SOFTWARE TOOLS NOTES

VOLUME 1 NUMBER 1 JANUARY 1983

THE AUSTRALIAN JOURNAL OF PROGRAMMING METHODS FOR TECHNICAL SOFTWARE

SOFTWARE TOOLS NOTES

Volume 1 Number 1 January 1983

Contents

The focus of *Software Tools Notes* is on programming methods for technical and engineering software.

It is particularly orientated towards the use of software tools to enhance the Fortran language.

The large base of public domain work on Software Tools in Ratfor, based on the book *Software Tools* by Kernighan and Plauger and developed by the Software Tools Users Group in the U.S.A. is drawn upon for material and inspiration.

Additionally, however, *Software Tools Notes* will cover the application of these techniques to scientific/engineering computing applications.

Contributions of a non-academic nature are encouraged. Where possible a methodology, technique or utility should be described so that readers may glean enough knowledge to be able to apply it in their own work if it seems useful. Source code for utilities or programs is welcomed and may be published if it is of sufficient quality and merit.

Address all correspondence to

The Editors,
Software Tools Notes,
197 Alma Road,
East St Kilda, Victoria, 3182.
Australia.
Telephone (03) 527 5881

Editors

Desmond FitzGerald, PhD.
Paul Howson, B. Eng.

Subscription rates:

Australia $40 for 4 issues.
Overseas $aus50 for 4 issues.

Editorial

WHILST THE "commercial" computing sector is experiencing a lot of innovation with new techniques and "fourth generation" languages, the "technical" computing field seems largely unaware of how it might refine its methodologies, be more productive and provide a more satisfying type of work for the practitioners involved.

The Bell Laboratories Unix™ operating system and its derivatives, in particular the *Software Tools* methodology, based on the book of the same name by Kernighan and Plauger, is increasingly a guiding direction in technical computing. Its adoption leads progressively towards machine independent, highly readable and maintainable code.

As our experience with Software Tools has grown and we have met others interested in this field and have had to communicate the philosophies and methods of Software Tools to them, the idea has kept recurring that some sort of regular publication discussing this area would be worthwhile.

Consequently, we have decided to publish this newsletter four times per year. Typesetting and printing are not cheap nowadays, so we have to ask for a subscription fee to cover production costs.

In the technical field, development and maintenance of code often rests with people having a minimum of programming training. The highly secretive and competitive approach widespread in the computer industry, acts to promote ignorance and confusion regarding improved programming methodoligies. We now live in an age of rapid technological change in computing hardware, with software flexibility and portability being increasingly important in ensuring continuity of service. Seldom are issues in programming discussed in a straightforward and down to earth way, and this severely limits opportunities for programmer education. (The efforts of some of the Bell Laboratories people and those that have followed in their footsteps are an encouraging stand for a common sense approach.)

To us, one of the most appealing things about Unix and Software Tools is the educational aspect. We see these methods as an opportunity to refine our practice of computer programming by learning from the examples of others and by constant experimentation of our own.

This first issue necessarily leans heavily on our own contributions, but we hope that in time it will come to reflect the efforts of others following these approaches in Australia. We therefore invite contributions in the form of articles or short communications or letters to the editor, with the emphasis on practical experience or the relevance of theoretical matters to practical experience.

Please let us know if you have any suggestions for what could be covered in forthcoming issues. (We have included a list of our own on the inside back cover.) The main reason for publishing *Software Tools Notes* is to provide information which will be genuinely useful to programmers, engineers and scientists and make them aware of new approaches that they *can* adopt and benefit from immediately.

We would be willing to act as a clearing house for the distribution of the standard Software Tools Users Group tapes in Australia if this is a useful service. It is to be noted that the current distribution tape includes a *Cookbook* for installing Software Tools on a new system as well as a *User Manual* that documents all tools and subroutines.

* * *

Recommended Literature

For readers who are unfamiliar with the history of the Software Tools system, here are some standard references:

Software Tools, by B.W. Kernighan and P.J. Plauger, published by Addison-Wesley, 1976.

Software Tools in Pascal, by B.W. Kernighan and P.J. Plauger, published by Addison-Wesley, 1981.

Software Tools Communications, published by the Software Tools User Group, 1259 El Camino Real, #242, Menlo Park, California 94025, U.S.A.

Ratfor — A Preprocessor for a Rational Fortran, by B.W. Kernighan, Software Practice and Experience, Volume 5, 1975.

The Unix Programming Environment, by B.W. Kernighan and J.R. Mashey, Software Practice and Experience, Volume 9, 1979.

Poetic Programming

Dr Desmond FitzGerald

THE OXFORD DICTIONARY defines the term poetry as the expression of beautiful or elevated thought, imagination or feelings, in appropriate language. Poetry is usually succinct but adequate, sometimes hiding all its meanings until pushed by the intellect. I believe with programming that there should always be a motivation towards the poetic.

Programs and programming methods are generally not the products of any higher guiding philosophy than getting the job done in the *quickest* possible time. There is little reflection on what and how things were done yesterday with a view to improving productivity tommorrow. This limited approach to improving the working environment means that programmers meet constraints that curtail their ability to grapple with real and complex systems.

The concept of poetic programming can extend beyond isolated utility and/or application programs into the entire user environment — forming a co-operating set of tools. In general, most environments are not of this ideal nature.

Human Beings and Their Potential for Programming

There are many failings or constraints that seem to crop up in programming that limit the amount and scope of work that can effectively be achieved. Among the constraints are

1 A maximum three to five hour concentrated burst of effort is the limit of most people's capability.

2 A distracted environment amounts to little real work being possible. A poor computing environment leads to frustration and distraction. Happy and harmonious surroundings are essential for productive work.

3 Lack of a *meaningful* goal causes a lack of motivation. Even with a worthy goal, lack of recognizable progress towards that goal causes difficulties. This general question of motivation is one of the more difficult areas to manage in a working environment.

4 Humans do not think in a logical sense at a very great speed. One good creative and intuitive idea may take a day or two of working and reworking at the logical level to form a workable framework.

5 The characteristic of the human mind of failing to comprehend increasing volumes of logic can become a major constraint. As an example of this, it is often acknowledged that the sheer volume of code in most operating systems makes it impossible for one person to at any one time be able to *keep on top of the code*. It is suggested that a limit of about five to seven competing factors is all that can be coped with at any one time.

6 Being attached to a way of working can be very counterproductive. Programmers often need to co-operate to achieve a goal and silly personal quirks do not help.

It can be seen that *HUMANS* have some very real problems and shortcomings when the task of programming is being considered.

Artificial Intelligence and Expert Systems

The case for establishing and maintaining high programming (coding) standards is made much stronger by looking to the future of programming. While not suggesting that programming as it is now known will become an obsolete profession, it seems likely that

1 Computers are going to take over more of the hum-drum lower level programming. Programmers will be using more abstract and concise methods.

2 *Expert Systems* will be developed that can accept input to a project from diverse sources and produce an integrated and consistent system. This process can be likened to a person reading a programming text and making very reasonable deductions about the code and the purpose of it without ever formally knowing the rules of the languages.

Good coding techniques *now* are an investment for the future. The quality of present thought ought to be preservable and translatable at quite an abstract level by future generations of expert system software. Poor software will be scrapped.

Making the Most of Personal Creativity

One or two people always give the creative insight and input into a well integrated system. The oft quoted "1% inspiration, 99% perspiration" applies very much to programming. While remaining in inspiration-mode is often an aspiration, we are mostly involved in hack-work.

Having painted this picture of where things stand, are there any elegant ways of coping with the 99% of time in programming that we are doing fairly mechanical work in perspiration-mode?

Learning from Other Disciplines

It should be generally recognized that some of the time honoured engineering principles of design are equally applicable in programming (hence the term *Software Engineering*).

Some basic engineering design tenets are

1 Modularity of components.

2 Starting from a broad over-view design.

3 Adopt a policy of early prototyping and testing with subsequent refinements.

4 Coming to some agreed standards for the manufacture and use of commonly used components.

On this last point, we can make an analogy between programming computers and designing cars. The car industry by now uses standardized parts, optimized for form, function and robustness. Attention is paid to efficiency — minimizing energy usage. Care is also put into the car's appearance. Cars have standardized user interfaces. The brake, clutch accelerator and steering wheel are positioned uniformly and respond in a standard manner. Very few of these attributes apply in most computing environments.

Speed of Thought

If you can work somewhere near to the speed of your thoughts at a computer terminal, getting good turnaround on each module or group of modules that you write, then you can conceive of and debug a great deal of code in a very short time. This amounts to individual programmer productivity that is a hundred times the "norm". A three to five hour productive session can then achieve a major development. The "system" must not impinge and interfere with the thought process. It ought to remain

in the background of consciousness. Working at or near the speed of thought keeps a project moving and motivation high. It encourages revision and refinement of ideas since changes will not take very long to implement. As an example, the Prime Source Level Debugger deserves high praise in being a tool that really helps in this area.

Being Smart vs Being Readable

It has been found that being overly clever and/or obscure in programming is basically non-productive in the long term. Even the person who wrote the code can be quite mystified as to how it works after six months or more. There is always a simpler way of expressing the algorithm or data construct. If a mutual review of on-going work is followed, the benefits are great and there is an added stimulus towards improving existing code.

Some of the more common problems in this general catagory are

1 Avoid the confusing use of temporary variables.

2 Do not use "tricks" which are dependent on your private knowledge of a particular machine.

3 Consistent indenting around control flow statements, uniform spacing around operators and a common style of comments should be followed.

4 Keep internal documentation up to date. A well written program should be almost self-documenting (even without the use of comments). Additional documentation should not repeat what is obvious.

Development of Tools and Languages

The fact of experience is that Fortran will remain the bread and butter of technical computing. A good case can be made to leave Fortran alone and to develop preprocessor techniques to aid in the more succinct and abstract statement of the logic required.

The *Software Tools* method with its associated libraries and utilities can make a very big impact on a user environment and personal productivity. Standardized library calls can be made available which give the application programmer a very dense fabric of programming support. It is possible to liken this support to standard "sub-assemblies" of parts. With a few lines of initialization and the appropriate library calls, the programmer can call on some very powerful workhorses.

Using a Preprocessor in Large Technical Applications

A Case Study

Dr Desmond FitzGerald and Paul Howson

PRIOR TO 1980, our main background had been in technical software — mainly for the mining industry. The engineering profession (at least here in Australia) is not exactly a torch bearer in innovative software methodology and most programmers working in an engineering environment follow software methods and philosophies of the sixties. Our engineering software was written in Fortran IV.

For a beginner, struggling with the complexities of Fortran (writing an engineering program) from the ground up can be exciting. But you can soon find yourself sitting at the keyboard tackling programming tasks which seem familiar. These tasks can seem *similar* to something you've done before, or that somebody else has done before, but they are not really what is wanted for the current project. How nice to be able to start with something that's already written and use parts from it for the new project.

Three years ago, we came in contact with "Software Tools" — through reading the original text by Kernighan and Plauger. At that time we procured the tape from Addison-Wesley here in Australia and commenced the gradual implementation of some of the more useful utilities on a Prime 400 system. These early attempts at implementing the utilities were rather crude, for at that stage we knew little about the Software Tools programming environment and did not fully understand its usefulness. So Ratfor was relegated to the back seat.

Around the same time we began to write applications software using Fortran 77. The coming of Fortran 77 probably kindled an interest in programming standards and software portability, for by this time we had begun the first of a series of programs for one of Australia's largest and oldest underground mines, situated at Broken Hill in central New South Wales.

Initially a geological database system had to be developed on a Prime machine for end use on a Perkin-Elmer machine, and the problems of portability began to rear their ugly heads. On this project there was some finger burning — and it drove home the lesson that attention to issues of software standards was important.

As our experience of Fortran 77 grew, we found that despite the fact that Fortran 77 had cleaned up Fortran a great deal, it still lacked good looping constructs, a flexible compile-time symbol substitution facility and had rigid formatting requirements. We found ourselves re-examining Ratfor and became convinced that Fortran 77 could benefit from preprocessing just as Fortran IV had done.

We then began a data entry program for 3D mine models, writing in Ratfor, but using the Fortran 77 character data type for characters (rather than integers). This unorthodox flavour of Ratfor was termed Rat77. (Actually no change was necessary to the Ratfor preprocessor itself. All the character declaration and handling fell into the category of Ratfor's LEXOTHER statement class and passed straight through into Fortran).

One of the advantages of using Fortran 77 characters was that suddenly quoted strings became scalar constants of character type and the division in Ratfor between *quoted strings* (or *Hollerith* strings) and *character variables* disappeared — they became compatible. The remark primitive was not required to output quoted strings since putlin did the same job, more or less.

Duplicate versions of all the software tools libraries were made using hand translations into the Fortran 77 character type. In retrospect, maintaining duplicate libraries has been very troublesome.

When we delivered the first application program written in Rat77 in July 1981, there was not unexpectedly, hesitation expressed by some of the staff at the mine. They felt that Ratfor was an unnecessary complication and that any programming task could be satisfactorily handled in Fortran. Of course, they were right inasmuch as it is *possible* to do any programming task in Fortran.

There are aesthetic and pragmatic reasons for using Ratfor and a Software Tools environment. These are that Ratfor looks good, is much terser than Fortran, and is much easier to maintain and develop for complex systems.

Within the last year, several more large mining application programs have been written in Rat77. These include (i) a line assay and entry system; (ii) an on-line data acquisition system; and (iii) an ore-reserves package (containing a simple relational database system).

WHAT HAVE WE LEARNED from this experience with Ratfor and Software Tools?

We have found that Ratfor's concise control constructs and compile time symbol substitution (macro processor) encourage the writing of code which is *parameterized* allowing a more generalized expression of an algorithm that the equivalent Fortran code.

Ratfor's existing base of software tools has provided examples of *modular programming* where each distinct job in a program is handled by a separate procedure. By programming in this way, you soon build up a collection of useful, generalized procedures. Such procedures are *tools* and in time can be built into a *library*. When *documented properly*, these can be passed on and used by other programmers, saving them the effort of re-invention.

We have borrowed numerous routines from and found much inspiration in the Software Tools libraries. This means less work to do. It encourages a consistency of style amongst software from different programmers. For instance, there is little noticeable difference in style between programs written in Ratfor by us in Melbourne and programs contributed by members of the Software Tools Users Group in the U.S.A. We benefit from what they've done and hopefully can in time contribute something back.

The appearance of a program is important too. A program put together with careful attention to consistency of style — elegant, economical, lucid, tidy and aesthetically pleasing to look at — reassures the reader of the overall integrity of the work.

When you *see* carelessness in a program, you suspect that the same carelessness will manifest in more subtle ways also.

Although is is possible to write in a confused way in any language, Ratfor has at least *allowed* us to write software in a cleaner and simpler way than is possible in Fortran.

Another principle which has become clear to us is *keep it simple.*

Almost without fail, when we've tripped ourselves up in a project, its because we've written code in a complicated or fancy way (perhaps thinking at the time how clever we were). Yet, revisit it in a year's time, and that complexity becomes unintelligible. You often find yourself going back over work, revising, rewriting, trying to cut it down to the simplest expression of the problem.

During the course of programming, repetitive code often emerges and we have learned to recognize this and invent *macros* to encapsulate it.

Consider a for loop to scan a linked list. In raw Ratfor it might be

```
for (ptr = start; ptr != 0; ptr = array (ptr))
    body of loop
```

This construct was turning up so often that we invented a macro for it

```
AlongList (start, ptr)
    body of loop
```

This has the advantages of less keystrokes and a clearer expression of what's happening. In this way you make building blocks from building blocks and so can build larger programs faster.

It has become apparent that data structures and run-time dynamic memory allocation are essential for list processing, interactive graphics, table-driven user interfaces and generalized database libraries. The macro processor has become an essential tool in extending the Ratfor language to provide these facilities. These methods are topics in themselves and will be explained in later issues.

In mid-1982, we received a tape from the Software Tools Users Group in the U.S.A. This group has developed and enhanced most of the programs presented in the original book by Kernighan and Plauger — generally moving them closer to the equivalent tool in the Unix operating system.

Future Directions

Ratfor is an enormous improvement on Fortran, but still lacks many facilities for engineering programming.

There are many changes and improvements we would *like* to make to Ratfor if we were given free reign to do so.

We have resisted making any changes (other than internal "speed-up" changes), for we believe that if we are to write programs which we describe as being "Ratfor" (or "Rat77") programs, then they must be compatible with the original Ratfor and Macro processors.

It seems that perhaps the time is ripe for a new Fortran preprocessor, based on Ratfor, but incorporating many more of the good ideas from C, and perhaps Pascal and Ada or whatever. Programming for real-world technical/engineering applications cries out for standardized language shorthands, facilities and building blocks more than the often simplistic "systems" programs.

We strongly feel that the appropriate way to change Ratfor is not to change it at all — leave it as it was originally designed and known and if necessary invent a *new* preprocessor with a new name, so there is no confusion about different versions of Ratfor.

From our experience we feel that areas of study and development which would greatly benefit the technical programming community are

1 Software tools to translate between commonly used programming languages. After all, most programmers can convert a program from Ratfor to C, or from Fortran to Pascal, and the rules are mostly (if not completely?) mechanical. Surely this sort of mindless drudgery is what the computer is for? In *Software Tools Communications* Number 9, the idea is raised of developing from Ratfor a preprocessor/language which would be translatable into other commonly used languages. This idea has practical merit.

2 Language extensions or libaries to provide shorthands for: character handling, formatted i/o that is predictable and sensible, dynamic memory allocation and list manipulation. Most of these facilities merely require consolidation into a standard from their current prototype form.

3 Language extensions or libraries to implement the needs of the technical/engineering programmer. These include interactive user interfaces, graphics library interfaces and a portable database system. Technical programming has specialized needs and could benefit from specialized code generation tools just as Cobol has benefitted from the so-called "fourth generation" languages.

This may sound overwhelming, but the experience with the Software Tools package so far indicates that by properly following principles of care and discipline in *how* we work, it can be achieved with a minimum of mindless, repetitive effort.

Ratfor and Fortran 77 Character Usage

Paul Howson

THE MOST SIGNIFICANT new feature of Fortran 77 (for program portability) is the character data type.

Earlier Fortran standards provided no machine-independent method of handling character data — in fact if you adhered strictly to the Fortran IV standard, you couldn't handle character data. There were data types for all kinds of numerical applications — integer, real, double precision, complex and logical — but there was no character data type as such. Characters were tolerated by assuming them to be typically 8-bit bytes and, by having a knowledge of the word length of your particular computer, you could resort to mysterious techniques to pack 8-bit characters into 16 or 32-bit integers (or reals).

However, this was all very dependent on the exact architecture of the computer. Some computers had 36-bit or 60-bit words and it was common practice to pack six or ten 6-bit characters into these. This style of handling characters produced programs which were locked-in to a particular machine architecture.

In *Software Tools*, Kernighan and Plauger adopted a machine independent way of handling characters in Ratfor by treating them as small integers. (A technique also used in Tektronix's Plot10 library.) Character strings were treated as one-dimensional arrays of integers with one character per array element.

They in fact said: "since Fortran IV doesn't support a character data type, let us use a data type that it *does* support to represent characters" — and Fortran has always supported the integer data type well.

Fortran IV did not officially recognize quoted strings, except perhaps in format statements. Even though most compilers would allow quoted strings (as for example arguments to subroutines), there was no machine independent way of manipulating the string components once inside the subroutine.

Not having any quoted string facility is crippling and in "Software Tools", Kernighan and Plauger requested a special i/o primitive called remark to put out to the standard error output a quoted string passed to it as an argument. They ensured that all their quoted strings were terminated with a period, so that remark could know when it had reached the end of the string. This technique worked for the frequent occasions when some message or other was to be sent to the user's terminal, but in order to send literal strings to an arbitrary device or file (via putlin), it was necessary to declare the string as an integer array and initiallize it, a slot at a time, using data statements. Hence to put out the string "fred" to file unit fd, one had to write

```
character fred (5)
data fred (1) /LETF/
data fred (2) /LETE/
data fred (3) /LETR/
data fred (4) /LETD/
data fred (5) /EOS/

call putlin (fred, fd)
```

whereas it would have been nice to be able to write simply

```
call putlin ("fred", fd)
```

Clearly, any program which required a lot of literal strings soon became very full of these long declarations. The Ratfor preprocessor is a case in point — it contains quite large block data areas to declare all the Ratfor keywords it needs to know.

To alleviate this, later versions of Ratfor provided the string statement which translated a line of the form

```
string fred "fred"
```

into those longhand array declaration and data statements required by Fortran.

With the coming of the Fortran 77 standard, Fortran responded to the glaring neglect of characters in the original standard by at last providing a character data type. No longer did you have to resort to any trickery to handle characters in Fortran programs. The new standard at last did something predictable with quoted strings by defining them to be scalar constants of type character.

And how did this change in Fortran affect Ratfor? One possible approach is to use the Fortran 77 character data type in place of Ratfor's "characters as integers".

By making Ratfor's "character" data type translate into the Fortran 77 character data type, quoted strings suddenly cease to be anything special. You can thus freely use quoted strings as arguments to subroutines (such as putlin or file handling routines). For example

```
fd = open ("myfile", READ)
```

or

```
call putlin ("a line of text" // NEWLINE, STDOUT)
```

However, despite this bonus of gaining quoted strings, there are disadvantages.

Consider firstly how Fortran 77 indicates the end of a character string. Fortran 77 character string variables are fixed length strings — they must have a declared length and storage is allocated for them at compile time. (This in itself is no different from using characters as integers.) In practice however you usually want to have strings whose contents vary in length during program execution.

This can be implemented by using a special string ending character which we may term symbollically EOS ("End-Of-String"). This is typically an element of the character set not normally used. Ratfor uses such a sentinel character for terminating strings — typically a small negative integer. In Fortran 77 style Ratfor, all characters must be elements of the machine's character set, so a non-printing character may be used as the end-of-string convention.

The problem is that such an end-of-string convention is not recognized by Fortran 77 which prefers to pad strings out to their declared length with blanks. A user implemented EOS convention invalidates many of the nice facilities that Fortran 77 provides for character handling.

For instance, Fortran 77 automatically passes into a subroutine the address *and length* of a character data type used as a calling argument. This length can be recovered inside the subroutine using the len intrinsic.

There arises a complicating ambiguity however. If the string being passed is a variable, *containing a terminating* EOS *character*, then its *length* is up to but not including the EOS character — even though this may be less than the declared length of the variable given by the len intrinsic. But if the string is passed as a *quoted string* in the calling routine (and hence *containing no* EOS *character*), then the length must be determined using the len intrinsic.

So the way to do an end-of-string test (when scanning a string) is not clear cut. Sometimes there's an EOS character, sometimes there's not. (The C language shows the preferred behaviour by encoding quoted strings with an EOS character at the end.) Complications like this are something we can afford to do without!

Consider now some of Fortran 77's intrinsic character functions.

If you declare

```
character name*20
```

and then use Fortran 77's convenient character assignment statement

```
name = "fred"
```

you get in name the four characters "fred" followed by 16 blanks. As far as Fortran 77 is concerned, *those 16 blanks are valid parts of* name. But what if you wanted to store the letters "fred", followed by just *one* blank? Well, there is no way in the standard to do this without the blank being "lost" in the blank padding.

We would need instead to write

```
name = "fred" // EOS
```

to get the desired result of a string (possibly containing blanks) with a recognizable terminator.

Character concatenation falls down most of the time also. For instance

```
character name1*20, name2*20, name*20

name1 = "fred" // EOS
name2 = "smith" // EOS
name = name1 // name2
```

won't give us name containing "fredsmith". In fact because Fortran 77 does not understand an end-of-string character convention, name will contain only "fred" followed by EOS followed by 15 blanks.

To achieve the desired concatenation we must write

```
name3 = name1 (1:length (name1)) //
    name2 (1:length (name2)) // EOS
```

where length is a function which returns the length of a string up to but not including the EOS. You will no doubt agree that the clarity hoped for by

providing an intrinsic concatenation function in Fortran 77 is somewhat lost through these undesirable but necessary subterfuges. In a similar way, the intrinsic function index is thwarted in its effectiveness.

In programs which do any serious character handling, Fortran 77 's blank padding scheme is so imprecise as to be un-usable. A user-implemented EOS convention is the only way out and alas we pay the price of forfeiting Fortran 77 's character intrinsics.

The second shortcoming is that in Fortran 77 *strings* are seen as *scalars*, different from *vectors* of characters — and the manipulation of strings requires a special notation. The reason, it seems, is that if you want you can have vectors (or arrays) of *strings*. In our experience such a facility is not worth the complexity of notation which it introduces. Yes, it is nice sometimes to be able to specify substrings with the colon notation, but most references to character variables are of two types: references to the entire string or references to *one single character* within the string. It is clumsy to have to write string(i:i) to get at the i-th character. It would have been far better to allow string(i) for this ubiquitous case. (Of course, if you want the single subscript notation, you can use *character vectors*, but these are not of the same type as quoted strings — which are character scalars.)

Despite having used Ratfor with Fortran 77 characters for over two years in large technical applications programs where the ability to use quoted strings in context was valuable, we have come to the conclusion that the best long-term strategy is to use the traditional Ratfor technique of representing characters as integers, for the following reasons:

1. It is compatible with the large existing base of Software Tools programs and libraries written in Ratfor.
2. It avoids the headaches associated with the widely differing ways that Fortran 77 characters have been implemented on different machines. The methods of passing a Fortran 77 character argument *with its length* to a procedure are many and varied and seem to trip one up constantly when porting software between machines. Integers are traditional, simple, less restrictive and reliable.
3. It is closer notationally to the flavour of other languages such as C and Pascal.
4. Since the Fortran 77 character design, with its lack of EOS convention, makes the Fortran 77 character intrinsics rather unusable, little is lost by not having them. Preprocessor enhancements could restore the *desired* functionality in character handling.

Forthcoming in *Software Tools Notes*

Here is a list we have made of some of the interesting topics which have arisen during our programming experience and some ideas for articles. Perhaps readers would care to suggest some more?

Report on the January 1983 Software Tools Users Group meeting in San Diego.

Using preprocessors.

Macro processing.

Ratfor programming course.

Software portability.

User interfaces in interactive programs.

Implementing list structures in Fortran with a macro processor.

The visual appearance of programs — aesthetic considerations.

Simulating a pointer facility in Fortran.

The Software Tools libraries.

Building on previous work — the concept of *tools*.

Automating i/o in Fortran by the use of tables.

Software development environments.

Plot files.

Architecture of data acquisition software.

What makes a good text editor.

Standards in writing software.

Programs for spelling mistake detection.

A set of object file primitives for Software Tools.

A lexical analyser program.

Typesetting by computer using macro processors.

This issue was prepared using Software Tools utilities and typeset from computer produced paper tape by Stock and Ritchie, Melbourne.

Published by D. FitzGerald & Associates, Melbourne, Australia.