An Automatically Generated, Realistic Compiler for an Imperative Programming Language

Uwe F. Pleban
Tektronix Laboratories
P.O. Box 500, M/S 50-662
Beaverton, OR 97077
uwe%tekchips.tek.com@relay.cs.net

Peter Lee
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
Peter.Lee@cs.cmu.edu

Abstract

We describe the automatic generation of a complete, realistic compiler from formal specifications of the syntax and semantics of Sol/C, a nontrivial imperative language "sort of like C." The compiler exhibits a three pass structure, is efficient, and produces object programs whose performance characteristics compare favorably with those produced by commercially available compilers. To our knowledge, this is the first time that this has been accomplished.

1 Introduction

We report on our experimentation with the semantics directed compiler generator MESS. In earlier papers [LeP86] [LeP87] [PlL87] we have already described the semantic foundations of our *high-level semantics* approach to language specification, and the principles embodied by our system. MESS automatically produces realistic compiler implementations from the high-level semantics of a language. The generated compilers are realistic in the following sense:

1. They compile nontrivial imperative programming languages into object code for standard machine architectures.
2. Their internal structure resembles that of conventional non-optimizing compilers. Specifically, MESS-generated compilers consist of three passes which communicate via intermediate languages. They perform the usual compile time computations, such as type checking, at compile time.
3. Both the compilers and the object code they produce exhibit good performance characteristics. In particular, the size and speed of the object programs compare favorably with the size and speed of code produced by commercially available non-optimizing compilers.

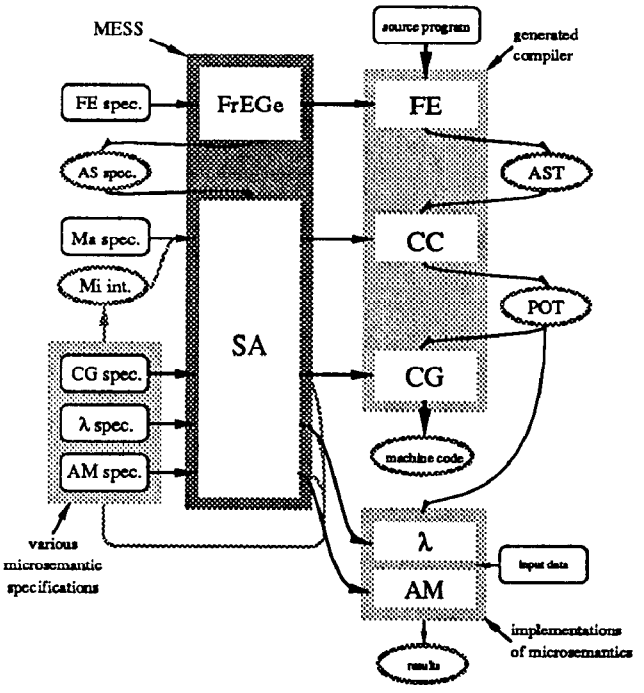
The structure of this paper is as follows. The next section briefly reviews the architecture of the MESS system. Section 3 informally describes the language Sol/C, and sketches small portions of the various specifications from which the Sol/C compiler is obtained. Section 4 discusses the generation of the compiler. In Section 5, the transformation of a small source code fragment into assembly code is traced through the compiler passes. Section 6 evaluates the performance of the Sol/C compiler and the object code it produces. Section 7 discusses our extensive experience with the MESS system during the past two years. Finally, we summarize related work in Section 8, and sketch future research endeavors.

2 The Semantics Directed Compiler Generator MESS

Figure 1 shows the overall structure of the MESS system, which runs on an IBM PC. It consists of two major components, the front-end generator FrEGe [Ple87], and the semantics analyzer SA. FrEGe processes a context-free grammar together with tree-building rules and declarations of syntactic domains (FE spec.). It produces LALR(1) parse tables, syntax error recovery tables, and tree building tables for a skeletal compiler front-end (FE), along with a description of the abstract syntax interface (AS spec.) for the semantics analyzer. This interface is used by the semantics analyzer to ensure that the abstract syntax expressions appearing in semantic equations match the tree shapes produced by the compiler front-end. The generated front-ends and FrEGe itself are written in TURBO Pascal [Bor85].

The semantics analyzer SA is written in TI PC SCHEME [TI87]. It processes both the semantic and microsemantic specifications, written in an extension of the applicative subset of Standard ML [Mil85]. A microsemantic specification (Mi spec.) defines the names and types of operators (signature) in a semantic algebra of actions, together with an interpretation for the algebra. The signature is exported to a (macro)semantic specification via a microsemantics interface file (Mi int.). The implementation of the operators is transformed into an equivalent SCHEME program. Note that any two microsemantic specifications which yield identical signatures are "plug compatible." In this paper, we describe a formal specification for

Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation
Atlanta, Georgia, June 22–24, 1988



This picture of MESS shows the various phases of compiler generation. The semantics directed compiler writer provides specifications for the front-end, the macrosemantics, and the microsemantics (FE spec., Ma spec., and one of CG spec., λ spec., or AM spec.). The specification of the abstract syntax (AS spec.) is automatically provided by the front-end generator FrEGe.

The semantics analyzer (SA) processes the semantic descriptions and generates the compiler core (CC). In addition, it produces an implementation of the microsemantic operators in a form corresponding to the type of microsemantics specified: either a code generator (CG), a set of functions written in the λ-calculus (λ), or an abstract machine (AM). A microsemantics interface file (Mi int.) is produced as a by-product of microsemantics analysis, and this is used by the SA to process the macrosemantics.

If a code generator is produced, the combination of FE+CC+CG constitutes a realistic compiler. Otherwise, FE+CC is a compiler that produces prefix-form operator terms (POTs). These can be interpreted by either a λ-calculus machine enriched with the λ-functions, or the abstract machine (AM).

Figure 1: Das MESS.

a code generator (CG spec.), and the resulting code generator implementation (CG).

The (macro)semantic definition (Ma spec.) of a language uses both the abstract syntax and microsemantic interface files. It describes the static semantic aspects of the language and a translation to terms of action operator applications. These prefix form operator terms (POTs) may quite comfortably be viewed as sentences in a high-level intermediate language directly based on the formal semantics of the language. The semantics analyzer generates the compiler core (CC), written in SCHEME, from the semantics.

The resulting compilers consist of three passes. The front end parses the source file and writes a SCHEME S-expression representing an abstract syntax tree (AST) to a file. The compiler core translates the AST to a POT, which is traversed by the code generator, yielding assembly code. With the exception of some constant folding and branch optimization performed by the code generator, optimizations are ignored.

3 A Compiler Specification for Sol/C

Informal Overview of Sol/C

Sol/C is a strongly typed, imperative language "sort of like C." It features two-level binding, recursive procedures with value and reference parameters, zero-based multidimensional arrays, integer, boolean, and character data types, arithmetic, relational, and boolean expressions, simple input and output, and the usual complement of control structures. Open (*i.e.*, conformant) array parameters are always passed by reference.

Figure 4 shows a fragment of a sample Sol/C program for multiplying matrices.

In the following we sketch small portions of the various specifications from which the compiler is obtained. Due to the limited space, only the handling of procedures is described.

Grammar and Tree-Building Rules

Figure 5 shows an excerpt from the specification of the Sol/C grammar and tree-building rules. This specification is processed by FrEGe into a compiler front-end (parser and tree-builder) for Sol/C. The specification is approximately 600 lines long, 200 of which are either commentary or blank.

Semantics

The high-level semantic specification for Sol/C describes the static semantic constraints for Sol/C and the translation from syntax trees into terms belonging to a semantic algebra of *actions*. Actions may be declarative (belonging to domain `DACTION`), imperative (`IACTION`), or value-producing (`VACTION`). The exact interpretation of the actions is given in a suitable microsemantic specification, such as the code generator specification discussed in the next subsection. All static semantic errors are flagged by the compiler generated from the high-level semantics.

Figures 7 and 8 give a fragment of the semantic specification. The entire specification is about 1,300 well commented lines long.

It is important to note that, given a suitable interpretation for the microsemantic operators, this specification *formally* describes the semantics of Sol/C. The semantic foundations of high-level semantics are discussed in [PlL87] and [Lee87].

Code Generator Specification

The microsemantic specification fragment given in Figure 6 is one of several plug compatible interpretations we have written for the semantic algebra of actions. It is by far the most involved, as it describes the generation of machine code for the iAPX8086 processor. Other interpretations, e.g. a continuation-style denotational microsemantics, have also been specified and (automatically) implemented. These are discussed in [Lee87].

In its entirety, the specification for the Sol/C code generator consists of almost 300 functions, and is roughly 3,500 well commented lines long (about 2,100 lines without comments). Almost 500 of the 3,500 lines deal with the vagaries of code output.

4 Generation of a Compiler for Sol/C

The Front End

The front end generator FrEGe converts the Sol/C grammar into Pascal tables which are incorporated into a skeletal parser and tree builder. The table sizes are as follows: 3.4K for parse tables, 0.75K for tree building tables, 0.75K for error recovery tables, and 2.6K for tables with symbolic information, such as grammar symbol names used in syntax error messages, and the names of node constructors used by the tree dumper. Front-end generation takes approximately 49 seconds. All timings given in this summary were taken on an IBM PC running with an 8MHz iAPX80286 processor, 640K of main memory, and a 20MB hard disk with 65 ms access time.

Although the front end is written in TURBO Pascal, it is loaded as part of the Sol/C compiler when invoking the SCHEME system. The remainder of the compiler calls the front end like a library routine. This is made possible through the external language calling interface provided by Version 3 of TI PC SCHEME.

The Compiler Core

The MESS semantics analyzer takes 5 minutes and 45 seconds to produce a compiler core, written in SCHEME, from the Sol/C macrosemantics. This includes roughly 53 seconds for parsing the specification and writing its syntax tree to a file.

The Code Generator

Finally, the code generator specification is converted into a code generator for the 8086 in roughly 16 minutes. This time includes about 2 minutes and 45 seconds for parsing the specification and writing its abstract syntax tree to a file. We estimate that about one fifth of the time is spent in garbage collections, due to the limited amount of memory for the SCHEME heap (fewer than 290K bytes after loading MESS).

5 Compilation of Sol/C Programs

Invoking the Compiler

The DOS command `solc` invokes the SCHEME system and causes it to load the three passes of the generated Sol/C compiler into memory. Note that TI PC SCHEME produces byte codes rather than native 8086 code. The various components of the Sol/C compiler occupy the following amounts of memory (in bytes):

Front end	89.5K
Compiler core	26.0K
Code generator	58.0K
MESS runtime library	12.0K
MESS skeletal compiler	4.5K
TOTAL	190.0K

This leaves about 160K for the SCHEME heap. It is then possible to compile Sol/C programs into assembly code by typing `(compyle` *file-name*) at the SCHEME system.

Parsing and Tree Building

Figure 9 shows the abstract syntax tree produced by the front-end for the matrix initialization procedure discussed earlier. The tree is written to a file as a SCHEME S-expression, for subsequent analysis by the compiler core.

Producing Intermediate Code

The compiler core reads the AST from the file, performs type checking, and converts it to a POT. Note that the front-end and the compiler core must communicate via a disk file because the front-end (written in Pascal) does not have access to the SCHEME heap. The intermediate code produced by the Sol/C compiler core for the initMatrix procedure is given in Figure 10.

Code Generation for the iAPX8086 Processor

Finally, the code generator produces 8086 assembly code from the POT. The code for the matrix multiplication program is given in Figure 11.

6 Performance Evaluation of the Compiler

Program Compilation

For the matrix multiplication program, the generated Sol/C compiler exhibits the following compile times:

parsing and tree building	0.95 secs
translation to POT	3.80 secs
code generation	11.95 secs
TOTAL	16.70 secs

The above times do not include the I/O overhead due to writing and reading the syntax tree. If the generated compiler core and code generator were compiled to native machine code (recall that in the TI PC SCHEME system, programs are compiled to interpreted byte codes), we believe they would run at least three times faster. We estimate that such a native code compiler would compile the matrix multiplication program in about 5 seconds.

Object Code Execution

We now compare the performance of four Sol/C object programs with that of corresponding TURBO Pascal (Version 3.0) programs. All of the Pascal programs were compiled with options for suppressing index and stack checking. The programs are:

fib:	compute the 22nd Fibonacci number 50 times
sort:	bubble sort 1000 numbers
sieve:	30 iterations of the sieve of Eratosthenes
matMult:	100 iterations of multiplying two 20x20 matrices

Figure 2 shows the object code sizes for the programs.

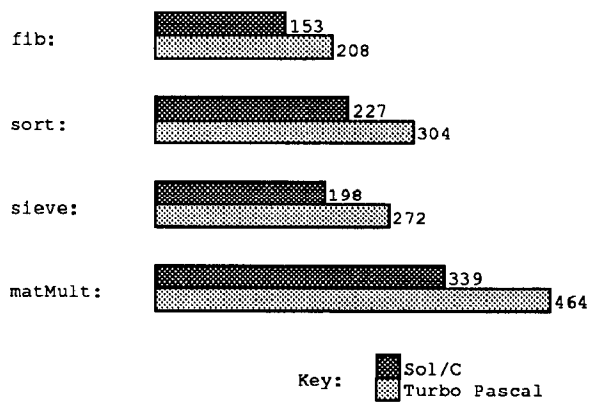


Figure 2: Object code sizes (in bytes).

The execution times for the three programs also see the Sol/C generated code outperform the TURBO Pascal code, as shown in Figure 3.

We can see that the Sol/C compiler performs quite well in comparison with a hand written compiler. Similar comparisons against other hand written compilers are given in [Lee87].

7 Correctness Concerns

Is the Sol/C compiler provably correct with respect to the standard denotational semantics of Sol/C? Unfortunately, only the compiler core is. The correctness of the code generator would have to be established by a tedious congruence proof involving the microsemantics which defines the operators as higher order

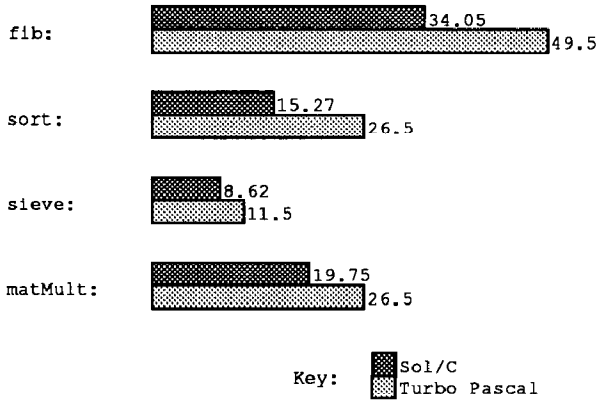


Figure 3: Execution times (in seconds).

functions in the denotational style. However, we now sketch ways of making this task less formidable.

In general, the code generator specification needs to be broken up into several passes, each of which implements a small set of transformations on the POT. For example, in a first pass, terms involving high level operators such as the `While` operator can be transformed into lower level terms involving labels and jumps. Such transformations are easy to validate with respect to a denotational interpretation of the `While` operator, as has been done in Stoy's book, for example. A second pass could process the declaration operators and perform storage allocation. It should not be very difficult to validate this pass with respect to a denotational model which reflects certain characteristics of an "abstract target machine." Finally, instruction selection and register allocation should occur in the last pass. Proving their correctness would be the most involved task. On the whole, however, the indicated separation of the code generator specification into several modules would vastly simplify a proof that the resulting code generator, and therefore the entire compiler, is correct.

8 A Critique of MESS

We have been experimenting with MESS for almost two years now. Although the system was initially constructed as a testbed of some of the principles of high-level semantics, its usefulness for experimentation with compiler specifications has exceeded all of our expectations. However, there are a number of shortcomings which a more complete system must remedy in order to be a truly useful semantics directed compiler generator.

1. The system needs to support ML style modules. At the present time, both the macrosemantic and microsemantic specifications must be written in one monolithic piece. More importantly, MESS does not ensure that microsemantic names which are not exported to the macrosemantics are placed in a separate name space. Several times, this has led to interference when loading the code generator and compiler core together.

2. The metalanguage should include ML references and assignments. The inability to define and maintain a global compile time state forces one to pass state information as arguments to most functions and return all updated information as results, even though this information is rarely accessed. The restriction to a purely applicative style of programming has resulted in unnecessary clutter in the code generator specification.

3. The messages issued by the MESS type checker are not very illuminating when subtle type errors are detected. Also, when a type mismatch is detected in a deeply nested expression, a cascade of messages inundates the user.

4. The clausal style of function definitions is compiled into very poor SCHEME code. For example, a first version of the code generator specification was written using the clausal form, and resulted in a SCHEME program which was almost twice as large as the present version.

5. The initial MESS environment should include more primitives for compiler writing, such as functions for generating labels, code output, etc. This would make specifications more abstract and compact.

6. The present implementation of MESS handles in a clean manner only those microsemantic models which provide a functional interpretation for the algebraic operators. For the Sol/C compiler, this has resulted in "kludging" the interface between the code generator and the compiler core.

7. A truly useful system must allow the compiler writer to specify optimization passes of a compiler. MESS has to date purposely ignored this issue.

8. Finally, the IBM PC environment is rather cramped. The time for iterating through the debugging loop for the Sol/C code generator was often larger than 30 minutes. Fortunately, a prototype version of the specification was developed using TURBO Prolog, with a debug cycle time of fewer than 5 minutes. We are presently considering a port of MESS to the Mac II, and expect at least a fourfold performance improvement.

9 Related Work and Future Research

The work of Mosses and Watt on *action semantics* [MoW86a] bears strong similarities with high-level semantics. In particular, their choice of "special" operators for the Pascal action semantics [MoW86b] is surprisingly similar to to the choice of operators for our Sol/C semantics. There are four principal differences between high-level semantics and action semantics: (1) operators in action semantics are defined by means of standard algebraic operators; (2) compile time and runtime aspects are not distinguished in action semantics; (3) transformations on the static environment in action semantics are also formulated via semantic operators rather than by means of λ-abstractions; and (4) data flow within an action semantics is expressed by explicit naming whereas in high-level semantics the nesting of prefix terms determines the data flow.

There are three semantics-based compiler generators which are similar in spirit to our approach. The CERES system of Jones and Christiansen [JoC82] accepts semantic specifications expressed in a small number of action-oriented operators inspired by those of Mosses. Sethi's system [Set82] generates efficient compilers by treating fundamental "runtime" operators in the semantic specification as uninterpreted symbols. His work is also motivated by that of Mosses, but still refers to microsemantic concepts such as continuations and stores. Both systems have only been used for generating compilers for languages with control structures for sequencing, looping and decision making, and simple expressions. Also, the intermediate code produced by the generated compilers must be translated by a code generator in an *ad hoc* manner. Finally, the work by Appel [App85] extends the work of Sethi to include procedures and more complex data flow. However, the code generators produced by his system are very slow and consume large amounts of storage space.

We are currently planning the design of a *Language Designer's Workbench* [Lee87] based on our experience with MESS. Our hope is that such a workbench will facilitate the development and use of modules of (macro and micro) semantic specifications. Presently, we are experimenting with such modularity by utilizing the `functor` construct of Standard ML. We have been able to make some early tests of our ideas by implementing the functor-based modules in the Standard ML compiler produced by Appel and MacQueen [ApM87]. Our results thus far have been encouraging, but still quite preliminary.

We have found the MESS system to be a useful tool for semantics directed compiler generation. In particular, the strong polymorphic type checking and pattern matching features of the ML-based MESS metalanguage make the process of writing semantic descriptions much easier than with previous systems.

References

[ApM87] A Standard ML compiler. Technical Report CS-TR-097-87., Dept. of Computer Science, Princeton University, Princeton, 1987.

[App85] Compile-time evaluation and code generation for semantics-directed compilers. Ph. D. Dissertation, Carnegie-Mellon University, July 1985.

[Bor85] *Turbo Pascal Reference Manual (Version 3.0)* Borland International, Inc., 1985.

[JoC82] Jones, N. D., and Christiansen, H. Control flow treatment in a simple semantics-directed compiler generator. Formal Description of Programming Concepts II, IFIP IC-2 Working Conference, North Holland, Amsterdam, 1982.

[Lee87] Lee, P. The automatic generation of realistic compilers from high-level semantic descriptions. Ph. D. Dissertation, The University of Michigan, April 1987.

[LeP86] Lee, P., and Pleban, U. F. On the use of LISP in implementing denotational semantics. Proc. 1986

ACM Conf. LISP and Functional Programming, Cambridge, MA, August 1986, 233-248.

[LeP87] Lee, P., and Pleban, U. F. A realistic compiler generator based on high-level semantics. Proc. 14th Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, Munich, W.-Germany, January 1987, 284-295.

[Mil85] Milner, R. The standard ML core language. Polymorphism II, 2 (Oct. 1985).

[MoW86a] Mosses, P., and Watt, D. A. The use of action semantics. Technical Report DAIMI PB-217, Aarhus University, Aarhus, Denmark, August 1986.

[MoW86b] Mosses, P., and Watt, D. A. Pascal: Action semantics. Draft — Version 0.3). Computer Science Dept., Aarhus University, Aarhus, Denmark, Sept. 1986.

[Ple87] Pleban, U. F. *Reference Manual for the Front End Generator FrEGe*. PhiloSoft, November 1987.

[PlL87] Pleban, U. F., and Lee, P. High-level semantics: an integrated approach to programming language semantics and the specification of implementations. To appear in Proc. of 3rd Workshop on the Math. Found. of Progr. Lang. Semantics, New Orleans, LA, April 8-10, 1987, Springer-Verlag, 1988.

[Set81] Sethi, R. Control flow aspects of semantics directed compiling. Tech. Rep. 98, Bell Labs., 1981; also in Proc. SIGPLAN '82 Symp. Compiler Construction, SIGPLAN Notices 17, 6 (June 1982), 245-260.

[TI87] *TI PC SCHEME Reference Manual*, Texas Instruments, Inc., August 1987.

```
program matrixMultiply ( "stdin", "stdout" ) is

const N = 20, iter = 100;
int A [N, N], B [N, N], C [N, N];   int k;

proc initMatrix (ref int mat [n, m]) is
int i, j;
begin
   i := 0;
   loop while i < n
      j := 0;
      loop while j < m
         mat [i, j] := 2 * i - j; j := j + 1
      endloop;
      i := i + 1
   endloop
end; /* initMatrix */

proc matMult (ref int Z [xul, zu2],
              ref int X [xul, xu2],
              ref int Y [yul, yu2])
is
int i, j, k, sum;
begin
   i := 0;
   loop while i < xul
      j := 0;
      loop while j < xu2
         k := 0;  sum := 0;
         loop while k < yu1
            sum := sum + X [i, k] * Y [k, j];  k := k + 1
         endloop;
         Z [i, j] := sum;  j := j + 1
      endloop;
      i := i + 1
   endloop
end; /* matMult */

begin
   call initMatrix (A);  call initMatrix (B);  k := 0;
   loop while k < iter
      call matMult (C, A, B);  k := k + 1
   endloop
end /* matrixMultiply */
```

Figure 4: A Sol/C program for multiplying matrices.

```
-- after the declaration of terminals follow the productions
&productions
...
procedure_declaration ->
      "proc" 'identifier' "(" list_of_parameters ")" "is"
            procedure_body /\ [[ ... ]]
                        -- [[ ... ]] means: use 'standard' rule
   -- node labels are derived from the rule
list_of_parameters ->
      list_of_parameters "," parameter /\ [[ @list ... ]]
    | parameter                        /\ [[ @unitlist ... ]]
    .                        -- @list/@unitlist build n-ary nodes
parameter ->
      val_parameter /\ @propagate
    | ref_parameter /\ @propagate
    .                -- propagate tree built so far
val_parameter ->
      "val" simple_type 'identifier'  /\ [[ ... ]]
    | simple_type 'identifier'        /\ [[ "val" simple_type 'identifier' ]]
    .
ref_parameter ->
      "ref" simple_type 'identifier'  /\ [[ ... ]]
    | "ref" array_name                /\ [[ "ref" @splice array_name ]]
    .                -- @splice splices the descendants into the new node
array_name ->
      simple_type 'identifier' "[" list_of_upper_bound_names "]" /\ [[ ... ]]
    .

list_of_upper_bound_names ->
      list_of_upper_bound_names "," 'identifier'  /\ [[ @list ... ]]
    | 'identifier'                                /\ [[ @unitlist ... ]]
    .

procedure_body ->
      list_of_local_declarations "begin" list_of_statements "end" /\ [[ ... ]]
    .
...
&domains
Body     = { program_body, procedure_body } .
Decls    = { list_of_global_declarations, list_of_local_declarations } .
Decl     = { global_declaration, variable_declaration,
             simple_variable_declaration, array_variable_declaration,
             constant_declaration, procedure_declaration, local_declaration } .
Params   = { list_of_parameters } .
Param    = { parameter, val_parameter, ref_parameter } .
UboundIds = { list_of_upper_bound_names } .
...
&end
```

In this excerpt from the FrEGe grammar for Sol/C, terminal symbols are enclosed in single or double quotation marks. The text following the symbol /\ specifies tree building rules.

Figure 5: Fragment of the Sol/C grammar.

```
microsemantics solc_cg

semantic domains
   LEVEL    = union localL | globalL.
   INTLIST  = INT list.
   REGISTER = union          (* the iAPX8086 registers used by the code generator *)
      ax | al | bx | cx | cl | dx | si | di | sp | bp.
   VALUEINREGISTER = union   (* value posted in register ax *)
      cont of NAME |            (* 1. variable *)
      unknown.                  (* 2. too complex to keep track of *)
   (* The code generator name environment.  Note that all names are distinct due to
    *  the renaming performed by the macrosemantics. Local and global
    variables are
    * accessed differently.
    *)
   ASSOC       = NAME -> LEVEL.
   REGCONTENTS = REGISTER -> VALUEINREGISTER.
   PARAMOFFSET = INT.              (* next available stack offset for parameters *)
   LOCALOFFSET = INT.              (* next available stack offset for locals *)
   ENV         = ASSOC * LEVEL * REGCONTENTS * PARAMOFFSET * LOCALOFFSET.
   LABELENV    = LABELNUMBER * RETURNLABEL * EXITLABEL *
                 PROGRAMNAME * ROUTINENAME * STRINGTABLE.
   ...
action domains  (* and operators *)

   DACTION = union
      BindProc of (NAME * IACTION) |
      BindValParam of (NAME * STYPE) |
      BindRefParam of (NAME * STYPE) |
      BindRefArrayParam of (NAME * DACTION * STYPE) | (* bounds decls embedded *)
      DeclSimpleVar of (NAME * STYPE) |
      DeclArrayVar of (NAME * INTLIST * STYPE) |      (* integers are bounds *)
      DeclSeq of (DACTION * DACTION) |
      NullDecl.
   ...

auxiliary functions
   ...
   eMoveRegReg (r1: REGISTER, r2: REGISTER) = putOpMov2 (regOp (r1), regOp (r2)).

   eMoveWRegInt (r: REGISTER, i: INT) =
      if i = 0 then putOp2 ("xor", regOp (r), regOp (r))
      else putOpMov2 (regOp (r), intOp (i)).

   ePushReg (r: REGISTER) = putOp1 ("push", regOp (r)).
   ...
   (* Procedure entry *)
   procEntry (localSize: INT) =
      (*
       * After caller has pushed arguments and return address:
       * (1) Save old frame pointer bp.   (2) Set new value of bp.
       * (3) Allocate local storage block.
       *)
      ( ePushReg (bp). eMoveRegReg (bp, sp). eSubRegInt (sp, localSize) ).

   (* Procedure exit *)
   procExit (localSize: INT) =
      (*
       * (1) Deallocate local storage block.   (2) Restore old frame pointer.
       * (3) Return to caller.
       * NOTE: Caller will deallocate storage for argument values.
       *)
      ( eAddRegInt (sp, localSize). ePopReg (bp). eRet () ).
   ...
```

Figure 6: Fragment of the Sol/C code generator specification.

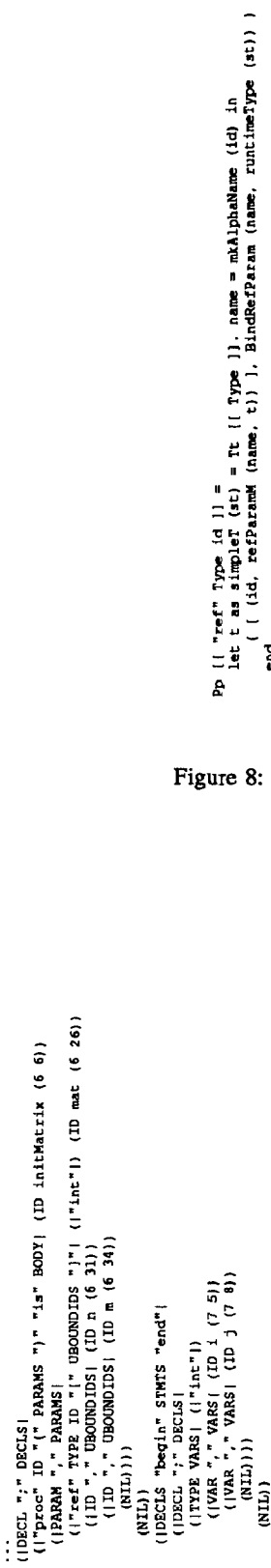

```
(***************** Code generation for declarations *****************)

genDecl (decl: DACTION, env: ENV, lEnv: LABELENV): (INT * ENV * LABELENV) =
   case decl of

   NullDecl => (0, env, lEnv) |

   DeclSeq (decl1, decl2) =>
      let (size1, env, lEnv) = genDecl (decl1, env, lEnv).
          (size2, env, lEnv) = genDecl (decl2, env, lEnv) in
        (size1 + size2, env, lEnv)
      end |

   BindProc (name, stmt) => genBindProc (name, stmt, env, lEnv) |

   BindValParam (name, typ) => genBindValParam (name, typ, env, lEnv) |

   BindRefParam (name, typ) => genBindRefParam (name, typ, env, lEnv) |

   BindRefArrayParam (name, boundlist, typ) =>
      genBindRefArrayParam (name, boundlist, typ, env, lEnv) |

   DeclSimpleVar (name, typ) => genDeclSimpleVar (name, typ, env, lEnv) |

   DeclArrayVar (name, intlist, typ) =>
      genDeclArrayVar (name, intlist, typ, env, lEnv)
and

genBindProc (procName: NAME, stmt:IACTION, env: ENV, lEnv: LABELENV)
 : (INT * ENV * LABELENV) =
   ((* Compile parameter declarations and procedure body;
     * save name of procedure in order to generate a label later. *)
    ...
    let lEnv = saveProcName (procName, lEnv).
        env = forgetRegisterContents (env).
        (env, lEnv) = genStmt (stmt, localLevel (env), lEnv)
    in (
        ...
        (0, globalLevel (env), deleteProcName (lEnv)) )
   end
  )
and

genBindRefArrayParam (name: NAME, boundsDecls: DACTION, typ: STYPE,
                      env: ENV, lEnv: LABELENV)
 : (INT * ENV * LABELENV) =
   (* Allocate a new slot on the stack for the array pointer. *)
   let loc = currentParameter (env).  env = addName (name, env).
       size = pointerSizeOf (typ).    env = nextParameter (size, env)
   in (
       equate (name, loc).
       (* Process bounds names. *)
       let (boundsSize, env, lEnv) = genDecl (boundsDecls, env, lEnv) in
         (size + boundsSize, env, lEnv)
       end
      )
   end
and
...
```

```
semantics SolC
interface   "solc-fe"    (* 'solc-fe.asl': abstract syntax interface *)
microsemantics "solc-cg"  (* 'solc-cg.mic': interface to code generator *)
semantic domains
  (* The compile-time environment keeps track of the statically-determined information
   * (the MODE) for each declared identifier.  Also, it remembers the current block
   * level, and maintains flags indicating whether a loop or a procedure is being
   * processed, in order to check for the legality of exit and return statements.
   *)
  ENV       = ASSOC * BLOCK_INFO.
  ASSOC     = ID -> (MODE * LEVEL).
  BLOCK_INFO  = LEVEL * INLOOP * INPROC.
  LEVEL     = union  global_L | local_L of NAME.
  INLOOP    = BOOL.     INPROC    = BOOL.
  MODE = union                              (* The various modes:          *)
    noneM |                                 (*   undeclared                *)
    labelM of NAME |                        (*   label                     *)
    constM of CONST |                       (*   constant                  *)
    varM of VAR |                           (*   variable                  *)
    procM of (NAME * NUM_PARAMS * MODELIST) |  (*   procedure              *)
    refParamM of PARAM |                    (*   reference parameter       *)
    valParamM of PARAM |                    (*   value parameter           *)
    constParamM of PARAM.                   (*   constant parameter        *)
                                            (*   (=open array bound)       *)
  VAR   = NAME * TYPE.
  PARAM = VAR.
  TYPE = union                          (* The various compile time types: *)
    noneT |                             (*   no type                       *)
    simpleT of SIMPLE_TYPE |            (*   unstructured                  *)
    arrayT of (INT * TYPE) |            (*   multidimensional array        *)
    openArrayT of (MODE * TYPE).        (*   multidimensional open array  *)
  SIMPLE_TYPE = union int_type | bool_type | char_type.
  ...
auxiliary functions
  ...
  (* ****** Environment Handling ****** *)

  val emptyAssoc: ASSOC = fn id. (noneM, global_L).

  val initialBlockInfo: BLOCK_INFO = (
                          (* program block at global level *)
        global_L,         (* not inside a loop *)
        false,            (* not inside a procedure *)
        false).

  val emptyEnv: ENV = (emptyAssoc, initialBlockInfo).

  (* Retrieve the mode of the given identifier from the given environment. *)
  lookup (id: ID, (assoc, (currentLevel, _, _)): ENV): MODE =
    let (mode, declLevel) = assoc id in
      if (declLevel = currentLevel) orelse (declLevel = global_L) then mode
      else noneM
    end.

  (* Check that an identifier is not yet declared in the current block. *)
  notDeclared (id: ID, (assoc, (currentLevel, _, _)): ENV): BOOL =
    let (mode, declLevel) = assoc id in
      (mode = noneM) orelse not (currentLevel = declLevel)
    end.

  (* Add a new association pair to the environment. *)
  addAssoc (id: ID, mode: MODE, (assoc, info as (level, _, _)): ENV): ENV =
    ((id => (mode, level)) assoc, info).
  ...
```

```
semantic functions
  Mm:  AST -> OUTPUTFILE.                                   (* program meanings *)
  Dd:  AST -> ENV -> (ENV * DACTION).                       (* declarations *)
  Pp:  AST -> (ID_MODELIST * DACTION).                      (* parameters *)
  PpA: AST -> TYPE -> (TYPE * ID_MODELIST * DACTION).       (* array parameter bounds *)
  BbR: AST -> ENV -> IACTION.                               (* routine blocks *)
  Ss:  AST -> ENV -> IACTION.                               (* statements *)
  Ll:  AST -> ENV -> (TYPE * VACTION).                      (* l-value expressions *)
  Ee:  AST -> ENV -> (TYPE * VACTION).                      (* expressions *)
  ...
semantic equations

  (* ------------------------  Declarations  ------------------------ *)

  Dd [[ decl ";" decls ]] env =
    let (env1, dA1) = Dd [[ decl ]] env.   (env2, dA2) = Dd [[ decls ]] env1 in
      (env2, DeclSeq (dA1, dA2))
    end.

  Dd [[ ]] env = (env, NullDecl).

  Dd [[ "proc" id "(" params ")" "is" body ]] env =
    if notDeclared (id, env) then
      let
        procName = name (id).
        (ims, bindingActions) = Pp [[ params ]].
        paramList = extractParamModes (ims).
        numParams = length (paramList).
        (*
         * Build the environment in which to process the body by extending
         * the existing environment with:
         * - the binding for the procedure identifier, and
         * - the bindings for the parameter identifiers.
         *
         * Since the binding for the procedure identifier id is visible inside
         * the body, the procedure may be called recursively.
         *)
        mode = procM (procName, numParams, paramList).
        procEnv = addAssoc (id, mode, env).
        bodyEnv = addAssocList (ims, enterProc (procName, procEnv)).
        bodyAction = BbR [[ body ]] bodyEnv
      in
        ( procEnv,          (* the resulting compile time environment *)
          BindProc (procName, Proc (bindingActions, bodyAction))
        )
      end
    else
      declError env [[ id ]] "Procedure already declared.".
  ...
  (* ------------------------  Parameters  ------------------------ *)

  Pp [[ param "," params ]] =
    let (ims, dAs)  = Pp [[ param ]].  ps as (ims', dAs') = Pp [[ params ]] in
      if checkModeLists (ims, ims') then (ims @ ims', DeclSeq (dAs, dAs'))
      else bindError ps [[ param ]] "Parameter will be redeclared."
    end.

  Pp [[ ]] = (nil, NullDecl).

  Pp [[ "val" Type id ]] =
    let t as simpleT (st) = Tt [[ Type ]]. name = mkAlphaName (id) in
    ( [ (id, valParamM (name, t)) ], BindValParam (name, runtimeType (st)) )
    end.
```

In this fragment of the semantic specification, note that the microsemantic operators are spelled with the first letter capitalized. Due to space constraints, this excerpt is presented in a more compressed format than the original specification.

Figure 7: Fragment of the Sol/C semantics (part 1).



```
Pp [[ "ref" Type id ]] =
  let t as simpleT (st) = Tt [[ Type ]]. name = mkAlphaName (id) in
    ( [ (id, refParamM (name, t)) ], BindRefParam (name, runtimeType (st)) )
    end.

Pp [[ "ref" Type id "[" uboundids "]" ]] =
  let componentType as simpleT (st) = Tt [[ Type ]]. name = mkAlphaName (id).
      (arrayType, ims, dAs) = PpA [[ uboundids ]] componentType.
      im = (id, refParamM (name, arrayType))
  in
    ( im :: ims, BindRefArrayParam (name, dAs, runtimeType (st)) )
  end.

(* ---------------  Open array parameter bounds  ------------------ *)

PpA [[ id "," uboundids ]] componentType =
  let name = mkAlphaName (id). param = constParamM (name, simpleT (int_type)).
      (subArrayType, idModes, dAs) = PpA [[ uboundids ]] componentType
  in
    ( openArrayT (param, subArrayType), (id, param) :: idModes,
      DeclSeq (BindValParam (name, runtimeType (int_type)), dAs) )
  end.

PpA [[ ]] componentType = (componentType, nil, NullDecl).
...
```

Figure 8: Fragment of the Sol/C semantics (part 2).

```
...
(|DECL ";" DECLS|
 (|"proc" ID "(" PARAMS ")" "is" BODY| (ID initMatrix (6 6))
  (|PARAM "," PARAMS|
   (|"ref" TYPE ID "[" UBOUNDIDS "]"| (|"int"|) (ID mat (6 26))
    (|ID "," UBOUNDIDS| (ID n (6 31))
     (|ID "," UBOUNDIDS| (ID m (6 34))
      (NIL))))
   (NIL))
  (|DECLS "begin" STMTS "end"|
   (|DECL ";" DECLS|
    (|TYPE VARS| (|"int"|)
     (|VAR "," VARS| (ID i (7 5))
      (|VAR "," VARS| (ID j (7 8))
       (NIL))))
     (NIL))
    (|STMT ";" STMTS|
     (|LVAR ":=" EXPR| (ID i (9 4)) (INT 0 (9 9)))
     (|STMT ";" STMTS|
      (|"loop" "while" EXPR STMTS "endloop"|
       (|EXPR1 "<" EXPR2| (ID i (10 15)) (ID n (10 19)))
       (|STMT ";" STMTS|
        (|LVAR ":=" EXPR| (ID j (11 7)) (INT 0 (11 12)))
        (|STMT ";" STMTS|
         (|"loop" "while" EXPR STMTS "endloop"|
          (|EXPR1 "<" EXPR2|
           (ID j (12 18)) (ID m (12 22)))
          (|STMT ";" STMTS|
           (|LVAR ":=" EXPR|
            (|ID "[" EXPRS "]"| (ID mat (13 10)) (|EXPR "," EXPRS| (ID i (13 15))
                                                   (|EXPR "," EXPRS| (ID j (13 18))
                                                    (NIL))))
             (|EXPR1 "-" EXPR2|
              (|EXPR1 "*" EXPR2|
               (INT 2 (13 24)) (ID i (13 28)))
              (ID j (13 32))))
             (|STMT ";" STMTS|
              (|LVAR ":=" EXPR| (ID j (13 35))
                                (|EXPR1 "+" EXPR2| (ID j (13 40)) (INT 1 (13 44))))
               (NIL))))
            (|STMT ";" STMTS|
             (|LVAR ":=" EXPR| (ID i (15 7))
                               (|EXPR1 "+" EXPR2| (ID i (15 12)) (INT 1 (15 16))))
              (NIL)))))
           (NIL))))
...
```

Abstract syntax trees are given as SCHEME S-expressions. Nodes with labels ID and INT are leaf nodes. They carry lexical information in the form of line and column numbers with them.

Figure 9: Abstract syntax tree for matrix initialization routine.

```
...
(!DECLSEQ
  (!BINDPROC (NAME INITMATRIX)
    (!PROC
      (!DECLSEQ
        (!BINDREFARRAYPARAM (NAME MAT_5)
          (!DECLSEQ (!BINDVALPARAM (NAME N_6) !INTTYPE)
            (!DECLSEQ (!BINDVALPARAM (NAME M_7) !INTTYPE)
              (!DECLSEQ (!BINDVALPARAM (NAME I_8) !INTTYPE)
                (!DECLSEQ (!BINDVALPARAM (NAME J_9) !INTTYPE)
                  !NULLDECL))
                !INTTYPE)
              !NULLDECL)
            (!BLOCK
              (!DECLSEQ
                (!DECLSEQ (!DECLSIMPLEVAR (NAME I_8) !INTTYPE)
                  (!DECLSEQ (!DECLSIMPLEVAR (NAME J_9) !INTTYPE)
                    !NULLDECL))
                !NULLDECL)
              (!STMTSEQ
                (!STMTSEQ
                  (!ASSIGN (!VAR (NAME I_8) !INTTYPE) (!INTEGER 0))
                  (!STMTSEQ
                    (!WHILE
                      (!LESS (!RVALUE (!VAR (NAME I_8) !INTTYPE))
                        (!RVALUE (!VALPARAM (NAME N_6) !INTTYPE)))
                      (!STMTSEQ
                        (!STMTSEQ
                          (!ASSIGN
                            (!INDEX
                              (!REFARRAYPARAM (NAME MAT_5)
                                (!EXPRSEQ
                                  (!RVALUE (!VALPARAM (NAME N_6) !INTTYPE))
                                  (!EXPRSEQ
                                    (!RVALUE (!VALPARAM (NAME M_7) !INTTYPE))
                                    !NULLEXPR)))
                              !INTTYPE)
                            (!EXPRSEQ
                              (!RVALUE (!VAR (NAME I_8) !INTTYPE))
                              (!EXPRSEQ
                                (!RVALUE (!VAR (NAME J_9) !INTTYPE))
                                !NULLEXPR))))
                          (!SUB (!MULT
                                  (!INTEGER 2)
                                  (!RVALUE (!VAR (NAME I_8) !INTTYPE)))
                                (!RVALUE (!VAR (NAME J_9) !INTTYPE)))))
                        (!STMTSEQ
                          (!ASSIGN
                            (!VAR (NAME J_9) !INTTYPE)
                            (!ADD (!RVALUE (!VAR (NAME J_9) !INTTYPE))
                                  (!INTEGER 1)))
                          !NULLSTMT)))
                      !NULLSTMT))))
                (!STMTSEQ
                  (!ASSIGN
                    (!VAR (NAME I_8) !INTTYPE)
                    (!ADD (!RVALUE (!VAR (NAME I_8) !INTTYPE))
                          (!INTEGER 1)))
                  !NULLSTMT))))))
...
```

All action operator names are prefixed with exclamation marks in order to avoid name clashes with SCHEME system names. Also, all program variable names have been α-converted by attaching unique numeric suffixes.

Figure 10: Intermediate code for matrix initialization routine.

```
; FILE:   mat.a
; BY:     Sol/C code generator for iAPX 8086 (V2.0, WESS)
; ON:     03/14/88 at 21:52
        dseg
SCinput_  db 'stdin',0          ; input file
SCoutput_ db 'stdout',0         ; output file
        cseg                    ; public symbols:
        public  SCmain_         ;   entry point for main program
        public  SCinit_         ;   library routine
        ...
        public  SCwriteln_      ;   library routine
        dseg
A_1   rb  800                   ; global storage for A
B_2   rb  800                   ; global storage for B
C_3   rb  800                   ; global storage for C
K_4   rb  2                     ; global storage for k
        dseg                    ; beginning of proc initMatrix
MAT_5   equ  4                  ;   1st parameter
N_6     equ  6                  ;   2nd parameter (1st upper bound)
M_7     equ  8                  ;   3rd parameter (2nd upper bound)
I_8     equ  -2                 ;   1st local
J_9     equ  -4                 ;   2nd local
        cseg
INITMATRIX_:
        push  bp                ; entry point for initMatrix
        mov   bp, sp            ; link previous frame
        sub   sp, 4             ; point to current frame
        mov   WORD [bp + I_8], 0 ; allocate local storage
        jmp   _L1002            ; i := 0
_L1003:
        mov   WORD [bp + J_9], 0 ; body of outer while
        jmp   _L1005
_L1006:                         ; body of inner while
        mov   ax, WORD [bp + I_8]
        shl   ax, 1
        sub   ax, WORD [bp + J_9]
        mov   cx, ax            ; cx = 2 * i - j
        mov   ax, WORD [bp + I_8]
        imul  WORD [bp + M_7]
        add   ax, WORD [bp + J_9]
        shl   ax, 1
        mov   bx, WORD [bp + MAT_5]
        add   bx, ax            ; bx = @mat + (i*m+j)*2
    store result indirect
        mov   WORD [bx], cx     ; *bx := cx
        inc   WORD [bp + J_9]   ; j := j + 1
_L1005:                         ; inner while test
        mov   ax, WORD [bp + J_9]
        cmp   ax, [bp + M_7]
        jl    _L1006            ; repeat if j < m
_L1007:                         ; end inner while
        inc   WORD [bp + I_8]   ; i := i + 1
_L1002:                         ; outer while test
        mov   ax, WORD [bp + I_8]
        cmp   ax, [bp + N_6]
        jl    _L1003            ; repeat if i < n
; end while
_L1004:                         ; end outer while
        add   sp, 4             ; deallocate local storage
        pop   bp                ; unlink activation record
        ret                     ; go home
SCmain_:
        ...                     ; main program follows here
        end
```

Figure 11: iAPX 8086 assembly code for matrix initialization routine.