The PCAT Programming Language Reference Manual

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

September 27, 1995

Revisions: Harry Porter

September 22, 2005

1 Introduction

The **PCAT** language 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 of the language. Fragments of syntax are specified in BNF throughout this document as needed; the complete context-free grammar of PCAT is given in Section 12.

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.

Whitespace (blank, tab, or newline characters) may be used to separate tokens and to improve readability, but whitespace is otherwise ignored. To eliminate ambiguity however, whitespace is required (1) between two adjacent keywords or identifiers, (2) between a keyword and a number following it, and (3) between an identifier and a number following it. No whitespace characters are required between a number and a keyword following it, since this causes no ambiguity. Delimiters and operators don't need whitespace to separate them from their neighboring tokens on either side. Whitespace characters may not appear within any token except a string.

Comments are enclosed in the pair `(*` and `*)`; they cannot be nested. Any character is legal in a comment, including new-line characters. 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

The following are reserved keywords. They must be written in lower case. (For clarity, keywords are shown in boldface throughout this document.)

```
and       do        for       not       read      type
array     else      if        of        record    var
begin     elseif    is        or        return    while
by        end       loop      procedure then      write
div       exit      mod       program   to
```

Constants are either integer, real, or string. *Integers* contain only digits; they must be in the range 0 to 2^{31}-1. *Reals* contain a decimal point; a digit is required before the decimal point, but not afterwards. *Strings* begin and end with a double quote (") and contain any sequence of printable ASCII characters, except double quotes. Strings may not contain unprintable characters such as tabs or newlines. String literals are limited to 255 characters in length, not including the delimiting double quotes.

Using a regular expression notation in which "|" represents set union, "{}" represents Kleene closure, NOT represents set complement, and literal symbols are enclosed in single quotes ('), the above definitions may be made more precise:

```
letter  = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' |
          'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' |
          'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' |
          'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' |
          'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' |
          's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
digit   = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' |
          '9'
INTEGER = digit {digit}
REAL    = digit {digit} '.' {digit}
STRING  = '"' {NOT('"')} '"'
```

Note that neither an integer nor a real can be negative, since there is no provision for a minus sign.

Identifiers are strings of letters and digits starting with a letter, excluding the reserved keywords. Identifiers can be specified as follows, where `RESERVED` represents the set of reserved keywords listed above:

```
ID      = ( letter {letter | digit} ) - RESERVED
```

Identifiers are limited to 255 characters in length.

The following are the remaining operators and delimiters:

```
operator  =  ':=' | '+' | '-' | '*' | '/' | '<' | '<=' |
             '>' | '>=' | '=' | '<>'
delimiter =  ':' | ';' | ',' | '.' | '(' | ')' | '[' | ']' |
             '{' | '}'
```

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
```

(In the context-free grammar notation used in this document, braces {} are used to mean "zero or more occurrences" and brackets [] are used for optional items. The entire grammar is repeated for easy reference in Section 12.)

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

Each file processed by the compiler will consist of exactly one complete 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 5.2), and `nil` (see Section 5.4).

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 field name (which live in separate name spaces; see Section 5.4).

```
Declaration       → var VarDecl {VarDecl}
                  → type TypeDecl {TypeDecl}
                  → procedure ProcedureDecl {ProcedureDecl}
```

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 procedure (for local declarations) or to the end of the program (for global declarations). A local declaration of an identifier hides any outer declarations and makes them inaccessible in the inner scope. Within a single scope, all variables, types, and procedures must have unique names. Thus, within a single procedure, the same identifier cannot be used to name both a variable and a type.

To handle recursive declarations, this scoping rule is modified slightly for types (see Section 5) and procedures (see Section 8).

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 may be *basic types* or may be produced from existing types using the *type constructors* **`array`** or **`record`**. PCAT uses the *name equivalence* model for types: two types are equal if and only if they have the same name.

5.1 Basic Types

There are three built-in, basic types: `integer`, `real`, and `boolean`; these can not be redefined by type declarations. Integer constants all have type `integer`, real 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 may be used whenever a `real` is expected; the compiler will automatically insert a conversion 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 Constructed Types and Type Declarations

Each array type or record type must be defined and given a name in a *Type Declaration*. For example:

```
type MyArr is array of MyRec;
     MyRec is record
                id: integer;
                next: MyRec;
              end;
```

Here are the grammar rules relating to type declarations:

```
Declaration       → type TypeDecl {TypeDecl}
TypeDecl          → TypeName is CompoundType ';'
TypeName          → ID
CompoundType      → array of TypeName
                  → record FieldDecl {FieldDecl} end
FieldDecl         → ID ':' TypeName ';'
```

Only array types and record types may be defined in type declarations. Types (whether basic or compound) may not be renamed or given aliases.

5.2.1 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. (The creation and allocation of new arrays at runtime is discussed in Section 10.7.) It is a checked runtime error to dereference outside the bounds of an array.

5.2.2 Record Types

A record type is a structure consisting of a fixed number of *fields* of generally different types. The record type declaration specifies the name and type of each field. Field names are used to initialize and access the record's components; the fields for each record type form a separate namespace, so different record types may reuse the same field names.

The special built-in value `nil` belongs to every record and array type. It is a checked runtime error to dereference a field from a nil record reference. The creation (i.e., allocation) of new records at runtime is discussed in Section 10.6.

5.2.3 Array and Record 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 that appears to contain other records as values of its fields actually contains pointers to these records. In particular, a record may contain (a pointer to) itself in one of its fields, i.e., the record type may be recursive.

To permit mutually recursive types, the set of all type declarations in a single body (i.e., within a single procedure or within the declaration section of the program body) is taken to be a recursive set; the scope of all the declarations in the set begins at the *first* declaration.

Records and arrays have infinite lifetimes; the heap object containing a record or array exists from the moment it is allocated 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 would be 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 and array type. There is no provision for user-defined constants.

7 Variables

Variables are declared thus:

```
Declaration       → var VarDecl {VarDecl}
VarDecl           → ID { ',' ID} [ ':' TypeName ] ':=' Expression ';'
```

Every variable must have an initial value, given by `Expression`. The optional `TypeName` can be omitted whenever the variable's type can be deduced from the initial value, i.e., in all cases except when the initial value is `nil`.

Variable initializing expressions are evaluated one at a time, in order; they are never recursive although the initializing expression in one variable declaration may make use of another variable declared previously in the same scope.

8 Procedures

Procedures are declared thus:

```
Declaration       → procedure ProcedureDecl {ProcedureDecl}
ProcedureDecl     → ID FormalParams [':' TypeName] is Body ';'
FormalParams      → '(' FormalSection {';' FormalSection} ')'
                  → '(' ')'
FormalSection     → ID {',' ID} ':' TypeName
Body              → {Declaration} begin {Statement} end
```

Procedures encompass both *proper procedures*, which are invoked by the execution of a procedure "call statement" and do not return a value, and *function procedures*, which are invoked by the evaluation of a "function 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 the **`return`** statement discussed in 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 called. The scope of formal parameters is the body of the procedure (including its local declarations). Parameters are always passed by value.

The set of all procedures declared in a single namespace (i.e., the main program for globally declared procedures or the surrounding procedure for nested procedures) 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 namespace, and includes the bodies of all the procedures in the namespace 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 to. Variables, procedure parameters, record fields, 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 field dereferencing; the identifier after the dot must be a field 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      → UnaryOp Expression
                → Expression BinaryOp Expression
UnaryOp         → '+' | '-'
BinaryOp        → '+' | '-' | '*' | '/' | div | mod
```

Operators `+`, `-`, and `*` 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.

10.3 Logical operators

```
Expression      → UnaryOp Expression
                → Expression BinaryOp Expression
UnaryOp         → not
BinaryOp        → or | and
```

These operators require Boolean operands and return a Boolean result. The operators **`or`** and **`and`** are "short-circuit" operators; they will not evaluate the right-hand operand if the result is determined by the left-hand one.

10.4 Relational operators

```
Expression        → Expression BinaryOp Expression
BinaryOp          → '>' | '<' | '=' | '>=' | '<=' |  '<>'
```

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 two arrays are the same instance, not whether they have the same contents).

10.5 Function call

```
Expression        → ID Arguments
Arguments         → '(' 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 until a **`return`** statement is executed, returning a value.

10.6 Record construction

A record constructor expression is used to allocate a new record on the heap, initialize each of its fields, and return a pointer to the newly created record.

```
Expression       → ID FieldInits
FieldInits       → '{' ID ':=' Expression { ';' ID ':=' Expression} '}'
```

If *typeid* is the name of a record type, then the record constructor *typeid* `{`id_1`:=`$expr_1$`,` id_2`:=`$expr_2$`, ...}` evaluates each expression left-to-right, and then creates a new record instance of type *typeid* with its named fields initialized to the resulting values. The names and types of the field initializers must match those of the record type named *typeid*, though they need not be listed in the same order.

10.7 Array construction

An array constructor expression is used to allocate a new array on the heap, initialize each of its elements, and return a pointer to the newly created array.

```
Expression        → ID ArrayValues
ArrayValues       → '{' '{' ArrayValue { ',' ArrayValue} '}' '}'
ArrayValue        → [ Expression of ] Expression
```

If *typeid* is the name of an array type, then the array constructor *typeid* `{{` $expr_{n1}$ **`of`** $expr_{v1}$`,` $expr_{n2}$ **`of`** $expr_{v2}$`, ... }}` first evaluates each pair of expressions in left-to-right order to yield a list of pairs of integer counts *ni* and initial values *vi*, and then creates a new array instance of *typeid* whose contents consist of *n1* copies of *v1*, followed by *n2* copies of *v2*, etc. If any of the counts is 1, it may be omitted. For example, the specification `{{1, 2 of 6, 3 of 9, 5}}` yields an array of length 7 with contents 1, 6, 6, 9, 9, 9, 5.

10.8 Precedence and Associativity

Function call and parenthesization have the highest (most binding) precedence; followed by the unary operators; followed by `*`, `/`, **`mod`**, **`div`**, and **`and`**; followed by `+`, `-`, and **`or`**; followed by the relational operators.

The arithmetic binary operators are all left-associative.

11 Statements

11.1 Assignment

```
Statement          → LValue ':=' Expression ';'
```

The expression is evaluated and stored in the location specified by the L-Value.

Assigning a record or array value actually assigns a pointer to the record or array variable on the left-hand side.

11.2 Procedure Call

```
Statement          → ID Arguments ';'
Arguments          → '(' 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 until a **`return`** statement (with no expression) is executed.

11.3 Read

```
Statement          → read '(' LValue {',' LValue} ')' ';'
```

Executing this statement reads numeric literals from standard input, evaluates them, and assigns the resulting values into the locations specified by the given L-Values. 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 WriteArgs ';'
WriteArgs          → '(' WriteExpr {',' WriteExpr} ')'
                   → '(' ')'
WriteExpr          → STRING
                   → Expression
```

Executing this statement writes the values of the specified expressions (which must be simple integers, reals, Booleans, or string literals) to standard output, followed by a newline character.

11.5 If-Then-Else

```
Statement        → if Expression then {Statement}
                       { elseif 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 sequence, 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 LValue ':=' Expression to Expression
                       [ by Expression ]
                       do {Statement} end ';'
```

Executing the statement **`for`** *index* **`:=`** $expr_1$ **`to`** $expr_2$ **`by`** $expr_3$ **`do`** *stmts* **`end;`** is equivalent to the following steps: (i) Evaluate expressions $expr_1$, $expr_2$, and $expr_3$ in that order to values v_1, v_2, v_3 (which must be integers); (ii) Assign v_1 to *index*; (iii) If the value of *index* is less than or equal to v_2, execute *stmts*; otherwise terminate the loop; (iv) Set *index* := *index* + v_3 and repeat from step (iii).

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

`LValue` is the index variable and must have type integer. The index variable can be inspected or set above, within, or below the loop body. The `LValue` expression is evaluated only once to determine the location of the index variable, before $expr_1$, $expr_2$, and $expr_3$ are evaluated. Note that any changes to the index variable from within the loop body may affect the number of iterations.

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 may be multiple **return**s within a single procedure body. The last executable statement in a procedure must be a **return**; execution may not "fall out the bottom" of a procedure. A **return** within a function procedure must include an expression of the correct return type; a **return** statement within a proper procedure must not include an expression. The main program body must not include any **return** statements.

12 The Syntax of PCAT

```
Program         → program is Body ';'
Body            → {Declaration} begin {Statement} end
Declaration     → var VarDecl {VarDecl}
                → type TypeDecl {TypeDecl}
                → procedure ProcedureDecl {ProcedureDecl}
VarDecl         → ID { ',' ID} [ ':' TypeName ] ':=' Expression ';'
TypeDecl        → TypeName is CompoundType ';'
TypeName        → ID
CompoundType    → array of TypeName
                → record FieldDecl {FieldDecl} end
ProcedureDecl   → ID FormalParams [':' TypeName] is Body ';'
FieldDecl       → ID ':' TypeName ';'
FormalParams    → '(' FormalSection {';' FormalSection} ')'
                → '(' ')'
FormalSection   → ID {',' ID} ':' TypeName
Statement       → LValue ':=' Expression ';'
                → ID Arguments ';'
                → read '(' LValue {',' LValue} ')' ';'
                → write WriteArgs ';'
                → if Expression then {Statement}
                   {elseif Expression then {Statement}}
                   [else {Statement}] end ';'
                → while Expression do {Statement} end ';'
                → loop {Statement} end ';'
                → for LValue ':=' Expression to Expression [ by Expression ]
                   do {Statement} end ';'
                → exit ';'
                → return [Expression] ';'
WriteArgs       → '(' WriteExpr {',' WriteExpr} ')'
                → '(' ')'
WriteExpr       → STRING
                → Expression
Expression      → Number
                → LValue
                → '(' Expression ')'
                → UnaryOp Expression
                → Expression BinaryOp Expression
                → ID Arguments
                → ID FieldInits
                → ID ArrayValues
LValue          → ID
                → LValue '[' Expression ']'
                → LValue '.' ID
Arguments       → '(' Expression {',' Expression} ')'
                → '(' ')'
FieldInits      → '{' ID ':=' Expression { ';' ID ':=' Expression} '}'
ArrayValues     → '{' '{' ArrayValue { ',' ArrayValue} '}' '}'
ArrayValue      → [ Expression of ] Expression
Number          → INTEGER  |  REAL
UnaryOp         → '+' | '-' | not
BinaryOp        → '+' | '-' | '*' | '/' | div | mod | or | and
                → '>' | '<' | '=' | '>=' | '<=' |  '<>'
```