# The Programming Language SuperPascal

PER BRINCH HANSEN [1]

*School of Computer and Information Science*
*Syracuse University, Syracuse, NY 13244, USA*

November 1993

**Abstract:** This paper defines *SuperPascal*—a secure programming language for publication of parallel scientific algorithms. *SuperPascal* extends a subset of IEEE Standard Pascal with deterministic statements for parallel processes and synchronous message communication. A parallel statement denotes parallel execution of a fixed number of statements. A *forall* statement denotes parallel execution of the same statement by a dynamic number of processes. Recursive procedures may be combined with parallel and *forall* statements to define recursive parallel processes. Parallel processes communicate by sending typed messages through channels created dynamically. *SuperPascal* omits ambiguous and insecure features of Pascal. Restrictions on the use of variables enable a single-pass compiler to check that parallel processes are disjoint, even if the processes use procedures with global variables.

**Key Words:** Programming languages, Parallel programming, Recursive parallelism, Synchronous communication, SuperPascal.

## 1 Introduction

This paper defines *SuperPascal*—a secure programming language for publication of parallel scientific algorithms. *SuperPascal* extends a subset of IEEE Standard Pascal with deterministic statements for parallel processes and synchronous message communication. A parallel statement denotes parallel execution of a fixed number of statements. A *forall* statement denotes parallel execution of the same statement by a dynamic number of processes. Recursive procedures may be combined with parallel and *forall* statements to define recursive parallel processes. Parallel processes communicate by sending typed message through channels created dynamically. *SuperPascal* omits ambiguous and insecure features of Pascal. Restrictions on the use of variables enable a single-pass compiler to check that parallel processes are disjoint, even if the processes use procedures with global variables.

This paper defines the parallel features of *SuperPascal* using the terminology and syntax notation of the Standard Pascal report [IEEE 1983]. Brinch Hansen [1993a] illustrates *SuperPascal* by examples. The syntactic checking of parallel statements is discussed further in [Brinch Hansen 1993b].

A *portable implementation* of *SuperPascal* has been developed on a Sun workstation under Unix. It consists of a compiler and an interpreter written in Pascal.

To obtain the *SuperPascal* software, use anonymous FTP from the directory *pbh* at *top.cis.syr.edu*.

## 2 Processes and Variables

command =
    variable-access | expression | statement | statement-sequence .

The evaluation or execution of a *command* is called a *process*. A structured process is a sequential or parallel composition of processes. The components of a parallel composition are called *parallel processes*. They proceed independently at unpredictable speeds until all of them have terminated.

In a program text an *entire variable* is a syntactic entity that has an identifier, a type, and a scope.

During program execution a *block* is activated when a process evaluates a function designator or executes a procedure statement or program. Every activation of a block $B$ creates a new instance of every variable that is local to $B$. When an activation terminates, the corresponding variable instances cease to exist.

During recursive and parallel activations of a block, multiple instances of the local variables exist. Each variable instance is a dynamic entity that has a location, a current value, and a finite lifetime in memory.

The distinction between a *variable* as a syntactic entity in the program text and a class of dynamic entities in memory is usually clear from the context. Where it is necessary, this paper distinguishes between *syntactic variables* and *variable instances*.

Parallel processes are said to be *disjoint* if they satisfy the following condition: Any variable instance that is assigned a value by one of the processes is not accessed by any of the other processes. In other words, any variable instance that is accessed by more than one process is not assigned a value by any of the processes.

## 3 Type Definitions

Every type has an identifier. Two types are the same if they have the same identifier and the same scope.

*Examples:*

The following types are used in the examples of this paper:

```
type
  vector = record x, y: real end;
  body = record m: real; r, v, f: vector end;
  system = array [1..n] of body;
  channel = *(body);
  net = array [0..p] of channel;
  mixed = *(body, integer);
  two = array [0..1] of mixed;
  four = array [0..1] of two;
```

### 3.1 Channel Types

Processes communicate by means of values called *messages* transmitted through entities called *channels*. A *communication* takes place when one process is ready to *output* a message of some type through a channel and another process is ready to *input* a message of the same type through the same channel.

Processes create channels dynamically and access them by means of values known as *channel references*. The type of a channel reference is called a *channel type*.

channel-type =
  "*" "(" message-type-list ")" .
message-type-list =
  type-identifier { "," type-identifier } .

A channel type

$$*(T_1, T_2, \ldots, T_n)$$

denotes an unordered set of channel references created dynamically. Each channel reference denotes a distinct channel which can transmit messages of distinct types $T_1, T_2, \ldots, T_n$ only (the *message types*).

A type definition cannot be of the recursive form:

$$T = *(\ldots, T, \ldots)$$

*Examples:*

```
*(body)
*(body, integer)
```

# 4 Variables

## 4.1 Entire Variables

An *entire variable* is a variable denoted by one of the following kinds of identifiers:

1. A variable identifier introduced by a variable declaration or a *forall* statement.
2. A function identifier that occurs as the left part of an assignment statement in the statement part of the corresponding function block.

*Examples:*

The following entire variables are used in the examples of this paper:

```
var
  inp, out: channel;
  c: net;
  a: system;
  ai, aj: body;
  left: mixed;
  top: four;
  i, j, k: integer;
```

A *variable context* is associated with each command $C$. This context consists of two sets of entire variables called the *target* and *expression variables* of $C$. If the process denoted by $C$ may assign a value to an entire variable $v$ (or one of its components), then $v$ is a target variable of $C$. If the process may use the value of $v$ (or one of its components) as an operand, then $v$ is an expression variable of $C$.

## 4.2 Block Parameters

Consider a procedure or function block $B$ with a statement part $S$. An *implicit parameter* of $B$ is an entire variable $v$ that is global to $B$ and is part of the variable context of $S$. If $v$ is a target variable of $S$, then $v$ is an *implicit variable parameter* of $B$. If $v$ is an expression variable of $S$, then $v$ is an *implicit value parameter* of $B$.

A *function* block cannot use formal variable parameters or implicit variable parameters.

A *recursive procedure* or *function* block cannot use implicit parameters.

## 4.3 Target Variables

An entire variable $v$ is a target variable of a command $C$ in the following cases:

1. The variable identifier $v$ occurs in an assignment statement $C$ that denotes assignment to $v$ (or one of its components).

2. The variable identifier $v$ occurs in a *for* statement $C$ that uses $v$ as the control variable.

3. The variable identifier $v$ occurs in a procedure statement $C$ that uses $v$ (or one of its components) as an actual variable parameter.

4. The variable $v$ is an implicit variable parameter of a procedure block $B$, and $C$ is a procedure statement that denotes activation of $B$.

5. The variable $v$ is a target variable of a command $D$, and $C$ is a structured command that contains $D$.

### 4.4 Expression Variables

An entire variable $v$ is an expression variable of a command $C$ in the following cases:

1. The variable identifier $v$ occurs in an expression $C$ that uses $v$ (or one of its components) as an operand.

2. The variable identifier $v$ occurs in the element statement $C$ of a *forall* statement that introduces $v$ as the index variable.

3. The variable $v$ is an implicit value parameter of a function block $B$, and $C$ is a function designator that denotes activation of $B$.

4. The variable $v$ is an implicit value parameter of a procedure block $B$, and $C$ is a procedure statement that denotes activation of $B$.

5. The variable $v$ is an expression variable of a command $D$, and $C$ is a structured command that contains $D$.

### 4.5 Channel Variables

A *channel variable* is a variable of a channel type. The value of a channel variable is undefined unless a channel reference has been assigned to the variable.

channel-variable-access =
    variable-access .

*Examples:*

```
inp
c[0]
top[i,j]
```

# 5 Expressions

## 5.1 Channel Expressions

channel-expression =
    expression .

A channel expression is an expression of a channel type. The expression is said to be *well-defined* if it denotes a channel; otherwise, it is *undefined*.

*Examples:*

```
out
c[k−1]
```

## 5.2 Relational Operators

If $x$ and $y$ are well-defined channel expressions of the same type, the following expressions denote boolean values:

```
x = y      x <> y
```

The value of x = y is true if $x$ and $y$ denote the same channel, and is false otherwise. The value of x <> y is the same as the value of

**not** (x = y)

*Example:*

```
left = top[i,j]
```

# 6 Message Communication

The required procedures for message communication are

open  send  receive

## 6.1 The Procedure Open

open-statement =
    "*open*" "(" open-parameters ")" .
open-parameters =
    open-parameter { "," open-parameter } .
open-parameter =

channel-variable-access .

If $v$ is a channel variable, the statement

open(v)

denotes creation of a new channel.

The *open* statement is executed by creating a new channel and assigning the corresponding reference to the channel variable $v$. The channel reference is of the same type as the channel variable. The channel exists until the program execution terminates.

The abbreviation

$$\text{open}(v_1, v_2, \ldots, v_n)$$

is equivalent to

**begin** open($v_1$); open($v_2, \ldots, v_n$) **end**

*Examples:*

```
open(c[k])
open(inp, out)
```

## 6.2 The Procedures Send and Receive

send-statement =
    "*send*" "(" send-parameters ")" .
send-parameters =
    channel-expression "," output-expression-list .
output-expression-list =
    output-expression { "," output-expression } .
output-expression =
    expression .
receive-statement =
    "*receive*" "(" receive-parameters ")" .
receive-parameters =
    channel-expression "," input-variable-list .
input-variable-list =
    input-variable-access { "," input-variable-access } .
input-variable-access =
    variable-access .

The statement

send(b, e)

denotes output of the value of an expression $e$ through the channel denoted by an expression $b$. The expression $b$ must be of a channel type $T$, and the type of the expression $e$ must be a message type of $T$.

The statement

receive(c, v)

denotes input of the value of a variable $v$ through the channel denoted by an expression $c$. The expression $c$ must be of a channel type $T$, and the type of the variable $v$ must be a message type of $T$.

The *send* and *receive* operations defined by the above statements are said to *match* if they satisfy the following conditions:

1. The channel expressions $b$ and $c$ are of the same type $T$ and denote the same channel.

2. The output expression $e$ and the input variable $v$ are of the same type, which is a message type of $T$.

The execution of a *send* operation delays a process until another process is ready to execute a matching *receive* operation (and vice versa). If and when this happens, a *communication* takes place as follows:

1. The sending process obtains a value by evaluating the output expression $e$.

2. The receiving process assigns the value to the input variable $v$.

After the communication, the sending and receiving processes proceed independently.

*Communication Errors:*

1. *Undefined channel reference*: A channel expression does not denote a channel.

2. *Channel contention*: Two parallel processes both attempt to send (or receive) through the same channel at the same time.

3. *Message type error*: Two parallel processes attempt to communicate through the same channel, but the output expression and the input variable are of different message types.

The abbreviation

$$\text{send}(b, e_1, e_2, \ldots, e_n)$$

is equivalent to

$$\textbf{begin}\ \text{send}(b, e_1);\ \text{send}(b, e_2, \ldots, e_n)\ \textbf{end}$$

The abbreviation

$$\text{receive}(c, v_1, v_2, \ldots, v_n)$$

is equivalent to

$$\textbf{begin}\ \text{receive}(c, v_1);\ \text{receive}(c, v_2, \ldots, v_n)\ \textbf{end}$$

*Examples:*

```
send(out, ai)
receive(inp, aj)
send(top[i,j], 2, ai)
```

# 7 Statements

## 7.1 Assignment Statements

If $x$ is a channel variable access and $y$ is a well-defined channel expression of the same type, the effect of the assignment statement

```
x := y
```

is to make the values of $x$ and $y$ denote the same channel.

*Example:*

```
left := top[i,j]
```

## 7.2 Procedure Statements

The *restricted actual parameters* of a procedure statement are the explicit variable parameters that occur in the actual parameter list and the implicit parameters of the corresponding procedure block.

*Restriction*: The restricted actual parameters of a procedure statement must be distinct entire variables (or components of such variables).

A procedure statement cannot occur in the statement part of a function block. This rule also applies to a procedure statement that denotes activation of a required procedure.

## 7.3 Parallel Statements

parallel-statement =
    "**parallel**" process-statement-list "**end**" .
process-statement-list =
    process-statement { "|" process-statement } .
process-statement =
    statement-sequence .

A *parallel* statement denotes parallel processes. Each process is denoted by a separate process statement.

The effect of a parallel statement is to execute the process statements as parallel processes until all of them have terminated.

*Restriction*: In a parallel statement, a target variable of one process statement cannot be a target or expression variable of another process statement.

*Example:*

```
parallel
  source(a, c[0]); sink(a, c[p])|
  forall k := 1 to p do
    node(k, c[k−1], c[k])
end
```

## 7.4 Forall Statements

forall-statement =
    "**forall**" index-variable-declaration "**do**"
        element-statement .
index-variable-declaration =
    variable-identifier ":=" process-index-range .
process-index-range =
    expression "**to**" expression .
element-statement =
    statement .

The statement

**forall** $i := e_1$ **to** $e_2$ **do** $S$

denotes a (possibly empty) array of parallel processes, called *element processes*, and a corresponding range of values, called *process indices*. The lower and upper bounds of the process index range are denoted by two expressions, $e_1$ and $e_2$, of the same

simple type (the *index type*). Every index value corresponds to a separate element process defined by an *index variable* $i$ and an *element statement* $S$.

The *index variable declaration*

$$i := e_1 \textbf{ to } e_2$$

introduces the index variable $i$ which is local to the element statement $S$.

A *forall* statement is executed as follows:

1. The expressions $e_1$ and $e_2$ are evaluated. If $e_1 > e_2$, the execution of the *forall* statement terminates; otherwise, step 2 takes place.

2. $e_2 - e_1 + 1$ element processes run in parallel until all of them have terminated. Each element process creates a local instance of the index variable $i$, assigns the corresponding process index to the variable, and executes the element statement $S$. When an element process terminates, its local instance of the index variable ceases to exist.

*Restriction*: In a *forall* statement, the element statement cannot use target variables.

*Examples:*

```
forall k := 1 to p do
  node(k, c[k−1], c[k])

forall i := 0 to 1 do
  forall j := 0 to 1 do
    quadtree(i, j, top[i,j])
```

## 7.5 Unrestricted Statements

unrestricted-statement =
    sic-clause statement .
sic-clause =
    "[" "**sic**" "]" .

A statement $S$ is said to be *unrestricted* in the following cases:

1. The statement $S$ is prefixed by a *sic* clause.

2. The statement $S$ is a component of an unrestricted statement.

All other statements are said to be *restricted*.

Restricted statements must satisfy the rules labeled as *restrictions* in this paper. These rules restrict the use of entire variables in procedure statements, parallel statements, and *forall* statements to make it possible to check the disjointness of parallel processes during single-pass compilation (see 7.2, 7.3 and 7.4).

The same rules do *not* apply to unrestricted statements. Consequently, the programmer must prove that each unrestricted statement preserves the disjointness of parallel processes; otherwise, the semantics of unrestricted statements are beyond the scope of this paper.

*Examples:*

```
[sic] { i <> j }
  swap(a[i], a[j])

[sic] { i <> j }
  parallel a[i] := ai|a[j] := aj end

[sic] { disjoint elements a[i] }
  forall i := 1 to n do a[i] := ai
```

### 7.6 Assume Statements

```
assume-statement =
    "assume" assumption .
assumption =
    expression .
```

The effect of an *assume* statement is to test an assumption denoted by a boolean expression. If the assumption is true, the test terminates; otherwise, program execution stops.

*Example:*

```
assume i <> j
```

## 8 SuperPascal versus Pascal

The following summarizes the differences between *SuperPascal* and Pascal.

### 8.1 Added Features

Table 1 lists the *SuperPascal* features that were added to Pascal.

## 8.2 Excluded Features

Table 2 lists the Pascal features that were excluded from *SuperPascal*.

## 8.3 Minor Differences

*SuperPascal* differs from Pascal in the following details:

1. *Program parameters* are comments only.

2. A multi-dimensional *array type* is defined in terms of one-dimensional array types.

3. The required type *string* is the only string type:

   $$\text{string} = \textbf{array}\ [1..\text{maxstring}]\ \textbf{of}\ \text{char}$$

   A character string with $n$ string elements denotes a string of $n$ characters followed by maxstring$-n$ *null* characters, where

   $$2 \leq n \leq \text{maxstring} \qquad \text{maxstring} = 80 \qquad \text{null} = \text{chr}(0)$$

   The default length $n$ of a write parameter of type string is the number of characters (if any) which precede the first null character (if any), where $0 \leq n \leq$ maxstring.

4. The required textfile *input* is the only input file. The file identifier is omitted from *eof* and *eoln* function designators and *read* and *readln* statements. The input file is an implicit value parameter of the eof and eoln functions and is an implicit variable parameter of the read and readln procedures (see 4.2).

5. The required textfile *output* is the only output file. The file identifier is omitted from *write* and *writeln* statements. The output file is an implicit variable parameter of the write and writeln procedures (see 4.2).

## 8.4 Required Identifiers

Table 3 lists the required identifiers of *SuperPascal*.

Table 1: Added features

| Language concepts | Required identifiers |
|---|---|
| channel types | null |
| structured function types | maxstring |
| parallel statements | string |
| forall statements | open |
| unrestricted statements | send |
| assume statements | receive |

Table 2: Excluded features

| Language concepts | Required identifiers |
|---|---|
| labels | text |
| subrange types | input |
| record variants | output |
| empty field lists | page |
| set types | reset |
| file types | get |
| pointer types | rewrite |
| packed types | put |
| nameless types | new |
| renamed types | dispose |
| functions with side-effects | pack |
| functional parameters | unpack |
| procedural parameters | |
| forward declarations | |
| goto statements | |
| with statements | |

Table 3: Required identifiers

| | | |
|---|---|---|
| abs | maxint | round |
| arctan | maxstring | send |
| boolean | null | sin |
| char | odd | sqr |
| chr | open | sqrt |
| cos | ord | string |
| eof | pred | succ |
| eoln | read | true |
| exp | readln | trunc |
| false | real | write |
| integer | receive | writeln |
| ln | | |

## 8.5 Syntax Summary

The following grammar defines the complete syntax of *SuperPascal*.

program =
    program-heading “;” program-block “.” .
program-heading =
    “**program**” program-identifier [ “(” program-parameters “)” ] .
program-parameters =
    parameter-identifier { “,” parameter-identifier } .
program-block =
    block .
block =
    [ constant-definitions ] [ type-definitions ]
        [ variable-declarations ] [ routine-declarations ]
            statement-part .
constant-definitions =
    “**const**” constant-definition “;” { constant-definition “;” } .
constant-definition =
    constant-identifier “=” constant .
constant =
    [ sign ] unsigned-constant .
sign =
    “+” | “−” .
type-definitions =
    “**type**” type-definition “;” { type-definition “;” } .
type-definition =

```
    type-identifier "=" new-type .
new-type =
    enumerated-type | array-type | record-type | channel-type .
enumerated-type =
    "(" constant-identifier-list ")" .
constant-identifier-list =
    constant-identifier { "," constant-identifier } .
array-type =
    "array" index-range "of" type-identifier .
index-range =
    "[" constant ".." constant "]" .
record-type =
    "record" field-list "end" .
field-list =
    record-section { ";" record-section } [ ";" ] .
record-section =
    field-identifier-list ":" type-identifier .
field-identifier-list =
    field-identifier { "," field-identifier } .
channel-type =
    "*" "(" message-type-list ")" .
message-type-list =
    type-identifier { "," type-identifier } .
variable-declarations =
    "var" variable-declaration ";" { variable-declaration ";" } .
variable-declaration =
    variable-identifier-list ":" type-identifier .
variable-identifier-list =
    variable-identifier { "," variable-identifier } .
routine-declarations =
    routine-declaration ";" { routine-declaration ";" } .
routine-declaration =
    function-declaration | procedure-declaration .
function-declaration =
    function-heading ";" function-block .
function-heading =
    "function" function-identifier [ formal-parameter-list ]
        ":" type-identifier .
formal-parameter-list =
    "(" formal-parameters ")" .
formal-parameters =
    formal-parameter-section { ";" formal-parameter-section } .
```

formal-parameter-section =
    [ "**var**" ] variable-declaration .
function-block =
    block .
procedure-declaration =
    procedure-heading ";" procedure-block .
procedure-heading =
    "**procedure**" procedure-identifier [ formal-parameter-list ] .
procedure-block =
    block .
statement-part =
    compound-statement .
compound-statement =
    "**begin**" statement-sequence "**end**" .
statement-sequence =
    statement { ";" statement } .
statement =
    empty-statement | assignment-statement |
    procedure-statement | if-statement |
    while-statement | repeat-statement |
    for-statement | case-statement |
    compound-statement | parallel-statement |
    forall-statement | unrestricted-statement |
    assume-statement .
empty-statement = .
assignment-statement =
    left-part ":=" expression .
left-part =
    variable-access | function-identifier .
procedure-statement =
    procedure-identifier [ actual-parameter-list ] .
actual-parameter-list =
    "(" actual-parameters ")" .
actual-parameters =
    actual-parameter { "," actual-parameter } .
actual-parameter =
    expression | variable-access | write-parameter .
write-parameter =
    expression [ ":" expression [ ":" expression ] ] .
if-statement =
    "**if**" expression "**then**" statement
        [ "**else**" statement ] .

```
while-statement =
    "while" expression "do" statement .
repeat-statement =
    "repeat" statement-sequence "until" expression .
for-statement =
    "for" control-variable ":=" expression
        ( "to" | "downto" ) expression "do" statement .
control-variable =
    entire-variable .
case-statement =
    "case" expression "of" case-list "end" .
case-list =
    case-list-element { ";" case-list-element } [ ";" ] .
case-list-element =
    case-constant { "," case-constant } ":" statement .
case-constant =
    constant .
parallel-statement =
    "parallel" process-statement-list "end" .
process-statement-list =
    process-statement { "|" process-statement } .
process-statement =
    statement-sequence .
forall-statement =
    "forall" index-variable-declaration "do"
        element-statement .
index-variable-declaration =
    variable-identifier ":=" process-index-range .
process-index-range =
    expression "to" expression .
element-statement =
    statement .
unrestricted-statement =
    "[" "sic" "]" statement .
assume-statement =
    "assume" expression .
expression =
    simple-expression
        [ relational-operator simple-expression ] .
relational-operator =
    "<" | "=" | ">" | "<=" | "<>" | ">=" .
simple-expression =
```

[ sign ] term { adding-operator term } .
adding-operator =
“+” | “−” | “**or**” .
term =
factor { multiplying-operator factor } .
multiplying-operator =
“∗” | “/” | “**div**” | “**mod**” | “**and**” .
factor =
function-designator | variable-access |
unsigned-constant | “(” expression “)” |
“**not**” factor .
function-designator =
function-identifier [ actual-parameter-list ] .
variable-access =
entire-variable { component-selector } .
entire-variable =
variable-identifier .
component-selector =
field-selector | indexed-selector .
field-selector =
“.” field-identifier .
indexed-selector =
“[” index-expressions “]” .
index-expressions =
expression { “,” expression } .
unsigned-constant =
character-string | unsigned-real |
unsigned-integer | constant-identifier .
character-string =
“’” string-elements “’” .
string-elements =
string-element { string-element } .
string-element =
string-character | apostrophe-image .
apostrophe-image =
“” ” .
unsigned-real =
unsigned-integer real-option .
real-option =
“.” fractional-part [ scaling-part ] | scaling-part .
fractional-part =
digit-sequence .

```
scaling-part =
    "e" scale-factor .
scale-factor =
    [ sign ] unsigned-integer .
unsigned-integer =
    digit-sequence .
digit-sequence =
    digit { digit } .
identifier =
    letter { letter | digit } .
```

## Acknowledgements

I thank Jonathan Greenfield and Peter O'Hearn for their helpful comments.

## References

[1] IEEE (1983). *IEEE Standard Pascal Computer Programming Language*. Institute of Electrical and Electronics Engineers, New York, NY.

[2] Brinch Hansen, P. (1993a) SuperPascal—a publication language for parallel scientific computing. School of Computer and Information Science, Syracuse University, Syracuse, NY.

[3] Brinch Hansen, P. (1993b) Interference control in SuperPascal—a block-structured parallel language. School of Computer and Information Science, Syracuse Syracuse University, Syracuse, NY.