | LE | NE | PLUS | MINUS | TIMES | SLASH
                -> DIV | MOD | AND | OR
unop            -> UPLUS | UMINUS | NOT
line            -> INTEGER
```

Here `line` is the source line number to use in error messages referring to this construct.