An experimental optimising compiler for Modula-X

Darko Stefanović
Department of Computer Science
University of Massachussets

June 1994

Abstract

We describe the organisation and implementation of an optimising compiler for Modula-X, and discuss peformance results. We first describe the language and its origins. We then describe our compilation model, and discuss the construction of the front-end and the back-end. Finally we give some performance measurements.

1 The language

Modula-X is the name given to a small, Pascal-like subset of Modula-3, which has been used in the basic compiler construction course at UMass and also in the advanced language implementation classes at UMass and CMU. A testbed for optimisation techniques based on this language has also been developed and tried,[5, 2, 1] but it is not used in the compiler discussed here.

The language inherits procedural style and block structure and a solid repertoire of control flow constructs from Modula-3. (It lacks modules and exceptions, however.) It provides records and fixed-bound arrays as type constructors, and integer subranges and booleans as base data types.

In this implementation, we decided to limit procedure parameter modes to pass-by-value (VALUE) and pass-by-reference (VAR), and eliminate the READONLY mode because of its obscure semantics. It would have been nice to be able to eliminate VAR parameters as well, since their less obscure but nevertheless unpleasant semantics is said to be the "result of improper language design".[3] We felt that this mode is frequently used in practice and could not be eliminated from the language. Also, for practical reasons (see below) we disallowed nested comments.

We added reference types and dynamic storage allocation for values of such types. To make reference types useful, we added a new kind of declaration for recursive types, much in the style of ML. Declaration TYPEREC can be used to declare several types simultaneously, provided that mutual recursion is limited to reference types. We believe this to be a good compromise between the approach of Modula-3, which allows unrestricted declaration ordering, and the Pascal style which requires cumbersome forward declarations. Our reference types are strictly monomorphic, in keeping with the rest of the language. In implementing reference types, we decided to use garbage collection rather than explicit deallocation, because of safety concerns, and because we could draw on our experience in garbage collector implementation. Note, however, that some optimisations do not handle reference types at present. See Appendix for the abstract syntax of the language and some concrete program examples.

2 Compilation model

The compiler is multi-pass; the front end comprises the parser and the annotator. There are two alternate back ends. The old one is a non-optimising translator to GNU C, which is described elsewhere[4] and will not be further mentioned here.[1] The new back end is itself multi-pass and manipulates a control-flow graph based internal representation which will be detailed below. There is an inlining (procedure integration) pass and an expression propagation pass (the latter bundled with several kinds of code elimination). The compiler itself is written in Standard ML. Some repercussions of this decision are mentioned below. It is a rather large program, with about 23 000 lines of code.

2.1 Front end

For the parser we decided to use ML-Yacc and ML-Lex, and we took and augmented the existing C-Yacc source for Modula-X and an ML-Lex source for Pascal (here we had to sacrifice nested comments). The parser builds an abstract syntax tree. The annotator analyses the abstract syntax tree, building a fat annotated syntax tree. It resolves identifier references, builds type structures and static environments, and type-checks all expressions. Except for correcting some problems, we left the front end unchanged from the old compiler.

Source file prefixes:
`moduLaX.absyn`,
`modulaX.annotate`,
`modulaX.annotsyn`,
`modulaX.env`,
`modulaX.grm`,
`modulaX.ints`,
`modulaX.lex`,
`modulaX.objects`,
`modulaX.types`,
`modulaX.values`.

2.2 Intermediate form

The annotated syntax tree is translated into the intermediate form. The representation of a program has three parts: block hierarchy, symbol tables, and code.

The language allows nested procedures, and moreover, nested blocks with declarations inside statements in procedure bodies (like Modula, but unlike Pascal). In particular, FOR loops implicitly introduce such blocks. The block hierarchy data structure keeps track of the nesting of blocks. Each block has a level (a display level), and may have a name if it's a procedure. Each block is given a unique number. The block hierarchy is implemented as a forest.

A block introduces new scope. A symbol table lists the names of (local) procedures and variables that become visible inside the block, together with the "allocation" of these names in the store as <*display*, *offset*> pairs. Blocks which are procedures will also introduce parameters and possibly a return result. The module symbol table is implemented as a mapping (`intmap`) from block numbers to block symbol tables.

[1] It may or may not work at present.

Block structure determines visibility of names. It is not, however, appropriate for representing control flow. The natural boundary for control flow is not the (static) block entry / block exit pair, but the (dynamic) procedure call / return pair. Therefore, in our intermediate form, the main body of a procedure is represented as one control-flow graph. Internal blocks are assumed to be flattened out onto this graph. On the other hand, local procedures, including procedures declared inside statements, are represented separately. Within a procedure, computation is carried out on the visible part of the store. A local scratchpad is assumed to be available for temporary storage of computed results. This takes the form of an unlimited supply of "registers". Note that the registers are purely local to each procedure. The language of control-flow graph nodes is low-level, typed, and mostly monomorphic. The computation model is load-store. The values in the store and in the registers are allowed to have arbitrary type, so that any representation (memory layout) decisions can be postponed for a later code generation phase for any given hardware. The full description of the code for a procedure consists of the control-flow graph, a type map of the registers used, and a roadmap. The roadmap is meant to carry all information which is needed for code transformations. Currently, it lists the entry and exit nodes of the procedure (for various dataflow problems; entry node is also needed for code execution) and a call sites list (used in inlining). We envisage that the roadmap would be essential for loop transformations, where the roadmap information would let the transformer know the original loop structure rather than have to reconstruct it. We didn't work on loop transformations, however. For the entire compilation unit, there is a mapping (`intmap`) from block numbers to full procedure representations. The appendix has the signature of the functor implementing the intermediate form.

Source file prefix: `interm`.

2.3 Conversion to intermediate form

Given an annotated syntax tree, conversion to the intermediate form is relatively simple. It is performed by a traversal of the tree, similar to grammar attibution, wherein environment information is passed down (starting from an outermost predefined environment), augmented, and passed back up. When closure is needed (as in recursive procedure calls), we use the pseudo-updatable nature of our `digraph` representations. The translation of all language constructs is straightforward, with the exception of the FOR statement, which is very complicated. The FOR statement introduces a new scope, or block in our intermediate representation. This new scope contains the control variable of the loop. The FOR statement computes the loop start, stop, and step values, once only. It is natural to place the computed values in three additional variables in the local block. Lastly, the FOR statement implicitly requires code to test the control variable against the loop stop value, and the code to increment the control variable. All of the above code must be hand-crafted and glued into the control-flow graph. As a result, the FOR loop results in a lot of code expansion. We've found that constant propagation removes a significant part of this overhead. However, procedures with deeply nested FOR loops have caused severe performance problems in the propagator itself. A redesign of the FOR loop translation may be in order to avoid bloated code in the first place.

Source file prefix: `to_interm`.

2.4 Inlining

The inliner computes an intermediate representation, where certain procedure call sites have had the called procedure expanded in-line. Inlining is greatly complicated by the presence of nested procedures, and nested blocks. Here is how the inliner works.

The roadmaps for individual procedures contain the call site information. We collect all this information. For each call site we need to know the calling procedure, the innermost enclosing block of the call site, the exact place of the call site (the control-flow graph node), and the called procedure.

We construct the procedure call graph. We compute from this graph which procedures are leaves, and which procedures form recursive call cycles. We filter the set of all call sites according to some inlining criterion. The criterion is adjustable using a "compile-time" switch (see below), which gives some familiar inlining strategies, with the flexibility of adding any other, by means of an "up-called" predicate which the user may supply.

Once we know which call sites are to be inlined, we must determine an order of inlining. The policy implemented is *cascading* inlines: if A calls B and B calls C, and both call sites should be inlined, then C is first inlined into B, and then the resulting procedure is inlined into A. The cascading order of call sites is determined by (topologically) sorting them according to the must-be-after relation: site s1 (P1 calls Q1) must be inlined after site s2 (P2 calls Q2) if P2 is in the set of procedures reachable from Q1 in the call graph. Obviously, an *anticascading* order may be of interest as well. It will not lead to excessively large procedures in the root regions of the call graph, but will also prevent optimisations across multiple call layers which the cascading order allows. This question should be explored further, and the choice of strategy should be made adjustable.

Finally, we inline each call site. In the presence of nested procedures and blocks, when the called procedure has local procedures, an entire subtree in the block hierarchy must be copied to the call site, or more precisely, to the innermost enclosing block (the calling block). Furthermore, it cannot be copied intact, since the static scope (display) level where the called procedure is declared need not be the same as that of the call block. Accordingly, some variable references need to be changed. The details are too tedious to describe here. Then we must similarly transform the main body of the called procedure. In addition to considerations just described, references to any parameters (to the called procedure) and any return result are converted to references to fresh variables in the calling block. Finally, the call node in the calling block is replaced by code which moves values from registers (where they normally reside prior to the call) to the new variables, and the call-return node is replaced by code which loads the return result into its target register. Then the nodes of the called procedure are spliced between the former call and call-return. All block hierarchy and block symbol table information must be adjusted properly.

2.4.1 How to use

The following switch is available in structure `Inliner`:

```
datatype inline_policy =
   INLINE_POLICY_NONE
  | INLINE_POLICY_LEAVES_ONLY
  | INLINE_POLICY_ALL_NONREC
  | INLINE_POLICY_NAMELIST of string list
  | INLINE_POLICY_EXTERNAL_FUNCTION of
      (Intermediate.call_site
       * Intermediate.program_representation
       -> bool)
val inline_policy: inline_policy ref =
   ref INLINE_POLICY_NONE
```

The meaning is as follows:

- INLINE_POLICY_NONE — no call sites inlined,
- INLINE_POLICY_LEAVES_ONLY — all calls to leaf procedures inlined,
- INLINE_POLICY_ALL_NONREC — all calls inlined except for cycles of recursive calls,
- INLINE_POLICY_NAMELIST *names* — all calls to the named procedures are inlined (like the C inline directive),
- INLINE_POLICY_EXTERNAL_FUNCTION *function* — apply given function to each call site to determine whether to inline (requires understanding of compiler internals to use).

Source file prefix: `inliner`.

2.5 Expression propagation

This is currently the only phase where individual procedures are transformed (optimised). The complexity of the source language and the desire for ambitious optimisation led to a design that is quite complex and in its present implementation inefficient. The approach is as follows: perform a dataflow analysis to find out the *values* which are available in various *cells* at various points in the program.

By values we mean abstract values, which may be constants or symbolic expressions. (If constituents of a symbolic expression are found to be constants, or if algebraic rules so allow, an expression may reduce to a constant.) The constants may be of any type in the language, either a ground type (integer and boolean) or a structured type (record, array) or even a reference type. In the present implementation, symbolic expressions are not developed, and are treated as $\perp$ in the stub code. Therefore, expression propagation reduces to constant propagation.

By cells we mean all the visible storage of the procedure at hand, which includes its registers, its local variables, its local parameters and (possibly) return result, local variables of any blocks nested inside the procedure, all variables and parameters of blocks external to the procedure. The cells are typed, so each may only contain one type of value; the types may be structured.

The goal of this design was to permit tracking values assigned to arrays and records and their components. It turned out that the complexity of implementation was great precisely beacuse of this goal. It would have been much simpler to restrict attention to scalars. Having to determine for structures in variables (on the stack) what exactly is being assigned to, and then propagating the structured abstract values was difficult to code, and probably the source of gross inefficiencies in dataflow. On the other hand, it is not at all clear that the utility of this goal is significant.

Once the expression propagation dataflow problem is solved, we know for each node in the control-flow graph what is known about values of variables and registers just before it is executed and just after. We then examine each node and modify those that can be simplified. For example, if the value of a target register of that node is known to have a constant value after the execution of the node, then the node can be turned into a load-immediate (if the register is of a ground type). Many other cases of redundant computation, unnecessary loads and stores are handled similarly.

Following this transformation, many nodes remain in the graph whose results are nowhere used. Dataflow analysis is used again to compute reaching definitions and to compute reaching uses. With this information we turn useless computations into dummy nodes. We found that keeping use-def information accurate through a sequence of such modifications is an unpleasant task, and we should look for better ways to represent this information. Finally, all dummy nodes are removed from the control-flow graph.

Source file prefix: `commonsubexpression`.

2.5.1 How to use

The following knobs are available in structure `Commonsubexpression`:

```
val max_procedure_nodes: int ref = ref 100
val max_seconds_per_procedure = ref 600
```

The expression propagation dataflow phase runs unreasonably long for procedures with more than about 200 original nodes. This is the single worst performance problem in the compiler. These controls should be adequate to keep the compiler in check, so that the worst offenders can be avoided, while we still get other procedures optimised.

2.6 Abstract machine

The intermediate form is directly executable on an abstract machine which we also implemented. The machine is equipped with a store where both ordinary and heap variables live, a block activation stack through which the variables are accessed, a display through which the activation stack is accessed, a procedure invocation stack, and an unbounded supply of register windows each with an unbounded number of registers. The "program counter" keeps track of the current procedure and the current control-flow graph node. The interpreter is rather complex, because the values in registers and store cells can be structured. Profiling information is kept for each instruction kind, and also for lower-level operations, such as stack allocations and store reads and writes. Given a cost assignment, which is also adjustable, a final cost measure is computed as the weighted sum of the instruction counts. The default cost assignment is as follows:

```
val cost_polymorphic_eq = ref 1
val cost_deep_read = ref 30
val cost_deep_write = ref 20
val cost_allocate = ref 10
val cost_allocate_and_write = ref 10
val cost_deallocate = ref 10
val cost_cycle = ref 0
val cost_PROCEDUREENTRY = ref 1
val cost_PROCEDUREEXIT = ref 1
val cost_BLOCKENTRY = ref 1
val cost_BLOCKEXIT = ref 1
val cost_IMMEDIATE_INT = ref 1
val cost_IMMEDIATE_BOOL = ref 1
val cost_LOCATE_VARIABLE = ref 0
val cost_LOCATE_PARAMETER = ref 0
val cost_LOCATE_RETURNRESULT = ref 0
val cost_LOAD = ref 1
val cost_STORE = ref 1
val cost_CALC_ADD = ref 1
val cost_CALC_SUB = ref 1
val cost_CALC_MUL = ref 1
val cost_CALC_DIV = ref 1
val cost_CALC_MOD = ref 1
val cost_CALC_UMINUS = ref 1
val cost_CALC_EQ = ref 1
val cost_CALC_NE = ref 1
val cost_CALC_GT = ref 1
```

```
val cost_CALC_LT = ref 1
val cost_CALC_GE = ref 1
val cost_CALC_LE = ref 1
val cost_CALC_AND = ref 1
val cost_CALC_OR = ref 1
val cost_CALC_NOT = ref 1
val cost_RANGECK = ref 1
val cost_LOCATIVE_INDEX = ref 1
val cost_VALUE_INDEX = ref 1
val cost_LOCATIVE_RECORDSELECT = ref 1
val cost_VALUE_RECORDSELECT = ref 1
val cost_TEST = ref 1
val cost_CALL = ref 1
val cost_CALL_RETURN = ref 1
val cost_WRITE = ref 1
val cost_READ = ref 1
val cost_JOIN = ref 0
val cost_DUMMY = ref 0
val cost_HEAP_NEW = ref 20
val cost_HEAP_NIL = ref 1
val cost_HEAP_DEREF = ref 1
val cost_MOVE = ref 0
```

When execution of a program completes, the instruction profile is printed out. It will look like this:

```
--------------- Execution profile ---------------
polymorphic_eq  (cost 1) = 1008
deep_read  (cost 30) = 54067
deep_write  (cost 20) = 8189
allocate  (cost 10) = 4619
allocate_and_write  (cost 10) = 4655
deallocate  (cost 10) = 9274
cycle  (cost 0) = 103018
PROCEDUREENTRY  (cost 1) = 665
PROCEDUREEXIT  (cost 1) = 665
BLOCKENTRY  (cost 1) = 1004
BLOCKEXIT  (cost 1) = 1004
IMMEDIATE_INT  (cost 1) = 6486
IMMEDIATE_BOOL  (cost 1) = 126
LOCATE_VARIABLE  (cost 0) = 21959
LOCATE_PARAMETER  (cost 0) = 3379
LOCATE_RETURNRESULT  (cost 0) = 346
LOAD  (cost 1) = 20226
STORE  (cost 1) = 7451
CALC_ADD  (cost 1) = 3487
CALC_SUB  (cost 1) = 878
CALC_MUL  (cost 1) = 525
CALC_DIV  (cost 1) = 227
CALC_MOD  (cost 1) = 34
CALC_UMINUS  (cost 1) = 26
CALC_EQ  (cost 1) = 1008
CALC_NE  (cost 1) = 0
CALC_GT  (cost 1) = 1828
CALC_LT  (cost 1) = 2014
CALC_GE  (cost 1) = 1515
```

```
CALC_LE  (cost 1) = 1446
CALC_AND  (cost 1) = 2998
CALC_OR  (cost 1) = 1741
CALC_NOT  (cost 1) = 414
RANGECK  (cost 1) = 2544
LOCATIVE_INDEX  (cost 1) = 1210
VALUE_INDEX  (cost 1) = 1334
LOCATIVE_RECORDSELECT  (cost 1) = 1618
VALUE_RECORDSELECT  (cost 1) = 4391
TEST  (cost 1) = 3226
CALL  (cost 1) = 664
CALL_RETURN  (cost 1) = 664
WRITE  (cost 1) = 87
READ  (cost 1) = 0
JOIN  (cost 0) = 1638
DUMMY  (cost 0) = 4190
HEAP_NEW  (cost 20) = 0
HEAP_NIL  (cost 1) = 0
HEAP_DEREF  (cost 1) = 0
MOVE  (cost 0) = 0
-------------------------------------------------
Total cost = 2043784
-------------------------------------------------
```

In addition to execution, which takes one configuration to the next, there are provisions for tracing, which constructs a full map of an execution, with all intermediate configurations intact. In an imperative language, keeping around multiple copies of the store would be extremely expensive. In SML, as long as incremental changes to data structures are very small, non-destructive updating makes tracing more sensible, although very long traces still cause memory blowup.

Source file prefix: `machine/modulaX_regwindows`.

2.7 Utilities

Several underlying utilites support the main algorithms above. Heavy use is made of integer sets and integer maps, both defined in the Standard ML Library. We added functionality to these structures with `IntSet_Extra` and `IntMap_Extra`. Directed graphs with labelled nodes and edges, and basic graph algorithms are implemented in the structure `Digraph`, using integer sets and maps. Trees and forests in structure `Forest` are used for hierarchical structures, and are built on top of digraphs. We have two dominator algorithms implemented, Purdom-Moore and Aho-Ullman, but not the fast Lengauer-Tarjan algorithm. We are not using these algorithms but we will be once symbolic expressions are in place. There is a generic dataflow solver based on the (corrected) model from Allen/Rosen/Zadeck. It is used directly by the Aho-Ullman dominator algorithm. The reaching definitions and uses solvers are wrapped around the dataflow algorithm. They remain generic in that they require the node graphs to be interpreted for them. Thus, none of the utility codes is specific to the Modula-X compiler.

Source file prefixes:
`util/aho_ullman_dominators`,
`util/digraph`,
`util/dominators`,
`util/dominators_aux`,
`util/edge_dataflow`,

```
util/forest,
util/intmap_extra,
util/intset_extra,
util/list_fns,
util/purdom_moore_dominators,
util/reaching_definitions,
util/reaching_uses.sig,
util/stringmap.
```

3 How to use

To use, one must have SML/NJ, version 0.93, with SourceGroup, and the Standrad Library, version 0.2. The source files are listed in `files.txt`; a building and experimentation script is in `prova.sml`. The structure `ModulaX`, defined in `main.sml`, provides functions `compile` and `run` which one would ordinarily invoke. Source files must lie in the subdirectory `test/`. It also defines a substructure Options, which contains several compile-time and run-time switches:

```
structure Options =
   struct
      val inline = ref false
      val cse = ref false
      val print_compileinfo = ref false
      val print_execinfo = ref false
   end
```

One will also want to control the individual optimisations as described above.

4 Performance

We shall only discuss the performance of compiled code, not of the compiler itself. In the implementation, little attention was paid to performance, and as a result very nonlinear time is spent in expression propagation (so much that some procedures ran out of time – see question marks in the table below), although other parts of the compiler are well-behaved.

The test programs were as follows. From the joint test suite:[1][2]

- `joint-matmult.m` — Matrix multiplication.
- `joint-fortest.m` — Trivial FOR loop test.
- `joint-fib.m` — Non-recursive Fibonacci sequence calculator.
- `joint-pano.m` — Small contrived program offering many optimisation opportunities.

Other programs:

[2] The remaining program from that suite, `joint-arrays.m`, is not a valid program in this language, since it has an EXIT without a corresponding LOOP.

- cse4.m, cse17.m, cse18.m, cse19.m, cse20.m — Small contrived programs to verify constant propagation.
- comparison-matrixinit.m — Small contrived program to test inlining and expression propagation: elements of an array are initialised to a constant value in a procedure.
- comparison-matrixop.m — Small contrived program to test inlining and expression propagation: elements of an array are computed in a procedure.
- stchain.m — A program that tests the functioning of strangely nested procedures. Good test of inlining correctness.
- g.m — originally written in Pascal, a large integer arithmetic implementation, which uses records and arrays, but no dynamic allocation. The experience with the old back end was that a large fraction of time was spent on array subscript range checks. This is the largest program existing in Modula-X. Unfortunately, the compiler is not sufficiently robust to compile this program with optimisations; it either takes too long or it runs out of memory.

4.1 Comparisons

Here we compare the cycles and costs when a program is compiled with or without inlining and expression propagation. Inlining uses the leaves-only policy. The results are summarised in table 1.

Program	Size (lines)	Original (cycles / cost)	Cse only (cycles / cost)	Inline only (cycles / cost)	Inline & cse (cycles / cost)
joint-fortest.m	16	18 047 / 167 542	11 515 / 103 222	n/a	n/a
joint-fib.m	38	4 200 / 41 653	2 495 / 28 978	4 206 / 41 671	2 446 / 27 930
joint-pano.m	33	3 666 / 40 475	2 122 / 20 746	n/a	n/a
joint-matmult.m	45	1 048 470 / 763 638 217	?1 048 470 / 763 638 217	n/a	n/a
cse4.m	19	65 / 851	7 / 187	n/a	n/a
cse17.m	21	402 / 6 243	390 / 6 213	422 / 6 248	317 / 5 594
cse18.m	21	59 / 1 333	52 / 1181	63 / 1 350	11 / 410
cse19.m	17	265 / 3 161	253 / 3 131	285 / 3 166	180 / 2 512
cse20.m	35	5 821 / 348 352	5 520 / 348 352	6 421 / 354 152	4 385 / 341 042
comparison-matrixinit.m	21	6 095 / 385 466	5 853 / 385 136	6 695 / 385 666	?6 695 / 385 666
comparison-matrixop.m	23	10 495 / 1 092 906	10 253 / 1 092 576	11 095 / 1 093 106	9 055 / 1 080 522
stchain.m	148	894 / 8 787	874 / 8 757	914 / 8 922	872 / 8 750
g.m	1 424	103 018 / 2 043 784	?	104 902 / 2 063 881	?

Table 1: Execution times of programs under different optimisations

As can be seen, with our implementation of inlining and cost assignment, inlining by itself increases the execution cost. However, it is seen to offer opportunities for expression propagation.

We tried limited experimentation with inlining policies. We wanted first to compare leaves-only and all-nonerecursive policies. It turns out that comparisons are only meaningful for programs that have chains of calls; the only ones we had were stchain.m and g.m. The latter was too big to compile. For stchain.m with all nonrecursive sites inlined, we got 948/9139 without cse, and 932/9139 with cse. Thus on that one program, expanding leaves only was a better choice. We wrote cse20.m as a contrived (but plausible) example, and obtained 6821/353872 without cse, 4285/340762 with cse, so for this program it was better to inline all call sites, provided this was followed by cse.

References

[1] Brendon Cahoon, Stephanie Coleman, John Ridgway, Sharad Singhai, and Glen Weaver. 710 compiler experiment. University of Massachusetts, Amherst, 1994.

[2] Eliot Moss. CMPSCI 710 project compiler description. University of Massachusetts, Amherst, 1994.

[3] David Schmidt. *Denotational Semantics*. Allyn and Bacon, Inc., Boston, first edition, 1986.

[4] Darko Stefanović. Implementing a small imperative language with safe dynamic allocation. Memo, April 1993.

[5] Norm Walsh. CS791V project report: Constructing a framework for compiler optimization. University of Massachusetts, Amherst, 1992.

A Abstract syntax

```
(* The abstract syntax of Modula-X, expressed
  as several recursive data types in SML
*)

signature MODULAX_SYNTAX =

sig

datatype Program =
             PROGRAM of Module

and      Module =
             MODULE       of (* id: *) string
                           * (* block: *) Block

and      Block =
             BLOCK        of (* OptDecls: *) Declaration list
                           * (* Stmts: *) Stmt list

and      Declaration =
             DECLCONST    of (* id: *) string
                           * (* type: *) Typ option
                           * (* Expr: *) Expr
           | DECLTYPE     of (* id: *) string
                           * (* type: *) Typ
           | DECLTYPEREC  of ((* id: *) string
                           * (* type: *) Typ) list
           | DECLVAR      of (* idlist: *) string list
                           * (* type: *) Typ option
                           * (* Expr: *) Expr option
           | DECLPROC     of (* id: *) string
                           * (* signature: *) Signature
                           * (* Blocknoscope: *) Block

and      Signature =
             SIGNATURE    of (* Optformals: *) Formal list
                           * (* OptDeclType: *) Typ option

and      Formal =
             FORMAL       of (* idlist: *) string list
                           * (* optMode: *) Mode option
                           * (* type: *) Typ

and      Mode =
             VALUE
           | VAR
           | READONLY

and      Stmt =
             STMTBLOCK    of Block
           | STMTASSIGN   of Expr * Expr
           | STMTCALL     of ProcCall
```

```
          | STMTEXIT
          | STMTEVAL     of Expr
          | STMTFOR      of (* id: *) string
                          * (* from: *) Expr
                          * (* to: *) Expr
                          * (* optby: *) Expr option
                          * (* stmts: *) Stmt list
          | STMTIF       of (Expr * (* then part stmts: *) Stmt list)
                          (* some ELSIF clauses: *)
                          * (Expr * Stmt list) list
                          (* and an ELSE clause: *)
                          * (Stmt list) option
          | STMTLOOP     of Stmt list
          | STMTREAD     of (* format: *) string
                          * (* read list: *) Expr list
          | STMTREPEAT   of Stmt list
                          * (* until: *) Expr
          | STMTRETURN   of Expr option
          | STMTWHILE    of Expr
                          * Stmt list
          | STMTWRITE    of (* format: *) string
                          * (* write list: *) Expr list

and      Typ =
            TYPARRAY     of Typ list * Typ
          | TYPRECORD    of Fields
          | TYPSUBRANGE  of Expr * Expr
          | TYPID        of string
          | TYPREF       of Typ

and      Fields =
            FIELDS       of ((string list) * Typ) list

and      Expr =
            EXPROR       of Expr * Expr
          | EXPRAND      of Expr * Expr
          | EXPRNOT      of Expr
          | EXPREQUAL    of Expr * Expr
          | EXPRNEQUAL   of Expr * Expr
          | EXPRLESS     of Expr * Expr
          | EXPRGREATER  of Expr * Expr
          | EXPRLE       of Expr * Expr
          | EXPRGE       of Expr * Expr
          | EXPRADD      of Expr * Expr
          | EXPRSUB      of Expr * Expr
          | EXPRMUL      of Expr * Expr
          | EXPRDIV      of Expr * Expr
          | EXPRMOD      of Expr * Expr
          | EXPRUPLUS    of Expr
          | EXPRUMINUS   of Expr
          | EXPRID       of string
          | EXPRINT      of int
          | EXPRTEXT     of string
          | EXPRRECSEL   of Expr
                          * (* id: *) string
```

```
             | EXPRARRAYREF of Expr
                             * Expr list
             | EXPRPROCCALL of ProcCall
             | EXPRDEREF    of Expr
             | EXPRNEW      of Typ
             | EXPRNIL      of Typ

and      ProcCall =
             PROCCALL     of Expr
                             * Expr list

end
```

B Intermediate form

```
signature INTERMEDIATE =
   sig
      structure ModulaX_Types: MODULAX_TYPES
      structure ModulaX_Ints: MODULAX_INTS
      structure Digraph: DIGRAPH
      structure Forest: FOREST

      datatype blocks_hierarchy_node =
         BLOCKS_HIERARCHY_NODE of
         {
           level: int,
           is_procedure: bool,
           name: string
           }

      datatype blocks_hierarchy =
         BLOCKS_HIERARCHY of blocks_hierarchy_node Forest.forest

      datatype parameter_mode = PASS_BY_VALUE | PASS_BY_REFERENCE

      datatype block_symboltable =
         BLOCK_SYMBOLTABLE of
         {
           level: int,
           procedures: (string * int) IntMap.intmap,
           variables: (string * ModulaX_Types.ty) IntMap.intmap,
           parameters:
           (string * ModulaX_Types.ty * parameter_mode) IntMap.intmap,
           procedures_byname: int StringMap.dict,
           variables_byname: int StringMap.dict,
           parameters_byname: int StringMap.dict,
           result: ModulaX_Types.ty option
           }

      datatype module_symboltable =
         MODULE_SYMBOLTABLE of block_symboltable IntMap.intmap

      datatype register =
         REGISTER of int
         (* this abstraction will allow handling FP registers in the future *)

      datatype variable =
         VARIABLE of {display: int, offset: int}

      datatype parameter =
         PARAMETER of {display: int, offset: int}

      datatype procedure =
         PROCEDURE of {display: int, offset: int, block: int}

      datatype flowgraph_node =
         FLOWGRAPH_PROCEDUREENTRY
       | FLOWGRAPH_PROCEDUREEXIT
```

```
| FLOWGRAPH_BLOCKENTRY of {block_number: int}
| FLOWGRAPH_BLOCKEXIT of {block_number: int}

| FLOWGRAPH_IMMEDIATE_INT of
{reg: register, const: ModulaX_Ints.integer}
| FLOWGRAPH_IMMEDIATE_BOOL of
{reg: register, const: bool}

| FLOWGRAPH_LOCATE_VARIABLE of
{reg: register, variable: variable}
| FLOWGRAPH_LOCATE_PARAMETER of
{reg: register, parameter: parameter}
| FLOWGRAPH_LOCATE_RETURNRESULT of
{reg: register, display: int}

| FLOWGRAPH_LOAD of {loc_reg: register, value_reg: register}
| FLOWGRAPH_STORE of {value_reg: register, loc_reg: register}

| FLOWGRAPH_CALC_ADD of
{source1_reg: register, source2_reg: register, result_reg: register}
| FLOWGRAPH_CALC_SUB of
{source1_reg: register, source2_reg: register, result_reg: register}
| FLOWGRAPH_CALC_MUL of
{source1_reg: register, source2_reg: register, result_reg: register}
| FLOWGRAPH_CALC_DIV of
{source1_reg: register, source2_reg: register, result_reg: register}
| FLOWGRAPH_CALC_MOD of
{source1_reg: register, source2_reg: register, result_reg: register}
| FLOWGRAPH_CALC_UMINUS of
{source_reg: register, result_reg: register}

  (* EQ and NE are polymorphic *)
| FLOWGRAPH_CALC_EQ of
{source1_reg: register, source2_reg: register, result_reg: register}
| FLOWGRAPH_CALC_NE of
{source1_reg: register, source2_reg: register, result_reg: register}

| FLOWGRAPH_CALC_GT of
{source1_reg: register, source2_reg: register, result_reg: register}
| FLOWGRAPH_CALC_LT of
{source1_reg: register, source2_reg: register, result_reg: register}
| FLOWGRAPH_CALC_GE of
{source1_reg: register, source2_reg: register, result_reg: register}
| FLOWGRAPH_CALC_LE of
{source1_reg: register, source2_reg: register, result_reg: register}

| FLOWGRAPH_CALC_AND of
{source1_reg: register, source2_reg: register, result_reg: register}
| FLOWGRAPH_CALC_OR of
{source1_reg: register, source2_reg: register, result_reg: register}
| FLOWGRAPH_CALC_NOT of
{source_reg: register, result_reg: register}

| FLOWGRAPH_RANGECK of
{low: ModulaX_Ints.integer, high: ModulaX_Ints.integer,
```

```
   source_reg: register, result_reg: register}

 | FLOWGRAPH_LOCATIVE_INDEX of
 {array_loc_reg: register, index_value_reg: register,
   elem_loc_reg: register}
 | FLOWGRAPH_LOCATIVE_RECORDSELECT of
 {record_loc_reg: register, field_offset: int,
   field_loc_reg: register}
 | FLOWGRAPH_VALUE_INDEX of
 {array_value_reg: register, index_value_reg: register,
   elem_value_reg: register}
 | FLOWGRAPH_VALUE_RECORDSELECT of
 {record_value_reg: register, field_offset: int,
   field_value_reg: register}

 | FLOWGRAPH_HEAP_NEW of {ref_loc_reg: register}
 | FLOWGRAPH_HEAP_NIL of {ref_loc_reg: register}
 | FLOWGRAPH_HEAP_DEREF of {ref_reg: register, object_reg: register}
    (* DEREF converts a REF 't   into a  't loc   *)

 | FLOWGRAPH_CALL of {procedure: procedure,
   actuals: register list, return_reg: register option}
 | FLOWGRAPH_CALL_RETURN of {procedure: procedure,
   actuals: register list, return_reg: register option}

 | FLOWGRAPH_TEST of {reg: register, on_true: int, on_false: int}

 | FLOWGRAPH_WRITE of {format: string, values: register list}
 | FLOWGRAPH_READ of {format: string, locatives: register list}

 | FLOWGRAPH_JOIN
 | FLOWGRAPH_DUMMY
 | FLOWGRAPH_MOVE of {source_reg: register, result_reg: register}
                      (* NOT USED EXCEPT IN OPTIMISATIONS *)
datatype call_site =
   CALL_SITE  of {calling_proc: int, calling_block: int,
     call_node: int, called_proc: int}

datatype roadmap =
   ROADMAP of {entrypoint: int,
               exitpoint: int,
               callsites: call_site list
               }

datatype extended_type =
   VALUE of ModulaX_Types.ty
 | LOCATIVE of extended_type

datatype procedure_flowgraph =
   PROCEDURE_FLOWGRAPH of
   {
     graph: (unit, flowgraph_node, unit) Digraph.digraph,
     registers: extended_type IntMap.intmap,
     roadmap: roadmap
     }
```

```
   datatype module_flowgraphtable =
      MODULE_FLOWGRAPHTABLE of procedure_flowgraph IntMap.intmap

   datatype program_representation =
      PROGRAM_REPRESENTATION of
      {
        blocks_hierarchy: blocks_hierarchy,
        module_symboltable: module_symboltable,
        module_flowgraphtable: module_flowgraphtable
        }

   val etystring: extended_type -> string
   val get_new_register: extended_type IntMap.intmap * extended_type
   -> extended_type IntMap.intmap * register

   val display_blocks_hierarchy: blocks_hierarchy -> unit
   val display_block_symboltable: block_symboltable -> unit
   val display_module_symboltable: module_symboltable -> unit
   val flowgraph_node_as_string: flowgraph_node -> string
   val display_cfg: (unit, flowgraph_node, unit) Digraph.digraph -> unit
   val display_procedure_flowgraph: procedure_flowgraph -> unit
   val display_module_flowgraphtable: module_flowgraphtable -> unit
   val display_program: program_representation -> unit

end
```

C Example programs

C.1 cse4.m

```
MODULE cse4;
VAR
    i,j,k,l,m,n,o,p,q: INTEGER;

BEGIN
   i := 50;              (*50*)
        j := i * 2;      (*100*)
        k := j;          (*100*)
        l := i+k;        (*150*)
        m := l DIV k;    (*1*)
        n := i DIV i;    (*1*)
        o := n + m + l;  (*152*)
        p := o - 3;      (*149*)
        q := p MOD 5;    (*4*)

   WRITE ("i=%d, q=%d\n", i, q);
END cse4.
```

C.2 cse17.m

```
MODULE cse17;
CONST size = 5;
TYPE matrix = ARRAY [1..size] OF INTEGER;
VAR i: INTEGER;
    A, B, C: matrix;

PROCEDURE calc_elem (ii: INTEGER) =
   BEGIN
      C [ii] := A[ii] * B[ii];
   END calc_elem;

BEGIN
      WRITE ("[ ");
      FOR i := 1 TO size DO
         A [i] := i;
         B [i] := i;
         calc_elem (i);
         WRITE ("%d ", C [i]);
      END;
      WRITE ("]\n");
END cse17.
```

C.3 cse18.m

```
MODULE cse18;
CONST size = 5;
TYPE matrix = ARRAY [1..size] OF INTEGER;
VAR i: INTEGER;
    A, B, C: matrix;

PROCEDURE calc_elem (ii: INTEGER) =
```

```
   BEGIN
      C [ii] := A[ii] * B[ii];
   END calc_elem;

BEGIN
      WRITE ("[ ");
    i := 3;
         A [i] := i;
         B [i] := i;
         calc_elem (i);
         WRITE ("%d ", C [i]);

      WRITE ("]\n");
END cse18.
```

C.4 cse19.m

```
MODULE cse19;
CONST size = 5;
TYPE matrix = ARRAY [1..size] OF INTEGER;
VAR i: INTEGER;
    C: matrix;

PROCEDURE calc_elem (ii: INTEGER) =
   BEGIN
      C [ii] := ii;
   END calc_elem;

BEGIN
      FOR i := 1 TO size DO
         calc_elem (i);
         WRITE ("%d\n", C [i]);
      END;
END cse19.
```

C.5 cse19.m

```
MODULE cse20;

(* modelling a threshold circuit *)

CONST size = 100;
CONST threshold = 5;

TYPE matrix = ARRAY [1..size] OF INTEGER;
VAR i: INTEGER;
    C: matrix;

PROCEDURE sq (x: INTEGER): INTEGER =
   BEGIN
      RETURN x * x;
   END sq;

PROCEDURE calc_elem (ii: INTEGER) =
   BEGIN
```

```
      IF ii <= threshold THEN
         C [ii] := 2 * sq (ii);
      ELSE
         C [ii] := 2 * sq (threshold);
      END;
   END calc_elem;

BEGIN
      i := 1;
      WRITE ("[");
      WHILE i <= size DO
         calc_elem (i);
         WRITE ("%d ", C [i]);
         i := i + 1;
      END;
      WRITE ("]\n");
END cse20.
```

C.6 comparison-matrixinit.m

```
MODULE comparison_matrixinit;
CONST size = 10;
TYPE matrix = ARRAY [1..size] OF ARRAY [1..size] OF INTEGER;
VAR i, j: INTEGER;
    C: matrix;

PROCEDURE calc_elem (ii, jj: INTEGER) =
   BEGIN
      C [ii, jj] := 5;
   END calc_elem;

BEGIN
   FOR i := 1 TO size DO
      WRITE ("[ ");
      FOR j := 1 TO size DO
         calc_elem (i, j);
         WRITE ("%d ", C [i, j]);
      END;
      WRITE ("]\n");
   END;
END comparison_matrixinit.
```

C.7 comparison-matrixop.m

```
MODULE comparison_matrixop;
CONST size = 10;
TYPE matrix = ARRAY [1..size] OF ARRAY [1..size] OF INTEGER;
VAR i, j: INTEGER;
    A, B, C: matrix;

PROCEDURE calc_elem (ii, jj: INTEGER) =
   BEGIN
      C [ii, jj] := A[ii, jj] * B[ii, jj];
   END calc_elem;
```

```
BEGIN
   FOR i := 1 TO size DO
      WRITE ("[ ");
      FOR j := 1 TO size DO
         A [i, j] := i;
         B [i, j] := j;
         calc_elem (i, j);
         WRITE ("%d ", C [i, j]);
      END;
      WRITE ("]\n");
   END;
END comparison_matrixop.
```

C.8 joint-pano.m

```
MODULE PANO;

(* A panoramic test designed to test all five optimizations *)

VAR
  start,stop,step, y, z : INTEGER;
  cptest, cse, licm: INTEGER;

BEGIN
  start := 1;
  stop  := 100;
  step  := 2;
  y := 0;
  z := 0;
  licm := 0;
  cptest := 0;

  IF y # y THEN
    WRITE("should not be printed\n");      (* dead code *)
  END;

  FOR i := start TO stop BY step DO
    y := y + step;
    z := y + step;    (* useless code since z is never printed out *)
    cptest := i * step;   (* for constant propagation and strength reduction *)
    cse := i + (i * step);  (*  i*step should be caught by cse *)

    licm := start + stop;    (* should be caught by licm *)

    WRITE("i=%d, y=%d, cptest=%d, cse=%d, licm=%d\n",i, y, cptest, cse, licm);
  END;

END PANO.
```

C.9 joint-matmult.m

```
MODULE matmult;
  (* This module does matrix multiply and prints out the result at the end *)
  (* The result will be z = x * y *)
TYPE
```

```
  multdim_array =
  ARRAY [0..25], [0..25]  OF INTEGER;

VAR
  i, j, k : INTEGER;
  x,y,z : multdim_array;

BEGIN
  FOR i:= 0 TO 25 DO
    FOR j := 0 TO 25 DO
        x[i][j] := 1;
        IF j > 4 THEN
          y[i][j] := 2;
        ELSE
          y[i][j] := j * 2;
        END;
    END;
  END;

  FOR i:= 0 TO 25 DO
    FOR j := 0 TO 25 DO
        FOR k := 0 TO 25 DO
           z[i][j] := x[i][k] * y[k][j];
        END;
    END;
  END;

  WRITE ("Contents of Z are: \n");
  FOR i:= 0 TO 25 DO
    FOR j := 0 TO 25 DO
        WRITE ("%d ", z[i][j]);
    END;
    WRITE("\n");
  END;
END matmult.
```

C.10 joint-fortest.m

```
MODULE Fortest;

VAR
  start,stop,step, y : INTEGER;

BEGIN
  start := 1;
  stop  := 1000;
  step  := 2;

  FOR i := start TO stop BY step DO
    y := y + step;
    WRITE("i=%d\n",i);
    WRITE("y=%d\n",y);
  END;
END Fortest.
```

C.11 joint-fib.m

```
MODULE Fib;

PROCEDURE FigureFib(N: INTEGER): INTEGER =
VAR
  Prev  : ARRAY[1..2] OF INTEGER;
  FibNum: INTEGER;
  Loop  : INTEGER;
BEGIN
  IF (N = 0 OR N = 1) THEN
    RETURN 1;
  END;

  Prev[1] := 1;
  Prev[2] := 1;
  FOR Count := 2 TO N DO
    FibNum := 0;
    FOR Loop := 1 TO 2 DO
      FibNum := FibNum + Prev[Loop];
    END;
    Prev[1] := Prev[2];
    Prev[2] := FibNum;
  END;

  RETURN FibNum;
END FigureFib;

VAR
  N : INTEGER;
BEGIN
  IF (FALSE) THEN
    WRITE("This dead code will be removed by the optimizer.");
    WRITE("Note: the tuples remain in the source.");
  ELSE
    WRITE("Finding Fib of 30\n");
    N := 30;
    WRITE("Fib(%d) is %d\n",N,FigureFib(N));
  END;
END Fib.
```

C.12 stchain.m

```
MODULE phase5test;

PROCEDURE printscores(a,b,c:INTEGER) =
BEGIN
  IF a = b THEN
    WRITE ("\t:-) GOOD : got full %d pt(s).       TOTAL SCORE: %d\n", a,c);
  ELSE
    WRITE ("\t:-( SORRY: missed %d pt(s).         TOTAL SCORE: %d\n", a,c);
  END;
END printscores;
VAR
  score := 0;
```

```
  local_score :INTEGER;


(* 1. Simple static chain checks:   (10 pts)
        a. using variables from outer levels
        b. using arguments from outer levels
        c. calling one level within.

   2. Twisted static chain: testing the two types of calling : (20 pts)
                b. same level
                c. levels enclosing current lexical level.
*)


PROCEDURE scope1 (l,m,n,o:INTEGER):INTEGER =
  VAR
     a :INTEGER;
  PROCEDURE scope2 (m,n,o:INTEGER) =
    VAR
      b :INTEGER;
    PROCEDURE scope3 (n,o:INTEGER) =
      VAR
        c :INTEGER;
      PROCEDURE scope4 (o:INTEGER) =
        VAR
          d :INTEGER;
      BEGIN
        d:=l*o;
        IF (b = 2*a) AND (c = 2*b) AND (d = 2*c) THEN
          local_score := 10;
        ELSE
          local_score := 0;
        END;
          WRITE("\ta: %d   b: %d   c: %d   d: %d\n",a,b,c,d);
      END scope4;

    BEGIN
      c := l*o;
      scope4(2*o);
    END scope3;
  BEGIN
    b := l*o;
    scope3(n,2*o);
  END scope2;
BEGIN
  a:= l*o;
  scope2(m,n,2*o);
  RETURN(local_score);
END scope1;

PROCEDURE twisted_chain(l,m:INTEGER): INTEGER =
  VAR store : INTEGER;
      count := 1;
      local_score:=0;
```

```
  PROCEDURE a1(i,j:INTEGER) =
    VAR b :INTEGER;
    PROCEDURE a2 (k,l:INTEGER) =
    VAR a:INTEGER;
    BEGIN
      a := i;  (* checking to see if the arguments from outer scope work
                right *)
      b:= l+1 ; (* checking to see if variables from the outer scope work *)
      WRITE("a : %d, l : %d\n", a,l);
      WRITE("k : %d, b : %d\n", k,b);
      IF (a = store) AND (k = store-1) AND (l = count) AND (b = count+1)
      THEN local_score := local_score+2;
           store := store - 2;
           count := count + 2;
      END;
      IF (i = 0) OR (k = 0) THEN RETURN;
      ELSE a1(k-1,l+1);
      END;
    END a2;
  BEGIN
    a2(i-1,j+1);
  END a1;

  VAR outer_k, outer_l : INTEGER;

  PROCEDURE b1 (i,j:INTEGER) =
    PROCEDURE b2a(k,l:INTEGER) =
    BEGIN
      WRITE("k : %d , l: %d\n",k,l);
      outer_k := k; outer_l := l;
      IF (outer_k = store) AND (outer_l = count)
      THEN local_score := local_score+1;
           store := store - 1;
           count := count + 1;
      END;
      IF (k = 0) THEN RETURN
      ELSE b1(k-1, l+1);
      END;
    END b2a;
    PROCEDURE b2b(k,l:INTEGER) =
    BEGIN
      b2a(k,l);
    END b2b;
  BEGIN
    b2b(i,j);
  END b1;

BEGIN
  WRITE("TEST a : 10 pts\n");
  store := l;
  a1(l,m);
  store := l-2;
  count := 0;
  outer_k:= l-2;
  outer_l:= 0;
```

```
  WRITE("TEST b : 10 pts\n");
  b1(l-2,m);
  RETURN local_score;
END twisted_chain;


BEGIN
  WRITE("TESTING STATIC CHAINS:\n");
  score := score + scope1(2,3,4,5);
  WRITE("TEST : simple static chain \n");
  printscores(10,local_score, score);

  WRITE("TEST : 'twisted' static chain \n");
  local_score := 2*twisted_chain(5,0);
  score := score + local_score;
  printscores(20,local_score, score);

END phase5test.
```