University of Glasgow
Department of Computing Science
Lilybank Gardens
Glasgow G12 8QQ

University of St Andrews
Department of Computational Science
North Haugh
St Andrews KY16 9SS

Lgen, Pgen and Sgen - Language Development Tools for a Persistent Environment

S. M. Blott and J. Campin

Persistent Programming
Research Report 52
November 1987

Alan Dearle

Lgen, Pgen and Sgen

Language Development Tools for a Persistent Programming Environment

Stephen Michael Blott
Jack Campin

Department of Computing Science
17 Lilybank Gardens
University of Glasgow
Glasgow G12 8QQ

Contents

0: Introduction

This paper describes three high level programming tools which have been developed to aid in general language manipulation. All three have been implemented in PS-algol to be compatible with each other and make extensive use of PS-algol's programming facilities. The tools are

- **Lgen**, a lexical analyser generator.
- **Pgen**, a parser generator.
- **Sgen**, a structure editor generator.

Lexical analysis is the process of taking a stream of characters and breaking it into semantically meaningful units or tokens. Parsing takes these tokens and assembles them into a structure that represents the meaning of the entire string. A structure editor allows a user to edit the structure represented by the string and therefore it is impossible (in theory) to generate a meaningless string.

Lgen and Pgen are extremely general in their potential use. They can be used for applications varying in scale from a single key response to an interactive query to the analysis of a script in a high level programming language. Sgen has a more specialised role. The current implementation is directed very much toward texts of programming languages and other applications of that scale.

0.1: Notation and typographical conventions

The bulk of the text of this report is in the Palatino font (this one). PS-algol source code, as well as isolated keywords, identifiers and string literals, are in the `Courier` font; keywords are distinguished by being in **`boldface`**, identifiers in *`italics`*, and string literals in `plain text`. (Bold face is also used for emphasis

occasionally). Metasyntactic expressions are in *Zapf Chancery* and Unix commands or filenames, including the names of PS-algol source files, are in Helvetica.

0.2: Historical Notes

These tools were all implemented by Stephen Blott in twelve weeks in the summer of 1987, as a student vacation project. The specifications for the tools were derived from the Unix tools lex and yacc and from the Cornell Synthesizer Generator. The project was supervised by Malcolm Atkinson and Jack Campin.

1: A User's Guide to Sgen Editors

An Sgen editor allows the editing of a script of a language that will always maintain a meaningful string of the language. All editing actions are in terms of the abstract syntax tree (AST) of the script. The abstract syntax of a language is a syntax that reflects the semantics of the language while ignoring the concrete syntax required for parsing. In order to use any editor generated by Sgen it is essential that the abstract syntax of the language be understood.

1.1: The Editing Paradigm

During an editing session the screen is split into two parts. The top four lines are used for commands and messages while the lower window is for displaying the abstract syntax tree. There are several editing commands, all of which are described below, but first I will describe the general editing paradigm.

1.1.1: Selecting Nodes of the Tree

There is always a current selected node. This is indicated by the inverted video section of the screen. Nodes of the AST may be selected by the mouse. Click the left-hand mouse button over a section of the script and the lowest enclosing node of the tree will be selected as the new current node.

1.1.2: Selecting Commands

There are two ways of selecting each command. They can be selected using a menu which pops up whenever the right-hand mouse button is clicked. The required command can then be selected by the left-hand mouse button. All

commands are also available from the keyboard by a single letter command selection mechanism.

1.1.3: Entering Strings

Whenever the user is required to enter a string the system will produce a string editor in the command area of the screen. If it is appropriate the editor will already contain a string to edit. The string editing operations that are available are

- entering and deleting characters at the current cursor position
- deleting all the characters to the left of the cursor with the "oops" key
- positioning the cursor with the left-hand mouse button.

To return the current string type <return>. One point to be careful of is that the string editor returns only those characters to the left of the cursor at the end of the string edit. Therefore, before ending the edit, the cursor should always be at the right-hand end of the string.

1.1.4: AST Display

The display on the screen of the AST is limited to only those parts of the tree that are "near" to the current node. Any part of the tree which is too far away to be displayed is folded into an icon. The icon is in the form of the name of the type of node which was folded enclosed within curly brackets. The definition of "near" is given in terms of two numbers. One indicates how high up the tree to look to find the root of the display. The other indicates how many levels of the tree below the current node should be displayed. These two values can be altered by the **control** command thereby changing the display characteristics at runtime.

1.2: Commands

This is a list of all the editing commands that are available in any editor generated by Sgen. With each command is given the single key that can be used to get the command. If the reason for the choice of key is opaque then the word that it is supposed to abbreviate is also given.

up: Changes the currently selected node of the AST. The new current node becomes the parent of the currently selected node. The key is "**u**".

down: Also changes the currently selected node of the AST. This time the new node is one of the children of the current node. The user is asked which child to select. The number of the child should be indicated by the appropriate digit on the keyboard. There is no need to press <return>. This decision implies that no AST node may have more than nine children. The increased ease of use was felt to out-weigh this limitation. Any situation where an AST node has more than 9 children is likely to be unmanageable for the user. The key is "**d**".

right: Advances the cursor in a root-to-left-to-right traverse of the nonterminal nodes of the AST. The key is **<return>**.

left: Applies the same cursor movement as the command above except in the opposite order. The key is **<space bar>**.

del: Deletes the current nonterminal AST node from the abstract syntax tree. The old limb of the AST is, however, still used for attribution. The key is "**k**" to stand for "kill node".

undel: Reverses the effect of **del**. The section of tree that was removed by the delete at that node is replaced. The key is "**l**" to stand for "live" which is (almost) the opposite of "kill".

add: Allows text to be added at a deleted node. The editor must have a parser for that type of node. The user is prompted to enter text. The text is parsed. If the parse is successful then the generated AST is inserted at the current node and re-attribution takes place. Otherwise no change is made. The key is "**a**".

root: Sets the current node to be the root node of the tree. The key is "**r**".

start: Sets the current node to be the leftmost node of the tree. The key is "**s**".

end: Sets the current node to be the rightmost node of the tree. The key is "**e**".

change: The specialisation of add for terminal symbols. Any terminal symbol can be changed. The user is prompted for a string. This string is sent through the lexical analyser. If the entire string corresponds to a lexical token in the same class as the original token then the replacement is made followed by re-attribution. The key is "**c**".

pack: Copy to the clipboard. The Sgen editor maintains a clipboard of named pieces of AST. These fragments can later be placed in any position compatible with that type of node. The user is asked for a name to store the current node under. If there is already something stored under that name, it will be overwritten. The key is "**p**".

unpack: Retrieves a fragment of AST from the clipboard. The current node must have been deleted. A menu appears of all the AST fragments that are held. The user can choose a fragment using the left mouse button. The fragment of AST associated with that name in the clipboard is then put in place in the current tree if it is of the correct type. The key is "**o**" to stand for "open".

quit: Quits the editor. The key is "**q**".

control: Gives the user the option of changing various details of the display. See the section below on display parameters. The key is "^" which is intended to represent a rise in level.

2: Programmer's Guide to Lgen, Pgen and Sgen

This section gives a programmer's guide to the three programming tools Lgen, Pgen, and Sgen. Lgen and Pgen are extremely general purpose programming tools suitable for all applications from a small scale user response to an interactive prompt to a full scale program in a high level language.

All the procedures can be found in the PS-algol database `"steve"`, with password `"steve"`, which should be made when the system is built. For precise details on how to use these programs to create lexical analysers, parsers or editors, examine the examples in the "tester" directories (see sections **2.1.4**, **2.2.3** and **2.3.4**).

Details of the implementation and loading of the systems are given in the hacker's guide (section **3.2**).

2.1: Lgen - a Lexical Analyser Generator

Lexical analysis is the process of taking an input stream of characters and breaking it into groups of characters or tokens. The characters of the token (or lexeme) form one semantically meaningful unit of the language. The input could be anything from a high level programming language to a user response to a simple interactive prompt. If the abstraction at this lexical level is not made, code tends to become cluttered and opaque with the important features hidden. If the lexical abstraction is rewritten for each application, programmers find themselves repeatedly rewriting very similar code. The solution is to generalise the lexical abstraction by writing a lexical analyser generator.

Lgen is such a tool for building lexical analysers. When provided with a specification (described below) it will produce a function for breaking up a string into tokens. The schematic type of Lgen is

lexical specification → (character stream → token stream).

There are several benefits from using such a system rather than a programming solution:

• The implementation time is decreased. The specification can be written quickly and correctly.

• Given that the specification is correct the lexical analyser will be correct. All debugging and modification work can be done at a specification level.

• A standard type of lexical analyser is produced. This allows for flexibility in that analysers may be interchanged and programmers coming across an analyser for a new purpose will already be familiar with the interface.

Lgen is stored under the key `"lgen"` in a structure class

```
        structure lgen.box (
                proc (*pntr -> proc (pntr -> pntr)) lgen.place)
```

where the parameter **`*pntr`** is the specification and the **`proc (pntr -> pntr))`** returned is the analyser generated.

2.1.1: Lgen Specification

An Lgen specification is a vector of token declarations. Each token must be declared using a structure of the class

```
        structure t.spec (
                string t.pattern,       ! (a)
                       t.token;         ! (b)
                bool return)            ! (c)
```

The meaning of each field is as follows (note that I have changed the order in the discussion of the fields):

(b) *`t.token`*: This is the name that will be associated with every instance of this class of token. The name of the semantically meaningful unit. Some examples of

token names might be `"identifier"` or `"begin.token"`.

(c) *`return`*: This indicates whether this token is to be exported or not. Tokens can be specified in terms of other tokens. It is therefore desirable that a user may define a token purely for the purpose of use in a future token declaration. In this case it is not required that this token ever be recognised in itself by the analyser. By setting this field to **`false`** the user can specify that this is a private token. i.e. the token is not to be exported. A typical example of this might be in declaring a token to represent identifiers. It may be convenient to introduce a token to represent letters. Clearly the 'letters' token is not required ever to be recognised in its own right.

(a) *`t.pattern`*: This is the description of all character sequences that can be instances of this class of token. The description is in the form of a stylised regular expression (RE). An RE can be either a) a literal or b) a structured combination of regular expressions. In BNF notation an RE is

$\mathcal{RE}$ ::=	*literal*	(1)
	| " *named_RE* "	(2)
	| **[** $\mathcal{RE}$ **]**	(3)
	| **<** $\mathcal{RE}$ **>**	(4)
	| **(** *RE_list* **)**	(5)
	| $\mathcal{RE}_1$ $\mathcal{RE}_2$	(6)

(1) **literals**: These can be any single character. Because of the eight operator characters `"`, `[`, `]`, `(`, `)`, `|`, `<`, and `>`, an escape character is needed to generate these as literals. The escape character is `'` (an apostrophe, as in PS-algol). You escape the escape character to get the literal apostrophe.

(2) **names**: The regular expression associated with another token can be used by enclosing the token name of the required pattern in quotes, for example, `"letter"` to use the regular expression for letters. A linear scoping scheme is implemented. i.e. a token name t_i can only be quoted in

a token definition t_j such that $i < j$.

(3) **repetitions**: These indicate that the given regular expression can appear any number of times (including zero). For example, `"aaa"`, `"a"`, and `""` all match the pattern `[a]`. In this case the inner regular expression is trivial but, in general, it can be arbitrarily complicated.

(4) **options**: These define the inner expression to be optional; it can appear one or zero times.

(5) **alternatives**: This type of definition indicates that one of a series of regular expressions can appear. Each option is separated by a vertical bar `|`. For example, this mechanism would be used in the definition of the token `"digit"` as `(0|1|2|3|4|5|6|7|8|9)`.

(6) **sequences**: These mean that the regular expression $\mathcal{RE}$ is obtained by any instance of $\mathcal{RE}_1$ followed by an instance of $\mathcal{RE}_2$. As such they are the most commonly used combination mechanisms. Note that there must be no spaces between the regular expressions, as the space itself would be treated as a literal expression.

Note that there are no facilities for specifying ranges of ASCII characters.

We are now in a position to give a specification example; a language of positive integer expressions with the four main operators, brackets and identifiers. The identifiers are as used in PS-algol. Note that much of the input is distorted by the need for PS-algol escape characters. This problem is overcome by use of the program Lexan described in section **2.1.3**.

```
let specification = @1 of pntr [
        t.spec  ("( |)", "whitespace", true), ! tab or space
        t.spec  ("(0|1|2|3|4|5|6|7|8|9)", "digit", false),
        t.spec  ("'"digit'"['"digit'"]", "number", true),
        t.spec  ("'(", "open bracket", true),
        t.spec  ("')", "close bracket", true),
```

```
        t.spec  ("(+|-|*|/)", "operator", true),
        t.spec  ("(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)",
                "small", false),
        t.spec  ("(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)",
                "big", false),
        t.spec  ("('"big'"|'"small'")", "letter", false),
        t.spec  ("('"letter'"|'"digit'")", "character", false),
        t.spec  ("'"letter'"[('"character'"|'"digit'")]",
                "identifier", true)]
```

2.1.2: The Lexical Analyser

The lexical analyser produced is a procedure which will generate a specific analyser given an input stream. The input stream is an object of class

```
structure l.stream (
        proc (-> string) ls.gtok;       ! the "get" procedure
        proc (string) ls.ptok)          ! the "put" procedure
```

There are two procedures; one to deliver the next character of input and one to receive a character back. Any characters sent back should be subsequently returned again on future calls to *ls.gtok*. These procedures are responsible for maintaining an input buffer. The problem of providing accurate error information was considered too difficult to generalise so has been left to the user. Typically this will consist of two extremes, one where the job is so trivial that error diagnostics are unnecessary and the other where the application is so complicated that the requirement has to be tailored to this specific case. The "get" procedure should return `""` repeatedly when the end of stream is reached.

Given an instance of *l.stream*, a lexical analyser will be produced. The lexical analyser is an instance of a structure class

```
structure l.pack (
        proc (-> pntr) gtok;    ! (a)
        pntr spellings)         ! (b)
```

(a) ***gtok***: This always provides the next lexeme descriptor that can be matched on the input stream. If two tokens can be matched, the longest is returned. If there are two (or more) of the same length then the one that was declared last (nearest the bottom) in the specification is returned.

(b) ***spellings***: This is a table of all the lexemes seen. The table is indexed by lexeme to the lexeme descriptor returned by *gtok*. There is a single entry in the spelling table for each lexeme and thus the same instance of *s.entry* (see below) is returned every time the same lexeme is recognised. In particular, if an identifier has been seen once then all subsequent times it is recognised the same descriptor will be returned.

A lexeme descriptor (that which is returned by *gtok* and held in the *spellings* table) is an object of structure class

```
        structure s.entry (string s.lexeme, s.token; pntr s.tail)
```

The first field is the lexeme that was matched - a copy of that section of the input stream that matched a pattern. The second field is to indicate the class of token of which this lexeme is a member. The third field is unused but is there to enable a user to attribute information to lexemes, for example symbol table entries; in Sgen (see section 3.7) it is used to point to the parent of a terminal node.

2.1.2.1: Whitespace

The name "whitespace" is a special token name which the analyser will skip if it recognises it on the input and will continue searching for the next token. This corresponds to the notion of whitespace in most lexical analysis applications. Note: this does not mean that in its declaration the return field is set to **false**. It is of lexical significance and therefore has the return field set to **true**. Whitespace is handled at a higher level than basic lexical analysis.

2.1.2.2: Errors

In the event of the input stream not matching any pattern in the analyser, the machine searches for the next segment of the input which will match a pattern. It then returns a token with the *s.token* field "error.token" along with a lexeme that is the entire section of input which would not match any token.

2.1.2.3: End of File

When the stream function returns "" to indicate that the end of file has been reached, the analyser returns a token with *s.token* field "_eof". All subsequent calls to the analyser will produce this same token.

2.1.2.4: Warning

A warning should be given here about the dangers of defining a token that can be matched by the empty string. If the token is ever matched then it will be matched infinitely many times and the rest of the input stream will be lost.

2.1.3: Lexan

Lexan is an auxiliary program that can be used to aid the building of lexical analysers. Because of the use of control characters necessary in PS-algol it can become quite confusing to enter a specification within the body of a program. It can also be quite messy as each individual declaration has to be boxed up in a structure. For these reasons the program Lexan, stored in a structure

```
        structure lexan.builder.box (
              proc (string -> pntr) lexan.builder.place)
```

under the name "`lexan.builder`", can be used. It takes in a filename in which each declaration is given over three lines. The first is the name of the token. The second is the pattern. The third indicates whether the the token is returnable or not ("`y`" indicates that it is and "`n`" indicates that it is not). There are some special cases built into the program.

The character "`\`" is used as the escape character within patterns.

`\n`	is the newline character
`\t`	tab
`\b`	space
`\\`	backslash
`\string.any`	any character (or sequence such as 'n) that can appear in a PS–algol string
`\comment.any`	any characters that can appear in a PS-algol comment

The last two were included to aid in the building of the PS-algol test analyser and remained since they were of such practical use in other places.

The structure returned by Lexan is of class

```
structure lexan.box (proc (pntr -> pntr) lexan.place)
```

and contains the required lexical analyser.

2.1.4: Tester

There is a test lexical analyser that comes with the system. It is an analyser for simple positive integer expressions. A shell script, LOAD.tester, is provided to load the system. It is useful to experiment with this example.

2.1.5: Lgen and Lex

Lex (see [LESK79] in the references) is the standard Unix tool for creating lexical analysers. Lgen provides all the essential features of Lex in a cleaner way, in that PS-algol provides higher order functions and persistence, so there is no need to output source code as Lex does. This leads to a more appealing implementation of a lexical analyser generator.

Another difference from Lex is that the finite state machine generated is interpreted rather than compiled; this could be avoided by adding a back end using the callable compiler, but this is not likely to lead to a large increase in speed.

2.2: Pgen - a Parser Generator

Pgen is a parser generator. It can be used to generate parsers which take the stream of tokens and generate a structure representing the semantics of the stream. The lexical analyser should be produced by Lgen. Pgen is an SLR(1) parser generator. The reasons for wishing to use a parser generator are exactly as indicated for Lgen. The schematic view of Pgen is also similar:

syntactic specification → (token stream → parse tree)

Pgen is stored under the key "pgen" in a structure of class

```
        structure pgen.mk.box (
                proc (*pntr, *string -> pntr) pgen.mk.place)
```

where the `*pntr` and `*string` parameters are the two components of the specification (see **2.2.1**) and the `pntr` returned is a table of parsers (see **2.2.2**).

As before, I will start with a description of the specification process and then detail the workings of the generated parsers.

2.2.1: Pgen Specification

The motivation for the design of Pgen was initially to aid in the development of Sgen. As such parsers were required for various parts of the language as well as for the language as a whole. For this reason the parser generator was required to return a table containing parsers for all the nonterminals that the specification required.

A Pgen specification is in two parts. (1) A BNF grammar and (2) a vector of strings naming the nonterminals for which parsers are required.

The BNF specification takes the form of a vector of pointers each of which represents a rule of the grammar. Each is a structure of class

```
        structure p.spec (
                string ps.rule;
                proc (*pntr -> pntr) ps.action)
```

Each `ps.rule` should be a string containing

1) the name of the nonterminal that this is a rule for;
2) the symbol "::=", pronounced "derives";
3) a list (possibly empty and separated with blanks) of names. These names should be either the names of nonterminals of the grammar or names of terminal symbols to be recognised. A nonterminal is in the lexical class of PS-algol identifier while terminals are the same but prefixed with the symbol '#'. A list of occurrences of these symbols will be used to form an instance of the nonterminal on the left.

LR parsing naturally supports the evaluation of one synthesized attribute. Typically this would be the abstract syntax tree of the current parse. To support this, along with each nonterminal node of the parse tree, there is space for one pointer value. The procedure `ps.action` is executed when all the subtrees for a nonterminal have been completed. The value it returns is the value that is associated with this instance of the nonterminal. The arguments that are passed to it are the value associated with each of its children. If the subtree is a literal then the lexeme descriptor (an `s.entry`) is passed instead.

Generally the `ps.action` attribute is the abstract syntax tree but it can be used for anything. The following is an example specification. It is a continuation of the integer expression example given in the last section. Here the synthesized attribute is used not as an AST but to evaluate the expression. We will require that the value at the root node is an instance of

```
        structure a.number (int the.number)
```

with the correct integer value in place. The first step is to write the procedures that will be used in the evaluation. The specification is then created in two variables *spec* and *required*, *required* being the vector of nonterminal names that need to have parsers created for them.

```
let do.operation = proc(*pntr v -> pntr)
          case v(2)(s.lexeme) of
                    "+":  a.number(v(1)(the.number) + v(2)(the.number))
                    "-":  a.number(v(1)(the.number) - v(2)(the.number))
                    "*":  a.number(v(1)(the.number) * v(2)(the.number))
                    "/":  a.number(v(1)(the.number) / v(2)(the.number))
                    default:   {abort; nil}      ! error
! assume a procedure eval of type proc (string -> int)
! which finds out the number represented by the string
let do.number =
          proc (*pntr v -> pntr);
                    number (eval (v (1) (s.lexeme)))
let do.bracket =
          proc(*pntr v -> pntr);
                    v(2)
let spec = @1 of pntr [
          p.spec ("exp ::= exp #operator exp", do.operation),
          p.spec ("exp ::= #number", do.number),
          p.spec ("exp ::= #open exp #close", do.bracket)]
let required = @1 of string ["exp"]
let editors = pgen (spec, required)
let expression.parser.pack = s.lookup ("exp", editors)
structure a.parser (proc (pntr -> pntr) the.parser)
let expression.parser = expression.parser.pack (the.parser)
```

This specification is particularly simple-minded. I have ignored the identifiers defined in the lexical specification, and also the fact that this grammar for integer expressions is ambiguous. In this example only one nonterminal, *exp*, is declared and all the other symbols are terminals.

2.2.2: Pgen Parsers

Pgen returns a pointer. This pointer is to a PS-algol table indexed by nonterminal name onto structures of the class

```
          structure a.parser (proc (pntr -> pntr) the.parser)
```

The parameter expected is a pointer to an `l.pack` as produced by `lgen`. This is used to generate the stream of tokens. The pointer returned is to the parse tree (or derivation tree) that was produced. The type of this tree is

```
          structure p.tree (
                    pntr p.gs;          ! a grammar.symbol; see section 3.6, III
                    *pntrp.children;
                    pntr p.action, p.parent, p.value;
                    bool p.deleted)
```

The field that is of direct interest is the `p.value` field. This contains the value of the synthesized attribute at this node. The whole parse tree is made available as the derivation may be of use. The field `p.children` contains all the children of this nonterminal node. Each one is either another instance of `p.tree` or an instance of `s.entry` to indicate a terminal node.

When Pgen is used to parse a token stream, the token stream is exhausted. It is not possible to reuse that stream afterwards.

2.2.2.1: Specification Errors

If the specification is invalid in any way then Pgen will return **nil** instead of the table of parsers. Some internal errors (in the specification of the grammar) will cause Pgen to abort.

2.2.2.2: Error Recovery

Pgen does not attempt any error recovery. Generalising error recovery in LR parsing is beyond the scope of this project and as such the problem was put to one side. In the event of an error, the Pgen parser will simply return `nil` rather than a pointer to a `p.tree`.

The willingness to accept this decision was influenced by the fact that Pgen was designed specifically for use in Sgen, the structure editor generator. When using a structure editor it is typical to type only small sections of text to be parsed at any time; in general, less than one line of characters. With such small strings, error recovery is not as significant.

2.2.2.3: End of Token Stream

Pgen assumes the same mechanism as Lgen uses to signify the end of file. It is, in theory, possible to plug a hand-crafted lexical analyser into Pgen but should not in general be necessary.

2.2.3: Tester

There is a test parser that comes with the system. It is a parser for simple positive integer expressions. The shell script LOAD.tester loads the system. As with Lgen, this is a useful tutorial example.

2.2.4: Pgen and Yacc

Yacc (see [JOHN79])is the standard parser generator tool in Unix. It implements parser generation for a slightly wider set of grammars (LALR(1)) than Pgen. However, Pgen makes use of higher order functions to clean up the implementation of many details of the specification, the generated parser and the synthesized attribute evaluation.

2.3: Sgen - a Structure Editor Generator

The initial motivation for Lgen and Pgen was to produce a language based editor generator, Sgen. Sgen should have the schematic type

language specification of L → ((sentence of L and input) → sentence of L).

The specification of a language to the degree that an editor can be generated involves the description of several features of the language:

- **Lexical syntax**: This is done simply by providing a lexical analyser constructed by Lgen.
- **Concrete syntax**: More than simply a parser for the language is required here. Each construct of the language that the user of the editor will be allowed to edit individually will need its own parser. This was the main motivation for Pgen being designed so that parsers are generated for all the nonterminals listed in the second field of the specification.
- **Abstract syntax**: In general the structure of the concrete input syntax is distorted by the needs of parsing. Therefore a syntax which reflects the semantics of the language is required. All editing is done in terms of the abstract syntax tree. Sgen requires that the abstract syntax be specified. Sgen itself does not build the AST as the construction of an AST node from possibly a collection of parse tree nodes is not an easy problem to describe in generality. Each parser given to Sgen is therefore required to produce, as its synthesized attribute, an abstract syntax tree. Sgen then provides an editing executive to edit and maintain this. A description of what the abstract syntax should be is in section **2.3.2**.
- **Unparsing scheme**: The translation from input stream to abstract syntax tree loses the original layout of the script on a page. Indeed, this layout may never have existed in its entirety if the script was generated entirely within the editor. It is therefore required that the user specify the layout of the script on the page.
- **Attribution**: While an AST is being maintained it is natural to wish to be able to analyse static semantic features of the language, as in type checking and scope checking of identifiers. A facility to do this is implemented by means of an attribute grammar that is specified along with the abstract syntax tree and whose evaluation is maintained by Sgen as the AST is updated.

Sgen is of PS-algol type

```
proc (pntr, pntr, *pntr, int, int -> proc (pntr -> pntr))
```

where all five parameters are required for specification (see **2.3.1**) and the generated editor is the **proc (pntr -> pntr)** (see section **2.3.3**). It is stored in a structure *sgen.box* with one field called *sgen.place* of the type above.

There follows a description in detail of the specification of an Sgen structure editor. Following that, the operation of the editor is described.

2.3.1: Sgen Specification

Sgen is of PS-algol type

```
proc (pntr,                  ! (a)
      pntr,                  ! (b)
      *pntr,                 ! (c)
      int, int               ! (d)
      -> proc (pntr -> pntr))
```

The parameters of the specification are as follows:

(a) **pntr**: This is to the lexical analyser. It should be stored in a

```
structure lexan.box (proc (pntr -> pntr) lexan.place)
```

and will be used to generate a lexical analyser every time a string needs to be parsed.

(b) **pntr**: This is to the parsers that are to be used. The parsers should be stored in a PS-algol table as returned from Pgen. The synthesized attribute should be of type AST which is described below.

(c) **`*pntr`**: This is the specification of the AST. This includes specification of the unparsing scheme and of the attribution. Each pointer should be to a node of structure class

```
structure ast.dec(
        string dec.spec;          ! (1)
        bool dec.parsable;        ! (2)
        string dec.parse.as;      ! (3)
        bool dec.is.root;         ! (4)
        string dec.unparse;       ! (5)
        pntr dec.attribution)     ! (6)
```

These fields are:

(1) a **`string`** for the AST rule specification (described below).

(2) a **`bool`** indicating if this type of node is parsable,

(3) a **`string`** indicating the parser to be used to create a node of this type.

(4) a **`bool`** to indicate if this is the type of the root of the tree.

(5) a **`string`** to indicate the unparsing scheme (described in **2.3.1.2**).

(6) a **`pntr`** to a specification of the attributes and evaluation rules that will be required at every instance of this type of node (see **2.3.1.3**).

(d) Two **`ints`** to describe the initial display parameters of the editor that is generated. See section **1.1.4** on display parameters.

2.3.1.1: AST Rule Specification

An AST rule specification is a **`string`**, which should contain

(a) the <u>name</u> of the abstract nonterminal that this rule is declaring a right-hand-side for. Abstract nonterminals all have the lexical syntax of PS-algol identifiers but must have capital first letters.

(b) the <u>derives symbol</u> "`::=`".

(c) the <u>constructor</u> for this rule. This is the name that identifies a node built

using this rule. The lexical syntax of a constructor is the symbol "`@`" followed by a string with the syntax of a PS-algol identifier.

(d) the <u>argument list</u> enclosed within round brackets. Each element of the argument list (which could be empty) must be either a node class of the AST or the name of a lexical token. Lexical tokens are specified by a PS-algol identifier prefixed with the "**#**" symbol as with literals in a Pgen specification.

As an example of a group of AST rule declaration strings, I will continue with the integer expression example. In the parser example I used a simple minded concrete syntax. The common unambiguous concrete grammar for expressions is extremely distorted and would certainly require an abstract syntax such as the one given below.

```
"Exp ::= @Number(#number)"
"Exp ::= @Id(#identifier)"
"Exp ::= @Bracket(Exp)"
"Exp ::= @PlusOp(Exp Exp)"
"Exp ::= @MinOp(Exp Exp)"
"Exp ::= @DivOp(Exp Exp)"
"Exp ::= @MultOp(Exp Exp)"
"Exp ::= @Let(#identifier Exp Exp)"
```

2.3.1.2: Unparsing Rule Specification

Each *`ast.dec`* will have a rule such as the ones above and also an unparsing (display) scheme for the particular form of node. This is also specified by means of a string. The string contains a list of tokens of five classes separated by whitespace (blanks or tabs). The five token classes are

• **literal**: Enclosed within quotes - "`"literal"`"

• **child**: To indicate that a child's display should be put at this place. The selection of which child is positional on the number of children already displayed. This is indicated by the token "@".

• **tabbing**: Tabbing is not done normally. Tabbing indicates the position of the left margin. This allows for features like indentation of blocks. To set the tab one more unit to the right use the token "->" and to the left use the token "<-".

• **attribute values:** To display the value of an attribute the format is `$.`*`name`* where *name* is the name of the attribute to be displayed. All attribute values are **`pntr`** but the value of any attribute to be printed must be of an instance of the structure class

```
        structure att.printable (string att.string).
```

Attribute values are not printed when the unparse is as a dump of the tree for output as opposed to going to the screen

• **hidden literals**: To indicate that this literal should be printed at this position during an unparsing for the screen but should be omitted if the text is for output. This allows literals to be used to guide editing or explain an attribute value that is being printed where the literal is no part of the syntax of the script. This type of unparse is obtained by prefixing a literal unparse with a colon character, for example

```
        :"value = "
```

An example of unparsing rules would be

```
@Number  " @ "
@Id      " @ "
@Bracket " '"('" @ '")'" "
@PlusOp  " @ '"+'" @ "
@MinOp   " @ '"-'" @ "
@DivOp   " @ '"/'" @ "
@MultOp  " @ '"*'" @ "
@Let     " '"let '" @ '" = '" -> '"'n'" @ '"'nin'n'" @ <- "
```

Note how these strings are confused by the requirement for PS-algol control characters. This is clearly an area for improvement.

2.3.1.3: Specification of Attribution

Attribute declarations are associated with each constructor for each type of AST node. The specification takes the form of a list of individual attribute specifications where the list must be a structure

```
        structure mk.Att.list (pntr Att.elem, Att.next)
```

Each attribute declaration takes the form of a pair held in an instance of the structure class

```
        structure Att (
              string Att.spec;
              proc (*pntr -> pntr) Att.eval)
```

The **`string`** specifies the name of this attribute and where the arguments for its evaluation should be found. The **`proc`** field is the evaluator for each instance of this attribute. The structure of the **`string`** is as follows:

- the name of the attribute being assigned to. This has the lexical syntax of a PS-algol identifier.
- an assignment token : ":="
- an open round bracket "(".
- a list (possibly empty) of other attributes and values each of which must be one of four classes (where *name* has the form of a PS-algol identifier)
    - `^.`*`name`* to indicate a parental attribute
    - `@.`*`name`* to indicate another attribute of this node
    - `$`*`i.name`* to indicate an attribute of the *i*'th child of this node. The number *i* must be a single digit. This effectively limits the number of children of an AST node to 9. I do not feel this to be a great limitation as an AST node should be simple enough to convey one semantic unit of the text.
    - `$`*`i`* to indicate that the *i*'th child is to be used as a parameter.

This allows, in particular, the selection of terminal nodes.

- The close round bracket token ")".

Some examples of attribute specification strings might be

```
"env := (^.env)"
"value := ($1.value, $2.value)"
```

Sgen performs a simple check for attribute circularity.

2.3.2: The Abstract Syntax Tree

The abstract syntax tree produced by the parsers must be an instance of the class AST. It must also satisfy the specification of the abstract syntax. If it does not then there is no guarantee on how it will be treated. The type is

```
structure AST (
        string AST.name;                ! the name of this type of node
        string AST.opt.name;            ! the name of the constructor
        *pntr AST.children;             ! the children of this node
        pntr AST.spec;                  ! the declaration of this type of node
        pntr AST.opt;                   ! the declaration of this constructor
        pntr AST.parent;                ! the parent of this node
        pntr AST.attribution;           ! attribute values for this node
        bool AST.deleted)               ! is this node deleted from tree?
```

The parser is only required to put values into the first 3 fields as the rest will be filled in by the system. That is, the user says what type of node it is and what constructor was used. The user also gives pointers to all the children of this node. Each child must be of the correct type for the rule specifying this constructor. Each child must also be either another AST node or a terminal instance of *s.entry*. Then the system will fill in all the other details in each AST node.

2.3.3: The Sgen Editor

The Sgen editor is simply an executive that allows the editing and maintenance of a consistently attributed abstract syntax tree. All alterations of the AST result in reattribution (on those parts that are now out of date) and redisplaying of the tree.

The editor is a procedure with the PS-algol type

```
        proc (pntr -> pntr)
```

which takes in and returns a structure in the class

```
        structure s.state (string s.string; pntr s.state)
```

A script produced during one execution of the Sgen editor may subsequently be re-edited. It is further possible that the editor specified may have incompatible input and output syntax. To cope with both these cases, the Sgen editor works in terms of a pair of values for each script:

- the unparsed script
- the abstract syntax tree.

On input to the editor, the the abstract syntax tree will be used if the pointer is not **nil**. Otherwise an attempt will be made to parse the given string. On output from the editor, *s.string* will be the unparsed version of the current AST and *s.state* will be the abstract syntax tree.

2.3.4: Tester

There is a test editor that comes with the system. It is an editor for a simple "**let** ... **in** ..." positive integer expression language with attribution to evaluate the expressions and thereby produce a calculator. The shell script LOAD.tester loads the system and a brief "play-about" session is most instructive in getting to grips with Sgen.

2.3.5: Sgen and CSG

The Cornell Synthesizer Generator (see [REPS83], [REPS85]) is another structure editor generator. Some of the notation (in the unparsing scheme for example) is very similar to that used by CSG. PS-algol, however, provides many facilities which lend themselves to generator programs such as Sgen. Two obvious ones are that its treatment of data by dynamic strong typing and persistence allows the building of large data structures to represent complex structures easily, and that it fully implements higher order functions. These features have lead to a more integrated and (potentially) more appealing implementation than one based on C and text files.

Using persistent higher-order functions also leads to "functional" behaviour in the editor (one stream in and one stream out), whereas the CSG is "Emacs-like" - many files may be "visited" and modified in the course of an editing session. Also like Emacs, CSG supports multiple buffers; the clipboard in Sgen provides some of the same capabilities, but all editing takes place in one buffer.

3 : Hacker's Guide to Lgen, Pgen and Sgen

This section is a hacker's guide to the programming tools. It contains a discussion of various features of the systems' implementation:

- the source files
- loading the system
- the databases and procedure hierarchy
- program descriptions

The level of detail given in this report should be sufficient to point a programmer in the right direction but is in no way a line-by-line (or even procedure-by-procedure) guide. I suggest a hunt for the authors as the best method of discovering the intimate details (about the programs, not the authors).

Some of the algorithms used are not immediately obvious. Most of them are given in [AHO85]. The relevant pages are cited at appropriate places.

3.1: Source Files

All the files are under the directory ..../Summer87. There are 6 directories, two for each program - a program directory and a tester directory. The directories are sgen and sgen.tester, pgen and pgen.tester and lgen and lgen.tester. Each program is entirely self-contained within its directory except that:

- Pgen requires all of Lgen to be already present in the database
- Sgen requires all of Pgen and Lgen to be present.

3.2: Loading the Systems

The facilities required to load the system are listed below. I have tried to include everything but have almost certainly overlooked something that has been taken for granted. If the system cannot find anything it needs it will give an error message and "down tools" in protest.

- PS-algol compiler, interpreter and databases;
- A database for the generators, which should be given the name `"steve"` and password `"steve"`;
- The `"rutilities"` database;
- The procedure `"chooser"` as prepared by Richard Cooper;
- The procedure `"seditor"` as prepared by Richard Cooper.

Shell scripts have been prepared for each system to compile and load the required procedures into the database. These scripts are each in the appropriate program directory and are called LOAD.lgen, LOAD.pgen and LOAD.sgen. Each tester program also has a LOAD script which will set up the test analyser/parser/editor.

At this point all three procedures *lgen*, *pgen* and *sgen* have been loaded into the database `"steve"` with password `"steve"` along with the auxiliary procedure *lexan*.

The current implementation contains at least two machine dependencies. Firstly, only one font size is supported and its dimensions are hardwired into the code. Secondly, editing a script involves the editor having total control of the screen. It is necessary that when a key is depressed on the keyboard the normal display of that character be suppressed. In the current implementation this is done by two procedures *mk.term.cool* and *mk.term.uncool* in the file sgen. These send appropriate control sequences to the screen which turn echo on and off. The standard Unix way of doing this is to use the stty command. For some reason this

did not work. This may be because PS-algol entirely controls the screen at runtime and someone has overlooked this possibility. This is, however, pure speculation.

3.3: The Database and the Procedure Hierarchy

The database is simply a PS-algol table indexed to one level containing all objects required to be stored in the database. Procedures are linked together in such a way that it is necessary to load them in a certain order. There is a hierarchical structure to them and in order for any change in a lower procedure to have any effect, all procedures between the new procedure and the root must be reloaded. The structure of the three hierarchies is given in the appendix to this section.

Most of the programs are used entirely to save procedures in the database. Although there are many procedures that would be shared by two or more processes using the system, there are no variables that are shared, so no concurrency problems should occur. However, there are clashes between these programs and others that use the same utility procedures, since some of the utility procedures require write access to the `"rutilities"` database. This problem will manifest itself in a message

```
can't get database ("steve", "steve") (?).
```

If this happens, try again when the other user has finished.

3.4: List Representation

In many places there was a need to have a list of pointers to objects. The structure class

```
structure mk.list (pntr list.entry, l.rest)
```

was used throughout to contain such lists.

3.5: Lgen Implementation

Lgen takes input in the form of a vector of regular expressions. The output is a minimised deterministic finite state machine which is interpreted at runtime. The main procedures are as follows

I: parse - This program stores a procedure `parser` in the database under the key `"Regular Expression Parser"` in a

```
structure RegExpParser (proc (string -> pntr) RegPars)
```

`parser` takes in a string and returns a pointer to the structure that represents the regular expression. The structure has various forms to indicate the various types of regular expression.

```
structure literal (string l.reg)
structure optional (pntr o.reg)
structure multiple (pntr m.reg)
structure alternative (*pntr a.regs)
structure sequence (pntr s.reg2, s.reg2)
structure named.string (string reg.name)
structure empty (int i)              ! must be at least one field
structure parse.error (string message)
```

These form a structure in which the terminals are always `literal`, `named.string`, `empty` or `parse.error`. The parser returns either an entirely valid structure or a `parse.error` with all the error messages collected to the topmost level. In the current implementation the error messages are no better than useless. There is an example of what a representation of a regular expression would look like in the diagrams section towards the end of this section.

II: ndfm - This program stores a procedure `build.automaton` in the database under the key `"auto.build"` in a

```
structure auto.box (proc (*pntr -> pntr) auto.build)
```

It takes in a vector of regular expressions and returns a structure representing the nondeterministic machine which is equivalent to the expressions. The algorithm ([AHO85] pp. 121) patches the sub-regular expressions together by a simple method based on creating new states and having empty transitions everywhere.

For example, (r_1 | r_2 | r_3) can be done by:

(1) deriving an NDFM for each r_i $(i = 1, 2, 3)$

(2) creating a new start and finish state; *(s)* and *(f)*

(3) adding an empty move from *(s)* to r_i $(i = 1, 2, 3)$

(4) adding an empty move from r_i to *(f)* $(i = 1, 2, 3)$

This is done recursively. The next step is to put in place the machines for the named patterns. This requires rebuilding the structure to represent the expression for the pattern that was named. After this all the empty actions are removed by propagating all actions to all places reachable on empty moves only from any state. Eventually the machines for each pattern are joined together to form a single machine. The data structure used to represent the machine is

```
structure ndfm (pntr n.first, n.last)
structure ndfm.state (
          int s.number;
          pntr s.shifts;
          *string s.class)
structure ndfm.action (
          string a.input;
          pntr a.new, a.next)
```

The structure `ndfm` is used to hold pointers to the first and last state of a linked structure representing the machine. The two fields both point to states. Each state has

(a) a unique number `s.number`;

(b) a list of actions `s.shifts`;

(c) a vector of strings `s.class`, to indicate what has been recognised on any input stream that leaves the machine in this state.

The list of actions is made up of a list built by *ndfm.action* where

(a) *a.input* indicates the character for this action

(b) *a.new* points to the new state for this action

(c) *a.next* indicates the next action (or **nil**) in the list.

There is an example of a nondeterministic machine given in the diagrams appendix towards the end of this section.

III: min - This program stores a procedure *minimise* in the database under the key "minimise" in a

```
structure minimise.box (proc (pntr -> pntr) minimise.place)
```

minimise takes in a deterministic machine and returns a new machine which represents the same language but has the minimum number of states possible ([AHO85] pp. 141). This minimum machine is unique down to state name. The algorithm partitions the machine as follows

- first split it into accepting and non-accepting states;
- split the accepting partition into a set of partitions each containing only one element;
- split the non-accepting partition into partitions containing states with exactly the same set of symbols with actions defined on them;
- if there is a partition with states with actions on the same input which take you to different partitions, then split the current partition;
- repeat the previous step until no more partitions can be made.

At this stage all the states in each partition form an equivalence class and a representative from each partition can be chosen to be the new state representing all the states in the partition.

IV: dfm - This stores the procedure *determine* in the database under the key

"determine" in a

```
structure determine.box (proc (*pntr -> pntr) determine.place)
```

determine takes in a vector of regular expressions and produces a minimised deterministic finite state machine. It first uses *ndfm* to build the (massive) nondeterministic machine. It then makes a new automaton which is deterministic. Each state in the new machine corresponds to a set of states in the old. This deterministic machine is then minimised using *min*. It is helpful to read [AHO85] pp. 117, though the algorithm is not exactly the same.

V: opt - This stores the procedure *optimise* in the database under key "opt" in a

```
structure opt.box (proc (pntr) opt.place)
```

optimise takes in a deterministic machine and alters the implementation of the actions from a list of actions to a table of actions. This is supposed to make the interpretation of the machine faster. Whether it does is open to question. It does, however, make the implementation more appealing and simplifies the code for the interpretation.

VI: lgen - This stores the procedure *lexgen* in the database. Its database key and containing structure were described in section **2.1**. *lexgen* takes in a vector of patterns and uses *dfm* to produce the equivalent deterministic machine. The implementation is then changed with *opt* and the procedure to do the lexical analysis is produced having been specialised to this machine. The procedure is an interpreter of the machine which requires two procedures to represent the input stream.

VII: lexan - This stores in the database an auxiliary procedure *lexan.builder* that can be used to aid the entering of an Lgen specification. Its database key and containing structure are descibed in **2.1.3**. It takes in a name and reads a specification out of a file with that name and then calls *lgen* to do the work.

3.6: Pgen Implementation

The input to Pgen is a vector of grammar rule specifications and a vector of required parsers. It then produces procedures to implement the parsers for each nonterminal required for the grammar. The parser constructed is SLR(1) ([AHO85] pp. 221) so the first step is to produce an SLR(1) automaton for each nonterminal for which a parser is required. Each of these automata can then be interpreted by the LR parsing algorithm. A blow by blow description follows.

I: **pgen.lex** - This file contains the specification of the lexical syntax used to enter the rules of the grammar.

II: **load.pgen.lex** - This program uses Lexan to build an analyser from the specification in pgen.lex and store it in the database under the key "pgen.lex" in a *lexan.box* structure (see section **2.1.3**).

III: **parse** - This program stores a procedure *build.grammar* in the database under the key "grammar.parse" in a

```
        structure parse.box (proc (*pntr -> pntr ) parse.place)
```

build.grammar parses the grammar rules and builds the structure that will represent the grammar internally. The grammar is represented by

```
        structure grammar.symbol (
                string gs.name;
                pntr gs.parser, gs.defn)
```

The field *gs.name* is the name of the nonterminal. The *gs.parser* field got designed out of use so it is currently redundant. The *gs.defn* field contains a list of right hand sides for this symbol. It is an instance of the structure class

```
        structure gs.opt (pntr ts, gs.rest)
```

where the second field is the rest of the list and the first is a list of tokens which make up this right hand side. The token list is an instance of

```
        structure token.seq (
                string ts.name;    ! (a)
                pntr ts.symbol,    ! (b)
                     ts.rest;      ! (c)
                bool assumable)    ! (d)
```

These fields are as follows

(a) the name of this token;

(b) a pointer (which is filled in after all rules have been parsed) to the grammar symbol representing this token; this is **nil** in the case of a terminal;

(c) the next token or the end of the list, that is, a *ts.end* (see below);

(d) a field which was used in an attempt to implement error recovery but has since become redundant.

The end of the list is not **nil** as with most linked lists. It is an instance of the class

```
        structure ts.end (
                string recognised;                   ! (a)
                int arity;                           ! (b)
                proc (*pntr -> pntr) ts.action)      ! (c)
```

where the fields are:

(a) the name of the grammar symbol that this is a right-hand-side of;

(b) the number of symbols on the right-hand-side;

(c) the procedure that will be used to evaluate the synthesized attribute that the parser will support.

The pointer that is returned to represent the grammar as a whole is then a pointer to the distinguished nonterminal of the grammar. This is the nonterminal that first had a rule declared for it. All the other nonterminals of the grammar should then be reachable from this. There is an example of the internal representation of a grammar symbol in section **3.8** (Diagrams).

IV: ff - ("first and follow" - [AHO85] pp. 188) This program stores a procedure *mk.follow.table* in the database under the key "first/follow" in a

```
structure ff.box (proc (pntr -> pntr) ff.place)
```

mk.follow.table produces a follow table for a given (sub-) grammar. The follow table is a table indexed by nonterminal name onto a list of terminal symbols that can immediately succeed the nonterminal. In order to produce the follow table it must first produce a table of possible first symbols for each nonterminal. The list of possible follow terminal names is contained in a list of instances of the structure class

```
structure follow.tok (string f.tok; pntr f.rest).
```

V: slr - This program stores a procedure *build.slr.machine* in the database under the key "build.slr.machine" in a

```
structure slr.box (proc (pntr, pntr -> pntr) slr.place)
```

build.slr.machine will take in a grammar symbol structure and a follow table and produce an SLR automaton for the grammar ([AHO85] pp. 221). The automaton is built by first building a set of closures. A closure is represented by the structure

```
structure closure (pntr c.items)
```

The field points to a list made up of instances of

```
structure mk.list (pntr list.entry, l.rest)
```

where each entry is an *item*, an instance of the structure class

```
structure item (
          pntr rule,          ! (a)
               i.ts,          ! (b)
               pos.in.ts)     ! (c)
```

where the fields are

(a) a pointer to the grammar object that this item is a right-hand-side of;

(b) a pointer to the head of the token sequence that this item represents;

(c) a pointer to the position of the "dot" on the right-hand-side. If the pointer is to a *ts.seq* then the dot is before the *ts.seq* being pointed at. If the pointer is to a *ts.end* then the pointer is to the far right of the current right hand side.

The SLR automaton is represented by a pointer to the first state of the automaton. All other states are therefore reachable. Each state is an instance of the structure class

```
structure slr.state (
          pntr slr.closure,        ! (a)
               slr.actions,        ! (b)
               slr.error,          ! (c)
               slr.reductions;     ! (d)
          int  slr.state.number)   ! (e)
```

The fields are as follows:

(a) a pointer to the closure that this state represents.

(b) a pointer to a table, indexed by input symbol, which gives the action to be taken given that input symbol.

(c) a pointer that was used while error recovery was being attempted (but since that was dropped, it has become redundant).

(d) a pointer to a table, indexed by input symbol, which gives the reductions to be performed on the input symbol if it is not possible to do an action on the input symbol. If the current input symbol appears in neither the action nor reduction table then it is an error token.

(e) a state number to uniquely identify each state.

VI: bpm - This program will store a procedure *build.parse.machine* in the database under the key "build.parse.machine" in a

```
structure bpm.box (proc (pntr -> pntr) bpm.place)
```

build.parse.machine will generate an SLR(1) parser automaton from a pointer to the structure representing the nonterminal of the grammar required. It first calls *ff* to produce the follow table for the required grammar symbol. *build.slr.machine* is then used to produce the parser automaton. A pair is then returned to to the caller (*pgen*) to give it the start and stop states of the machine.

VII: pgen - This program stores the procedure *pgen* in the database - its key and container structure are described in section **2.2**. *pgen* generates the parsers required for the given grammar. First the grammar is built using the grammar parser. For each of the required parsers the procedure *build.parse.machine* is used to build the automaton. For each automaton the procedure *pgen.apply* is specialised to produce a parser. This is put into a table to be returned to the caller of *pgen*. The procedure *pgen.apply* simply produces a procedure that, when given an instance of a lexical analyser, will interpret the automaton to parse it ([AHO85] pp. 216). In the event of any error, **nil** is returned.

3.7: Sgen Implementation

The input to Sgen is a specification in terms of a pointer to a lexical analyser, a pointer to a table of parsers and a vector of pointers describing the abstract syntax. Sgen examines (using the procedure *ast*) the abstract syntax specification and then specialises the editor executive to this specification.

3.7.1: The Editor State

The editor executive relies on one major structure to contain all the information it will ever need. This structure is an instance of the structure class

```
structure editor.state (
         pntr e.tree,            ! (a)
              e.current,         ! (b)
              e.parsers,         ! (c)
              e.la.gen,          ! (d)
              e.root.parser,     ! (e)
              e.command,         ! (f)
              e.ast.table,       ! (g)
              e.ast.root,        ! (h)
              e.screen,          ! (i)
              e.clip)            ! (j)
```

The fields are used as follows:

(a) A pointer to the root of the current abstract syntax tree.

(b) A pointer to the current node of the tree at any point during the editing session.

(c) A pointer to the parser table given to *pgen*.

(d) A pointer to the lexical analyser given to *pgen*.

(e) A pointer to the parser for the root of the tree.

(f) A pointer to the current command.

(g) A pointer to the table containing the internal representation of the specification of the abstract syntax.

(h) A pointer to the root of the internal representation of the specification of the abstract syntax.

(i) A pointer to the structure representing the screen for this edit. The structure will be described during the description of the file show.

(j) A pointer to the table being used as the clipboard.

3.7.2: The Editor Executive

This comprises four main parts:

I: show - This program stores all the screen manipulation functions in the database under the key "show" in a single structure:

```
structure show.box (
          proc (int, int -> pntr) initialise.screen.place;          ! (a)
          proc (pntr, string) message.place;                        ! (b)
          proc (pntr, string, string -> string) get.string.place;   ! (c)
          proc (pntr) show.place;                                   ! (d)
          proc (pntr, pntr -> pntr) get.mouse.node.place)           ! (e)
```

The functions provided are:

(a) *initialise.screen* - This procedure sets up the screen for this edit. It also notes various things about the size of the screen. The function returns a pointer to the screen descriptor which is an instance of the structure class

```
structure screen.box (
          pntr command.place;           ! enhanced command area of screen
          #pixel display.picture;               ! image for display area
          pntr display.place;           ! enhanced display area of screen
          #pixel spare.screen.place;          ! spare image same size as
                                                          ! display area
```

```
          pntr spare.packaged.place;         ! enhanced spare display area
          pntr screen.map.place;     ! map from mouse position to AST node
          int tree.up, tree.down)               ! how much screen to display
```

(b) *message* - Prints a message on the command area of the screen.

(c) *get.string* - Gets a string from the user. The string is displayed at the command area and is editable. This procedure uses the `seditor` written by Richard Cooper.

(d) *show* - A procedure which, when given an editor.state, will display the entire tree according to the unparsing scheme stored in the abstract syntax specification. This procedure also notes, in the `screen.map.place` field of the screen descriptor, what was printed where. This is to allow nodes of the tree to be selected by the mouse.

(e) *get.mouse.node* - A procedure which, when given an `editor.state` and a mouse position will return the lowest node of the abstract syntax tree that encloses the given position.

II: att - This program stores a procedure `attribute.tree` in the database under the key "attribute.tree" in a

```
structure att.tree.pack (proc (pntr) att.tree.place)
```

When `attribute.tree` is given a node of an abstract syntax tree it will attach all the attributes specified in the AST specification onto the tree. This is done for all descendant nodes in the tree. If any new value will propagate required changes up into the tree above the given node then all the propagation is done. If a circularity is come across then it will be detected. The procedure puts all the newly attached attributes (set to unevaluated) onto the subtree and constructs a queue of them. All the attributes that will be changed by propagation are also set to unevaluated and put onto the queue. It loops round this queue until all attributes have been evaluated. If all the arguments for an attribute are evaluated then the attribute itself can be evaluated, otherwise that attribute is replaced at the end of the queue. Attributes are represented by the structure:

```
structure att.instance (
        pntr att.value;             ! value assigned to this attribute
        pntr att.depend;     ! list of attributes depending on this one
        pntr att.node;       ! pointer to the node this attribute is on
        pntr att.declaration;     ! declaration of attribute in the AST
        bool att.valid)             ! has this attribute been evaluated?
```

The definition of the declaration of an attribute is given in section 3.7.3.

III: cmnd - This program stores two procedures in the database under the key "cmnds" in a

```
structure cmnd.box (proc (pntr, pntr) cmnd.get.place,
                                      cmnd.apply.place)
```

One gets and validates commands and the other executes them. Each command in the User's Guide (section 1.2) has a structure class associated with it, each class having fields containing attributes of the command. Look in cmnd for the grubby details.

IV: ast - This program stores a procedure *mk.ast* in the database under the key "mk.ast" in a

```
structure mk.ast.box (proc (*pntr, pntr -> pntr) mk.ast.place)
```

mk.ast takes in the specification of the abstract syntax tree and produces the internal representation. There are 6 sub-programs which load up lexical analysers and parsers used by *mk.ast*:

- for analysing specification of the abstract syntax rules
  - **load.ast.lex**
  - **ast.pgen**
- for analysing the specification of the attribution
  - **load.att.lex**

  - **att.pgen**
- for analysing the specification of the unparsing scheme
  - **load.up.lex**
  - **up.pgen**

3.7.3: Representing the AST Specification

The internal representation of the specification of the abstract syntax tree is contained in an instance of the structure class

```
structure ast.spec (
        string ast.name;         ! (a)
        *pntr ast.options;       ! (b)
        bool ast.parsable;       ! (c)
        string ast.parse.as;     ! (d)
        pntr ast.parser;         ! (e)
        bool ast.is.root;        ! (f)
        pntr ast.attributes)     ! (g)
```

(a) ***ast.name*** - This is a string containing the name of this nonterminal of the abstract syntax.

(b) ***ast.options*** - This is a vector of pointers to the various options for right-hand-sides for this type of abstract syntax tree node. Each one is an instance of class

```
structure ast.opt (
        string rhs.constructor;        ! (a)
        *pntr rhs.names;               ! (b)
        pntr rhs.unparse;              ! (c)
        pntr rhs.attributes)           ! (d)
```

(a) Name of constructor for this version of the node.

(b) There is one pointer (in order) for each grammar symbol on the right hand side of this rule. Each one is an instance of a structure in the class

```
structure rhs.object (
        string r.name;
        pntr r.where;
        bool is.terminal)
```

where the second field is **nil** if the symbol is a terminal and will be filled in with a pointer to the relevant *ast.spec* if it is not.

(c) This is a pointer to a list of things to print when unparsing this type of node. The list is in a structure

```
structure up.list (pntr up.action, up.rest).
```

Each element of the list is one of

```
structure up.literal (string upl)            ! a literal to print
structure up.attribute (string upa)   ! attribute value to print
structure up.child (int upc)       ! print unparse of child # upc
structure up.control (bool in.tab)            ! move tab in or out
structure up.hide (string uph)        ! as literal, but see below
```

The unparsing of attributes and *up.hide*s only takes place onto the screen during the editing. The final unparse to download the abstract syntax tree does not take any notice of these commands.

(d) This specifies the attributes that will be associated with each instance of this node. They are contained within a list

```
structure mk.Att.list (pntr Att.elem, Att.next).
```

each element being an instance of the class

```
structure att.rule (
        string att.name;              ! the name of this attribute
        pntr arg.list;                                ! see † below
        int arity;                                   ! rule's arity
        proc (*pntr -> pntr) att.eval)          ! rule's evaluator
```

† This is an ordered list indicating where the various arguments to the evaluator have to come from each time that an attribute of this type is evaluated. It is stored in a structure

```
structure mk.arg.list (pntr arg.where, arg.rest)
```

where each entry in the list indicates where the argument is to come from and is one of

```
structure arg.parent (string ap.name)
        ! attribute called ap.name of parent
structure arg.own (string ao.name)
        ! this node's attribute
structure arg.child.val (
        int acv.number;
        string acv.name)
        ! an attribute acv.name of child number acv.number
structure arg.child (int ac.number)
        ! child number ac.number
```

(c) ***ast.parsable*** - This indicates whether this type of node can be parsed or not.

(d) ***ast.parse.as*** - This indicates what to use to parse a node of this type, if it is in fact parsable.

(e) ***ast.parser*** - This is a pointer to the structure containing the parser.

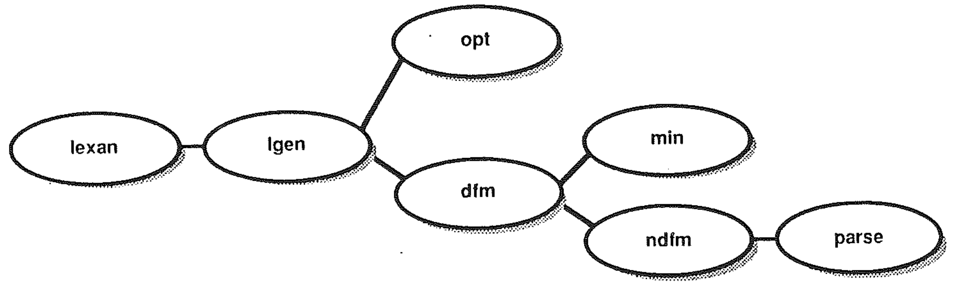
(f) ***ast.is.root*** - A **bool** to indicate whether this is the type of the root or not.

(g) ***ast.attributes*** - This is redundant. It was originally introduced to hold the specification of the attributes but that was long before any thought had actually been given to attributes. Clearly this is not the place to have this; it actually occurs in the *ast.opt* structure.

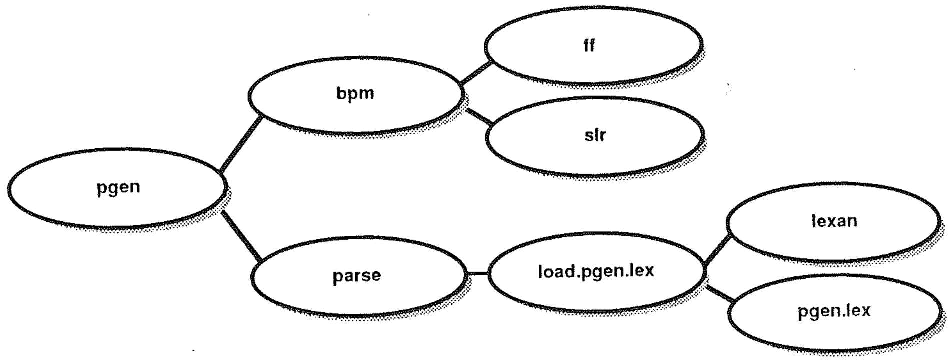
3.8: Diagrams

The diagrams here fall into two groups:

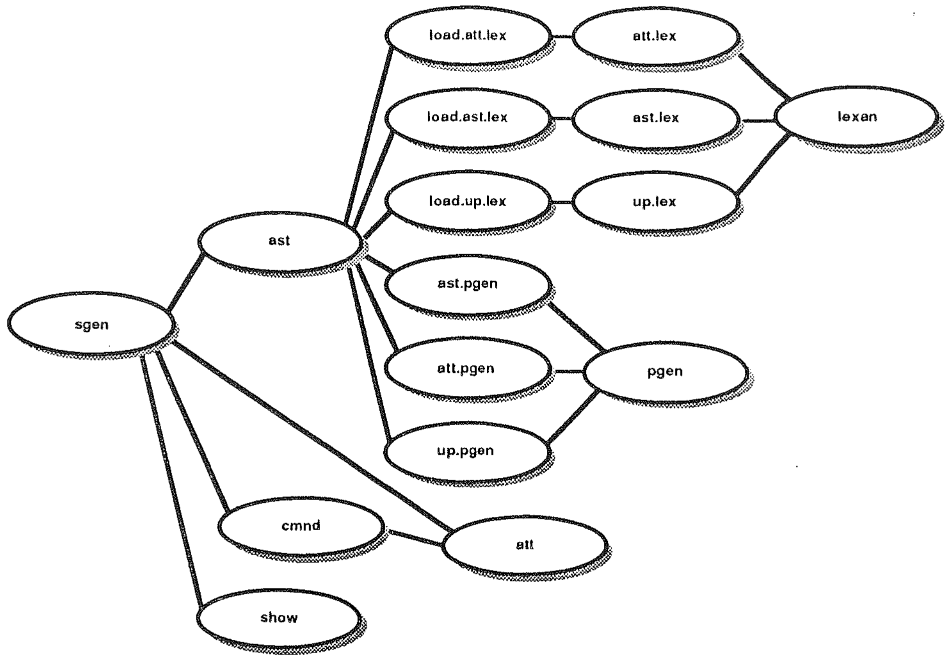
- The three procedure hierarchies;
- Some examples of data structures to clarify some of the explanations above.



The procedure hierarchy of Lgen

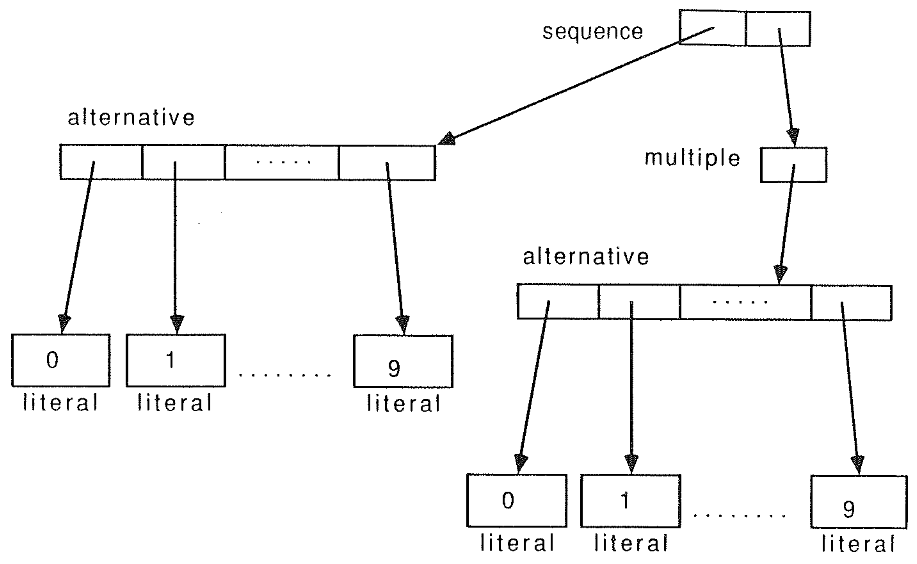


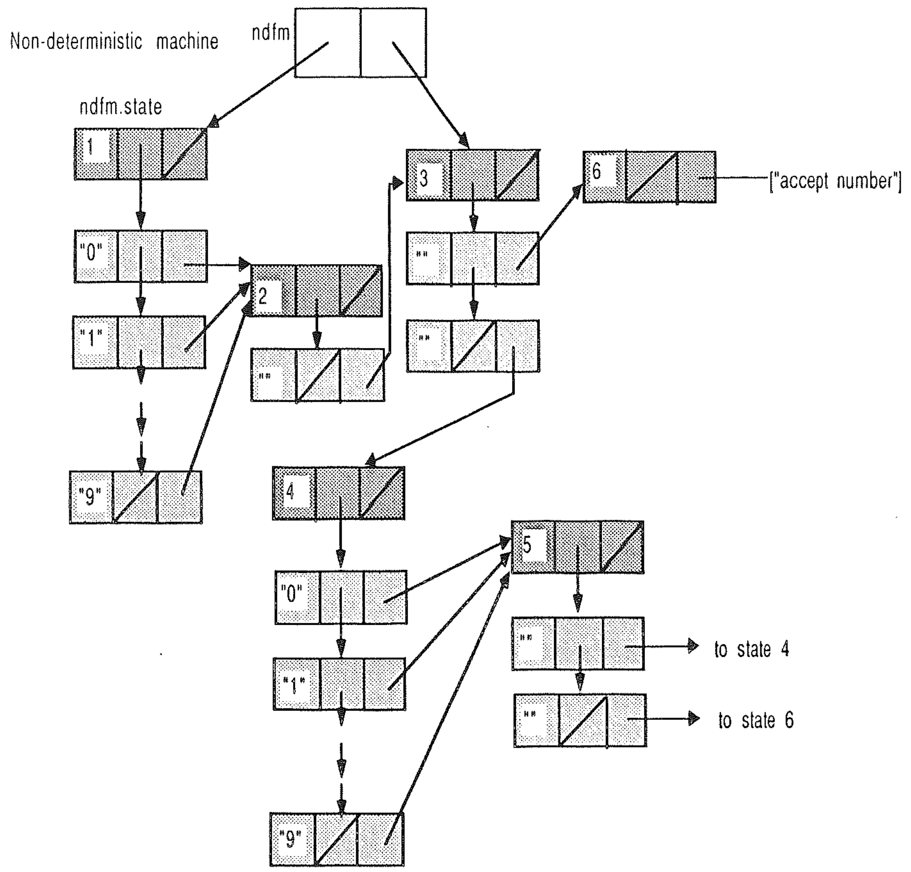
The procedure hierarchy of Pgen



The procedure hierarchy of Sgen

Example Regular Expression : positive integers





in the diagram above

▨ represents an instance on ndfm.state and

▨ represents an instance of ndfm.action

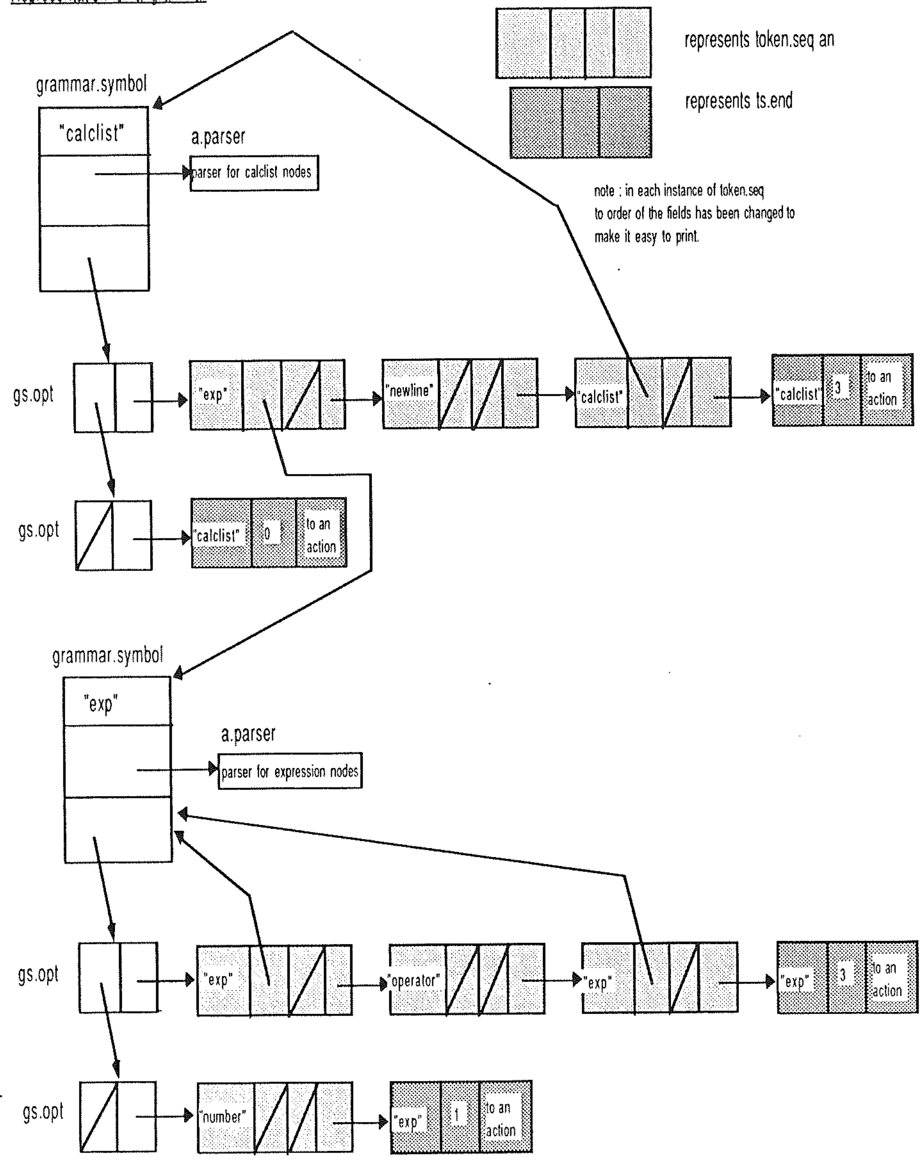
note : the order of the fields in ndfm.action has been switched to make it easier to draw.

Representation of a grammar

Instances of structure classes grammar.symbol and gs.opt are given in plain boxes. otherwise

▨ represents token.seq an

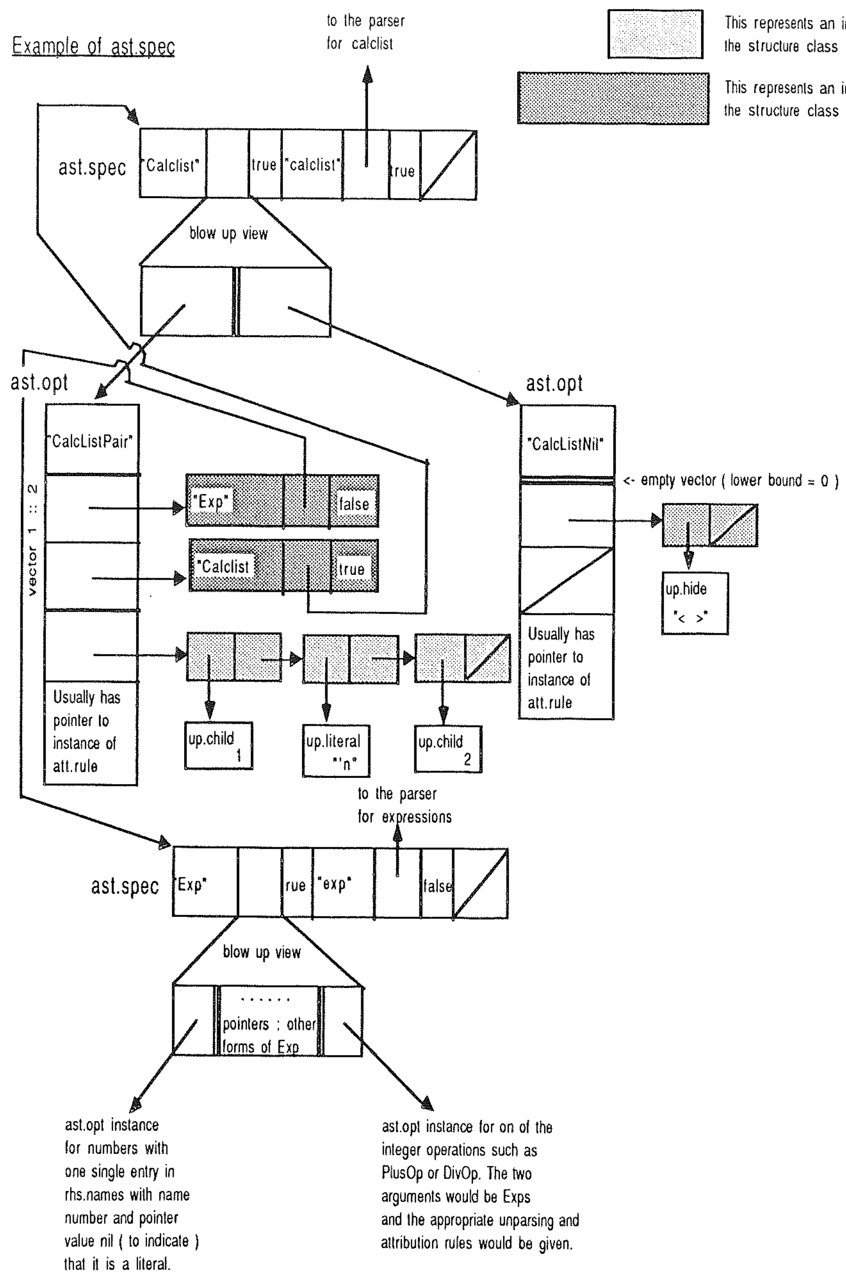
▨ represents ts.end


grammar.symbol
"calclist"
a.parser
parser for calclist nodes
note ; in each instance of token.seq to order of the fields has been changed to make it easy to print.
gs.opt
"exp"
"newline"
"calclist"
"calclist" 3 to an action
gs.opt
"calclist" 0 to an action
grammar.symbol
"exp"
a.parser
parser for expression nodes
gs.opt
"exp"
"operator"
"exp"
"exp" 3 to an action
gs.opt
"number"
"exp" 1 to an action

Example of ast.spec

▨ This represents an instance of the structure class up.list

▨ This represents an instance of the structure class rhs.object


to the parser for calclist
ast.spec
"Calclist"
true
"calclist"
true
blow up view
ast.opt
"CalcListPair"
vector 1 :: 2
"Exp"
false
"Calclist"
true
Usually has pointer to instance of att.rule
up.child 1
up.literal "'n"
up.child 2
ast.opt
"CalcListNil"
<- empty vector (lower bound = 0)
up.hide "< >"
Usually has pointer to instance of att.rule
to the parser for expressions
ast.spec
"Exp"
rue
"exp"
false
blow up view
......
pointers ; other forms of Exp
ast.opt instance for numbers with one single entry in rhs.names with name number and pointer value nil (to indicate) that it is a literal.
ast.opt instance for on of the integer operations such as PlusOp or DivOp. The two arguments would be Exps and the appropriate unparsing and attribution rules would be given.

4: Postscript on Applicability

4.1: Lgen and Pgen

Lgen provides a framework for building fast, easy to use, lexical analysers. It is relevant to all types of application where the input could be anything from a single line of text in response to an interactive prompt to an entire program in a high level programming language. In either case, the detail of establishing how to partition the input meaningfully is abstracted and the resulting code is clearer and more easily maintained. I would encourage programmers to familiarise themselves with the system as I have found it a great help in my subsequent programming work.

Similarly Pgen has wide applicability. Typically, given a string, which may be a single line or an entire program, the first step is to convert it into some internal representation of the meaning of the string. This can be done easily and quickly with parsers generated by Pgen and I would encourage its use in general programming. Again, it has been used extensively in the building of the Sgen system.

4.2: Sgen

Using Sgen is nontrivial. A detailed understanding of the underlying abstract syntax tree is required. However structure editors are clearly one way of controlling program development that may provide part of the key to improved programming techniques.

There is another use of structure editors that Sgen opens up as a field for future work. No-one seems to have used structure editors as part of a large interactive system where the structure editor generator was used by the programmer on frequent occasions when a structured response was required from the user. The current implementation of Sgen is too directed towards programming languages for

this application to be possible. Some simple changes could enable this. Among these we might mention:

(1) Passing the font and the display area as parameters. Other programming tools such as menu generators and string editors in PS-algol are made far more flexible by being generalised in terms of the screen; and Sgen also has the font size hard-coded in (in the screen descriptor described in section **3.7.2** part **I**). Abstracting these would be a (simple) significant step forward in the use of Sgen in large interactive systems.

(2) Transformation templates. The current set of commands available in Sgen is fixed. Transformation templates (see section 2.8 of [REPS85]) would provide a way for a programmer to specify new commands with the editor. These would take the form of "rewrite rule" procedures from one partial abstract syntax tree to another, each of which will always be valid in the same places. This idea has not received much thought but is clearly required if Sgen is to be used more generally.

(3) Specification checking. At the moment the amount of checking that is possible is tiny and the amount that is actually done is even less. The result is that specification errors lead to obscurely reported structure access errors at runtime.

Bibliography

Copies of documents in this list may be obtained by writing to:

The Secretary,
Persistent Programming Research Group,
Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ
Scotland.

or

The Secretary,
Persistent Programming Research Group,
Department of Computational Science,
University of St. Andrews,
North Haugh,
St. Andrews KY16 9SS
Scotland.

Books

Davie, A.J.T. & Morrison, R.
"Recursive Descent Compiling", Ellis-Horwood Press (1981).

Atkinson, M.P. (ed.)
"Databases", Pergamon Infotech State of the Art Report, Series 9, No.8, January 1982. (535 pages).

Cole, A.J. & Morrison, R.
"An introduction to programming with S-algol", Cambridge University Press, Cambridge, England, 1982.

Stocker, P.M., Atkinson, M.P. & Gray, P.M.D. (eds.)
"Databases - Role and Structure", Cambridge University Press, Cambridge, England, 1984.

Published Papers

Morrison, R.
"A method of implementing procedure entry and exit in block structured high level languages". Software, Practice and Experience 7, 5 (July 1977), 535-537.

Morrison, R. & Podolski, Z.
"The Graffiti graphics system", Proc. of the DECUS conference, Bath (April 1978), 5-10.

Atkinson, M.P.
"A note on the application of differential files to computer aided design", ACM SIGDA newsletter Summer 1978.

Atkinson, M.P.
"Programming Languages and Databases", Proceedings of the 4th International Conference on Very Large Data Bases, Berlin, (Ed. S.P. Yao), IEEE, Sept. 78, 408-419. (A revised version of this is available from the University of Edinburgh Department of Computer Science (EUCS) as CSR-26-78).

Atkinson, M.P.
"Progress in documentation: Database management systems in library automation and information retrieval", Journal of Documentation Vol.35, No.1, March 1979, 49-91. Available as EUCS departmental report CSR-43-79.

Gunn, H.I.E. & Morrison, R.
"On the implementation of constants", Information Processing Letters 9, 1 (July 1979), 1-4.

Atkinson, M.P.
"Data management for interactive graphics", Proceedings of the Infotech State of the Art Conference, October 1979. Available as EUCS departmental report CSR-51-80.

Atkinson, M.P. (ed.)
"Data design", Infotech State of the Art Report, Series 7, No.4, May 1980.

Morrison, R.
"Low cost computer graphics for micro computers", Software Practice and Experience, 12, 1981, 767-776.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"PS-algol: An Algol with a Persistent Heap", ACM SIGPLAN Notices Vol.17, No. 7, (July 1981) 24-31. Also as EUCS Departmental Report CSR-94-81.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Nepal - the New Edinburgh Persistent Algorithmic Language", in Database, Pergammon Infotech State of the Art Report, Series 9, No.8, 299-318 (January 1982) - also as EUCS Departmental Report CSR-90-81.

Morrison, R.
"S-algol: a simple algol", Computer Bulletin II/31 (March 1982).

Morrison, R.
"The string as a simple data type", Sigplan Notices, Vol.17,3, 46-52, 1982.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Progress with Persistent Programming", presented at CREST course UEA, September 1982, revised in "Databases - Role and Structure", see PPRR-8-84.

Morrison, R.
"Towards simpler programming languages: S-algol", IUCC Bulletin 4, 3 (October 1982), 130-133.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Problems with persistent programming languages", presented at the Workshop on programming languages and database systems, University of Pennsylvania. October 1982. Circulated (revised) in the Workshop proceedings 1983, see PPRR-2-83.

Atkinson, M.P.
"Data management", in Encyclopedia of Computer Science and Engineering 2nd Edition, Ralston & Meek (editors) January 1983. van Nostrand Reinhold.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Algorithms for a Persistent Heap", Software Practice and Experience, Vol.13, No.3, 259-272 (March 1983). Also as EUCS Departmental Report CSR-109-82.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"CMS - A chunk management system", Software Practice and Experience, Vol.13, No.3 (March 1983), 273-285. Also as EUCS Departmental Report CSR-110-82.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Current progress with persistent programming", presented at the DEC workshop on Programming Languages and Databases, Boston, April 1983.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"An approach to persistent programming", The Computer Journal, 1983, Vol.26, No.4, 360-365 - see PPRR-2-83.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"PS-algol a language for persistent programming", 10th Australian Computer Conference, Melbourne, Sept. 1983, 70-79 - see PPRR-2-83.

Morrison, R., Weatherill, M., Podolski, Z. & Bailey, P.J.
"High level language support for 3-dimension graphics", Eurographics Conference Zagreb, North Holand, 7-17, Sept. 1983. (ed. P.J.W. ten Hagen).

Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. & Morrison, R.
"POMS : a persistent object management system", Software Practice and Exerience, Vol.14, No.1, 49-71, January 1984.

Kulkarni, K.G. & Atkinson, M.P.
"Experimenting with the Functional Data Model", in Databases - Role and Structure, Cambridge University Press, Cambridge, England, 1984.

Atkinson, M.P. & Morrison, R.
"Persistent First Class Procedures are Enough", Foundations of Software Technology and Theoretical Computer Science (ed. M. Joseph & R. Shyamasundar) Lecture Notes in Computer Science 181, Springer Verlag, Berlin (1984).

Atkinson, M.P., Bocca, J.B., Elsey, T.J., Fiddian, N.J., Flower, M., Gray, P.M.D. - Gray, W.A., Hepp, P.E., Johnson, R.G., Milne, W., Norrie, M.C., Omololu, A.O., Oxborrow, E.A., Shave, M.J.R., Smith, A.M., Stocker, P.M. & Walker, J.
"The Proteus distributed database system", proceedings of the third British National Conference on Databases, (ed. J. Longstaff), BCS Workshop Series, Cambridge University Press, Cambridge, England, (July 1984).

Atkinson, M.P. & Morrison, R.
"Procedures as persistent data objects", ACM TOPLAS 7, 4, 539-559, (Oct. 1985) - see PPRR-9-84.

Morrison, R.,Bailey, P.J., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments", 8th International Conference on Software Engineering, Imperial College, London (August 1985), 166-172 - see PPRR-15-85.

Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 1-24 - see PPRR-16-85.

Davie, A.J.T.
"Conditional declarations and pattern matching", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 278-283 - see PPRR-16-85.

Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, proceedings of Data Types and Persistence Workshop, Appin, August 1985, 86-117 - see PPRR-16-85.

Buneman, O.P.
"Data types for data base programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 291-303 - see PPRR-16-85.

Cockshott, W.P.
"Addressing mechanisms and persistent programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 363-383 - see PPRR-16-85.

Norrie, M.C.
"PS-algol: A user perspective", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 399-410 - see PPRR-16-85.

Owoso, G.O.
"On the need for a Flexible Type System in Persistent Programming Languages", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 423-438 - see PPRR-16-85.

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
"A persistent graphics facility for the ICL PERQ", Software Practice and Experience, Vol.14, No.3, (1986) - see PPRR-10-84.

Atkinson, M.P. and Morrison R.
"Integrated Persistent Programming Systems", proceedings of the 19th Annual Hawaii International Conference on System Sciences, January 7-10, 1986 (ed. B. D. Shriver), vol IIA, Software, 842-854, Western Periodicals Co., 1300 Rayman St., North Hollywood, Calif. 91605, USA - see PPRR-19-85.

Atkinson, M.P., Morrison, R. and Pratten, G.D.
"A Persistent Information Space Architecture", proceedings of the 9th Australian Computing Science Conference, January, 1986 - see PPRR-21-85.

Kulkarni, K.G. & Atkinson, M.P.
"EFDM : Extended Functional Data Model", The Computer Journal, Vol.29, No.1, (1986) 38-45.

Buneman, O.P. & Atkinson, M.P.
"Inheritance and Persistence in Database Programming Languages"; proceedings ACM SIGMOD Conference 1986, Washington, USA May 1986 - see PPRR-22-86.

Morrison R., Dearle, A., Brown, A. & Atkinson M.P.; "An integrated graphics programming environment", Computer Graphics Forum, Vol. 5, No. 2, June 1986, 147-157 - see PPRR-14-86.

Atkinson, M.G., Morrison, R. & Pratten G.D.
"Designing a Persistent Information Space Architecture", proceedings of Information Processing 1986, Dublin, September 1986, (ed. H.J. Kugler), 115-119, North Holland Press.

Brown, A.L. & Dearle, A.
"Implementation Issuses in Persistent Graphics", University Computing, Vol. 8, NO. 2, (Summer 1986) - see PPRR-23-86.

Kulkarni, K.G. & Atkinson, M. P.
"Implementing an Extended Functional Data Model Using PS-algol", Software - Practise and Experience, Vol. 17(3), 171-185 (March 1987)

Cooper, R.L. & Atkinson, M.P.
"The Advantages of a Unified Treatment of Data", Software Tool 87: Improving Tools, Advance Computing Series, 8, 89-96, Online Publications, June 1987.

Atkinson, M.P, Morrison, R. & Dearle, A.
"A strongly typed persistent object store", 1986 International Workshop on Object-Oriented Database Systems, Pacific Grove, California (September 1986).

Atkinson, M.P., Morrison, R. & Pratten G.D.
"PISA : A persistent information space architecture", ICL Technical Journal 5, 3 (May 1987),477-491.

Atkinson, M.P. & Morrison, R.
"Polymorphic Names, Types, Constancy and Magic in a Type Secure Persistent Object Store". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Cooper, R. & Atkinson, M.P.
"Requirements Modelling in a Persistent Object Store". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Wai, F.
"Distribution and Persistence". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Philbrow, P.
"Associative Storage and Retrieval: Some Language Design Issues". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Guy, M.R.
"Persistent Store - Successor to Virtual Store". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Dearle, A.
"Constructing Compilers in a Persistent Environment". Presented at the 2nd Internaional Workshop on Persistent Object Stores, Appin, August 1987.

Carrick, R. & Munro, D.
"Execution Strategies in Persistent Systems". Presented at the 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

Brown, A.L.
"A Distributed Stable Store". Presented at the 2nd International Workshop on Persistent object Stores, Appin, August 1987.

Cooper, R.L., Atkinson, M.P., Dearle, A. & Abderrahmane, D.
"Constructing Database Systems in a Persistent Environment". Proceedings of the Thirteenth Internaional Conference on Very Large Databases, Brighton, September 1987.

Atkinson, M.P. & Morrison, M.
"Polymorphic Names and Iterations", presented at the Workshop on Database Programming Languages, Roscoff, September 1987.

Internal Reports

Morrison, R.
"S-Algol language reference manual", University of St Andrews CS-79-1, 1979.

Bailey, P.J., Maritz, P. & Morrison, R.
"The S-algol abstract machine", University of St Andrews CS-80-2, 1980.

Atkinson, M.P., Hepp, P.E., Ivanov, H., McDuff, A., Proctor, R. & Wilson, A.G.
"EDQUSE reference manual", Department of Computer Science, University of Edinburgh, September 1981.

Hepp, P.E. and Norrie, M.C.
"RAQUEL: User Manual", Department of Computer Science Report CSR-188-85, University of Edinburgh.

Norrie, M.C.
"The Edinburgh Node of the Proteus Distributed Database System", Department of Computer Science Report CSR-191-85, University of Edinburgh.

Theses

The following theses, for the degree of Ph. D. unless otherwise stated, have been produced by members of the group and are available from the address already given,

W.P. Cockshott
Orthogonal Persistence, University of Edinburgh, February 1983.

K.G. Kulkarni
Evaluation of Functional Data Models for Database Design and Use, University of Edinburgh, 1983.

P.E. Hepp
A DBS Architecture Supporting Coexisting Query Languages and Data Models, University of Edinburgh, 1983.

G.D.M. Ross
Virtual Files: A Framework for Experimental Design, University of Edinburgh, 1983.

G.O. Owoso
Data Description and Manipulation in Persistent Programming Languages, University of Edinburgh, 1984.

J. Livingstone
Graphical Manipulation in Programming Languages: Some Experiments, M.Sc., University of Glasgow, 1987.

Persistent Programming Research Reports

This series was started in May 1983. The following list gives those which have been produced at 17th September 1987. Copies of documents in this list may be obtained by writing to the addresses already given.

PPRR-1-83	The Persistent Object Management System - Atkinson,M.P., Bailey, P., Chisholm, K.J., Cockshott, W.P. and Morrison, R.	£1.00
PPRR-2-83	PS-algol Papers: a collection of related papers on PS-algol - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-5-83	Experimenting with the Functional Data Model - Atkinson, M.P. and Kulkarni, K.G.	£1.00
PPRR-6-83	A DBS Architecture supporting coexisting user interfaces: Description and Examples - Hepp, P.E.	£1.00
PPRR-7-83	EFDM - User Manual - K.G.Kulkarni	£1.00
PPRR-8-84	Progress with Persistent Programming - Atkinson,M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-9-84	Procedures as Persistent Data Objects - Atkinson, M.P. and Morrison, R.	£1.00
PPRR-10-84	A Persistent Graphics Facility for the ICL PERQ - Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. and Dearle, A.	£1.00
PPRR-11-85	PS-algol Abstract Machine Manual	£1.00
PPRR-12-87	PS-algol Reference Manual - fourth edition	£2.00
PPRR-13-85	CPOMS - A Revised Version of The Persistent Object Management System in C - Brown, A.L. and Cockshott, W.P.	£2.00
PPRR-14-86	An Integrated Graphics Programming Environment - 2nd edition - Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P.	£1.00
PPRR-15-85	The Persistent Store as an Enabling Technology for an Integrated Project Support Environment - Morrison, R., Dearle, A, Bailey, P.J., Brown, A.L. and Atkinson, M.P.	£1.00
PPRR-16-85	Proceedings of the Persistence and Data Types Workshop, Appin, August 1985 - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£15.00
PPRR-17-85	Database Programming Language Design - Atkinson, M.P. and Buneman, O.P.	£3.00
PPRR-18-85	The Persistent Store Machine - Cockshott, W.P.	£2.00
PPRR-19-85	Integrated Persistent Programming Systems - Atkinson, M.P. and Morrison, R.	£1.00
PPRR-20-85	Building a Microcomputer with Associative Virtual Memory - Cockshott, W.P.	£1.00
PPRR-21-85	A Persistent Information Space Architecture - Atkinson, M.P., Morrison, R. and Pratten, G.D.	£1.00
PPRR-22-86	Inheritance and Persistence in Database Programming Languages - Buneman, O.P. and Atkinson, M.P.	£1.00
PPRR-23-86	Implementation Issues in Persistent Graphics - Brown, A.L. and Dearle, A.	£1.00
PPRR-24-86	Using a Persistent Environment to Maintain a Bibliographic Database - Cooper, R.L., Atkinson, M.P. & Blott, S.M.	£1.00
PPRR-25-87	Applications Programming in PS-algol - Cooper, R.L.	£1.00
PPRR-26-86	Exception Handling in a Persistent Programming Language - Philbrow, P & Atkinson M.P.	£1.00
PPRR-27-87	A Context Sensitive Addressing Model - Hurst, A.J.	£1.00
PPRR-28-86b	A Domain Theoretic Approach to Higher-Order Relations - Buneman, O.P. & Ochari, A.	£1.00
PPRR-29-86	A Persistent Store Garbage Collector with Statistical Facilities - Campin, J. & Atkinson, M.P	£1.00
PPRR-30-86	Data Types for Data Base Programming - Buneman, O.P.	£1.00
PPRR-31-87	An Introduction to PS-algol Programming (third edition) - Carrick, R., Cole, A.J. & Morrison, R.	£1.00
PPRR-32-87	Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment - Morrison, R, Brown, A, Connor, R and Dearle, A	£1.00
PPRR-33-87	Safe Browsing in a Strongly Typed Persistent Environment - Dearle, A and Brown, A.L.	£1.00
PPRR-34-87	Constructing Database Systems in a Persistent Environment - Cooper, R.L., Atkinson, M.P., Dearle, A. and Abderrahmane, D.	£1.00
PPRR-35-87	A Persistent Architecture Intermediate Language - Dearle, A.	£1.00
PPRR-36-87	Persistent Information Architectures - Atkinson, M.P., Morrison R. & Pratten, G.D.	£1.00

PPRR-37-87	PS-algol Machine Monitoring - Loboz, Z.	£1.00
PPRR-38-87	Flexible Incremental Bindings in a Persistent Object Store - Morrison, R., Atkinson, M.P. and Dearle, A.	£1.00
PPRR-39-87	Polymorphic Persistent Processes - Morrison, R., Barter, C.J., Brown, A.L., Carrick, R., Connor, R., Dearle, A., Hurst, A.J.and Livesey, M.J.	£1.00
PPRR-40-87	Andrew, Unix and Educational Computing - Hansen, W. J.	£1.00
PPRR-41-87	Factors that Affect Reading and Writing with Personal Computers and Workstations - Hansen, W. J. and Haas, C.	£1.00
PPRR-42-87	A Practical Algebra for Substring Expressions - Hansen, W. J.	£1.00
PPRR-43-87	The NESS Reference Manual - Hansen, W. J.	£1.00
PPRR-44-87	Persistent Object Systems: their design, implementation and use. (proceedings of the Appin workshop August 1987) - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£20.00
PPRR-45-87	Delayed Binding and Type Checking in Database Programming Languages - Atkinson, M.P., Buneman, O.P. & Morrison, R.	£1.00
PPRR-46-87	Transactions and Concurrency - Krablin, G.L.	£1.00
PPRR-47-87	Persistent Information Space Architecture - PISA Club Rules - Atkinson, M.P., Lucking, J.R., Morrison, R. and Pratten, G.D.	£1.00
PPRR-48-87	An Event-Driven Software Architecture - Cutts, Q. and Kirby, G.	£1.00
PPRR-49-87	An Implementation of Multiple Inheritance in a Persistent Environment - Benson, P.J., D'Souza, E.B., Rennie, I.S., Waddell, S.J.	£1.00
PPRR-50-87	A Distributed Stable Store - Brown, A.L.	£1.00
PPRR-51-87	Constructing Compilers in a Persistent Environment - Dearle, A.	£1.00
PPRR-52-87	Lgen, Pgen and Sgen - Language Development Tools for a Persistent Programming Environment - Blott, S.M. and Campin, J.	£1.00
PPRR-53-87	Polymorphic Names and Iterations - Atkinson, M.P. and Morrison, R	£1.00
PPRR-54-87	A Requirements Modelling Tool Built in PS-algol - Cooper, R.L. and Atkinson, M.P.	£1.00
PPRR-55-87	A Persistent Software Database with Version Control - Cooper, R.L. and Atkinson, M.P.	£1.00
PPRR-56-87	User Interface Tools in PS-algol - Cooper, R.L., McFarlane, D.K. and Ahmed, S.	£1.00