A Context Sensitive Metalanguage for Intelligent Editors

SM Kaplan
University of Cape Town Cape Town

Abstract
The notion of and motivation for intelligent editor systems is introduced, specifically a language independent intelligent editor called "PASTOR". We also introduce and discuss the metalanguage used in the editor for grammar specification. We discuss the issues of context sensitivity checking in such an editor and show how a grammar specification may be enhanced with a high level "abstract machine language" to achieve these checks. These enhancements are illustrated by showing specifications of various contexts sensitive checks for grammar rules for PASCAL.

Introduction
The search for methods of increasing programmer productivity has led to the development of "intelligent editors". These editors provide a syntax-directed programming environment in which syntactic and context sensitive errors in programs are detected at source code input time. Further a set of "templates" or "rules" which define the grammar of the language allow the editor to insert reserved words and syntactic marks for the user, thus reducing many unnecessary and irritating errors.

By preventing the creation of syntactically incorrect programs the programmer is freed to concentrate on the more challenging aspects of programming.

One such editor is PASTOR, which originated at the University of Tasmania, and work on the editor is continuing there and at the University of Cape Town. It is designed to be used as a programming/teaching aid in a microcomputer laboratory and runs on the APPLE II microcomputer. When complete it will be a full editor in a style similar to the Cornell Program Synthesizer.10

PASTOR is different to the Cornell Program Synthesizer in that it is designed to be entirely language independent. The editor has a support environment in which a grammar specifier may define a language to PASTOR and then edit programs written in that language. The language is defined using a metalanguage similar to BNF. We begin by considering the differences between BNF and our metalanguage before moving on to the requirements of context sensitivity.

We assume the user is familiar with the BNF method of context free language specification, and list the differences between BNF and our metalanguage, which we will call CFF (for Context Free Form).

The CFF metalanguage
As mentioned in the previous section, CFF is very similar to BNF, with the following differences:

- The "bottom" level of the language is considerably higher than is conventional. The following CFF nonterminals are "understood" by PASTOR:
  - integer
  - number (* real or integer numbers *)
  - string
  - identifier (* a letter followed by a sequence of alphanumerics *)

- Nonterminals do not appear between angle brackets.

- The reserved words of a language appear in uppercase.

- Nonterminals appear in lowercase.

- All other characters are treated as terminal symbols of the language with the exception of the characters: =, %, <, _, >, ' = which are all used as metasymbols and so if the user needs to use them as terminal symbols they must appear in quotes.

- The spaces, linefeeds, and carriage returns that appear between symbols in the language specification have meaning. They indicate the prettyprinting conventions the user requires in his code, and are used by PASTOR when accepting code as well as when constructing code from the abstract syntax trees:
  - The percent sign % indicates the end of the right hand side of a nonterminal specification.
  - The exclamation sign and colon are used for prettyprinting purposes.
  - The hash sign # indicates a new left margin position. This feature is used to indent code eg the statements between a begin end pair in PASCAL can be indented using this feature.
  - Recursion is replaced by lists. The general form of a list is:

< symbols _ Symbols >

where

< means start of list

> means exit from list

-_ means end of list

and "symbols" may be anything, except the _ and > characters. If one of these is required, then it must be placed between quotes.

- Occurrences of nonterminals on the right hand side of a nonterminal expansion may be made optional by preceding them with a question mark?.

- Where a choice between alternatives is to be given, each terminal or non terminal in the list must be preceded by a backslash |

What, then, does a nonterminal specification in CFF look like?

We consider the definition of types in PASCAL. In standard BNF, type declarations could be defined as:

<types> = <typedefinition> <typedefinition>
<typedefinition> = < 1dshst > <type>
<type> = < id | array > | <record> | <set> |

In CFF, we shorten this to obtain:

"types"

TYPE < identifier > type_

> %

which we read as:

Types consists of the reserved word TYPES followed by a list of type declarations separated by semicolons. Each of these type declarations consists of a list of identifiers separated by commas, followed by a colon, followed by a type.
The idea of an option is fairly common in extended BNF notations. In his proposal PASCAL standard, Addyman enclosed options in square brackets so that [a] means expand the nonterminal a 0 or 1 times. He also does away with enclosing nonterminals in < > braces.

We must motivate choosing this rather unusual form of metalanguage

- Smaller parse trees will result a significant point on microcomputers. This is a result not only of the introduction of the list notation, which makes specification more compact, but also of the fact that many nonterminal expansions have been 'collapsed' in comparison with their normal BNF form (eg expression in CFF) is

\[
\text{expression } = \text{sign } \langle \text{factor \_ multop } \rangle \_ \text{addop } \rangle \_ \text{predeicate}\%
\]

The tree for an expansion of this nonterminal will be broad rather than deep, eg a conventional BNF tree would be a binary tree with an expression consisting of terms which in turn consist of factors which in turn could be a name or a parenthesised expression. The CFF notation would produce an n ary tree which is not deeply nested, making it easier for the user to keep track of what he is doing as code is entered, as the human mind cannot keep track of nesting levels as a computer can.

- Inclusion of pretty printing in the metalanguage allows the grammar specifier to specify how his code should be laid out.

- The metalanguage allows a language the size of PASCAL to be concisely specified. Since the specification needs to be held inside PASTOR as the template against which in turn is checked, there is also a space saving.

- The elimination of recursion and its replacement by lists makes the specifications less daunting to the user attempting to specify a language to PASTOR, and also makes the specification follow the syntax diagrams of a language more closely. The syntax diagram notation for an LL(1) language such as PASCAL allows PASTOR to easily determine a set of 'allowable next inputs' at each choice point, where all members of the set are mutually exclusive, and then use these to prompt the user.

This concludes the rather brief introduction to PASTOR and its metalanguage.

Towards Context Sensitivity

We now consider the introduction of context sensitivity into CFF. We want to be able to perform the following checks

- Verify the uniqueness of identifier declarations in a particular scope at declaration time,

- Verify that identifiers have been declared in such a way as to be visible in the environment in which the check is being made,

- Establish the type of an identifier

- Check type compatibility of factors in expressions,

- Check assignment compatibility between left and right hand sides of an assignment statement,

- Check correspondence (number and type) of actual and for mal parameters for procedure and function calls,

- Check correspondence (number and type) of actual and for mal subscripts in array references

There is no straightforward way of introducing these checks into a metalanguage. When compilers are written, these checks are "hard coded" into the compiler. Such a solution is obviously unacceptable in PASTOR because it is meant to be language independent.

Notations which allow the specification of context sensitivity are well known and include W grammars and Attribute grammars.

W grammars do not lend themselves to use in this environment because the rules of the grammar are meant for prompting the user or the editor, as well as guiding the editor in the layout of a user program.

"An Attribute grammar is a context free grammar augmented with certain formal devices (attributes, evaluation rules, conditions) that enable the non context free aspects to be specified by means of a powerful and elegant mechanism".

This is closer to what we require in that a context free grammar (which we already have in CFF) is "augmented with certain formal devices".

Attribute grammars appeared to be a possible answer to our problem, so we set out to see if attribute grammars could be adapted to our requirements. However, problems arose

- Attributes need to be inherited/synthesized through many nodes of a parse tree where they are not used. This increases tree size and the overheads of building such a tree.

- Specification of context sensitivity is unclear and gets lost in the maze of attribute passing.

- Language specifications become large unwieldy and more difficult to comprehend (see D. Watt's extended attribute grammar for PASCAL for an example of this).

- It becomes difficult to see how certain checks are explicitly carried out and on what data.

The idea of using attribute grammars was therefore shelved and we began to look at the possibility of developing our own notation around the general attribute grammar idea of augmenting a context free grammar.

An extremely important factor which guided the route we took is the data structure on which we proposed to operate which is an abstract syntax tree. The problem of scape of variables is solved for us by the shape of the tree. As each block has its own subtrees of local variables and code blocks on the same or "more nested" lexical levels are not visible, so all we can see are the variables at the local level as well as all the variables of all the blocks inside which the particular block we are considering is nested.

We therefore have a situation where appropriate algorithms can be constructed to handle searching the trees, and which can, by examining the contents of the nodes, determine the type of identifiers. We can also devise methods of determining the types of constants.

We can also introduce the notion of a stack on which data can be stored and on elements of which checks can be done. Expression evaluation, in particular, is easy on a stack, so it should be possible to stack the types of factors instead of the factors themselves and then check for type compatibility. This idea is easily extended for assignment compatibility checks.

We must also consider how to operate on the data which we can extract from the tree. Rather than use implicit operations, which the grammar specifier may find difficult to understand, we decided to develop a set of simple explicit commands which will allow the grammar specifier a great degree of control over the way context sensitive checks are to be executed. As the entire system is to run inside PASTOR, which is written in a high level language, implementation of tree searches will not be a problem.

We therefore suppose that inside PASTOR is an "abstract machine". When PASTOR wishes to perform a context sensitive check, it will do so by executing code on the abstract machine.

The machine is capable of tree searches, stack operations and comparisons as well as data transfer operations.

The machine supports a high level language which is to be specified in conjunction with the specification of the context free part of the users language. The context free part is compiled into an internal form understandable by PASTOR. At the same time that this operation is carried out on the context free part, the context sensitive part should be compiled into "abstract machine object code" which can be executed when PASTOR "runs" the abstract machine to perform a context sensitive check.

We will call the high level language AML (abstract machine language).

To insert an AML procedure into a CFF specification, we introduce the metasymbols [ ] and { }. Anything that appears between these two symbols will be considered to be AML and anything not within them is CFF.

Pretty printing is a function of CFF so spaces, carriage returns
and other separators inside the AML will be ignored for pretty printing.

Let us now consider the development of such a metalanguage illustrating such concepts as we introduce with PASCAL language definitions.

**Declarations**

As the space restrictions on a microcomputer based editor will be fairly stringent, we did not wish to add any data structures additional to the parse tree and the evaluation stack. However, the problems of scanning the parse tree to confirm uniqueness/declaration of identifiers, from the viewpoints of both speed and algorithmic complexity are such that we decided to introduce one more data structure. The grammar specifier declares sets in AML, and as each identifier is declared, it is inserted into a set (one set for, eg, labels, types, variables, subroutines). Uniqueness checks are then checks for non-membership of a set, declarations checks are checks for membership and the set gives us a pointer to the beginning of declarations for type evaluation purposes.

This may be illustrated by the declaration of types in PASCAL, using CFF enhanced with AML:

```
"types": TYPE < identifier[if uniquetypeset + varset + subset + constset]
then insert(typeset)
else error(2)

|---;> = type
```

Similar enhancements may be made for all declaration statements, although the enhancements do tend to become fairly complicated in places.

**Type Evaluation**

If it was decided not to hard code into PASTOR a type evaluation algorithm as this would not be practical in a language in dependant environment.

AML is a high level language with much capability including that of stack manipulation, tree traversal and procedure declarations/calls.

The grammar specifier may declare as many procedure declarations as he wishes and use them in any place where an AML statement may be used.

There is, however, one procedure that must be declared. This is called SUB, and it is called whenever a type evaluation is required. The grammar specifier must write the code for sub so that the abstract machine may be guided over the relevant sections of the parse tree an, by examining the contents of the nodes of the tree, determine the type required.

Tree traversal instructions in AML include the ability to detect the presence of a subtree (for optional nodes) and the presence of another occurrence of the same subtree at the same level (for lists), the ability to move from the current subtree to this next same subtree, the ability to move to an ancestor node and the ability to move down a specified subtree. The moves all work with reference to the template or rule describing the current node.

For example, if we are at a type node (using the rule for types defined above and we wish to evaluate the type of the identifier we may say "sibling(type)" which will move us down to the subtree defining the type A command ANCESTOR(types) issued anywhere on this subtree will return us to the ancestor node called "types", which is in this case the place from where we started.

**Statement Checking**

When one wishes to perform context sensitive checks in PASTOR, one must appreciate that in the same way as the syntax of a language is spread over many rules, so the development of context sensitive checks are spread over many nodes as well. Consider

```
"if"
IF expression[if stacktop < > 'boolean' then error(10)
THEN
statement?else%
```

We are assuming here that the rules for 'expression' will guarantee that the resultant type of the expression is left on the top of the evaluation stack. Most context sensitive checks require the development of such inter relationships between rules of the grammar.

Another example is the nonterminal "name" which is a factor of expressions.

```

  'name'
  identifier[if global(varset + funcset)
  then sub
  else error(8)

  ]?continuation%
```

If the identifier is not a variable or function then an error message is generated. If the identifier is in the correct class its type is found. If the type is an array or record or function type, then the user could be further prompted to enter the correct continuation.

**Conclusion**

AML is a valid and practical method of easily and concisely specifying all the context sensitive checks required by a language such as PASCAL. The procedural structure of AML makes implementations no more difficult than the implementation of a normal simple compiler a well understood process. This structure also reflects the way a grammar specifier thinks of checks as he specifies a grammar, making his task less daunting.

Work is at present proceeding on an AML specification of the context sensitive checks required for ADA. It is envisaged that the increased complexity of data types in ADA will lead to changes in the AML to handle these complexities.

**References**


Notes for Contributors

The purpose of this Journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review articles, exploratory articles of general interest to readers of the Journal. The preferred languages of the Journal will be the congress languages of IFIP although papers in other languages will not be precluded.

Manuscripts should be submitted in triplicate to: Prof. G. Wiechers at:
Department of Computer Science
University of South Africa
P.O. Box 592
Pretoria 0001
South Africa

Form of manuscript
Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins. The original ribbon copy of the typed manuscript should be submitted. Authors should write concisely.

The first page should include the article title (which should be brief), the author's name, and the affiliation and address. Each paper must be accompanied by a summary of less than 200 words which will be printed immediately below the title at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review categories.

Tables and figures
Illustrations and tables should not be included in the text, although the author should indicate the desired location of each in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Illustrations should also be supplied on separate sheets, and each should be clearly identified on the back in pencil with the Author's name and figure number. Original line drawings (not photoprints) should be submitted and should include all relevant details. Drawings, etc., should be submitted and should include all relevant details. Drawings, etc., should be about twice the final size required and lettering must be clear and "open" and sufficiently large to permit the necessary reduction of size in block-making.

Where photographs are submitted, glossy bromide prints are required. If words or numbers are to appear on a photograph, two prints should be sent, the lettering being clearly indicated on one print only. Computer programs or output should be given on clear original printouts and preferably not on lined paper so that they can be reproduced photographically.

Figure legends should be typed on a separate sheet and placed at the end of the manuscript.

Symbols
Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters between the letter O and zero; between the letter I, the number one and prime; between K and kappa.

References
References should be listed at the end of the manuscript in alphabetical order of author's name, and cited in the text by number in square brackets. Journal references should be arranged thus:


Proofs and reprints
Galley proofs will be sent to the author to ensure that the papers have been correctly set up in type and not for the addition of new material or amendment of texts. Excessive alterations may have to be disallowed or the cost charged against the author. Corrected galley proofs, together with the original typescript, must be returned to the editor within three days to minimize the risk of the author's contribution having to be held over to a later issue.

Fifty reprints of each article will be supplied free of charge. Additional copies may be purchased on a reprint order form which will accompany the proofs.

Only original papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters
A section of "Letters to the Editor" (each limited to about 500 words) will provide a forum for discussion of recent problems.

Hierdie notas is ook in Afrikaans verkrygbaar.
Contents/Inhoud

Very High Speed Graphics Work-Station* ........................................ 1
    J Jablonski and H J Dijkman

Documenting a Database System with a Database* .......................... 5
    D A Hunter

Program Design as the Tool of Preference in Computer
Literacy Education, Experience, Incentives, Implications*............... 9
    J M Richfield

A Virtual Multiprocessor Computer Design* .................................. 15
    T Turton

A Context Sensitive Metalanguage for Intelligent
Editors* ..................................................................................... 21
    S M Kaplan

Concurrency: An Easier Way to Program* .................................... 25
    C S M Mueller

*Presented at the Third South African Computer Symposium held on
14th to 16th September, 1983.