A compiler compiler and methodology for problem oriented language compiler implementors

R. V. Evans*, G. S. Lockington*, and T. N. Reid*

ICL Dataskil, Reading Bridge House, Reading RG1 8PN

The Program Synthesis system is a compiler compiler specially designed to enable the implementors of application systems to design, implement and issue language-based applications without having to concern themselves with the technical aspects of compiler construction; and also to ensure that the resulting systems are easily enhanced and easily portable. The background of the Program Synthesis system is described as well as its associated methodology and the internal structure of the system. The proposed enhancements to the system are described together with the reasons for their consideration. Finally some of the achievements of the system are described.

(Received October 1976)

1. Introduction

The Program Synthesis system (PS) is a compiler writing system specifically designed to aid the implementation of problem oriented language (POL) compilers for application systems.

PS provides many of the routines to be incorporated into such a compiler as well as a language in which the application designer can specify the remainder of the compiler.

The strategy adopted by PS is to view the process of compilation as a series of transformations. Some of these transformations have been written as invariant components of every compiler that the system produces; others are provided as table-driven processes for which the application designer supplies the tables, and the remainder are left to be written separately for each compiler.

PS provides a system implementation language (SIL) which has facilities for setting up the tables, coding the routines and collecting these with the standard parts to produce the POL compiler. SIL in addition has many of the properties of a general purpose language and can be used as such, although its main function is to enable the users of PS to construct compilers without requiring of them detailed understanding of compiling techniques.

As the PS system can itself be regarded as an application system within its own scope the implementation strategy has been to write the system in SIL and bootstrap it into existence via a hand compiled version.

2. Background

The design and implementation of user oriented application systems in a commercial environment is subject to many adverse influences that increase the resources required for the task and decrease the usability of the products.

Planning the production of the systems is often hampered by the adoption of ad hoc techniques that render difficult the breakdown and estimation of work involved; also the expertise requirements conflict as personnel are required with knowledge of the application problem area, system programming and the target machine architecture.

Designing the systems is hampered too by the absence of a well defined framework on which the design can be based, with the result that parts of the system which could be conceptually independent become inextricably interrelated with an attendant increase in complexity.

Implementing systems incurs considerable overheads as the common parts are rewritten for each system and application functions are written that already exist in other systems.

The user-images of the systems are often affected by the failure to apply sufficient application oriented expertise to their design, with the result that the systems appear too 'DP oriented'.

The versatility of the systems is hindered by the one-off approach which renders them non-portable, non-enhanceable and even non-correctable.

In view of all these factors, PS was designed as a system and an associated methodology for overcoming as many as possible of the various inefficiencies and difficulties in POL compiler design both simply and economically. The basic aim therefore of the Program Synthesis system is to simplify, standardise and render efficient the design, implementation and issue of user oriented application systems by personnel not versed in the arts of system programming and compiler construction. The design of PS has been according to its own philosophy, keeping the needs of the user uppermost. At the lexical level these needs are slight; simplicity and standardisation rather than flexibility being important as the user should not be concerned with the detailed treatment of identifiers, directives, etc. but merely needs a way of saying what certain things look like or how they are built up.

At the syntactic level the user's needs are greater: he should not have to worry about parsing strategy but will want to determine the syntax of the language he is designing fairly precisely and without hindrance, yet at the same time he must be protected from designing an inconsistent language or one with ambiguities. At the semantic level, the user needs almost complete control in order to make the semantics of the language problem oriented, although he should be discouraged from gross inefficiencies and encouraged to make the semantics match the syntax structurally. The user should not be concerned with details of identifier table maintenance, optimisation or the nature of the target machine—save that he may require control over the hardware representations of simple data types.

3. Application design using PS

The facilities of the PS system would mean little if presented on their own, so this section will describe them from the viewpoint of someone using PS in the intended way. Those parts of the process not involving the computer system will also be described.

3.1. *Vocabulary*

The first stage is to investigate the chosen application area with a view to compiling a vocabulary of its technical terminology. This is essential to the PS philosophy of talking to the user in his own terms; it may be a trivial task but when the application system designer is a tiro in the field of the application it is of great importance. This vocabulary should contain

*Now at ICL Dataskil, 6 The Forbury, Reading RG1 3EQ

for each word as complete a definition of the word as possible, including all implications, assumptions and related information; it will form the basis for the language syntax and its semantics.

3.2. *Language*

The appearance of the language can now be decided: it must be determined whether it will be a command language (algorithmic), a descriptive language (non-algorithmic) or whether it is to have features of both. The language design is essentially a process of deciding what sort of object can be talked about, what can be done to or with these objects and how these operations on them can be combined. The 'sentences' will probably be either mathematical or scientific equations of some sort or simple English (or whatever is the everyday language where the application is to be used) sentences using various of the vocabulary words in certain positions.

3.3. *Syntax*

Once the sorts of sentence or construct allowed in the language have been decided, the syntax can be formalised. SIL accepts a syntax definition in terms of productions made to look more like a programming language. The syntax of a SIL production, in BNF is:

⟨production⟩ ::= ⟨lhs⟩ → ⟨rhs⟩
⟨lhs⟩ ::= ⟨name⟩
⟨rhs⟩ ::= ⟨atom⟩ | ⟨rhs⟩ ⟨atom⟩
⟨atom⟩ ::= ⟨name⟩ | ⟨quote⟩ | ⟨repeat⟩ | ⟨prim⟩
⟨repeat⟩ ::= [⟨rhs⟩]
⟨prim⟩ ::= NM | ID | QU
⟨name⟩ ::= ⟨an identifier⟩
⟨quote⟩ ::= "⟨sequence of non-quote symbols⟩"

Thus the following are legal productions:

SUM → TERM "+" TERM
LIST → "("SUM [","SUM]")"
TERM → ID NM

The first of these corresponds to the BNF production:

⟨SUM⟩ ::= ⟨TERM⟩ + ⟨TERM⟩

and means that a SUM is a TERM followed by the symbol + followed by another TERM. The second means that a LIST is an open bracket, followed by a SUM followed by the contents of the square bracket (comma followed by a SUM) once or more or not at all, followed by a close bracket. The third means that a TERM is either an identifier or a number, as adjacent non-quotes are by convention alternatives. The three productions therefore define LISTs as things like:

(A + 1, B + 1, C + D)
(27 + 1)
(3 + 3, A + FRED)

In addition to the three primitive non-terminals ID NM and QU, meaning identifier, number and quote, the user may define his own; these are called reserved operands, to contrast normal reserved words that appear in quotes in productions and are regarded as operators. For example, a non-terminal BOOL could be defined by:

BOOL IS "TRUE", "FALSE", "UNDEFINED"

The examples of productions and reserved operands are not exactly as they would be written in SIL; a link is needed to the semantic definition. Any proper production, that is one having at least one quote in the rhs, must be followed by a colon, a number, a name in brackets and a quote, for example:

SUM → TERM "+" TERM : 37(RSUM) "SUM"

The number is for identification, the name specifies the relevant part of the semantic definition and the quote provides a user oriented name by which the construct can be referred to in messages (e.g. for automatically generated error messages).

Any reserved operand quote must also be followed by an identification number, for example:

BOOL IS "TRUE" : 1, "FALSE" : 0, "UNDEFINED" : 2

3.4. *Lexical*

Independently of the syntax, the lexical nature of the language can be defined. This includes the various sets of characters. PS needs to know which characters are allowed as:

(*a*) the first character of a name (identifier, reserved operand or operator)
(*b*) the subsequent characters of a name
(*c*) numeric characters (maybe including decimal point)
(*d*) quoted string delimiters
(*e*) layout: space and newline for example
(*f*) numeric signs (+ and − if required, or prefix meaning octal, say)

and also needs a list of all the symbols to be used, things like:

:= ** − (etc.

The other information needed for the lexical definition is:

(*a*) lengths of input and output lines
(*b*) page size and headings, footings
(*c*) appearance of comment indicator, continuation marker
(*d*) end-of-run and end-of-module directives.

3.5. *Testing*

At this stage, when the lexical and syntactic definitions are written, they can be input to PS and checked. The user is informed of any errors, ambiguities or omissions, and if there are no fatal ones (PS makes some small assumptions when minor errors occur) then the tables to drive the analyser of the POL compiler are created. The process of checking the definition may have to be repeated several times in the case of a large syntax as parts may have to be redesigned to remove ambiguities.

3.6. *Semantics*

After the language is defined loosely (3.2) and partially independently of the syntax, the semantics of the POL must be defined to the PS system. This involves two processes: the first consists of writing routines to perform the basic functions of the application and the second consists of writing, in SIL, routines to process the constructs that are defined in the syntax as being legal. Writing the basic application functions can proceed at any stage, it is really independent of PS and would probably be done any way even in an ad hoc implementation. The semantic processing routines must do two things: they must be capable of rejecting meaningless but syntactically valid constructs and they must be able to specify the meaning of semantically valid constructs.

SIL has as a subset a special programming language for writing these semantic routines. This semantic language provides most of the facilities of an ALGOL (Naur, 1963) or BCPL-like (Richards, 1969) language with the exception of block structure; it provides in addition several less common facilities:

1. Heap storage—for dynamically-created data items.
2. Pointers—generalised (typeless) reference variables to heap data.
3. Structures—similar to ALGOL 68 (van Wijngaarden *et al.*, 1969) but without its typed references.

The semantic definition consists of semantic routines and procedures: the latter are quite standard procedures with parameters passed by reference, the former have special significance.

Each name in brackets after a production is the name of a semantic routine that is responsible for processing any parsed construct matched by that production. The semantic processing is top-down, the routine corresponding to the top level (root) node of the parse tree is entered with the node as a parameter; subtrees are then processed by a special command in the semantic language, COMPILE, which causes the *N*th subtree (*N* being the operand of COMPILE) to be processed by entering its top level node's semantic routine. There are several other commands available within the semantic language:

GENERATE—this generates code for a high level abstract target machine (the ATM)

DECLARE —this implements data declaration in the ATM and takes care automatically of identifier table structures

PRESET —this sets up the structures for pre-setting declared data

SEMERROR—this causes an error message to be output; details of the erroneous construct, its line number and character position are incorporated automatically

Through these commands PS automatically, though still ultimately under the control of the user, generates and maintains the data structures required for the compilation and also generates the code for the ATM. The process of mapping from this on to an actual hardware is completely automatic save in one respect; the user can specify which actual machine representations are required for the various ATM data types. The user defines this in a part of SIL called the type table definition.

3.7. *Messages*

The actual text of messages output by the semantic routines, and also that produced on encountering standard syntactic errors, is specified in the 'messages' section of SIL. Each message text has an associated number (to identify it in the semantic routines or to the analyser) and a level number to indicate the seriousness of the cause of the message, one of: system crash, error, warning, comment and ignore.

3.8. *Testing*

The semantic routines and procedures, type table and messages, can be input to PS and compiled separately. When the whole definition has been processed the data and code modules produced are collected together with the invariant parts of the compiler to produce the finished POL compiler ready to be system tested.

4. The structure of PS

PS can be viewed at several different levels. Firstly the function of PS in its intended environment can be examined; secondly PS can be seen as a modular compiling system subject to its own design philosophy; thirdly the internal structure of the system's components can be looked at.

4.1. *PS in its environment*

The environment for which PS is intended is one in which a variety of application systems are to be produced and used with a variety of different machines. The total system in such a case will consist of the Program Synthesis System, a set of compiled language definitions for its applications, and a set of code generators for the different machines. In this environment, if a new application is designed and its formal definition submitted to PS, then the application is available for all the machines. Similarly if a new code generator is implemented it becomes available to all the applications, and if an enhancement is made to PS, or an increase in efficiency, then it will be available to all the applications on all the machines.

4.2. *PS as an application*

As PS can be regarded as an application compiler, it has been designed in such a way that its user interface is a special POL for compiler construction.

The basic functions of PS, included in every compiler produced by the system, are as follows: a table-driven lexical analyser, to read source text and split it into basic words and symbols; a table-driven syntax analyser to take the output of the lexical analyser and parse it into a tree structure; a facility to organise the processing of the parse tree; a control routine to provide working store, perform any initialisation, call the analysers and initiate the tree processing; a set of routines to provide name and identifier table access, type conversion, declarative and generative functions. These routines are part of every POL compiler including the SIL compiler.

The variable parts of PS can be further subdivided into the part concerning input and the part concerning output. The former consists of tables and routines formed from a particular POL definition by the SIL compiler; they are as follows: the tables to drive the lexical analyser; the tables to drive the syntax analyser; the routines and procedures for processing the parse tree; details of the various messages that the system may have to output. The latter part of the subdivision consists of the code generator, which varies with machines but not with applications, and the user-written type table defining the mapping from the ATM data types onto the target hardware which is clearly dependent both on machine and application.

4.3. *The internal structure of PS*

The standard PS lexical analyser is a simple, parametrised 'Glennie-type' syntax machine (Glennie, 1960) that processes its input serially. Elementary syntactic items are output as they are identified, except when two consecutive operands are detected, in which case the output must be modified to turn it into an operator form suitable for the parser by the insertion of a 'null' operator. The only complications are with numbers; in the expression $A := -3$ the minus sign is best regarded as part of the number, whereas in $A := A - 3$ and $A := -B$ it must be regarded as an operator in its own right. Consequently the number recogniser must be aware in a small way of the context before it can make a decision.

The PS syntax analyser is an operator precedence parser similar to that described by Floyd (1963). Its error recovery at present is rather primitive in that it can only deal with precedence clashes by the insertion or deletion of items, and it makes no attempt at a best fit when faced with an erroneous construct.

The semantic analysis is under the control of the user, although he is encouraged (not constrained) to traverse the parse tree in a logical top-down manner by the nature of the COMPILE command. The various declarative procedures provided set up and maintain identifier table entries and associated chains of information dealing with scope, type, structure and preset values. The generative procedures organise the generation of code for the abstract target machine, accessing the identifier table entries where necessary.

The code generation is of course different for all the possible actual target machines, although in each case will consist of several modules including the following: storage allocation, register allocation, local and global optimisation and module output.

In addition to the above components, PS has various ancillary procedures to do with input/output control. Although the basic I/O interface must be machine dependent, consisting of special machine instructions or supervisor calls, PS has standard procedures for text input, source and message listing, page

layout, etc. as well as for message construction. The last consists of attaching to a message its cause, that is the construct name and line number, and inserting parameters into a predefined string.

All the above are part of every PS-based compiler. In addition to these PS consists of the set of semantic routines for compiling SIL. The executable parts of the language are processed in quite traditional ways and consequently will not be described. The analysis of the syntax definition to produce precedence tables is a two-pass process: the first pass consists of traversing the productions to find the left and right terminals and non-terminals for each non-terminal in the definition, the transitive closures of these sets are then found; and the second pass consists of traversing the productions again using the left and right terminals to set up the precedence tables.

5. Enhancements to PS

The immediate future work on the system is to complete its implementation up to full specification; sufficient experience has however been derived from using the various parts of the system to enable several enhancements and alterations to be considered. Thus far none have been decided as certain but all are possible directions for future development.

The lexical analyser was designed to provide a certain set of facilities and this has been perfectly adequate for implementing the sorts of language intended. There are however many features of existing programming languages that cannot be implemented, for example hexadecimal numbers, Hollerith strings, and the use of compiler directives. Consequently it is being considered whether to adapt the lexical phase to a more general analyser driven by a special sort of syntax definition, yielding perhaps a 'Glennie machine' (Glennie, 1960) completely user-defined. It is also being considered whether to incorporate a macrogenerator into the PS system; although it is not clear whether this would be best as an integral part of the lexical analyser or as a freestanding entity.

There are two ways in which the parser could be adapted. In the present implementation the error recovery is rather naive; all that is needed however is further work as there are many good techniques that could be incorporated (Graham and Rhodes, 1975; James and Partridge, 1973). The main adaptation under consideration is the provision of a variety of different parsing strategies; in some circumstances there may be a need for more powerful though less efficient parsing when a top-down analyser would be preferable, or one of less pure design.

The semantic language part of SIL had its design frozen somewhat later than the analyser and consequently the enhancements considered up to that point have been incorporated. Any additions will now be in the form of ancillary procedures or amended versions of existing procedures. The grammar analysis part of the SIL compiler at present merely flags any errors or ambiguities in the language definition; it would be useful if in such cases fuller details could be given of how the errors can be avoided. In the case of correct language definitions it should be possible to produce a summary of the syntax, almost in the form of a little manual, together with precedence and other tables.

Taking a longer term view, it would be desirable for PS to be able to generate interactive compilers and maybe to work interactively itself. This possibility has yet to be considered in detail.

A list of possible enhancements, in order of increasing vagueness, could go on indefinitely. PS has turned out to be an excellent generator of potential research topics, as working with it involves contact with many fields, including language design, compiler error recovery and reporting, interactive compilation, optimisation, and machine architecture. It is hoped that as the system grows in size and power it will also become nicer to use.

6. Conclusions

The investigations concerned with the development of PS showed that much of the work in producing application systems was of a routine nature and could be performed either once for all or by machine. The PS system is the embodiment of these findings and the PS methodology developed along with the system. The PS system implementation language has been used to define the system itself and this has shown several things. The syntax definition part has proved capable of supporting the definition of a large and varied syntax; it has proved capable of detecting logic errors in that definition that were missed in a manual checking. The semantic language part of SIL has been shown quite adequate as a system implementation language for all the semantic routines written, and has been found to be clear and rapid to write although the use of a macrogenerator would considerably reduce the quantity of code written. The structure of PS has been found to encourage generality and transparency in the things defined with it, and the system itself has been able to accommodate several minor design changes in mid-implementation that would have caused far-reaching setbacks in a more ad hoc system. Although a complete application system has not yet been implemented by PS, several studies of extant and proposed systems have been carried out and it has been found that PS provides a good framework for rapid and fairly comprehensive design studies.

There have been many proposals for further work and enhancement on PS, the system having turned out to be a good research problem generator but the directions adopted will depend on how PS performs in its intended environment, that is, one of many systems being designed and implemented for many different application areas and on many different machines; in such an environment PS should free its users from routine and repetitive tasks, from reinventing the wheel, and allow them to concentrate their expertise on the application areas or on the target hardware, hopefully bringing the application user a more efficient, friendly system.

Acknowledgements

The authors would like to thank William Frazer-Campbell for programming and testing a considerable part of the PS system.

References

In addition to those mentioned explicitly in the text, several references are included to works that have to a lesser extent or in less well defined ways influenced the design and development of the PS system.

Languages

NAUR, P. (Ed.) (1963). Revised report on the algorithmic language ALGOL 60, *CACM*, Vol. 6, No. 1, pp. 1-17.

RICHARDS, M. (1969). BCPL reference manual, Tech. Monograph 69/1, University of Cambridge Computing Laboratory.

VAN WIJNGAARDEN, A., MAILLOUX, B. J., PECK, J. E. L., and KOSTA, C. H. A. (1969). Report on the algorithmic language ALGOL 68, *Numerische Mathematick*, Vol. 14, pp. 79-218.

Syntax analysis

BURGESS, C. J. (1972). Compile time error diagnostics in syntax directed compilers, *The Computer Journal*, Vol. 15, No. 4, pp. 302-307.

FLOYD, R. W. (1963). Syntax analysis and operator precedence, *JACM*, Vol. 10, pp. 316-333.

GLENNIE, A. E. (1960). On the syntax machine and the construction of a universal compiler, Carnegie Tech. Computation Centre Report No. 2.

KHABBAZ, N. A. (1974). Multipass precedence analysis, *Acta Informatica*, Vol. 1, pp. 77-85.

LEDGARD, H. F. (1970). Production systems, a formalism for specifying the syntax and translation of computer languages, Oxford University Computing Laboratory Programming Research Group Monograph PRG-1.

Error recovery

GRAHAM, S. L. and RHODES, S. P. (1975). Practical syntactic error recovery, *CACM*, Vol. 18, No. 11, pp. 639-650.

JAMES, E. B. and PARTRIDGE, D. P. (1973). Adaptive correction of program statements, *CACM*, Vol. 16, No. 1, pp. 27-37.

Semantics

ABRAMSON, H. (1973). *Theory and application of a bottom-up syntax-directed translator*, Academic Press.

GRIES, D. (1971). *Compiler construction for digital computers*, John Wiley.

MOSSES, P. (1974). The mathematical semantics of ALGOL 60, Oxford University Computing Laboratory Programming Research Group Monograph PRG-12.

Book review

Computational probability and simulation, by S. J. Yakowitz, 1977; 240 pages. (*Addison-Wesley*, £18·00 hard cover, £10·00 paper)

Mathematical Modelling and Digital Simulation for Engineers and Scientists, by J. M. Smith, 1977; 332 pages. (*Wiley*, £14·50)

Yakowitz v. Smith or Monte Carlo v. The Rest

These two books share a common aim in so far as both are aimed at finding a numerical solution to a range of problems intractible to the analytical methods of classical mathematics. However, the declared objectives of the two authors are quite different. Dr. Yakowitz is an academic and, armed with this information, your reviewer was unsurprised to find that his book is based on teaching material. (Let me quickly add, before my academic friends disown me, that I do not intend that to be a derogatory remark . . . read on). Mr. Smith, on the other hand, is in the world of commerce and although he does not say so, I assume his interest is in practical solutions. Now the paradox is that Mr. Smith has written an 'academic' book using mathematical methods of numerical analysis while Dr. Yakowitz has concerned himself much more with the practical aspects of simulation.

From the biographical data given, courtesy Library of Congress, I infer that both authors entered their careers in the early, golden, days of computing when there really was a man/machine interface, particularly in the scientific branch of computing. At that time, computers showed the way to seemingly endless vistas of computed solutions while causing immediate problems to arise from lack of the two main computer resources: space and time. Great ingenuity was shown then in overcoming these problems by the use of the more obscure methods of numerical analysis. At the same time, the shackles of manual computation were thrown off, a machine could be expected to perform the same arithmetic repeatedly for hours and Monte Carlo methods were born. Now we have two books describing precisely those techniques, or at least those techniques improved by 20 odd years of practical use. Have we at last passed from the euphoria induced by megabytes of core and picosecond processing times to being once again concerned in stretching the resources available to solve the real problems which face us? I fear the answer is No! Mr. Smith is preaching the word but Dr. Yakowitz is more concerned with using the power available to solve problems intractible by any other method.

Dr. Yakowitz starts his book by describing some of the problems associated with the generation of pseudo-random numbers. He highlights the problem of testing for randomness but concludes that the random number generator used in this book is adequate. So that the reader (student) may try out the examples, the FORTRAN code for a random number generator is given. (The FORTRAN used is for a CDC 6400 with a 64 bit word). The book then introduces probability theory through the medium of throwing dice. From here the simulation of other functions is only a short step with sample FORTRAN code for one function and sufficient information in the text to enable other functions to be simulated. The chapter concludes with justification of an algorithm and some exercises and references.

This format is repeated in chapters on random walks and gambler's ruin (proving, of course, by computer that you can't win). Up to this point the book justifies its claim to give an experimental introduction to probability. The last two chapters are somewhat heavier going, giving the theory of limiting processes and Monte Carlo integration without the leaven of sample programs. However, by this time the diligent reader will have learned enough of the techniques to write his own examples.

A criticism of this book might be that it contains assertions and justifications but no proof. It does succeed in giving a practical education to a reasonable standard which is its aim. It gives a wealth of references and is in the main, a readable book with an enjoyable practical content.

From a book whose subject was literally created by the computer to one whose subject appeared destined to become a Cinderella until the value of fast stable methods of numerical solution was appreciated. Mr. Smith writes his book from his experience gained in solving problems arising in engineering. The methods described in this book are not the classical methods I mentioned earlier but modern adaptations to suit a wider variety of conditions. For example, certain classical numerical integration methods are shown to be the same integrator 'tuned' to correct different errors.

Because of its mathematical approach, this book is not light reading, but the style is clear and subjects are presented in a logical way. The subject of mathematical modelling is handled in two stages. The first two chapters describe the problems arising in describing a physical system in mathematical terms, modelling techniques considered include Laplace transformations, signal reconstruction, and even use of simple linear systems. By following the methods for describing the object system, the modeller will then find methods for solution in the remainder of the book. This is, perhaps, a simple view of the problem, but the point being made by Mr. Smith is that a successful simulation starts with the original formulation and not at the stage of solving equations. The tools described include linear and non-linear integration methods, Chebyshev polynomials fast evaluation techniques and assessments of useful formulae. The emphasis is on speed with accuracy, stability of solutions, and control of the growth of errors. I would describe this as a very useful book both for reference and as an introduction to mathematical simulation.

Dr. Yakowitz, on page 100, states that the solution of a system of simultaneous linear equations can be found by a Monte Carlo method in one sixth of the time taken by Gaussian elimination. Since the two authors have not covered quite the same ground, perhaps a reader of this review will accept the challenge and model a system using both techniques, reporting back in this Journal . . . .

R. E. SMALL (London)