Working Paper Series
ISSN 1170-487X

Character-less Programming

Keith Hopper
Robert H. Barbour

Working Paper 94/1

February, 1994

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Character-less Programming

Keith Hopper
Computer Science, University of Waikato, Hamilton, New Zealand
email kh@athens.cs.waikato.ac.nz

Robert H Barbour
Computer Science, University of Waikato, Hamilton, New Zealand
email coms1034@lucifer.cs.waikato.ac.nz

Abstract

This paper proposes a mechanism for the definition and implementation of programming languages. Culturally-dependent aspects of the definition, such as the language and character set, can be separated from the process of translating programs and from the execution of a translated program.

Introduction

New techniques are urgently required to provide Information Technology to all nations and cultures. Attempts are under way to both codify those features of individual cultures for which corresponding hardware and software solutions need to be found, and to develop enabling techniques for the production of programs within, by and for people of all cultures (Barbour , Cunningham and Ford, 1992)

Underlying the urgency in the developing enabling technology is the universal need to be able to express, communicate and manipulate ideas -- and to provide culturally appropriate representations of these for passage to, from and between computer systems. The use of language of some kind for such purposes has been a common feature of all human cultures for millennia and is likely to continue to be so for the foreseeable future. The principal currently available technology for the expression and manipulation of ideas resides in the use of computer programming languages.

The intention of this paper is to briefly review the features of programming languages and their visible manifestation in order to provide a background to a proposed new mechanism for specifying two aspects of a programming language, together with a corresponding strategy for the implementation of programming language processing tools and the environment in which they may be used. It is suggested further, based upon experimental implementation, that these techniques are as applicable to those programming languages currently in use as to those yet to b developed.

Expression of Meaning

All languages use the notion of 'token in a shared context' to enable those communicating to determine the meaning of an utterance. This assertion applies whether it be two friends in idle conversation or a programmer constructing a compiler.

The context is provided in two parts :

a. That part which is contained in the structure of the utterance itself -- the syntax of the language.

b. That part which is drawn from the common milieux of those communicating.
For example, when using the English word 'path', the two friends in conversation would probably be referring to a walkway between two hedgerows "over there".

A programmer communicating with a computer almost certainly refers to a sequence of tokens used to identify some name space.

The token made visible as the word 'path' therefore has no intrinsic meaning of its own. All English speakers who are computer literate, however, have an internal map from visible (audible) token to (at least) the two meanings-in-context given as examples. To extend this notion further, homophones may have different mappings-in-context as may homographs. Providing the recipient of the utterance knows the context then all is well. However, if the shared context is not in fact determinate as, for example, on reading the isolated visible token' fin' -- [is it English and part of a fish, or French and this is the end!], then communication is not possible.

Token Abstraction

The realisation, well-known to linguists, that no token has an intrinsic meaning in isolation, seems not to have been applied in the design and implementation of computer programming languages in general (Barbour, 1993). This is understandable from an historical perspective as computer representations of visible characters were in use well before the development of language theory as it is known today.

It also reflects the natural attitude of programmers who are working within a single culture. A programmer would take for granted that other people would share both context and thus the meaning of the tokens used without further communication.

The first computer language to make use of the idea of token abstraction was APL, probably because of the richness of the visible forms of the tokens used to express it. No computer character encoding used at the time of its development had the necessary richness of variety of visible tokens. The latest version of this language, Extended APL (DIS 13751), follows its predecessor by naming tokens rather than giving them explicit visible expression in the standard (although, naturally, the names themselves are visible when printed).

An implementation of APL is required to provide a mapping from some external representation of a token to the language-specified token. No distinction is necessarily made, however, between the two conceptually different things in the language implementation.

The programming language Ada (including its latest revision as Ada-9X--CD8652:1993) goes a step further in indicating a possible solution to the overall problem (although it must be pointed out that this has not yet been taken to its inevitable conclusion as described later in this paper). The Ada language defines enumeration types, each value of which is itself a token, provided with a position (in the enumeration). In a real sense the value of this token is its position in the value domain defined by the type. However, Ada goes one step further -- it provides for an explicit representation to be defined for each value. Referring to the heading of this section and inverting the above statement, the Ada language provides an abstract value for each representation (visible/audible token) in an enumeration.

The Modula-2 programming language (DIS 10514) offers a variant though related definition of the pre-defined language type CHAR which it defines as an enumeration. Some of the names for the values of the type are pre-defined in the standard while additional names are (if provided) to be defined by an implementation. The important point about this for the purposes of this paper, however, is that Modula-2 leaves a person carrying out an implementation entirely free to choose any suitable representation desired for the transfer of values into and out of a program.

Language Tokens and Representation

While all programming language standards (or other specifying documents) are provided as written/displayable text, this is solely for the human reader. If it were necessary to provide standards documents for blind people -- unable to read text-- then it is likely that they would be prepared as audio recordings of the spoken word. The language tokens would then be phoneme rather than lexeme sequences.

Similarly, it is increasingly becoming evident that in the foreseeable future programming will become an activity in which some design tool generates the syntactic tokens of a language -- not a string of lexemes derived from a visible character representation. It is this notion, together with those ideas taken from APL, Ada and Modula-2 language definitions which has lead to the first part of the proposed solution to token rich but character-less programming! Provided that a programming language translator is passed a stream of syntax tokens in whatever internal representation it requires then it can attempt to carry out the translation desired.

The task of providing the internal representation of the syntax tokens is a simple one of mapping short sequences of lexemes/phonemes into a token. The encoding of the lexemes/phonemes is irrelevant -- provided that a mapping is provided to the translator for its use when translating!

What is needed, therefore, is a list of all of those terminal tokens which exist in a programming language -- a list specified as an ordered enumeration in the language standard (or other specifying document). A typical list would include such tokens as -- Full-stop, End-token, Line-mark, For-token, White-space, Procedure-token, Quote-mark. The list, apart from being ordered in the language specification must contain ALL the tokens specified by the language.

It is necessary to distinguish tokens in this list from those specified in an individual program, which programmer-defined tokens may be divided into the following classes :

a. Compositions of language defined tokens. In all known programming languages such compositions are restricted to the representation in visible form of a numeric value (which requires translator processing in order to determine its value). For languages in which such compositions are needed then the list of language tokens will necessarily include such additional language-defined tokens as Digit-zero, Digit-one, Exponent-mark, etc.

b. Bracketed data. For such data values a language need only define one or more marks to provide the brackets, even a single mark to 'toggle' between program source and data -- as, for example the Quote-mark as used in this sentence. Many existing language definitions identify strings of visible characters in this way (see Note).

c. All other programmer-defined tokens -- considered as identifiers. NOTE Where an existing language does interpret the contents of some character string in a particular way (eg when I/O formatting) there is always some 'escape' token followed by tokens known to the I/O formatting translator -- sometimes followed by a further separator to indicate the end of the 'nested' bracketing (dependent on the syntax of the formatting language). This is just another language for which syntax tokens need definition. The fact that it may form part of some programming language translator must not be allowed to obscure the fact that it is a different language and as such needs definition too.

Provided that a programming language translator can be given a mapping of the external representations for language tokens, including of course those necessary to provide for programmer composable numeric value representations then it has no need to be aware of the external form of representation of ANY token at all. It has no need to be aware of any character encodings used in its host environment -- for example any use of codes

which a suitable representation engine may produce as Arabic, English, Chinese or any other humanly readable/audible language 'words' or 'digits' or 'punctuation'.

A mapping provided for a translator in this way should be made available in the environment in which it is to execute, as part of the culture-dependent components of the host operating system. A multi-cultural operating system will provide, say, for Arab, English and Chinese programmers to work in their own natural language alongside each other on the same joint project and all be able to use the same programming language tools, including the translator.

The technique described, therefore, enables the initialisation phase of a translator to read such a mapping in order to initialise its lexical analyser. During subsequent translation of some unit or program the encodings received in the source stream are matched against the mappings and further lexical actions proceed in the way outlined below :

a. If a match is found then the appropriate token value (NOT the encoding) is used in further processing.

b. If the token is needed to start a composition then subsequent tokens are used in such composition as required by language-specific rules until a non-composing token is detected.

c. If the token is a start-data-delimiter then subsequent encodings (NOT tokens as there is no mapping for them) are collected as data until an appropriate end-data-delimiter token is detected.

d. Every other sequence of encodings is treated as an identifier and (most likely) looked-up/entered into the translator identifier table.

> NOTE Comments in the source of a translation unit are treated as data which is discarded for the purposes of the translator. Further translation syntactic and semantic processing can then take place in the normal way.
>
> Note that no characters at all are needed -- merely an encoding map together with three auxiliary lexical rules, only the composition and data delimiting rules being language-dependent as would be expected. A format for a universal file to contain such a mapping for any programming language and cultural environment is given, together with a skeleton example, in Annex A.

Data Manipulation

While the suggested language-defined translator map completely specifies a character-less (and hence considerably more portable and culturally-sensitive) translator mechanism, it does not, of itself, solve the allied problem of data manipulation by an executing program.

Where data appeared in the source of a program, the translator lexical analyser merely collected encodings in a totally transparent manner. After all, it is the manipulation of such unknown data, together with that read from some input channel during processing, which is the principal purpose of a program. The production of data through some output channel is only of related concern if it is intended for use by some other tool (eg a rendering engine producing sounds or visible marks for human users).

The encoding of data, whether as part of some program source or as data obtained/generated during program execution is therefore of major concern to the writer of a program -- but only insofar as the actual encoding employed may be 'understood'

by the program. The solution to this problem given above in the case of a programming language translator (which is, after all, only another program), relied upon providing a mapping from encodings to lexical tokens in a completely portable, transparent, manner.

While it would, perhaps, be ideal for the specification of every program to indicate its input/output languages in such a way, this is unlikely to be practical for some time to come (if ever). As an interim solution, therefore, it will be necessary to provide a simple, practical mechanism which will improve upon the present situation -- ie it must be possible to determine the incoming tokens from the external representation.

The current work being undertaken world-wide to collect a number of abstract concepts which are shared by more than one culture offers, it is suggested, a small step in the right direction. If, for each shared abstraction, a unique Abstract Data Type (ADT) is provided as part of the environment within which a program executes then the program may make use of such a type and its standard operations without any concern for the culturally specific interpretation of such a concept. Similarly, in addition to not needing to be concerned with local interpretation, the program does not need to "understand" the syntax nor encoding of any human interpretable form of the abstraction -- that is hidden inside the implementation of the lexical analyser/generator for import/export of value of the abstraction.

The implementer of the ADT facility for some specific culture and system will, for lexical analysis/generation purposes, need to read a mapping of a culturally-specific encoding in exactly the same way that a language translator has to read its lexical mapping. Once again, there is no need for the culturally-specific ADT implementer to be aware of what form an encoding may take.

Are Characters Really Necessary?

While the discussion in this paper has so far hardly referred to the word, characters ARE necessary although only in a very limited way -- far more limited than much current thinking about programming would indicate. A quick review of any average selection of programs written in whatever programming language will reveal that, with very few exceptions, characters are currently used for the following :

a. Expressing in visible form in program source some complete or partial message for export to a human (or other text reading entity)-- eg "Bad luck in the family!".

b. For use in converting to/from some internal token or value (converting a number or date, say).

c. Either individually or in combination as a substitute syntax token (in much the same way as programs are currently written, just as some form of textual data).

The relation of these three to comments/data, language tokens and value composition as specified for programming languages is obvious. What, perhaps, is not so obvious is that these have essentially all been eliminated from application programs which :

a. Provide for 'messages' to be external as a list of messages in the same order as some internal enumeration. Naturally this is done to permit the use of different natural languages and orthographic forms of message.

b. Use culturally shareable ADTs which embody conversion analyser/generator code.

c. Merely use encoding pattern matching to determine whether or not some encoding sequence matches a known syntax token. This uses encodings but not characters!

The only point where characters are actually needed, therefore is as visible forms (where an encoding uses a suitable rendering engine) for human interpretation/generation. This could occur, for example where a human user is preparing a list of messages for some program to use. Note that the list is prepared using the visible marks on a display, for example, but that what is really wanted is the encoding of the message. The program merely uses the encoding, not the characters.

In essence, programming is not a task which involves characters. In restricted circumstances detailed manipulation of character encodings may be required, but this is almost exclusively restricted to the use and preparation of mappings to lexical tokens, tokens which have some representation internal to a using program which is not the concern of any external entity.

Words and Strings

It is proposed, as a final part of the interim solution using ADTs suggested earlier, that two very special ADTs are mandatory -- Word and String, where a String is defined as a sequence of words separated by white space encodings.

From the point of view of the human user, the ability of a computer to organise visible text in accordance with some culturally appropriate rules is of paramount importance. The organised list of names, etc is a very important feature of many human activities. Strangely enough, these two ADTs (and others which may be derived from them) are the only ones which necessarily use the concept of character as understood by the human user in the semantics of their operations, which include among other things, ordering and equality/inequality testing.

Acknowledgments

Many thanks are owed to those kind people who have listened to our often tentative explanations of embryo ideas: D. Andrews, A. la Bonte, R. Hicks, H. Jespersen, R.J. Mathis, P.J. Plauger, K. Pronk, P. Rabin, K. Simonsen, M. Woodman and D. Wong. Much of the credit for the ideas expressed is due to them, any errors are ours.

Bibliography-

Barbour, R.H., Learning Strategies. Unpublished D.Phil thesis. University of Waikato, Hamilton, 1993.

Barbour R.H, Cunningham S.J. and Ford, G., Maori word-processing for indigenous New Zealand young children. British Journal of Educational Technology, Vol. 24, No. 2, p114-124 1993.

CD 13751 Information Technology - Programming Languages, their environments and system software interfaces - Programming Language Extended APL, ISO/IEC draft 26 Aug. 1993.

CD 8652:1993 Information Technology - Programming Languages, their environments and system software interfaces - Programming LanguageAda, ISO/IEC, draft 16 Sep. 1993.

DIS 10514 Information Technology - Programming Languages, their environments and system software interfaces - Programming LanguageModula-2, ISO/IEC draft 31 Jan. 1994.

Annex A PROPOSED MAPPING FILE FORMAT

The file format for a mapping file is based upon the fact that all standard character encodings occupy an integral number of octets.
File header

First octet -- an unsigned binary number which indicates the number of octets of which all other entries are a multiple -- called size in the following.

Next size octets -- a separator token which is not used elsewhere in the file as a component of any other token.

Next size octets -- an alternate token not used anywhere else in the file as a token, indicating that for one entry there are alternate external encodings.

Next size octets -- a separator token.

File Entries

Each entry in the file consists of one or more encoding sequences followed by a separator. Where there is more than one sequence in an entry then each sequence is separated from a following one by an alternate token as defined in the file header. Each encoding sequence must be a multiple of size octets.

There shall be as many entries as there are lexical tokens specified in the programming language concerned.

NOTE This careful form of expression indicates, for example, that Arabic digit or Chinese digit (as well as roman digit, etc) coding could be used in an appropriate culture (and made visible by an appropriate local rendering engine) for Digit-zero, Digit-one, etc!

Example Enumeration

The following listing of lexical tokens for the Modula-2 language is merely an exemplar of which the source material was readily available to the authors. Following the listing there is some discussion about the way in which the entries in the encoding file could be made.

Digit_0
Digit_1
Digit_2
Digit_3
Digit_4
Digit_5
Digit_6
Digit_7
Digit_8
Digit_9
Digit_10
Digit_11
Digit_12
Digit_13
Digit_14
Digit_15
Hex_number_mark
Octal_number_mark
Character_mark

Exponent_mark
Single_quote
Double_quote
White_space
Comment_start
Comment_end
Source_code_directive_start
Source_code_directive_end
Colon
Comma
Ellipsis
Equals
Period
Semicolon
Left_parenthesis
Right_parenthesis
Left_bracket
Right_bracket
Left_brace
Right_brace
Assignment_operator
Plus_operator
Minus_operator
Logical_disjunction
Multiplication_operator
Division_operator
Logical_conjunction
Logical_negation
Inequality
Less_than
Greater_than
Less_than_or_equal
Greater_than_or_equal
Dereferencing
AND_SY
ARRAY_SY
BEGIN_SY
BY_SY
CASE_SY
CONST_SY
DEFINITION_SY
DIV_SY
DO_SY
ELSE_SY
ELSIF_SY
END_SY
EXIT_SY
EXCEPT_SY
EXPORT_SY
FINALLY_SY
FOR_SY
FORWARD_SY
FROM_SY
IF_SY
IMPLEMENTATION_SY
IMPORT_SY
IN_SY
LOOP_SY

MOD_SY
MODULE_SY
NOT_SY
OF_SY
OR_SY
PACKEDSET_SY
POINTER_SY
PROCEDURE_SY
QUALIFIED_SY
RECORD_SY
REM_SY
RETRY_SY
REPEAT_SY
RETURN_SY
SET_SY
THEN_SY
TO_SY
TYPE_SY
UNTIL_SY
VAR_SY
WHILE_SY
WITH_Sy
ABS_Ident
BITSET_Ident
BOOLEAN_Ident
CARDINAL_Ident
CAP_Ident
CHR_Ident
CHAR_Ident
COMPLEX_Ident
CMPLX_Ident
DEC_Ident
DISPOSE_Ident
EXCL_Ident
FALSE_Ident
FLOAT_Ident
HALT_Ident
HIGH_Ident
IM_Ident
INC_Ident
INCL_Ident
INT_Ident
INTERRUPTIBLE_Ident
INTEGER_Ident
LENGTH_Ident
LFLOAT_Ident
LONGCOMPLEX_Ident
LONGREAL_Ident
MAX_Ident
MIN_Ident
NEW_Ident
NIL_Ident
ODD_Ident
ORD_Ident
PROC_Ident
PROTECTION_Ident
RE_Ident
REAL_Ident

SIZE_Ident
TRUE_Ident
TRUNC_Ident
UNINTERRUPTIBLE_Ident
VAL_Ident

These are the some hundred and thirty lexical tokens of the Modula-2 programming language. Some of them are permitted more than one encoding by the nature of the language, for example the visible marks '#' and '<>'are, as currently defined, valid alternatives in an 'English' representation of the Inequality lexical token.

White_space too has several possibilities including a space (' '), a horizontal tab and a line mark (however that may be indicated/detected in a particular operating system). Note that in an English version of a mapping file there are a couple of potentially awkward problems as the letter 'C' is used as a Character_mark as well as Digit-12. Such duplication is, however, peculiar to a particular form of visible representation which is always context dependent in a composition. Disambiguation of the context dependency is a simple problem within the value composing routine.