GENERATION OF INCREMENTAL INDIRECT THREADED CODE FOR LANGUAGE-BASED PROGRAMMING ENVIRONMENTS[1]

KHALID AZIM MUGHAL[2]

Abstract

We present an approach to generating incremental threaded code for language-based programming environments from the specification of the runtime semantics of the programming language. Language-based environments (LBEs) that support incremental code generation have usually done so using ad hoc techniques for incremental recompilation. Our aim is to provide one uniform operational model based on attribute grammars that allows the specification of the runtime semantics, and thus code generation, to be incorporated with the specification of the syntax and static semantics of the language.

The proposed semantic model of incremental code generation allows specification of compact code. It severely limits the repropagation of semantic information in the program tree due to changes in the code caused by modifications to the source program. Extended with interactive execution, this representation also facilitates the implementation of such debugging features as control-flow tracing, single-stepping and value-monitoring at the source level.

We have demonstrated the feasibility of the proposed model by implementing a program editor for Pascal. The specification is written in the Synthesizer Specification Language (SSL) and forms the input to the Cornell Synthesizer Generator (CSG).

1 Introduction

We are interested in providing code generation to support program interactive program and debugging. The main problem is mapping of the internal representation of the program maintained by the LBE to an executable form. The techniques are implemented in the incremental framework provided by CSG [Reps & Teitelbaum 1988].

---

[1] This work was partially supported by the Norwegian Research Council of Sciences and Humanities.
[2] This paper is based on excerpts from the author's dr.scient thesis [Mughal 1988].
Author's address:
UNIVERSITY OF BERGEN
DEPARTMENT OF INFORMATICS
ALLÉGT. 55, N-5007 BERGEN, NORWAY.
Phone: 47-5-212879 Telefax: 47-5-212857
Internet: khalid%eik.ii.uib.no@uunet.uu.net

We are concerned with *formal* specifications for *generating complete* LBEs. There are primarily three schools of formalisms for such specifications. The *semantic actions* approach as exemplified by the GANDALF editors [Habermann et al. 1982]. The *attribute grammar* [Knuth 1968] approach as exemplified by CSG [Reps & Teitelbaum 1988] and POEGEN [Fischer et al. 1984]. The third approach is based on *denotational semantics* [Stoy 1977] and is primarily employed in compiler generators: SIS [Mosses 1976], [Paulsen 1981]. EDS [Pal 1986] is a denotational semantics based generator for LBEs. However there are systems that employ a more hybrid approach for such formal specifications: PSG [Bahlke & Snelting 1985], PECAN [Reiss 1984a]. *Action equations* are advocated for such specifications in [Kaiser 1986]. A parallel development in this direction of tool generators can be seen in the field of compiler-compilers: GAG [Kastens et al. 1982], HLP [Räihä 1984], SIS [Mosses 1976].

2 Incremental Recompilation

In order to provide efficient execution facilities, we have to address the problem of incremental recompilation: minimize the amount of computation necessary to generate and maintain executable code in the face of source program modifications. We would like to reuse as much of the old code as possible after a program is modified. Before we look at some approaches to the incremental problem, it will be useful to distinguish between the *granularity of recompilation* and *extent of recompilation* [Reps & Teitelbaum 1988]. By granularity of recompilation we mean the *size* of program fragment which can be modified before recompilation is necessary to reflect the source program change in the generated code. The extent of recompilation refers to the amount of code that must be recompiled to reflect the change made to the program.

Hand-crafted LBEs that provide runtime facilities vary in their approach to providing incremental recompilation. In the Cornell Program Synthesizer [Teitelbaum & Reps 1981] incremental recompilation is dependent on the language construct modified. The granularity of recompilation coincides with the extent of recompilation for a control construct, an assignment or an expression. However modification of declarations requires more extensive reanalysis of the program. The extent of recompilation can thus be the entire procedure.

The DICE (Distributed Incremental Compiling Environment) system [Fritzson 1984] supports incremental recompilation at the statement level in a host-target machine configuration. DICE provides statement level granularity of recompilation and supports source-level debugging. The extent of granularity is primarily updating the code for the statement on the target machine. In the MAGPIE system [Delisle et al. 1984] [Schwartz et al. 1984], the granularity of recompilation coincides with the extent of recompilation which is the entire procedure. PSEP [Ford & Sawamiphakdi 1985] uses again a different approach to incremental recompilation where editing and code generation are concurrent processes. In the PSG system

only incremental compilation is possible, not incremental recompilation. PSG performs incremental compilation during top-down derivation of the abstract syntax tree (AST) by series of refinement steps. PECAN editors [Reiss 1984b] limit the recompilation to what is called a *compilation unit* which may be a statement or a procedure depending on the context.

In CSG, its attribute evaluation mechanism can be employed for incremental recompilation. Code can be defined as instances of attributes attached to nodes of the AST. After program modification, the attribute-updating mechanism also updates the code attributes that need reevaluation. Consequently the extent of recompilation in our model is dependent on the number of code attributes that need updating. CSG provides two (limited) ways for controlling the *granularity* of recompilation for creating editors with execution facilities. One is to specify that certain program fragments can only be edited as an entire unit. The second method makes use of *demand attributes*. Such attributes are given values only when a demand is placed on them. For example when such an attribute is an argument to another attribute which needs evaluation. We present ways in which to limit the *extent* of recompilation due to changes in the value of code because of program modifications. [Reps & Teitelbaum 1988] and [Reps & Teitelbaum 1987] are the authoritative references on CSG. [Reps & Teitelbaum 1988] and [Reps 1984] describe efficient methods for incremental attribute evaluation in LBEs.

3 Incremental Indirect Threaded Code

Our code generation model is based on incremental semantic analysis. [Reps 1983] is a discussion of static semantic analysis in LBEs. We are interested in applying the power of this incremental model for implementing runtime facilities for LBEs. Any efficient implementation realizing this goal must take the following two factors into consideration: it must minimize the change propagation affecting code generation due to program modifications, and secondly, it must provide an efficient execution of the generated code. The execution paradigm is interpretation of generated code on a von Neumann machine. Thus we provide immediate program translation and interactive execution.

Since we want to make use of the incremental attribute mechanism for code generation, we can define an attribute that represents the code for a node in the AST. The code for the program is thus fragmented and attached to various nodes of the AST as attribute values.

One scheme for maintaining this fragmented code would be to *synthesize* the code for the *root* of the AST from the code fragments at the subordinate nodes in the AST. Thus the attribute value of the code at the root of the AST would represent the code generated for the program under development. This approach is used in [Milos et al. 1984] for the generation of a Pascal P-code compiler. They describe a *semantic grammar* [Paulsen 1981] which is an amalgamation

of *extended attribute grammars* [Watt & Madsen 1983] and denotational concepts [Stoy 1977]. Appropriately transformed P-code is then run on the SECD machine [Mosses 1976]. This might provide for efficient execution of the code but is very inefficient for incremental updating of the generated code. In an incremental attribute evaluation scheme, any change to a code fragment of a subordinate node would mean propagation of this change to the root of the tree.

Another approach would be to coalesce or linearize this code just prior to execution time. This would introduce the overhead due to the extra step of coalescing the code every time the program was executed. It would avoid excessive change propagation but would entail additional book-keeping in mapping the code back to its respective node in the AST for debugging purposes. This extra book-keeping is also necessary for the previous approach.

We describe a scheme where the code fragments are "threaded". In order to thread the code fragments, we must know the entry point to a code fragment and this entry point must be made available to other code fragments that "exit" to it. This is modeled in our scheme by using two attributes **entry** and **completion** which explicitly define the entry to a code fragment and the entry to the next code fragment to execute on the completion of execution of the current one. **Entry** and **completion** attributes are used to thread the code fragments. Entry to and exit from code for language constructs is only possible via these two attributes respectively. Hence the term threaded code [Bell 1973] appropriately describes our code generation strategy

We will describe this approach in terms of code generation for statements found in a block-structured language like Pascal. For a *statement* node, we define a *synthesized* attribute **entry** that indicates the entry point corresponding to the code of that statement and whose value is then available to the statement's parents and siblings in the AST. These nodes can exit to the *statement* node using the **entry** attribute value. In addition, we define an *inherited* attribute **completion** whose value the *statement* node inherits from its parents or siblings and can incorporate in its own code. The value of the **completion** is effectively the entry point of the next code to be executed after the current statement is completed. These two attributes thread the fragmented code together while allowing the individual code fragments to remain attached to the nodes of the AST as attribute values.

Since we are using attribute grammars to specify generation of threaded code, we have to be careful about *circularities* in our specification. The diagram in Figure 1 shows the *flow of control* in the linked code structure for the **while** loop using **goto** instructions. This is based on the traditional semantics of a **while** loop:

a) evaluate the loop condition.

b) if condition is not true, go to d.

c) execute the loop body and go to a.

d) ...

A **while** loop is typically defined by the following production:

Statement$1 ::= **while** expr **do** Statement$2

In the AST representing the **while** loop, the code for the loop condition is generated in the subtree with root *expr*. The code for the loop body is generated in the subtree with root *Statement$2* and the code for the loop test is generated at the node (corresponding to the above production) for the nonterminal *Statement$1*. Figure 2 shows the flow of control in the **while** loop when we use the **entry** and **completion** attributes of the nonterminals *Statement$1*, *expr* and *Statement$2* to thread the **code** of these nonterminals. The flow of control in Figure 2 corresponds to the flow of control shown in Figure 1. However, in Figure 2, the threading of the code for the loop test, body and condition of the **while** loop via the **entry** and **completion** attributes leads to circularity in the *dependency graph* of these attributes. The dependencies are in reverse, opposite to the flow of control arcs shown in Figure 2.

The Cornell Synthesizer Generator however requires a noncircular attribute grammar as its input. In general, the circularities that cause problems in SSL are due to the circularities in the *attribute equations* of the specification. An attribute variable A on the right side of a semantic

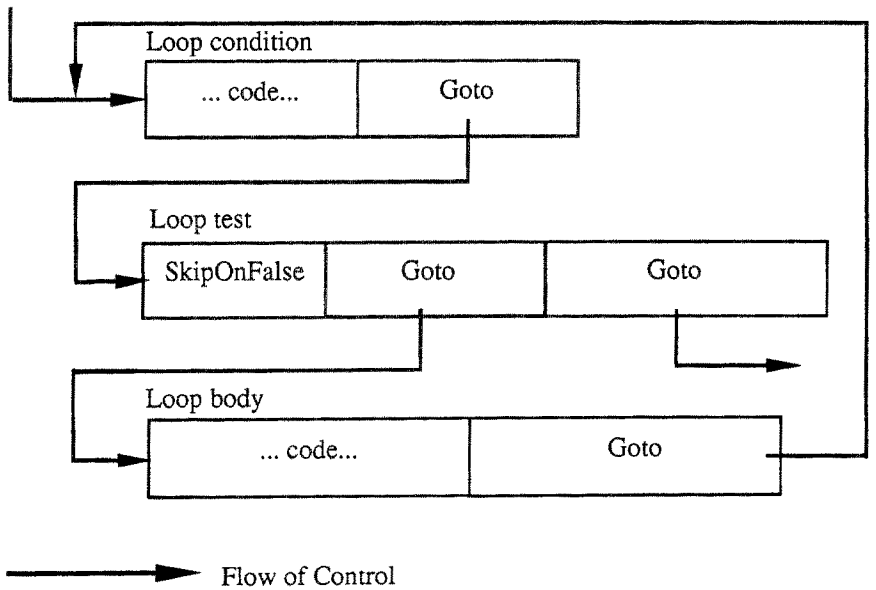
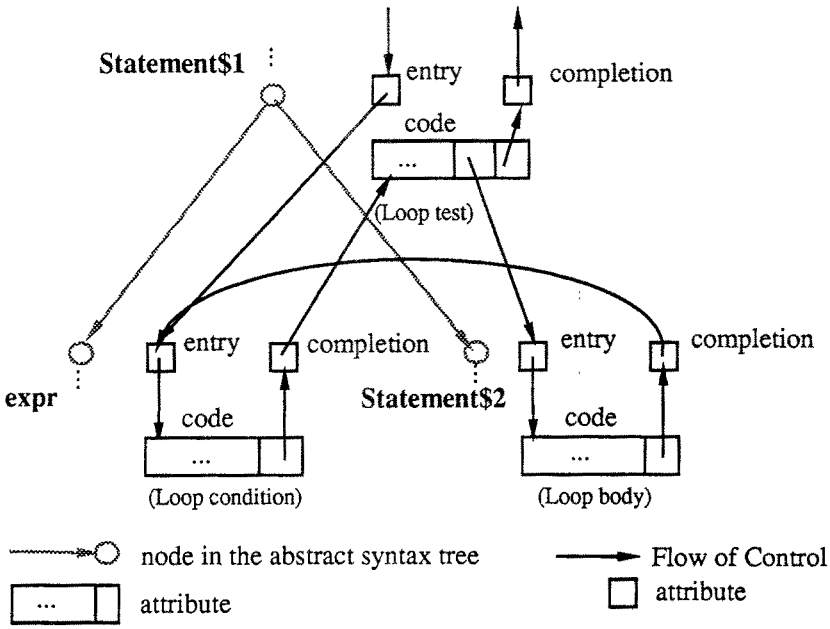


Figure 1: Control Flow Graph for the **While** Loop

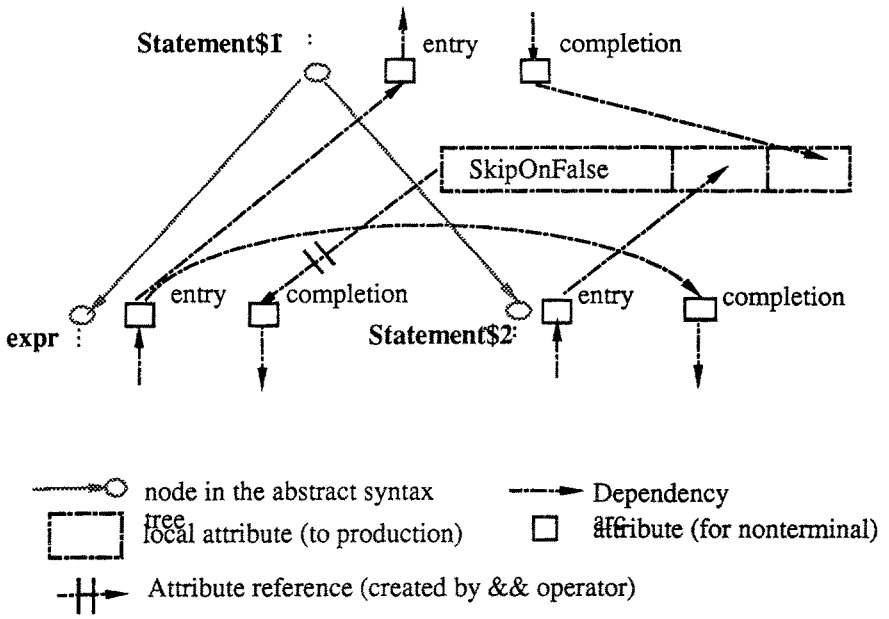


Statement$1 ::= **while** expr **do** Statement$2

Figure 2: Threaded Code for the **While** Loop

equation for attribute variable B, introduces a dependency from A to B. We overcome the circularity problem by introducing a level of indirection in the generated code by making use of the built-in, primitive "type" ATTR of *attribute references* in SSL. If A is an attribute variable, then &&A is an ATTR-valued expression whose value is a reference to A. &&A is called an *attribute reference*. The semantics of the && operator are such that a semantic equation defining an attribute B in term of &&A does *not* introduce any dependency from A to B.

We can represent the links in the generated code by ATTR-valued expressions. When a production is applied, an attribute occurrence of the code fragment is created. **Entry** and **completion** are then *attribute references* to this code created by the && operator. This strategy effectively breaks the circularity mentioned above as the values of these attributes are no longer dependent on the value of the code fragments. Hence we have dispensed with the use of a fix-point operator for mutually dependant code values by the use of attribute references. In Figure 2 we can think of the flow of control arcs (or threads) as being attribute references. SSL does not provide a standard dereferencing operator for ATTR-values. During execution, a special dereferencing operation performs the indirection required to access the next code fragment, given the ATTR value from the current one. Thus we have an execution scheme based on *indirect* threaded code, analogous to that described in [Dewar 1975].



Statement$1 ::= **while** expr **do** Statement$2

Figure 3: Dependency Graph for the **While** Production

We also define the code as a *local* attribute of the *production*, rather than that of the nonterminal *Statement*. Code as an attribute of a nonterminal would require specifying attribute equations even for cases where the nonterminal does not require the specification of code. Code as a local attribute thus allows for generation of much more compact code. Another major consequence of this approach, combined with the use of attribute references for linking the code, is that change propagation due to change in the value of the code is limited to the production in which the code fragment is defined.

Figure 3 shows the dependency graph for the attributes in the **while** *production* in the AST. In this particular production there is only one local code attribute which is threaded in the overall code for the **while** construct using **entry** and **completion** attributes. The arc ⫴→ does not represent a true dependency and breaks the circularity cycle mentioned above. The value of *expr*.**completion** is an attribute reference created by applying the && operator to the local code. Note also that the entry point to the **while** loop is the entry point of the *expression*. This is made available to whatever statement that precedes this production in the AST via *Statement$1*.**entry**. The interpreter is thus able to transfer control directly to the expression rather than leapfrogging first to this production and then jumping to the expression. This also allows for the generation and execution of compact code.

As pointed out earlier not all nodes need to specify new code. In many cases it is only necessary to thread the existing code properly to reflect the flow of control using their **entry** and **completion** attributes. A case in point are the grammar productions for a sequence of statements were the threading information in their **entry** and **completion** attributes is passed on to the other nodes but no new code is generated. Threading code in this manner allows for efficient execution as the sequencing of instructions is indicated by the code's threads.

Another advantage of this approach is that it also facilitates the tracing of control flow in the attributed syntax tree during program execution, requiring a simple mapping from the attribute reference to its production in the tree.

4 Continuations and Completions

The incremental indirect threaded code generation scheme described in the previous section can be thought of as a *completion semantics* [Henson & Turner 1982]. A *completion* is an operational representation of a *continuation* in the denotational semantic model [Stoy 1977]. A continuation specifies what function the "remaining program" computes, and is defined in reverse - opposite to the flow of execution in the program. A completion is a concrete representation of this function.

The correspondence between the continuation structure of a program and a flow graph of a program is expounded in [Sethi 1983]. Each statement defines a continuation in terms of a supplied continuation and the denotations of its subcomponents. A special notation for *pipes* is used to specify the construction of flow graphs for the control constructs of the C programming language. A special combinator *cycle* is used for recursively defined continuations as in the case of loops. The edges of the flow graph are analogous to the threads in the generated code in our scheme.

In [Reppy & Kintala 1984], a case is made for automatically generating complete *AG-based* LBE specifications from *continuation* semantics by techniques similar to those in [Sethi 1983]. The executable representation is similar to our scheme: *code trees* containing op-codes for an abstract machine are generated and executed by an interpreter for the corresponding abstract machine. In contrast to our scheme, it however uses semantic functions with side effects to mutate the internal values of generated code trees.

5 Implementation Status

A specification for a Pascal editor with full *static semantic checking* is distributed with the CSG release [Teitelbaum et al. 1987]. This specification has been augmented with the runtime semantics of Pascal to provide runtime facilities.

SSL Pascal-I, which is described in [Mughal 1988], is an implementation of a complete program editor for a subset of Pascal where both the runtime semantics and the runtime environment are realized in SSL. The generated code is P-code [Nori et al. 1975] defined and modified in SSL to produce incremental indirect threaded code. SSL Pascal-I demonstrates the implementation of non-trivial control flow aspects of language constructs such as loops, recursive procedure call, different modes of parameter passing (including procedure parameters), handling of labels and non-local gotos, and other structured statements. SSL Pascal-I supports only primitive data types in Pascal as these can be easily represented directly by data types of SSL. The runtime management includes a tail-recursive interpreter and a runtime shallow-binding store, also implemented in SSL. The program editor incorporates such debugging facilities as control-flow tracing, single-stepping and value-monitoring at the source level.

SSL Pascal-I demonstrated the feasibility of incremental indirect threaded code, but the implementation of the *runtime facilities* was not efficient due to the limitations of the specification language SSL. Experience gained from this implementation has been valuable in the next undertaking of providing efficient runtime facilities for editors generated in the framework of CSG [Teitelbaum et al. 1988]. The interpreter is written in C and the code instructions are defined as a *primitive data type* CODE of the specification language SSL. Constructors are provided for all P-code attribute values of the primitive data type CODE. Thus the specification of code generation from SSL Pascal-I carries over to this scheme almost in its entirety. However the execution environment is stack-frame based implementing deep-binding in contrast to the shallow-binding-by-name scheme of SSL Pascal-I. The deep-binding scheme calls for the calculation of address offsets of variables during incremental threaded code generation. The code generation scheme of SSL Pascal-I is thus modified with respect to the offset calculation. The new scheme also supports structured and dynamic data types of Pascal. Debugging features include flow tracing, single-stepping, limited execution resumption after program modification and a form of COME command [Alberga et al. 1984]. For obvious reasons, the interpretation in C is far more efficient than the one in SSL Pascal-I. It is hoped that this Pascal editor will be released with the next version of CSG.

6 Summary

We have presented a viable non-ad hoc scheme for the specification of runtime semantics based on indirect threaded code in the framework provided by the CSG. It allows the implementation of non-trivial control flow aspects of many language constructs and facilitates implementation of certain runtime features. We believe this approach can be generalized to generate LBEs with runtime facilities for other block-structured languages. Ultimately one would like to provide the

editor designer with a generic package for implementing runtime semantics and debugging facilities, with "hooks" into the system for tailoring and augmenting the runtime model. We believe this work to be a step in that direction.

7 Acknowledgements

I would like to thank Professors Tim Teitelbaum and Tom Reps for allowing me to work on the Cornell Synthesizer Generator Project. Support from the rest of the CSG Task Force, both past and present, is also deeply acknowledged.

8 References

[Alberga et al. 1984]
Alberga, C.N., Brown, A.L., Leeman,Jr. G.B., Mikelsons, M., and Wegman, M.N.
A Program Development Tool.
IBM J. Res. Develop., Vol. 28, No. 1, January 1984, 60-73.

[Bahlke & Snelting 1985]
Bahlke, R. and Snelting, G.
The PSG - Programming System Generator.
In Proceedings of the ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments, Seattle, WA., June 25-28, 1985, 28 - 33. (SIGPLAN Notices 20, 7, July 1985.)

[Bell 1973]
Bell, J.R.
Threaded Code.
Communications of the ACM 16, 6 (June 1973), 370 - 372.

[Delisle et al. 1984]
Delisle, N. M., Menicosy, D. E., and Schwartz, M. D.
Viewing a Programming Environment as a Single Tool.
In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Penn., April 23-25, 1984, 49 - 56. (Joint issue: SIGPLAN Notices 19, 5, May 1984, and Software Engineering 9, 3, May 1984.)

[Dewar 1975]
Dewar, R.B.K.
Indirect Threaded Code.
Communications of the ACM, 18, 6 (June 1975), 330 - 331.

[Fischer et al. 1984]
Fischer, C.N., Pal, A., Stock, D.L., Johnson, G.F., and Mauney, J.
The POE Language-based Editor Project.
In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Penn., April 23-25, 1984, 21 - 29. (Joint issue: SIGPLAN Notices 19, 5, May 1984, and Software Engineering 9, 3, May 1984.)

[Ford & Sawamiphakdi 1985]
Ford, R. and Sawamiphakdi, D.
A Greedy Concurrent Approach to Incremental Code Generation.
In Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, January 14 - 16, 1985, 165-178.

[Fritzson 1984]
Fritzson, P.
Preliminary Experience from the DICE system: a Distributed Incremental Compiling Environment.
In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Penn., April 23-25, 1984, 113 - 123. (Joint issue: SIGPLAN Notices 19, 5, May 1984, and Software Engineering 9, 3, May 1984.)

[Habermann et al. 1982]
Habermann, A. N., Ellison, R., Medina-Mora, R., Feiler, P., Notkin, D.S., Kaiser, G.E., Garlan D.B., and Popvich, S.
The Second Compendium of Gandalf Documentation.
Dept. of Computer Science, Carnegie-Mellon University, 1982.

[Henson & Turner 1982]
Henson, M.C.and Turner, R.
Completion Semantics and Interpreter Generation.
In Conference Record of the 9th. ACM Symposium on Principles of Programming Languages, Albuquerque, N.M., January 25-27, 1982, 242-254.

[Kaiser 1986]
Kaiser, G.E.
Generation of Runtime Environments.
In Proceedings of the SIGPLAN '86 SYMPOSIUM ON COMPILER CONSTRUCTION, Palo Alto, Calif., June 25-27, 1986, 51-57. (SIGPLAN Notices 21, 7, July 1986.)

[Kastens et al. 1982]
Kastens, U., Hutt, B., and Zimmermann, E.
Lecture Notes in Computer Science, vol.141: GAG: A Practical Compiler Generator.
Springer-Verlag, New York, 1982.

[Knuth 1968]
Knuth, D.E.
Semantics of context-free languages.
Mathematical Systems Theory 2, 2 (June 1968), 127 - 145.

[Milos et al. 1984]
Milos, D., Pleban, U., and Loegel, G.
Direct Implementation of Compiler Specifications or The Pascal P-Code Compiler Revisited.
In Conference Record of the 11th. ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah, January 15-18, 1984, 196-207.

[Mosses 1976]
Mosses, P.D.
Compiler Generation Using Denotational Semantics.
Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, Springer-Verlag, New York, 1976, 536-441.

[Mughal 1988]
Generation of Runtime Facilities for Program Editors.
Dr.scient. thesis, Dept. of Informatics, University of Bergen, Norway, May 1988.

[Nori et al. 1975]
Nori, K.V., Amman, U., Jensen, K., Nageli, H.H., and Jacobi, Ch.
The Pascal <P> Compiler: Implementation Notes.
Revised Edition.Instituts fur Informatik, Eidgenossische Technishe Hochschule, Zurich, 1975.

[Pal 1986]
Pal, A.A.
Generating Execution Facilities for Integrated Programming Environments.
Ph.D. dissertation, Dept. of Computer Science, University of Wisconsin - Madison, Wisc., 1986.

[Paulsen 1981]
Paulsen, L.
A compiler generator for semantic grammars.
Ph.D. dissertation, Dept. of Computer Science, Stanford University, Calif., 1981.

[Reiss 1984a]
Reiss, S. P.
Graphical Program Development with the PECAN Program Development Systems.
In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Penn., April 23-25, 1984, 30 - 41. (Joint issue: SIGPLAN Notices 19, 5, May 1984, and Software Engineering 9, 3, May 1984.)

[Reiss 1984b]
Reiss, S. P.
An Approach to Incremental Compilation.
In Proceedings of the SIGPLAN '84 SYMPOSIUM ON COMPILER CONSTRUCTION, Montreal, Canada, June 17-22, 1984, 144-156. (SIGPLAN Notices 19, 6, June 1984).

[Reppy & Kintala 1984]
Reppy, J.H.and Kintala, C.M.R.
Generating Execution Facilities for Integrated Programming Environments.
Technical Memorandum, AT&T Bell Laboratories, Murray Hill, 1984.

[Reps 1983]
Reps, T.
Static-semantic analysis in language-based editors.
In Digest of Papers of the IEEE Spring CompCon 83, San Francisco, Calif., Mar. 1983., 411 -414.

[Reps 1984]
Reps, T.
Generating Language-Based Environments.
M.I.T. Press, Cambridge, Mass., 1984.

[Reps & Teitelbaum 1987]
Reps, T. and Teitelbaum, T.
The Synthesizer Generator Reference Manual.
2nd Ed., Dept. of Computer Science, Cornell University, Ithaca, N.Y., July 1987.

[Reps & Teitelbaum 1988]
Reps, T. and Teitelbaum, T.
The Synthesizer Generator.
Springer-Verlag. To be published in fall 1988.

[Räihä 1984]
Räihä, K.J.
Attribute Grammar Design Using the Compiler Writing System HLP.
Methods and Tools for Compiler Construction, B. Lorho (ed.), Cambridge University Press, 1984.

[Schwartz et al. 1984]
Schwartz, M. D., Delisle, N. M., and Begwani, V. S.
Incremental Compilation in Magpie.
In Proceedings of the SIGPLAN '84 SYMPOSIUM ON COMPILER CONSTRUCTION, Montreal, Canada, June 17-22, 1984, 122-131. (SIGPLAN Notices 19, 6, June 1984).

[Sethi 1983]
Sethi, R.
Control Flow Aspects of Semantic Directed Compiling.
ACM TOPLAS 5, 4 (October 1983), 554-595.

[Stoy 1977]
Stoy, J.E.
Denotational Semantics.
MIT Press, Cambridge, Mass., 1977.

[Teitelbaum & Reps 1981]
Teitelbaum, T. and Reps, T.
The Cornell Program Synthesizer: a syntax-directed programming environment.
Communications of ACM 24, 9 (September, 1981) 563-573.

[Teitelbaum et al. 1987]
Teitelbaum, T., Mughal, K., and Ball, T.
A Pascal editor with full static-semantic checking.
Included with the Cornell Synthesizer Generator, Release 2.0.
Dept. of Computer Science, Cornell University, Ithaca, N.Y., July 1987.

[Teitelbaum et al. 1988]
Teitelbaum, T., Mughal, K., and Ball, T.,Belmonte, M. and Schoaff, P.
A Pascal editor with execution and debugging facilities.
To be included with the Cornell Synthesizer Generator release.
Dept. of Computer Science, Cornell University, Ithaca, N.Y., fall 1988.

[Watt & Madsen 1983]
Watt, D.A., and Madsen, O.L.
Extended Attribute Grammars
The Computer Journal, Vol. 26, No. 2, 1983, 142 - 153.