1984

Ratpas documents

Michael P. Shepanski
University of Wollongong

RATPAS DOCUMENTS

by

Michael Shepanski

Department of Computing Science

University of Wollongong

Preprint No. 84.5 March 7, 1984

Submitted in partial fulfillment of the requirements of CSCI 321 Softward Project at the University of Wollongong in 1983

P.O. Box 1144, WOLLONGONG.N.S.W. 2500 Australia

Tel: (042) 270 859
Telex: 29022

INDEX

Ratpas Documents

Michael Shepanski

Software Project
University of Wollongong
Department of Computing Science.

This is a collection of papers related to the *Ratpas* programming language and its first implementation. The language was created by Paul Bailes of the University of Wollongong in 1982 and is now implemented on the UNIX* operating system by translation into *Pascal*.

The first paper is "*A Tutorial Introduction to Ratpas*", which serves to introduce Pascal programmers to programming in Ratpas as quickly as possible. It only attempts a brief sketch of the language, so users who require a more rigorous specification are directed towards the second paper, "*The Ratpas Report*" which gives a more complete description. The third paper, an entry for the UNIX programmer's manual, gives details of how to invoke the translator, and lists the discrepancies between specification and implementation. This completes the user documentation.

The fourth paper, "*Design of a Translator from Ratpas to Pascal*", describes the rationale of decisions that went into the design of the translator. This is intended to give systems programmers an understanding of the concepts which the design embodies and the reasons for the code assuming the form that it does, so that any modifications can be made in this perspective.

Since Ratpas is still an active research topic, the language for which the translator was designed is not the same as that described in the latest reports by Bailes. The implemented language is that which was current at the time when the project was begun. This is midway between that introduced in Bailes' "*A Rational Pascal*" as it appeared in the University of Wollongong Computing Science Department preprint no. 82/20, and that described in the edition yet to be published. In areas of confusion, the definitive reference is *The Ratpas Report*.

*UNIX is a Trademark of Bell Laboratories.

A Tutorial Introduction to Ratpas

Michael Shepanski

Software Project
University of Wollongong
Department of Computing Science

ABSTRACT

Ratpas [1][2] is an introductory language. For this reason, most instruction on the use of the language will usually be done concurrently with the teaching of fundamental concepts of computer programming. Rather than attempting such an ambitious pedagogical exercise, this document addresses the more modest task of introducing Ratpas to those already familiar with programming in Pascal [3]. It may, therefore, be taken as a guide for Ratpas instructors.

Getting Started

The simplest Ratpas programs are those without declarations and with no control structures except for a sequential composition of statements. These can be written simply by listing the statements, one after another. There is no need for any semicolons, **begins**, **ends**, program headings or full stops. For example, this is a simple Ratpas program:

```
write ('hello everybody')
write (' this is my first Ratpas program')
writeln writeln ('and programming in Ratpas is FUN.')
```

Note that it's bad practice to put more than one statement on a line; we just did it to show that you *can* if you really want to. A corresponding Pascal program would be:

```
program HelloWorld (input, output);

begin
write ('hello everybody');
write (' this is my last Pascal program')
writeln; writeln ('and programming in Ratpas is FUN.')
end.
```

The statements of the program have not themselves changed; it is usually so that Ratpas is the same as Pascal except in declarations and that it requires you to leave out bits of syntax that Pascal demands.

Declarations

Doing anything more challenging than writing out sequences of messages requires the use of declarations. The biggest change in this regard is that, in Ratpas, declarations go *after* statements. Also, the policy of eschewing syntactic detail is carried on to declarations: there are no headings like *const* or *type*, and of course no semicolons.

A constant declaration is formed by simply writing the name of the constant followed by an equals-sign (`=') and the value it is to have. The same thing applies for types, except that instead of a value, a type description goes after the equals-sign. To declare variables, write the name of the variable, then an equals-sign followed by the word *var*, then the type that the variable is to have. (e.g. `x = var integer', which you would read as `x is a variable of type integer').

Therefore, a simple interactive Ratpas program might look like this:

```
writeln ('Hello, what is your name ?')
readln (name)
writeln ('How old are you, ', name)
readln (age)
write ('You know, ', name, ', I think ')
writeln (date - age, ' was a good year for humans.')

name = var string
age = var 0..maxint
date = 1983
maximum_string_length = 60
string = array [1..maximum_string_length] of char
```

There a couple of additional points you may have noticed in this example. First, the declarations occur in a haphazard order. The fact is, when you are writing a list of declarations to go with a Ratpas block, you can write them down in any order you like. Second, one of the identifiers is rather long and contains the underscore character. You can always use identifiers like this in Ratpas, and it is a good idea to do so rather than using unreadable acronyms and abbreviations.

In Pascal, it is permissible to declare any number of variables by listing them all, separated by commas, on the left-hand side of a **var** declaration. You can do exactly the same thing in Ratpas, but you can also do it with constants and types, e.g.

```
a, b, c = 5
x, y = set of char
```

has exactly the same effect as:

```
a = 5
b = 5
c = 5
x = set of char
y = set of char
```

Procedures and Functions

The Ratpas syntax for declaring procedures is very different --much simpler-- from that of Pascal. For example,

```
write_results (i = val integer, x, y = val real) =
        {
        writeln ('And the answer is....')
        write (i, ':', x, ',', y, '.')
        }
```

declares a procedure **write_results** which takes two value parameters and performs the indicated statements. All of the formal parameters must be declared with either the keyword **val** (as shown) for value parameters, or the keyword **var** (as in Pascal) for reference parameters.

The same is true for function declarations. The only difference (since Ratpas has no keywords **function** and **procedure**) is that functions are declared with a result type, and return a value of that type. e.g.:

```
factorial (n : integer) : integer =
        {
        if n = 0
                return 1
        else
                return n * factorial (n-1)
        }
```

The **if**-statement will be explained shortly - the things to notice are:

(1) The result-type is declared after the formal parameters, just as in Pascal.

(2) Rather than assigning to the function-name, the value to be returned is indicated by an explicit **return** statement. This statement not only defines the value that results from the function being called, but also forces the function to stop immediately to return that value.

(3) The syntax of the function *call* is the same as in Pascal.

The latest Ratpas dialects (those post-dating the design of the current implementation) have abandoned the return statement altogether, and follow the Pascal convention of assigning to the function name and continuing execution. A convenient side-effect of the current implementation strategy is that assignments to the function name *are* permitted, so you have the choice.

An additional feature of Ratpas is the so-called *simple-function*. This is just a convenient way of entering functions that do not involve any significant processing steps, but are just shorthand for expressions. They correspond to FORTRAN's statement-functions. Instead of writing:

```
cotangent (theta = val real) : real =
        {
        return cos (theta) / sin (theta)
        }
```

write:

```
cotangent (theta = val real) : real =
        cos (theta) / sin (theta)
```

In general, a function whose body is just an expression with no braces is a short form for a function whose body (inside braces) is simply a return-statement with that expression.

As was noted before, declarations occur after statements. This applies, not only to the Ratpas program in which functions and procedures may be declared, but also within the functions and procedures themselves. For example, a Ratpas program to print cotangent tables may be written as follows:

```
for i := 1 to max
        writeln (i, cot (i))

cot (degrees = val integer) : real =
        {
        r := radians (degrees)
        return cos (r) / sin (r)

        r = var real
        radians (i = val integer) : real =
                {
                return i * pi / 180

                pi = 3.14159265358
                }
        }
i = var integer
max = 360
```

Ratpas Program Structure

This last example illustrates another important property of Ratpas; Unlike Pascal, Ratpas programs of any length very quickly assume a highly structured hierarchy of procedures and functions. This is due, not only to the desirability of structured programming, but also to Ratpas' lack of *begin*s and *end*s.

Each Ratpas block can be either a sequence (one or more) of simple statements (procedure-calls, assignments, etc), or a loop with a simple-statement body, or a conditional statement with only simple statements at each alternative. This means that sequences of loops and conditionals, or loops and conditionals over sequences, or nested loops and conditionals cannot be done `in line', and must be factored into procedures, which can be nested to any depth.

Differences in Statement Syntax.

As already illustrated, Ratpas *for*- and *while*-loops differ from their Pascal counterparts in that there is no keyword **do**. Conditional statements differ more substantially. The full form of the Ratpas *if* is like this:

```
if exprn1
        stmt1
elif exprn2
        stmt2
elif exprn3
        stmt3
...
else
        stmtn
```

which is equivalent to Pascal's :

```
if exprn1 then
        stmt1
else if exprn2 then
        stmt2
else if exprn3 then
        stmt3
...
else
        stmtn
```

The else-part (i.e. the word **else** and the statement following it) may be omitted, as may be any of the elif-parts.

The Ratpas equivalent of Pascal's **case** statement is still more different. In place of:

```
case exprn of
        const-list1:
                stmt1;
        const-list2:
                stmt2;
        ....
        const-listn:
                stmtn
        end
```

write:

```
switch exprn
        case const-list1
                stmt1
        case const-list2
                stmt2
        ....
        case const-listn
                stmtn
```

Skip, Abort and Goto

Ratpas does not have an empty statement in the style of Pascal's. The equivalent construct is **skip**, which does nothing when executed.

More significantly, Ratpas does not have a **goto** statement. Other than **return**, the only unstructured branch available in Ratpas is **abort**. This statement, when executed, simply causes the program to stop. Of course, since there are no explicit **goto** statements, there is no need of label declarations.

Other Differences between Ratpas and Pascal

There are three other areas in which Pascal and Ratpas are different.

Firstly, in Ratpas expressions, the lowest priority is given to the **or** operator, followed by **and**, then **not**, and then then the other Pascal operators. This means that the Ratpas expression

```
0 <= i and i < n or a[i] = ' ' or not flag
```

means:

```
((0 <= i) and (i < n)) or (a[i] = ' ') or (not flag)
```

Secondly, Ratpas *record* types have a different notation, following the trends of the above syntactic changes. Fields are declared with the equals-sign rather than the colon, as in variable declarations, and the syntax of variant records corresponds to that of **switch** statements above. A typical Ratpas type declaration might be like the following:

```
list = ^record
        mark = 0..3
        switch atomtag = boolean
                case true
                        car, cdr = list
                case false
                        prname = alfa
        end
```

This example illustrates the other difference between Ratpas and Pascal; it is not necessary to declare extra types to avoid self-reference. As long as types are not declared to contain themselves, and if all the types they reference are declared somewhere in scope, then the declarations are legal.

REFERENCES

[1] P. A. Bailes, *A Rational Pascal*, University of Wollongong Computing Science Dept. preprint no. 82/20.

[2] M. P. Shepanski, *The Ratpas Report*, ``The Ratpas Documents'', University of Wollongong Computing Science Preprint.

[3] K. Jensen and N. Wirth, *The Pascal User Manual and Report*, second edition, Springer-Verlag 1978.

The Ratpas Report

Michael Shepanski

University of Wollongong
CSCI321 - Software Project

ABSTRACT

In *A Rational Pascal* [1], Bailes put forward the rationale for a new programming language, for which an informal description was provided. This Report serves to amplify that description in a more formal --albeit less tutorial-- fashion.

1. Program Structure and Scope Rules:

Ratpas is a language for the expression of the creation of algorithms. A complete Ratpas program represents a history of programming refinements, such that each operation is either primitive or defined by another such refinement. The executable commands which thus refine a task are together called a *statement*. A statement will usually use names for data types, variables, constant values, expressions or other tasks which belong to it. The information which gives meaning to these names (also called *identifiers*) is provided in *declarations*, and a statement together with its declarations is called a *block*. A Ratpas program is one instance of a block*, i.e.

program -> block
block -> statement declaration*

In addition to the statement *using* some identifiers and the declarations *declaring* some, the declarations may also use identifiers (the meaning of `declare' will be explained by enumeration in section 5. Any occurrence of an identifier, other than a declaring occurrence, is a use of that identifier).

Because the refinement of tasks is hierachical, the construction of a block is necessarily recursive. This is embodied in certain declarations (see sections 5.6 & 5.7) containing blocks.

The scope of a declaration is the set of uses of identifiers which are bound to that declaration. Scope is defined as follows:

For any block B containing a statement S and declarations D_1, D_2, D_3,...D_n*:

(a) Any use in S of an identifier declared by some D_i is bound to the declaration D_i.

(b) Any use in some D_i of an identifier declared by some D_j is bound to the declaration D_j, unless it is bound in some block contained in D_j.

(c) Any use which is bound to any declaration D_i is said to be bound in B.

It is illegal for any identifier to be bound to more than one declaration. It is illegal for any identifier in a program not to be bound at all, unless it is a predefined identifier, in which case it assumes its predefined meaning (see section 6).

2. Statements:

A statement is a construct which iterates an action, or chooses among actions, or performs a sequence of actions. A single action is the simplest executable command.

statement -> conditional | loop | compound

---

* This syntax rule, and others throughout the report, use the metalanguage described in appendix A.

* Not all of these declarations need be present in the text of the block; there may be implicit declarations which also number among the D_i, i.e. for val parameters - see sections 3.2 & 5.4.

2.1. Conditional Statements:

Conditional statements choose and execute one action from a set.

```
conditional -> if-statement ! switch-statement
if-statement -> if if-branch (elif if-branch)*
        [else action]
if-branch -> expression action
switch-statement -> switch expression
        (switch switch-branch)* [default action]
switch-branch -> const-list action
const-list -> constant (, constant)*
```

The if-statement with branches $\langle e_1\ a_1\rangle$, $\langle e_2\ a_2\rangle$,.... $\langle e_3\ a_3\rangle$ evaluates (see section 4) e_1, then e_2 and so on until some e_i evaluates to the intrinsic Boolean constant **true**. Then a_i is executed. However, if no e_i evaluates to **true** then, if there is an **else**, the action following it is executed, otherwise no action is taken. It is illegal for any of the expressions to evaluate to any type other than **Boolean**.

The switch-statement with branches $\langle l_1\ a_1\rangle$, $\langle l_2\ a_2\rangle$,.... $\langle l_3\ a_3\rangle$ evaluates its expression and determines whether its value is equal to any constant in any l_i. If it is, then a_i is executed. If it is equal to none of the constants, then if there is a **default** then the action following it is executed, otherwise no action is taken. The expression must evaluate to a scalar type other than **real**, and all of the constants must be of that same type.

2.2. Loop Statements:

Loop statements perform an action repetitively.

```
loop -> while-loop ! repeat-loop ! for-loop
while-loop -> while expression action
repeat-loop -> repeat action until expression
for-loop -> for identifier := expression
        (to ! downto) expression action
```

A while-loop evaluates its expression. If it does not evaluate to a Boolean, the expression is illegal. If it evaluates to **false**, nothing is done. If it evaluates to **true**, the action is executed, and the while-loop is executed again.

The statement

repeat <a> until <s>

executes a, then executes the statement:

while not (<s>) <a>

The statement

for <i> := <e1> to <e2> a

evaluates the expressions e1 and e2, and assigns the value of e1 to the variable i (see section 3.1). If this value is greater than e2 (according to the `>' operator for the types involved), then i is assigned the value of e2. Otherwise, a is executed, followed by the statement:

for <i> := succ (<e1>) to <e2> a

The statement

for <i> := <e1> downto <e2> a

evaluates the expressions e1 and e2, and assigns the value of e1 to i. If this value is less than e2 (according to the `<' operator for the types involved), then i is assigned the value of e2. Otherwise, a is executed, followed by the statement:

for ‹i› := pred (‹e1›) downto ‹e2› a

It follows from these definitions that, in any for-loop, the identifier must be bound to a variable declaration (see section 5.3) with a type which is in the domain of the functions **succ** and **pred**, i.e. a scalar type other than **real**, and that the expressions must be of types which are the same as the identifier's type, or of which the identifier's type is a subrange.

Moreover, if the execution of a alters the value of i, or the the value of the evaluation of either e1 or e2, then the result is undefined.

2.3. Compound Statements:

compound -> action+

A compound is executed by executing each of its actions in textual sequence.

3. Actions:

An action is a step in processing which does not, *a priori*, involve any control structure.

action -> skip ! abort ! assignment !
procedure-call ! return-action

The action **skip** does nothing.

The action **abort** causes execution of the entire program to cease unconditionally, breaking any promises this report may have made about the behaviour of whatever statements are pending.

3.1. Assignment Actions:

An assignment action assigns a value to a variable.

assignment -> variable := expression

A variable is an expression which designates a storage location's identity, rather than its contents.

variable -> identifier selector*
selector -> [expression (, expression)*] !
. identifier ! ^

Furthermore, a variable must satisfy the same type rules as it would to be a legal expression. The variable and the expression must be of the same type, or one a subrange of the other, or one real and the other either integer or a subrange thereof.

If a variable has selectors which contain expressions, these are evaluated before the assignment is done.

The effect of the assignment statement is to ensure that next time (dynamically) the variable is evaluated as part of an expression, its value will be that which was assigned.

3.2. Procedure Calls:

A procedure call executes a named statement, possibly with parameter substitution.

procedure-call -> identifier [actual-parameter-list]
actual-parameter-list -> (expression (, expression)*)

The identifier must be bound to a procedure declaration (see section 5.4)*. Iff

---

* except for calls of predefined procedures, which behave quite differently, see below.

there is no actual parameter list, then there must be no formal parameter list in the procedure's declaration. If there is an actual parameter list, its expressions must be the same in number and types as the formal parameters in the procedure declaration. Where there are **var** parameters in the formal parameter list, the corresponding expressions in the actual parameter list must be variables (see section 3.1).

To execute the procedure call, the block in the procedure, modified as follows, is executed:

(1) For each **val** parameter p of type t, the declaration

<p> = var <t>

is added to the block's declaration list and, before execution of the procedure's statement, p is assigned (see section 3.1) the value of the evaluation of the corresponding actual parameter.

(2) For each **var** parameter in the formal parameter list, the corresponding actual parameter has any contained expressions evaluated and replaced by the constants resulting from their evaluation. This modified actual parameter is substituted for all occurrences of the formal parameter in the procedure's statement, with these exceptions:

(a) Identifiers' occurrences in the substituted actual parameters retain the bindings of their occurrences in the procedure-call.

(b) Where a for-loop variable is replaced by an actual-parameter which contains selectors, the resulting statement is legal despite the constraint that a for-loop variable be an identifier.

Where the call is of an intrinsic procedure, the rules for number and types of parameters, and the operations which will be done to them, depend on the procedure (see section 6).

3.3. Return Actions:

return-action -> *return* expression

A return-action evaluates its expression and returns its value (see section 5.5)

4. Expressions:

An expression is a rule by which a value can be computed from constants and the values of variables, using standard operators and named functions.

Ratpas expressions are very similar to the expressions in Pascal [2] (more precisely, the input strings in the category named `expression' in Pascal syntax rules). The only difference is that Ratpas gives lower priority to logical operators. Although the set of syntactically correct strings is extensionally the same, this semantic change is readily expressed by the following changes in syntax:

expression -> disjunct (*or* disjunct)*
disjunct -> conjunct (*and* conjunct)*
conjunct -> [*not*] nonlogical-expression

`Nonlogical expression' represents the same syntactic category as [2]'s `expression', with the following modifications:

(1) Any production which generates something containing the symbols **or**, **and** or **not**, is excluded.

(2) Any pattern including the name `expression' is understood to refer to the definition above.

The function of all the operators, including **or**, **and** and **not**, is retained in type, value and domain.

5. Declarations:

A declaration determines how a set of identifiers may be used in all occurrences which are bound to that declaration according to the scope rules (see section 1).

```
declaration -> constant-declaration | type-declaration |
        variable-declaration | procedure-declaration |
        function-declaration
```

5.1. Constant Declarations:

```
constant-declaration -> IDlist = constant
IDlist -> identifier (, identifier)*
```

A constant declaration declares the identifiers to the left of the `=', so that every bound occurrence will evaluate to the value specified by that constant. Constants are as specified under the name `constant' in [2].

5.2. Type Declarations:

```
type-declaration -> IDlist = type
```

A type declaration declares the identifiers to the left of the `=', so that each represents the specified data type. If the type is not a simple identifier, then the identifiers represent a type different from those bound to any other declaration whose type is not an identifier. If the type is an identifier, then the identifiers on the left-hand-side are interchangeable aliases for that type. In any case, they are interchangeable aliases for each other.

Types are as specified under the name `type' in [2], with the exception of record types, where the following syntactic changes apply (preserving semantics in the obvious ways):

(1) The production

```
record-section -> field-identifier
        (, field-identifier)* : type
```

is replaced by:

```
record-section -> field-identifier
        (, field-identifier)* = type
```

(2) The productions

```
field-list -> fixed-part ; variant-part
fixed-part -> record-section (; record section)*
```

are replaced by:

```
field-list -> fixed-part variant-part
fixed-part -> record-section+
```

(3) The productions

```
variant-part -> case tag-field = type-identifier of
        variant (; variant)*
variant -> case-label-list : ( field-list )
```

are replaced by:

```
variant-part -> switch tag-field type-identifier
        variant+
variant -> case case-label-list field-list
```

5.3. Variable Declarations:

variable-declaration -> IDlist = *var* type

A variable declaration declares the identifiers to the left of the `=' to be variables of a given type, so that each may be assigned values, be passed as a **var** parameter, and occur in expressions, evaluating to the last assigned value.

5.4. Procedure Declarations:

procedure-declaration -> identifier [formal-parameter-list]
= (block)
formal-parameter-list -> (formal-parameter-group
(, formal-parameter-group)*)
formal-parameter-group -> formal-parameter
(, formal-parameter)* = (*val*|*var*) formal-type
formal-parameter -> identifier
formal-type -> identifier

A procedure declaration declares the procedure identifier (the first identifier in the declaration) to be a name for a statement with declarations, i.e. a block. The type of a formal parameter is the type specified in its formal parameter group. Formal parameters in a group which contains the keyword **val** are referred to as *val-parameters*; those in a group which contains the keyword **var** are referred to as *var-parameters*.

Within the scope of the procedure identifier, calls may be made according to the semantics given in section 3.2.

5.5. Function Declarations:

function-declaration -> general-function-declaration |
simple-function-declaration
general-function-declaration -> identifier
[formal-parameter-list] : functype = (block)
simple-function-declaration -> identifier
[formal-parameter-list] : functype = expression
functype -> identifier

A function declaration declares the function identifier (the first identifier in the declaration) to name a computational rule which yields a result.

If an identifier is declared by a general function declaration, then any bound use in an expression (see section 4), possibly followed by an optional parameter list, is called a *function call* and is evaluated by executing the statement contained in the function-declaration's block, under the same substitutions and restrictions as for a procedure (see section 3.2), until a return-action is executed. Then, regardless of any statements pending, execution of the function terminates and the returned value is the value of evaluation of the function call. However, if one function's execution involves evaluating a function call which activates another function's execution, a return action will only terminate execution of the function most recently begun. It is illegal for a function to terminate without executing a return action.

Simple function declarations are a concise alternative form for functions which would only contain a return action. The declaration

<f> <p> : <t> = <e>

for some identifiers f and t, some expression e, and some p which is either empty or a formal parameter list, is equivalent to:

<f> <p> : <t> = (return <e>)

6. Intrinsic Identifiers:

The set of predefined, or *intrinsic* identifiers is the same in both content and meaning as that in [2].

7. Notation:

This section refers to the structure of language elements and the typographical rules for entering them.

7.1. Identifiers:

An identifier is any string of letters, digits and the underscore character (`_') which is not one of the reserved words in appendix B.

7.2. Keywords and Special Symbols:

The symbols italicised in all this report's syntax rules, and those in the cited rules of [2], are represented by the sequence of characters of which they are composed.

7.3. Numbers and Strings

Numbers and strings are represented the same as specified in [2].

7.4. White Space

Any number of spaces, tabs and advancements to new lines may be inserted between identifiers, keywords, special symbols, numbers and strings. They may not be inserted between characters of any identifier, keyword, special symbol or number. Strings may not be broken across lines. At least one space, tab, change of line or comment (see 7.5) must be inserted between identifiers, keywords, numbers and strings whenever its absence would yield an ambiguous sequence.

7.5. Comments

At any place where white space is permitted, the special symbol `(*', followed by any sequence of characters, followed by the special symbol `*)', is permitted, provided the sequence of characters does not include the special symbol `*)'. This construction has no effect on the meaning of the program.

7.6. Miscellaneous

Any sequence of characters which is not permitted by sections 7.1...7.6, is illegal.

REFERENCES

[1] P. A. Bailes, *A Rational Pascal* University of Wollongong Computing Science Dept. preprint no. 82/20.

[2] K. Jensen & N. Wirth, *The Pascal User Manual and Report* second edition, Springer-Verlag 1978.

Appendix A – Metalanguage Used in Ratpas Syntax Rules

Ratpas syntax rules are always written as a name, followed by the symbol `->', followed by a pattern. The name represents a category of input strings which are recognised as some meaningful component of the language, and the rule means that the strings in that category are those which match the pattern. `Match' is defined as follows:

(1) An italicised string is matched only by occurrences of that string.

(2) A name (not italicised is matched by any string in the category which the name represents.

(3) The juxtaposition of two patterns is matched by the juxtaposition of any string which matches the first with any string which matches the second.

(4) Two patterns with a `|' symbol between them are matched by any symbol which matches the first, and are also matched by any symbol which matches the second pattern.

(5) A pattern followed by the symbol `*' is matched by any (possibly empty) concatenation of strings, all of which match the pattern.

(6) A pattern followed by the symbol `+' is matched by any nonempty concatenation of strings, all of which match the pattern.

Where ambiguity arises in the application of these rules, parentheses (`(', `)') are used in the usual way.

Appendix B – Reserved Word List

abort and array case default div downto elif end file for if in mod nil not of or packed record repeat return set skip switch to until val var while

NAME

rp, rpi, rpix - Ratpas-Pascal translator

SYNOPSIS

rp [**-l**] name.r [**-f** filename ...]
rpi [**-l**] name.r [**-f** filename ...]
rpix [**-l**] name.r [**-f** filename ...]

DESCRIPTION

Rp translates the program in the file *name.r* leaving the Pascal code in the file *name*.p. The Pascal code can be further translated and executed using *pi* and *px*, or with *pix*.

Rpi performs a complete translation from Ratpas to Pascal interpreter code, invoking both *rp* and *pi*, and postprocessing the latter's diagnostics to make them correspond more closely to the Ratpas source. *Rpix* does all this and immediately executes the interpreter code for ``load and go'' Ratpas.

The `-l' option causes a program listing during translation. The `-f' flag permits a nominated list of filenames to be included in the program heading of the resulting Pascal program. This list should include the names of all files (other than the standard input and output) to which the source programs refers.

FILES

file.r	input file
file.p	Pascal equivalent of *file.r*
obj	interpreter code output (*rpi* and *rpix* only)

SEE ALSO

pi (1)
px (1)
``The Ratpas Report'' - unpublished report available in the Wollongong University Computing Science Department.

DIAGNOSTICS

Rp recognises and attempts recovery from lexical errors and a few semantic errors - those involving circular declarations.

Errors detected during parsing are output with a listing of the line and a flag indicating the point of the error.

The first character of each error message indicates its class:

F	Fatal error; translation cannot proceed.
E	Error; no code can be generated.
w	Warning - a potential problem.

No recovery is attempted after syntax errors, and most semantic errors are passed on to the Pascal implementation. The resulting diagnostics diagnostics are amiss in that:

(1) The identifiers have mutations, generated by rp to eschew underscores and avoid clashes with Pascal keywords.

(2) The line numbers bear no resemblance to those of the errors in the Ratpas source.

Rpi and *rpix* trap the Pascal implementation's diagnostics and make an effort to reverse these effects, but are not always successful - see ``BUGS'' below.

AUTHOR

Michael P. Shepanski.

BUGS

Default cases are not supported.

The `-f' option should not be needed - filename information is deducible from the source code. Moreover, filenames are restricted to legal Pascal identifiers, rather than to the wider class of legal Ratpas identifiers.

There is no recovery from syntax errors.

Those errors which are caught by the Pascal implementation produce diagnostics which are often confusing. Although *rpi* and *rpix* attempt to strip out line number references, some are left and the algorithm for restoring identifiers is easily confused by string literals.

Design of a Translator of Ratpas to Pascal

Michael Shepanski

CSCI 331 - Software Project,
Department of Computing Science,
University of Wollongong.

ABSTRACT

Ratpas [1], [2] is a programming language which embeds the semantics of Pascal [3] in a new syntax. This report first observes the key differences between Ratpas and Pascal, and proceeds to examine their implications for a translator. This is done to sufficient depth as to constitute both a design and a guide for those who wish to enter into the source code.

8. Preamble.

The translator is to embody a mapping between the input space of legal Ratpas programs and the output space of legal Pascal programs; in particular, to maintain semantic invariance. Both the input and output spaces are complex and consist in a large number of dimensions. Fortunately, the spaces are identical in so many of these that the semantic invariance of most dimensions is maintained without transformation. It is only those which differ significantly that bear on the translation task, and hence provide a rationale for design decisions.

9. Some Observations.

Ratpas differs from Pascal in six principal ways:

(1) There is less syntactic detail; in particular no semicolons.

(2) In each block, declarations occur after the executable statements.

(3) There are no keywords `const', `type', etc. to help classify declarations.

(4) In each block, declarations can occur in any order.

(5) Rules for inter-related type declarations are less strict.

(6) Identifiers may contain the underscore character.

(7) There are severe restrictions on the complexity of control structures allowable in a statement.

Of these, all but the last dictate approaches in the design of the translating process.

9.1. The Lack of Syntactic Detail.

Wirth [3] employed semicolons in Pascal to delimit logically separate program parts --especially statements-- in a way which is readily recognised by the parser (and perhaps also by the programmer). Not only semicolons, but also the keywords `do' and `then' serve to separate a construct ending with an expression, from the beginning of another statement. Moreover, the keywords `const', `type', `procedure' and `function' are director symbols which inform the compiler of the construct that is about to be parsed. In short, syntactic detail allows Pascal programs to be parsed deterministically top-down.

Ratpas, however, has a more austere syntax which can only be parsed bottom-up (this point is formally undecidable, but the difficulty of writing an LALR(1) grammar -- see below-- strongly suggests that top-down parsing would be extremely difficult; in particular that there is no LL(1) grammar). Fortunately, there is a bottom-up parser-generator, `yacc' [4], available. Yacc will generate parsers for any language for which an LALR(1) grammar is supplied. Yacc requires a sequence of production rules, and code to be executed as each rule is reduced. It has value-stacking facilities which make it particularly easy to build semantic trees.

Nevertheless, the Ratpas syntax is such that its obvious grammar, obtained by using non-terminal symbols for all the constructs that are meaningful to the Ratpas programmer, is not LALR(1). It is possible to write an LALR(1) grammar, but it must be allowed that parser actions will not all be semantically meaningful, and any `semantic tree' which is built will not faithfully reflect the structure of every construct in the source code.

9.2. The Placement of Declarations.

In each Ratpas block, declarations occur after the executable statements. This means that no type- or scope-checking can be done at each use of an identifier, because the declarations which would make the use legal have not yet been processed. In practice, this is not important because Ratpas identifiers can simply generate Pascal identifiers, so it can be left to the Pascal implementation to check the consistency of declaration with use.

Even so, the placement of declarations is still problematic because the object code has to be generated the other way around. Consider, for example, the pathological Ratpas program:

```
proc1
proc1 = (
    proc2
    proc2 = (
        proc3
        proc3 = (
            proc4
            proc4 = (
                proc5
                proc5 = (
                    skip
                )
            )
        )
    )
)
```

which corresponds to the Pascal program:

```
program pathological (input, output);
    procedure proc1;
        procedure proc2;
            procedure proc3;
                procedure proc4;
                    procedure proc5;
                    begin
                    end;
                begin
                proc5
                end;
            begin
            proc4
            end;
        begin
        proc3
        end;
    begin
    proc2
    end;
begin
proc1
end.
```

In this example, executable statements are generated in the order opposite to that in which they were read. Where such wholesale reshuffling of text is necessary, three classes of solution present themselves:

(1) The translator makes N passes through the source-file (or temporary files, or pipes), where N is dependent on the program being translated.

(2) Random access files are used.

(3) A structure is kept in memory, which can grow large if the translation task grows large.

The first of these is not preferred on the grounds of simplicity and efficiency. It is noted, however, that a completely general translator should take this approach.

The second possibility is rejected because it would be non-portable and, in all likelihood, quite complex.

Remark 1: In the search for an LALR(1) grammar, the following investigation was pursued: All the right-hand sides of productions were reversed, with a view to writing a scanner which would read the source program character-for-character backwards (possibly via a lower-level routine which would read block-for-block backwards, releasing a character at a time). It is interesting that this approach, if successful, would have simplified the translation of the above program. Unfortunately, though, there are other examples in which translation is more than just reversal, and so this would still not afford a one-pass solution. The investigation was abandoned because Ratpas is less susceptible to LARL(1) grammars than it is to LALR(1) ones. *End of remark 1.*

The third approach is accepted as an imperfect but workable solution. Since yacc parsers are adept at building trees, and a tree can be made amenable to all the declaration-sorting required (see sections 2.3 and 2.4), the parser will build a semantic tree which will be traversed later in such an order as to produce the Pascal code.

9.3. The Anonymity of Declarations.

Ratpas declarations are not tagged by illuminating keywords such as `const', `type', `var' or `procedure'. As already noted, this makes parsing more difficult. However, where this becomes crucial is in declarations like

a, b, c = d

which, depending on context (the declaration of d) could declare types or constants.

A necessary condition for a grammar to be LALR(1) is that it be context-free, so if a construct of this kind is to be reduced to a declaration, it can only be to one sort of declaration. Rather than reducing all such constructs to type-declarations or all to const-declarations, and coercing their internal representations afterwards, a more honest approach is to permit, for internal use only, a new class of declarations: renames. Thus, the internal representation (the semantic tree) is to contain six kinds of declaration: `const', `type', `var', `procedure', `function' and `ren' (for `rename'). After the tree is built, but before it is output, its renames must be eliminated. This is done by obtaining all necessary context information from the tree, and then replacing each rename by a const- or type- declaration as appropriate. The algorithm for doing this is discussed in section 6.

Remark 2: There is a different approach, which was considered at one stage: if a symbol table was structured so that the semantic subtree for each block referenced all occurrences of a given identifier through a common character-pointer, then a rename might simply change the character-pointers corresponding to the identifiers on the left-hand side, to point to the identifier on the right-hand side. This would effect a textual substitution, and no Pascal declaration would have to be generated in lieu of the rename. This approach was abandoned because it would be too difficult to prevent users from aliasing not only types and constants, but variables, procedures and functions as well. *End of remark 2.*

Remark 2a. It was considered that Ratpas should boast a new *feature*, permitting the aliasing of variables, procedures and functions. Discretion was thought the better part of valour. *End of remark 2a.*

9.4. The Ordering of Declarations.

The declarations in any Ratpas block may be entered in any order. In Pascal, they must be generated in the sequence:

```
const
type
var
function/procedure
```

This means only that the semantic tree should keep separate lists of these declarations for each block. More on this in section 4.

More importantly, const and type declarations must be generated with a certain ordering, so that no identifier is declared before those which on which its declaration depends. The const and type declarations must, therefore, be reordered. This is done after the renames have been eliminated, but before the tree is output. The algorithms are discussed in section 7.

Procedures and functions also have interdependencies, but this is easily overcome by Pascal's **forward** declaration feature. Forward declarations can be generated for every procedure and function before any procedures or functions are themselves output.

9.5. The Relaxation of Type-Dependency Restrictions.

Pascal has a rather obscure set of restrictions for type-declarations which refer to each other. For example, this is legal:

```
type
    list = ^node;
    node = record
        head : integer;
        tail : list
        end;
```

But this is not:

```
type
    list = ^record
        head : integer;
        tail : list
        end;
```

Ratpas requires only that the definitions be non-circular, and that all types used must be either primitive or declared somewhere within scope. Thus

```
    list = ^record
        head = integer
        tail = list
        end
```

is legal, but

```
    list = record
        head = integer
        tail = list
        end
```

is not.

The restrictions of Pascal are overcome by creating extra declarations wherever any nested structure is involved. Thus the legal Ratpas declaration shown above might, as an intermediate measure, generate:

```
type
    Y1 = record
        head : integer;
        tail : list
        end;
    list = ^Y1;
```

Of course, this creation of extra declarations must be done before they are re-ordered on dependencies. In fact, it need not be done as a separate pass, but can be accomplished during parsing so that all these artificial declarations appear in the semantic tree from its creation.

It is noted that creating extra declarations so liberally does much more than is necessary. The size of both the semantic tree and the generated code would be reduced if more intelligence were employed in deciding when to generate and when not. However, the above strategy is simple and has a high probability of correctness.

9.6. Underscores in Identifiers.

Ratpas identifiers are arbitrarily long sequences of letters, digits or occurrences of the underscore character (`_'), that begin with either a letter or an underscore. The version of Pascal which forms the output space (see [3]) permits identifiers, also of any length, but without underscores.

The conversion could be accomplished either through a symbol table, (which could map Ratpas identifiers into something like ("L00001", "L00002",...)), or by a one-to-one function from one set into the other, using a deterministic algorithm with no table. The latter alternative is preferred because:

(1) It is simpler, not requiring any new data structure.

(2) When the object identifiers are seen by a human, e.g. in error messages from the Pascal implementation, the correspondence can be more readily perceived, particularly if the conversion algorithm is written so as not to alter identifiers more than necessary.

(3) Possibly these error messages could be trapped by another process which would reverse the transformation. This is easier and more portable if a conversion table does not have to be passed between processes.

However, a mapping function alone is insufficient, because Ratpas and Pascal share a set of predefined identifiers, and care must be taken to ensure that they are not altered during translation. Therefore, a table will be needed, but it will be a simple static table of exceptional identifiers, which requires no runtime maintenance and may appear in any number of processes without runtime communication.

10. Conclusions So Far.

The translator is to center its attention on a data structure called the semantic tree, and will do its processing in four stages:

(1) Make one pass through the source code, creating the semantic tree.

(2) Traverse the semantic tree, eliminating all record of rename-declarations.

(3) Traverse the semantic tree, topologically sorting const- and type-declarations of each block according to their interdependencies.

(4) Traverse the semantic tree, generating Pascal code.

Stages 2, 3 and 4 could perhaps be integrated into one traversal, but it is thought worthwhile to keep this modularity, at least during program development.

Surprisingly, not much has been said yet about the task of actual translation of statements from Ratpas to Pascal. In fact, this is one of the simpler jobs, and is largely done in pass 1 (see below). Most of the difficulty of translation is in stages 2 and 3, and in pass 1 preparing the data structure to make these stages possible.

11. The Semantic Tree.

Syntactically, a Ratpas program is an instance of the *block* construct, which also occurs in procedures and functions. Therefore, the semantic tree of a Ratpas program will be of a general structure used for representing blocks:

The semantic tree for a block will consist of the semantic tree of the declarations local to that block, and a representation of the Pascal text of the executable statements of the block.

> *Note:* Representations of Pascal text are explained more fully later on. For the moment, the word *'picture'* will be used to denote internal representations of Pascal text, for reasons which will be revealed in section 4.2. *End of note.*

The semantic tree of a set of declarations in a block is, in some respects, a symbol-table local to that block. To avoid confusion with the symbol table described in section 9, however, it will be referred to as the semantic tree of a set of declarations. It consists of six lists of declarations' semantic trees: i.e. those belonging to consts, types, vars, rens, procedures and functions.

The semantic tree of a constant declaration consists of a list of constant-identifiers belonging on the left-hand side, at most one identifier on which the declaration depends, and a picture of the right-hand side.

The semantic tree of a type declaration consists of a list of identifiers belonging to the left-hand side, a list of type-identifiers on which the declaration depends, and a picture of the right-hand side.

The semantic tree of a variable declaration consists of a list of identifiers belonging to the left-hand side, and a picture of the right-hand side.

The semantic tree of a rename declaration consists of a list of identifiers on the left-hand side, and an identifier which belongs on the right-hand side.

The semantic tree of a procedure or function declaration consists of an identifier for the name of the function, a picture of the other information which is to be output when the procedure/function header is printed, and the semantic tree of the block which is contained in the procedure or function. The semantic tree of this block is, recursively, of the form just described.

A tree of this form, once created, furnishes stages 2, 3 and 4 with all the information they need. In order to implement this, however, it is necessary to further examine the data types *list* and *picture*.

11.1. Lists.

The Ratpas translator abounds with linear lists. Declarations contain lists of identifiers, the semantic tree contains lists of declarations and there are other other lists yet to be seen (see sections 5.2.1 and 9).

The implementation language (C [5]) does not support lists as primitive data types, so they must be implemented. However, if lists of every required type were to be implemented by a dedicated set of functions, then these functions would increase the size of the translator by no small amount. Therefore, it is proposed that a set of generic list macros be written which will perform the standard list functions on any desired type. How they are to be written is rather fiddly, and depends upon quirks of the C macro processor. Nevertheless, their performance may be succinctly specified thus:

For any type-identifier t occurring in the translator, the call:

LISTGEN (t);

will permit list operations to be done on that type throughout the C source file in which the LISTGEN occurs. In particular, lists of t's may be declared with the pseudo-type-identifier:

listof (t)

The function

nil (t)

returns an empty list of t's, and the pseudo-functions

cons (t, i, l)
car (t, l)
cdr (t, l)

perform the normal operations on an item i and list l, where t is, again, the item type. A list can be tested for emptiness by comparisons with *nil* (t). Moreover, *car* (t, l) and *cdr* (t, l) can be used as *lvalues*, i.e. the objects of assignment.

11.2. Pictures.

It has been observed (section 2.1) that the Ratpas text is to be parsed bottom-up, and that the reductions will not always be in semantically useful steps. Moreover, it is the task of the parser to do translation of executable statements and other parts which do not have to be reshuffled by stages 2 and 3. What, then, does this suggest regarding requirements of the representation of Pascal text being constructed?

The bottom-up phenomenon suggests that it will be necessary to create elementary representations (say, for Pascal tokens), and also possible to construct a larger representation out of smaller ones. However, since the order in which these constructions are done is virtually meaningless, it is thought prudent to be consistent and not make this information available outside the module which manipulates these structures.

It is for this reason that they are called pictures - a picture is not to be considered as a smaller semantic tree, but rather as an image of a sequence of tokens, similar to text on a page, which would have to be re-parsed if it were to to be understood. The only means of examining a picture is by sequential traversal of the tokens out of which it is constructed.

Therefore, pictures are encapsulated with access through only the following functions:

```
pic (s)
cat (p1, p2, p3, ...pn, 0)
traverse (p, f)
```

The function *pic* (s) returns a picture of the character string s. The function *cat* (p1, p2, p3, ...pn, 0) returns a picture of the concatenation of the pictures p1, p2, p3, ...pn (the 0 is needed for implementation reasons, since the function takes a variable number of parameters). The function *traverse* (p, f) calls the function f once for each token in the picture p, taken in sequence. At each call, the appropriate token (character string) is passed as an argument to f.

These functions serve all the requirements that the translator has of pictures.

12. The First Pass.

The first pass of the Ratpas translator is to read the input text and build a semantic tree of the form described in section 4, and with the constraint that none of the type-declarations will have nested structure.

12.1. The Scanner.

In order to read the text, it will be necessary to employ a lexical analyser, or scanner.

The interface between the scanner and the parser is quite strictly defined by the properties of yacc, and requires that the scanner be called as a function which returns the next token on the the input. Where more information is required (e.g. it it is insufficient to determine that a token is *an* identifier, the parser must be told *which* identifier), the convention will be maintained that the scanner will not attempt any translation, but will supply the token as read.

The interface between the source-file and the scanner must be such as to satisfy the following requirements:

(1) The scanner can `push back' up to two characters of input which are not white space (this follows from an examination of the regular expressions of numbers).

(2) The current line of text is available to be printed together with error messages.

The simplest way to supply these facilities is to have a module which buffers a line, and releases it one character at a time. This input-buffering module will give access to the line directly, i.e. by allowing other modules to use its name (this is necessary for sophisticated error-message output), but will provide the functions:

```
getch ()
ungetch (c)
echo ()
```

to get the next character (replacing the buffer if necessary), to push a character back, and to print out the current line respectively. Where possible, the discipline of only accessing the line through these routines will be maintained. In any case, the input-buffering module (in particular its internal routine for refreshing the buffer) will completely own the real input stream.

12.2. The Parser.

The semantic tree consists principally of pictures and declaration records. In the yacc environment, it is easy to construct pictures as rules are reduced, and to make the pictures reflect the comparatively trivial syntactic changes needed to effect much of the translation. For example, Ratpas has a syntax-rule:

while-stmt -> **while** exprn action

Since the parser operates bottom-up, the expression and action will already have been processed and will have made their pictures available. Therefore, on reducing this rule, the parser must just concatenate:

(1) A picture of the Pascal keyword `while'

(2) The picture of the expression, which has already been constructed

(3) A picture of the Pascal keyword `do'

(4) The picture of the action, which has also already been constructed.

and the picture thus formed will be automatically stacked and made available to the rule by which the while-stmt is further reduced. This mechanism can be similarly applied to all the simple syntax changes, and even to semantic change, i.e. inserting parentheses in expressions to reflect the difference in operator hierarchy.

12.2.1. Parsing Declarations.

Producing the semantic tree of declarations (see section 4) is more involved. In constant declarations and renames, the information needed for the record is all immediately available in the tokens from which the declaration is reduced. (Remembering that constant-declarations and renames (see section 2.3) are to be parsed as completely separate nonterminals, the grammar for the right-hand side of constant-declarations must be framed so as to avoid producing single identifiers).

Of all the different kinds of declaration record, types are probably the most difficult to create. The left-hand side identifiers are immediately available, and it is possible for a picture of the right-hand side to be made available by creating pictures as types are reduced. It is necessary to construct, as the right-hand side is parsed, a list of all dependent types. This list cannot be global because types can be nested. Therefore, each rule will stack a list of identifiers on which it depends. Since the productions must return both a picture and a list of identifiers, and since yacc only permits a single result to be stacked, all the rules which produce types, or parts thereof, must consistently stack a pair - which can be implemented in a list. Moreover, since type-declarations and ren-declarations are distinct, the grammar for the right-hand side of a type declaration must produce all Ratpas types *except* lone identifiers; the rules must be drafted carefully to enforce this.

The parser must also adhere to the constraint that the tree produced will not represent types with nested structure. In order to accomplish this, each time a structured type is parsed, an artificial type-declaration is created, and the pair returned is as though the invented type-identifier had occurred in place of the structured type. In order that it be possible to `create' declarations like this, at places in the grammar other than the productions of declaration-lists, it is necessary that the semantic tree of the current set of declarations be available for additions at all times. Because declarations can be nested, these semantic subtrees must be kept in an explicit stack (which can be implemented in a list). The stack will be pushed at the start of each block (it could be pushed at the start of each declaration-part, but this point is harder to recognise syntactically), and popped at the end of each block.

Var-declarations are simple to parse, once types have been handled. For procedure and function declarations, also, the information needed for the declaration records will all be readily available on the parsing stack.

13. The Second Pass.

The second pass of the translator must purge the semantic tree of all rename declarations. It does this at the root level of the tree (for declarations in the outermost block), and then recursively applies the mechanism to all the blocks in the functions and procedures declared in that block.

At each block, the algorithm is of the following general form:

```
while there are renames left:
    for each rename:
        try to eliminate it.
```

where `try to eliminate it' means to search for the right-hand side identifier in all the const- and type- declaration lists in the current block, and then in successive parent blocks; if this identifier is found, then the rename will be replaced by a const- or type-declaration accordingly (i.e. it will be deleted and an equivalent declaration of the nominated type will be added to the current block). The `for each rename:' loop must detect the condition that a complete scan of the renames is made without any progress. If this occurs, then none of the right-hand sides is known to be either a type or constant, so either the declarations are circular or a necessary identifier is undeclared, so an error is recognised.

Afterthought: Since it is necessary to chase declaration-lists in successive parent-blocks (to reflect scope-rules), these parent-blocks must be referencable. The second pass could keep an explicit stack of pointers to roots of trees and subtrees, but it should be simpler to alter the parser so as to provide a parent-link in the semantic subtree of each block. *End of afterthought.*

14. The Third Pass.

The third pass must reorder the const- and type- declaration lists so each declaration occurs before all those which depend on it. The traversal strategy will be the same as for pass 2, i.e. top-down. At each subtree, the following algorithm will be performed:

```
while there are const-declarations in the unsorted constant list:
    for each const-declaration in the unsorted constant list:
        attempt to move it into the sorted constant list.
while there are type-declarations in the unsorted type list:
    for each type-declaration in the unsorted type list:
        attempt to move it into the sorted type list.
```

Here, attempting to move something into a particular sorted list means searching that sorted list and successive corresponding parent lists for each identifier on which it depends; if it is found, then that declaration is moved into the sorted list. In each case, a check must be made that a traversal of an unsorted list is never completed without any progress, just as in the second pass.

15. The Fourth Pass.

The fourth pass must write out the Pascal program corresponding to the semantic tree. This is, for the most part, a simple top-down traversal; at each block, the declarations are printed out in the order

```
const
type
var
procedure forwards
function forwards
procedures
functions
```

(for debugging purposes, the fourth pass might also write any rens which are left), and then traverses the picture of the block's body. Some indentation can easily be provided, e.g. printing each procedure/function at one tab-stop further than its enclosing block. Formatting pictures is a more serious matter ([6]), since they do not provide any semantic structure. Here, all formatting must be done at a lexical level, e.g. by recognising semicolons.

Before any code is written out, Pascal requires a program heading, which is to contain a program name (an identifier with no semantics attached) and a list of all

variables of type `file' used by that program. The program name is easily created, either from information available to the translator at invocation or from more arbitrary sources. The list of file variables is somewhat harder and will, in this implementation, not be tackled adequately. Rather, the user must supply this information explicitly, either by editing the object program or, preferably, supplying the list to the translator on invocation. The second alternative is preferred, with the understanding that the filenames `input' and `output' will be taken as default.

16. The Symbol Table.

The design, to this point, is workable and a translator could be implemented without the need of a symbol table. However, since for each identifier occurrence in the output text a character-string is to be stored in the semantic tree, and since many of these will be the same in content, it would be wasteful of storage.

Therefore, a symbol table should be constructed, which contains a common copy of every identifier (both users' identifiers and generated ones). Then, every time a Pascal identifier is generated (either in the routine that translates Ratpas identifiers, or in the routine that generates unique ones for type declarations), a search is made of the symbol table. If the identifier already exists, then the symbol-table entry is used to reference it. If not, it is inserted.

Since the symbol-table is only a store-saving device, it will be oblivious to scope rules and it need not carry any information except the characters of which identifiers are composed. Nonetheless, this approach does have the advantage that, in passes 2 and 3, where searches are done to locate identifiers in the semantic tree, comparisons need only be done by character-pointer, not character for character.

The symbol-table implementation need only have one primitive, a function

insert (string)

which stores a given character string if it is new, and in any case returns a pointer to the common copy.

For sake of simplicity, implementation will be via a fixed-size hash table, with each position referencing a list of clashing identifiers.

REFERENCES:

[1] M. P. Shepanski, *The Ratpas Report*, submitted to P. A. Bailes, University of Wollongong Computing Science Dept. August 1983.

[2] P. A. Bailes, *A Rational Pascal*, University of Wollongong Computing Science Dept. preprint no. 82/20.

[3] K. Jensen and N. Wirth, *The Pascal User Manual and Report*, second edition, Springer-Verlag 1978.

[4] S. C. Johnson, *Yacc: Yet Another Compiler Compiler*, Computing Science Technical Report No 32, 1975, Bell Laboratories Murray Hill, New Jersey 07974 .

[5] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).

[6] P. A. Bailes and A. Salvadori, *A Semantically Based Formatting Discipline for Pascal*, to appear in "Software - Practice and Experience".