PascalJunior (PJ): A Simple Language for Practice Implementation

1 Introduction

This document describes the PascalJunior (PJ) programming language. PJ is a simple programming language designed for practice implementation. PJ is a simplified version of Pascal in which one may perform simple integer and floating point calculations, and simple function calls.

PJ supports three basic data types: *boolean*, *long integers*, and *floating point*. Each of these types may be aggregated into one-dimensional or two-dimensional arrays. A number of operators are defined for each type. You can assume that the underlying hardware supports long integers with 32 bit twos complement arithmetic and floating point with a 32 bit implementation of the IEEE floating point standard. Boolean is implemented using 32-bit integers with two possible values: 0 for false and 1 for true.

Control structures in PJ are limited. It has an *if* statement, a *while* statement and a *compound* statement. PJ only supports one level of nested functions which contain no arguments.

The language is intended to be *strongly typed*; that is, the type of each expression should be determinable at compile time. However, some *coercions* from one type to another will be permitted.

2 Lexical Properties of PJ

1. In PJ, blanks are significant.
2. In PJ, keywords always consist of capital letters. All keywords are reserved; that is, the programmer cannot use a PJ keyword as the name of a variable. The valid keywords are: `AND`, `ARRAY`, `BEGIN`, `DO`, `ELSE`, `END`, `EXIT`, `FALSE`, `FLOAT`, `IF`, `LONGINT`, `DIV`, `NOT`, `OR`, `PROGRAM`, `FUNCTION`, `READ`, `THEN`, `TRUE`, `VAR`, `WHILE`, `WRITE`, `WRITELN`. (Note that PJ is *case sensitive,* that is, the variable `X` differs from `x`. Thus, `END` is a keyword, but `end` can be a variable name. Pascal is indeed *case insensitive*.)
3. The following special characters have meanings in a PJ program. (See the grammar and notes for details.)
   `{ } ' < > = + - * [ ] ( ) . , : ;`
4. Comments are delimited by the characters { and }. A { begins a comment; it is valid in no other context. A } ends a comment; it cannot appear inside a comment. (This means comments may not be nested. { can appear in a comment; the first } closes the comment.) Comments may appear before or after any other token.
5. Identifiers are written with upper and lowercase letters and are defined as follows:
   $\langle Letter \rangle \rightarrow$ `a` | `b` | `c` | $\cdots$ | `z` | `A` | `B` | $\cdots$ | `Z`
   $\langle Digit \rangle \rightarrow$ `0` | `1` | `2` | $\cdots$ | `9`
   $\langle Identifier \rangle \rightarrow \langle Letter \rangle\ (\langle Letter \rangle \mid \langle Digit \rangle)^*$
   The implementor may restrict the length of identifiers so long as identifiers of at least 31 characters are legal.
6. Constants are defined as follows:
   $\langle Positive \rangle \rightarrow$ `1` | `2` | `3` | ... | `9`
   $\langle Sign \rangle \rightarrow$ `+` | `-` | ϵ
   $\langle Intnum \rangle \rightarrow \langle Positive \rangle\ \langle Digit \rangle^*$ | `0`
   $\langle Negnum \rangle \rightarrow -\langle Intnum \rangle$
   $\langle Floatnum \rangle \rightarrow \langle Intnum \rangle$ `.`
   | $\langle Intnum \rangle$ `.` $\langle Intnum \rangle$
   | $\langle Intnum \rangle$ `.` `E` $\langle Sign \rangle\ \langle Intnum \rangle$
   | $\langle Intnum \rangle$ `.` $\langle Intnum \rangle$ `E` $\langle Sign \rangle\ \langle Intnum \rangle$
   Special string constants are acceptable in WRITE and WRITELN statements. A PJ string is a sequence of non-single-quote characters enclosed by a pair of single quotes.
   TRUE and FALSE are special boolean constants.

3 PJ Syntax

This section gives a syntactical description of PJ. The sections following the grammar provide implementation notes on the various parts of the grammar.

3.1 BNF

The following grammar describes the context-free syntax of PJ:

⟨*Program*⟩ → `PROGRAM` ⟨*Identifier*⟩ `;`
⟨*Decls*⟩
⟨*SubprogramDecls*⟩
⟨*CompoundStatement*⟩ `.`

⟨*Decls*⟩ → `VAR` ⟨*DeclList*⟩
| ϵ

⟨*DeclList*⟩ → ⟨*IdentifierList*⟩ `:` ⟨*Type*⟩ `;`
| ⟨*DeclList*⟩ ⟨*IdentifierList*⟩ `:` ⟨*Type*⟩ `;`

⟨*IdentifierList*⟩ → ⟨*Identifier*⟩
| ⟨*IdentifierList*⟩ `,` ⟨*Identifier*⟩

⟨*Type*⟩ → ⟨*StandardType*⟩
| ⟨*ArrayType*⟩

⟨*StandardType*⟩ → `LONGINT`
| `FLOAT`
| `BOOLEAN`

⟨*ArrayType*⟩ → `ARRAY [` ⟨*Dim*⟩ `] OF` ⟨*StandardType*⟩
| `ARRAY [` ⟨*Dim*⟩ `,` ⟨*Dim*⟩ `] OF` ⟨*StandardType*⟩

⟨*Dim*⟩ → ⟨*Intnum*⟩ `..` ⟨*Intnum*⟩

⟨*SubProgramDecls*⟩ → ⟨ *SubProgramDecls* ⟩ ⟨*SubProgramDecl*⟩
| ϵ

⟨*SubprogramDecl*⟩ → ⟨*SubprogramHead*⟩ ⟨*Decls*⟩⟨*CompoundStatement*⟩

⟨*SubprogramHead*⟩ → `FUNCTION` ⟨*Identifier*⟩`:`⟨*StandardType*⟩

⟨*Statement*⟩ → ⟨*Assignment*⟩
| ⟨*IfStatement*⟩
| ⟨*WhileStatement*⟩
| ⟨*IOStatement*⟩
| ⟨*CompoundStatement*⟩

⟨*Assignment*⟩ → ⟨*Variable*⟩ `:=` ⟨*Expr*⟩

⟨*IfStatement*⟩ → `IF` ⟨*Expr*⟩
`THEN` ⟨*Statement*⟩
`ELSE` ⟨*Statement*⟩
| `IF` ⟨*Expr*⟩ `THEN` ⟨*Statement*⟩

WhileStatement → `WHILE` ⟨*Expr*⟩ `DO` ⟨*Statement*⟩

⟨*IOStatement*⟩ → `READ (` ⟨*Variable*⟩ `)`
| ⟨*WriteToken*⟩ `(` ⟨*Expr*⟩ ⟨*Format*⟩ `)`
| ⟨*WriteToken*⟩ `(` ⟨*StringConstant*⟩ ⟨*Format*⟩ `)`

⟨*WriteToken*⟩ → `WRITE`
| `WRITELN`

⟨*Format*⟩ → : ⟨*IntNum*⟩
| ϵ

⟨*CompoundStatement*⟩ → `BEGIN` ⟨*StatementList*⟩ `END`

⟨*StatementList*⟩ → ⟨*Statement*⟩
| ⟨*StatementList*⟩ `;` ⟨*Statement*⟩

⟨*Expr*⟩ → ⟨*Expr*⟩ ⟨*RelOp*⟩ ⟨*AddExpr*⟩
| ⟨*AddExpr*⟩

⟨*Relop*⟩ → `<` | `<=` | `>=` | `>` | `=` | `<>`

⟨*AddExpr*⟩ → ⟨*AddExpr*⟩ ⟨*AddOp*⟩ ⟨*MulExpr*⟩
| ⟨*MulExpr*⟩

⟨*Addop*⟩ → `+` | `-` | `OR`

⟨*MulExpr*⟩ → ⟨*MulExpr*⟩ ⟨*MulOp*⟩ ⟨*Factor*⟩
| ⟨*Factor*⟩

⟨*Mulop*⟩ → `*` | `DIV` | `AND`

⟨*Factor*⟩ → ⟨*Variable*⟩
| ⟨*Constant*⟩
| `NOT` ⟨*Factor*⟩
| ⟨*Identifier*⟩ `()`
| `(` ⟨*Expr*⟩ `)`

⟨*Variable*⟩ → ⟨*Identifier*⟩
| ⟨*Identifier*⟩ `[` ⟨*Expr*⟩ `]`
| ⟨*Identifier*⟩ `[` ⟨*Expr*⟩, ⟨*Expr*⟩ `]`

⟨*Constant*⟩ → ⟨*Intnum*⟩ | `-` ⟨*Intnum*⟩ | ⟨*Floatnum*⟩ | `TRUE` | `FALSE`

3.2 Section Notes

3.2.1 Declarations

PJ has three standard types: `BOOLEAN`, `LONGINT` and `FLOAT`. Booleans, integers and floats occupy in a single X86-64 machine "double word"/"long word" which consists of four bytes. (In a product compiler, a boolean usually occupies one byte.) These standard types may be composed into the structured `ARRAY` type. An identifier may represent one of six types of objects:

1. a boolean variable or array
2. an integer variable or array
3. a floating point variable or array

Identifiers are declared to be variables or arrays by a `VAR` declaration. For a one-dimensional and two-dimensional array, arbitrary upper and lower index bounds are permitted.

Example:

```
VAR x,y :  LONGINT;
    f1, f2, f3 :  FLOAT;
    a :  ARRAY [ 1 ..  15 ] OF LONGINT;
    s1, s2 :  ARRAY [0 ..  79 ] OF FLOAT;
```

3.2.2 Assignment Statement

The assignment statement requires that the *left hand side* (the ⟨*Variable*⟩ non-terminal) and *right hand side* (the ⟨*Expr*⟩ non-terminal) evaluate to have the same type. If they have different types, either coercion is required or a context-sensitive error has occurred. The coercion rules for assignment are simple. If both sides are numeric (of type `LONGINT` or `FLOAT`), the right hand side is converted to the type of the left hand side.

3.2.3 If Statement

The grammar for the `IF-THEN-ELSE` construct embodies one of the classical solutions to the dangling else ambiguity. It provides a unique binding of the *else-part* to a corresponding *if* and *then-part*.

To evaluate an if statement, the expression is evaluated. The expression must be of the boolean type.

Examples:

```
IF c=d THEN d := a
IF b=0 THEN b := 2*a ELSE b := b/2
```

3.2.4 While Statement

The while statement provides a simple mechanism for iteration. PJ's while statement behaves like the while statement in many other languages; it executes the statement in the loop's body until the controlling expression becomes false.

The controlling expression will be treated as a boolean value encoded as an `LONGINT`.

3.2.5 Expressions

PJ expressions compute simple values of type `BOOLEAN`, `LONGINT` or `FLOAT`. For both integer and floating point numbers, addition, multiplication, division, and comparison are defined. Boolean values can apply "and" and "or'.

Coercion: If an expression contains operands of only one type, evaluation is straight forward. When an operand contains mixed types, the situation is more complex. If an *Addop* or *Mulop* has an `LONGINT` operand and a `FLOAT` operand, the `LONGINT` operand should be converted to a `FLOAT` before the operation is performed.

Relational operators always produce a boolean value of false or true. Comparisons between integers and floats produce boolean results . To perform the comparison, the integer is converted to a float. For the numbers, comparison is based on both sign and magnitude.

Note: in an assignment, the value of a numeric expression gets converted to match the type of the variable that appears on its left hand side.

4 An Example Program

The following program represents a simple example program written in PJ. This program successively reads pairs of integers from the input file and prints out their greatest common divisor.

```
PROGRAM example;
  VAR x, y : LONGINT;
  FUNCTION gcd :LONGINT;
      VAR t : LONGINT;
      BEGIN
         IF y=0
          THEN EXIT(x)
          ELSE BEGIN
             t := x;
             x := y;
             y := t - y * (t DIV y);
             EXIT(gcd())
          END
      END;
  BEGIN
    READ (x);
    READ (y);
    WHILE (x <> 0) OR (y <> 0) DO
    BEGIN
       WRITELN (gcd());
       READ (x);
       READ (y)
    END
  END.
```