HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **B:** Technical Reports ISSN 0783–540X
No. **15**; March 1996 ISBN 951–22–2993–5

ReFlEx—AN EXPERIMENTAL TOOL FOR SPECIAL-PURPOSE PROCESSOR CODE GENERATION

Eero Lassila

Digital Systems Laboratory
Department of Computer Science
Helsinki University of Technology
Otaniemi, FINLAND

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
Otaniemi, Otakaari 1
FIN–02150 ESPOO, FINLAND

HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **B:** Technical Reports
No. **15**; March 1996

ISSN 0783–540X
ISBN 951–22–2993–5

ReFlEx—An Experimental Tool for Special-Purpose Processor Code Generation

Eero Lassila

Abstract: Code generation for embedded special-purpose processors is usually a difficult task for compiler writers as well as for assembly language programmers. This report describes an experimental demonstration prototype of a code generation tool. The tool is a retargetable assembly-code-level macro expander capable of program flow analysis. The main advantage offered by this macro expander is its strong support for macro hierarchy. The enhanced modularity provided by hierarchical macro libraries can make the code (produced either by the compiler writer or by the assembly language programmer) more easily readable, maintainable, and reusable. Still, a procedure written in the macro language retains its machine-specificity and, consequently, its efficiency.

Keywords: Code generation, special-purpose processors, macro expansion, program flow analysis.

Printing: TKK Monistamo; Otaniemi 1996

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
Otaniemi, Otakaari 1
FIN–02150 ESPOO, FINLAND

Phone: $\frac{90}{+358\text{–}0}$ 4511
Telex: 125 161 htkk fi
Telefax: +358–0–465 077
E-mail: lab@saturn.hut.fi

Contents

1 Introduction

In this report, we propose a novel method for machine code generation. Basically, the tool we have implemented is a retargetable assembly-code-level macro expander: it transforms a "higher-level" but still machine-specific program description into assembly language. Governed by user-written rules (i.e. macro definitions), the expander can generate code for different target processors. The prototype design documented here, ReFlEx 1.0, is intended for demonstrative purposes only; it is not yet meant for coping with real processor architectures. However, we believe that a similar but considerably more advanced tool might meet many of the challenges posed by modern embedded special-purpose processor architectures. Currently, embedded processor code generation [17] is a most demanding task for high-level language compiler writers as well as for assembly language programmers. Our ultimate goal is to support both of these two groups of software professionals in their work.

The name **ReFlEx** of the implemented program comes from a '**Re**targetable **Fl**ow-sensitive macro **Ex**pander'. The idea of making a macro expander perform full program flow analysis is the essence of our work. (To avoid confusion, we speak of 'macro expanders' instead of the more common 'macro processors'; by a 'processor' we always refer to computer hardware.)

1.1 Background

In computer programs, abstraction through modularity improves readability, maintainability, and reusability: the details of a module implementation become hidden from the clients of the module. Object-oriented programming is one of the popular modularity-promoting mechanisms at the high-level language level; we try to convince you that the use of relatively powerful modularity-promoting mechanisms may be justified even at the assembly language level.

Do we really need to bother much about the peculiarities of assembly languages? Is it not so that RISC-type microprocessors are specifically designed to be programmed with optimizing high-level language compilers? The answer to the latter question is yes [19], of course, but there are also many important special-purpose processors for which no compiler available seems to be good enough. Typically, such processors are used in embedded real-time computing systems: representative examples can be found among *digital signal processors (DSPs)* [14]. In many telecommunication systems, for instance, compiled code is currently ruled out by efficiency requirements. Thus, the programmers get themselves involved in the mess known as assembly language programming.

Special-purpose processor architectures present programmers with three inherent technical complications—all of which are closely related to the efficiency requirement:

- Unconventional functionality with both limitations and extensions.
- Architectural irregularity: a heterogeneous register set and an idiosyncratic instruction set.
- Fine instruction granularity.

Unconventional functionality and architectural irregularity are burdens to the compiler writer. They stem from the utmost adaptation to the narrow field of intended applications; the general-purpose RISC chips, in contrast, have regular architectures. Because of the irregularity, the case analysis needed in code generation soon becomes too enormous for compiler writers, as noticed by Wulf in 1981 [23].

Fine instruction granularity, on the other hand, is the feature that makes the work of the assembly language programmer laborious and error-prone: because individual instructions are relatively primitive, more code lines have to be written. A representative example of fine granularity is that in some processor architectures loads and stores are the only possible forms of memory access (for instance, prior to launching an addition instruction both the addends must be explicitly fetched from memory). Fine instruction granularity (with hardwired control) seems to correlate with high execution speed: RISC instructions are also fine-grained.

1.2 Notion of flow-sensitivity

Macro expansion [9, 18, 7, 4, 5] is a simple modularity-promoting mechanism traditionally used at the assembly language level [20]. However, conventional assembly language macros cannot be freely used hierarchically. For instance, if some macro M_1 internally uses a certain register R as a temporary data storage, the programmer has to take care when calling M_1: the macro expander is not able to issue a warning that the value possibly already stored in R will be lost during the execution of M_1 [13, pp. 15–17]. Additionally, if M_1, in turn, calls another macro, say M_2, then the original caller of M_1 must take into consideration even similar restrictions that perhaps concern M_2. In general, such implicit restrictions effectively prevent the introduction of macro hierarchy.

To support unrestricted macro hierarchy, the macro expander must not be a simple text string substitution engine but a more sophisticated tool aware of the control and data flow of the underlying program. Optimizing compilers [1] perform *program flow analysis* [11]; we apply it even to macro expansion. The usefulness of *flow-sensitivity* becomes apparent with the following C code fragment:

```
while (x < 0) {                        /* C code */
  y = x + y;
  w = liboper(z);
  x = z + w;
}
```

Let us assume that the C compiler we use can hold all the variables `x`, `y`, `z`, and `w` in registers. Furthermore, we assume that the compiler recognizes the `liboper` library function and is able to treat it as a macro, that is, to replace its call with inline assembly code [22, Ch. 13]. Finally, we suppose that the compiler contains a non-trivial "precompiler" that allocates the registers reserved for the register variables and expands such library macros as `liboper`. Now, flow-sensitivity would allow the precompiler to observe that when this `liboper` macro call is expanded, the register allocated for `x` is free to be used as a temporary storage but the one allocated for `y` is not (`x` is rewritten by the last statement of the loop body, but `y` has to retain its value throughout the last two loop body statements).

Accordingly, the essential advantage of flow-sensitivity is that the expander knows which registers are free at the point of each macro call. A register is *free* if it is guaranteed *not* to contain data that should *not* be overwritten, i.e. data earlier written into it and later to be read from it. Then, the suggested division of labor is as follows: the human programmer (e.g. a compiler writer) only chooses the optimum register class for each macro temporary; the macro expander then tries to find a free member of the chosen class. Thus, the programmer is released from a great deal of tedious bookkeeping.

If a macro expander could reach adequate flow-sensitivity, we could at least in principle construct hierarchical libraries with conditional macro definitions as building blocks. Such libraries would, in turn, make the assembly code more easily readable, maintainable, and reusable. Indeed, in later sections of this report we are able to present concrete examples of multilevel macro hierarchy.

Our most severe problem lies in monitoring all the versatile equipment that processors use to implement the control and data flow. In this report, we have had to adopt strong constraints on this equipment. Most importantly, we have excluded the possibility of indirect addressing. Note, however, that often only memory locations, and not CPU registers, can be addressed indirectly. (In the future, we aim at programmer-assisted mechanisms for *alias analysis* [1, Sec. 10.8].)

1.3 Overview of the proposed tool

We propose the flow-sensitive macro expander as a tool for both assembly language programmers and compiler writers—in other words, both as a stand-alone macro assembler and as the final part of a compiler back-end.

Flow-sensitive macro expansion, as well as macro expansion in general, requires two main components: the macro expander proper and a rule base consisting of macro definitions. Because the tool must be taught the target instruction set, we deem it appropriate to divide the rule base explicitly into *system macros* and *application macros*, as shown in Figure 1. The machine instructions of the target processor are visible to the application macro programmer only through the system macros, which constitute the basis of the macro hierarchy—they are actually not genuine macros but mere placeholders for individual machine instructions. Application macro definitions can be hierarchical: in the definition of an application macro A, you may call any macro B. The task of the macro expander is to convert each application macro call in the input program into a sequence of system macro calls.

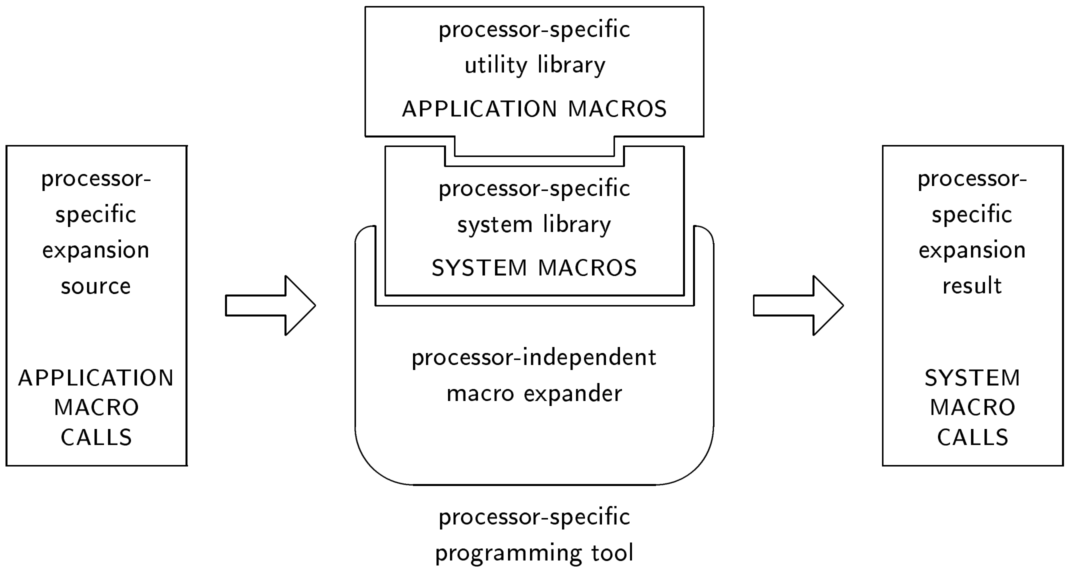

Figure 1: Operation of the proposed programming tool.

Although the macro expander proper is intended to be easily retargetable, we do not expect that application programs would be portable across different target hardware. As implicitly expressed above, the purpose of the system macro set is *not* to create a standard machine-independent programmer-visible layer by hiding the machine-specific details. On the contrary, the system macros exploit these very details in a controlled but still transparent fashion. There should be a natural one-to-one mapping between the system macros and the target machine instructions. For accuracy, the system macro set should be specified—and perhaps even the core of the application macro library should be written—by a single system manager (and not by application macro programmers).

Macros of any kind typically do not have any predefined semantics, contrary to the built-in elements of programming languages. The power of the proposed approach is in the generality resulting from this lack. For a simplified example, suppose that the target processor instruction set contains an *ADD* instruction that reads two registers and writes one register. Suppose also that the programmer uses the *ADD* instruction (i.e. the corresponding system macro) inside the definition of some application macro *MAC*. When the macro expander, then, tries to expand some call of *MAC*, all that it has to do concerning the *ADD* instance is to select three appropriate registers (in particular, it must take care that the data possibly already present in the output register is not prematurely overwritten). Thus, it can ignore a great deal of information relevant to any high-level language compiler. Most importantly, it need not even know whether *ADD* actually performs an addition or something wholly different.

To sum up, the lack of predefined semantics offers the following advantages:

- The macro expander proper can be made a relatively small program. It is the rule base consisting of macro definitions that drives the expansion.
- With suitable macro definitions, the tool can be tailored to fully exploit the particulars of the chosen target processor architecture.
- Operations not needed in the chosen application need not be supported.

1.4 Implications on target hardware

The flow-sensitive macro expansion is aimed at tackling the case analysis problem in code generation. Obviously, this problem is most severe when the target architecture is an irregular one—when the register set is heterogeneous and the instruction set is idiosyncratic. Accordingly, among the presently marketed processors we spot the fixed-point DSP chips as our most promising targets (admittedly, we need an additional postprocessor program for *code compaction* [10]). In particular, even among DSPs we focus on processors with narrowest functionality, highest speed, and lowest price (such as [3]).

We realize that the flow-sensitive macro expansion approach can, even as its best, be feasible only for a strongly restricted class of processor architectures (see [2] for some challenging ones of the numerous intricacies typically found in DSP architectures). Therefore, this class must be given a detailed characterization. We believe that a prospective macro expander implementor should at first be content with an unrealistically narrow characterization and only after a successful prototype strive for step-by-step extensions. A significant reason for not attacking directly the commercial architectures is that even a narrow characterization may perhaps serve as a constraint on future processor design; in a somewhat similar fashion, the RISC processors are designed specifically to be programmed with optimizing high-level language compilers.

Adopting additional architectural constraints of the suggested kind might pay especially in *rapid prototyping* [6] of embedded computer systems. Moreover, our approach promotes customized *application-specific instruction set processors* [16] as possible implementations for microelectronic systems with a tight development schedule. For such a processor, the proposed macro expander could provide an almost ready-to-use code generation tool; with such an application, the lack of portability across different target hardware is often insignificant.

1.5 Outline of this report

In Section 2 we present a tutorial that gives you a first impression of the implemented flow-sensitive macro expander ReFlEx 1.0. The overly simple target processor architectures supported by ReFlEx 1.0 are characterized in Section 3. Section 4 describes general notational conventions of the macro definition language. Before the features of the macro definition language are dealt with in detail in Section 6, Section 5 describes an essential supplementary facility: a user-extensible interpreter for evaluating arithmetic and flow-sensitive expressions at expansion time.

Section 7 discusses the general principles of the internal operation of ReFlEx. Section 8 gives a detailed description of the input and output of the expander: the expansion source and result. Section 9 describes how the expander is run, and what kind of problems may consequently emerge. Finally, Section 10 contains some technical information. In particular, it specifies how you can obtain a copy of an up-to-date revision of ReFlEx and installation instructions for it. The program can be installed and used for any non-commercial purposes without charge.

Appendices A and B serve as references covering the syntax of the macro definitions and the expansion source, respectively. Appendix C contains a sample set of macro definitions that constitutes a simple code generator. (This last appendix also includes an exercise that can be used as a diagnostic self-test.)

2 Tutorial

ReFlEx reads the macro definitions from a *rule file.* In this section, we write—step by step—a simple rule file. Then, in the last subsection we are able to run some elementary code generation examples. To be able to run the examples yourself, you should have the ReFlEx program successfully installed; see Section 10 for instructions.

This tutorial ignores some of the more advanced features of the macro definition language. Moreover, concerning even the basic features, the presentation here is only introductory. Having read this section, you have most likely still to learn more to be able to write your own rule files. A complete description of the macro definition language is given in Sections 5 and 6.

2.1 Overall rule base structure

The rule file should be a plain ASCII file. It is to be divided into four logical parts (each of which typically extends over several lines):

```
HEADER {      ...      }
STORAGE {      ...      }
SYSTEM {      ...      }
UTILITY {      ...      }
```

Figure 2: Overall structure of the rule file.

The `HEADER` part contains auxiliary definitions. In the `STORAGE` part we specify the available run-time data storage, i.e. the CPU registers and the data memory. The system and application macros are described in the `SYSTEM` and `UTILITY` parts, respectively.

Below, Section 2.2, which deals with the `HEADER` part, may well be read only cursorily. It is included only because it contains some definitions without which our rule file would be technically incomplete, even if still an intuitive one.

2.2 Expansion-time expression interpreter

In the `HEADER` part of the rule file, we may define new expressions for the built-in expansion-time expression interpreter. For instance, here is the obvious recursive definition for the factorial function:

```
Fact(a) = _If(_Lt(a,2), 1, _Mul(a,Fact(_Add(a,-1))));
```

The expressions recognized by the interpreter are called *forms.* The forms `_If`, `_Lt`, `_Mul`, and `_Add` are *primitive*, that is, predefined. (As a rule, if a primitive form has an alphanumeric name, then that name begins with an underscore '`_`'). Not surprisingly, `_Lt`, `_Mul`, and `_Add` are "less-than", multiplication, and addition, respectively. As for `_If`, if its first argument is nonzero, it returns the value of the second argument; otherwise, it returns the value of the third argument. This special primitive form supports lazy evaluation, because only the needed arguments are evaluated.

The forms defined by the user are called *compound* ones. A compound form is either *rigid* or *flexible.* A rigid form always expects a fixed number of arguments, whereas a flexible form may take any number of arguments. Like all the forms dealt with in this section, `Fact` is rigid.

Then, Figure 3 introduces some more compound forms:

`Not`	Is the argument zero?
`Gte`	Is the first argument greater than or equal to the second one?
`And2`	Are the two arguments both nonzero?
`Or2`	Is at least one of the two arguments nonzero?
`Eq`	Are the two arguments equal?
`Int`	Is the argument an integer?
`Zero`	Is the argument an integer equal to zero?

```
Not(a) = _Nand(a,a);
Gte(a,b) = Not(_Lt(a,b));
And2(a,b) = _If(a, _If(b,1,0), 0);
Or2(a,b) = _If(a, 1, _If(b,1,0));
Eq(a,b) = And2(Not(_Lt(a,b)), Not(_Lt(b,a)));
Int(a) = And2(Not(?(a)), Not(&(a)));
Zero(a) = And2(Int(a), Eq(a,0));
```

Figure 3: Definitions of rigid compound forms.

There is an essential difference between the two forms `Not` and `Zero`: if the argument turns out not to represent an integer, `Not` always returns a nonzero value, but `Zero` returns zero. The explanation is that the virtual form arguments (which are employed in the rule file) may, in addition to *integers*, designate *storage cells* or *instruction labels*, which are always interpreted as zeros if they appear where an integer is expected. The '`?`' and '`&`' primitive forms used in the `Int` definition check whether their arguments are storage cells or instruction labels, respectively.

2.3 Available data storage

Our imaginary target processor architecture is utterly simplified. As shown in Figure 4, there are 1024 memory locations `M`[0]–`M`[1023], four auxiliary registers `R`[0]–`R`[3], and a single accumulator `A` (due to the singularity, '`A`' can be used as a shorthand for '`A[0]`'). Thus, there are three distinct *storage classes*, whose members are collectively called *storage cells*, or simply *cells*. ReFlEx assumes that the members of each single storage class can be used fully interchangeably.

```
STORAGE {
  M[1024];
  R[4];
  A[1];
}
```

Figure 4: Declaration of data storage cells.

Note that ReFlEx itself need not to know the cell lengths (which may well be different for different classes) in terms of bit positions. On the other hand, any professional ReFlEx user should absolutely be familiar with this information. (In accordance with the established compiler terminology, we say that Figure 4 contains a "declaration". In general, a declaration introduces the name of an entity without specifying the internal structure of the entity.)

2.4 Machine instruction set

Next, in Figure 5 we declare the system macros, which represent the machine instructions of the target. What we define is actually only the interface seen by application macro programmers. ReFlEx cannot convert system macro calls into real machine instructions; what it is able to do, is to convert application macro calls into system macro call sequences. We claim that this latter task is the interesting one, while the former task should be a routine matter and therefore relegated to a ReFlEx-compatible assembler.

So there are, in all, ten system macros, among which there are one unconditional branch (`JUMP`) and three conditional branches (`BRANCH`). Most of the system macros are provided with a *test* (`TEST`) that constrains their use. ReFlEx does not know their full semantics, which must, of course, be known

```
SYSTEM {
  set(c > r) {
    TEST And2(And2(?R(r), Int(c)),
              And2(Gte(c,-1024), _Lt(c,1024)));
  }
  load(m > r) { TEST And2(?R(r), ?M(m)); }
  store(r > m) { TEST And2(?R(r), ?M(m)); }
  move(s > d) {
    TEST And2(Or2(?A(s), ?R(s)),
              Or2(?A(d), ?R(d)));
  }
  add(a,r > a) { TEST And2(?A(a), ?R(r)); }
  sub(a,r > a) { TEST And2(?A(a), ?R(r)); }
  JUMP goto() [l] { }
  BRANCH eq(a) [l] { TEST ?A(a); }
  BRANCH gt(a) [l] { TEST ?A(a); }
  BRANCH lt(a) [l] { TEST ?A(a); }
}
```

Figure 5: Declaration of system macros.

to the application macro programmer. With the information seen in Figure 5, even the following brief description should be fairly comprehensive for a prospective application macro programmer:

- `set` "reads" a signed 11-bit integer and writes it into one of the auxiliary registers (form `?R` checks whether its arguments belong to storage class `R`). Thus, `set` is for immediate addressing, for the value of the integer is fixed at expansion time.
- `load` copies the contents of a memory location into an auxiliary register, and `store` performs the opposite data transfer.
- `move` can move data between two storage cells, provided that each one of the cells is either the accumulator or an auxiliary register.
- `add` adds the contents of one of the auxiliary registers into the accumulator. Thus, it reads and writes the accumulator, and additionally reads an auxiliary register. Similarly, `sub` performs a corresponding subtraction.
- `goto` jumps to the specified instruction label.
- `eq`, `gt`, and `lt` branch to the specified label if the contents of the accumulator are, respectively, equal to zero, positive, or negative. Thus

they all read the accumulator.

From this rather restricted system macro set, we may infer that the actual target processor instruction set is similarly restricted. Note in particular that the accumulator cannot be loaded directly from memory.

2.5 Higher-level macros

Now we are ready to define application macros, without which the rule file would not be of any use. Our first application macro, `my_null` shown in Figure 6, is most simple: it does not do anything. Still, it is a useful macro, as you will see later. (Actually, the essential feature of `my_null` is the requirement that the input and output arguments must be the same cell.)

```
my_null(x > x) {
  null: { }
}
```

Figure 6: Definition of the `my_null` application macro.

The `my_move` macro in Figure 7, then, implements a general data transfer not subject to any storage class restrictions, contrary to the system macros declared above. The macro definition consists of seven alternative *versions*, each with a distinct name. The versions are listed in the order of decreasing priority, i.e. in the order in which ReFlEx should try to apply them to each `my_move` call.

Note that the `as_set` version, for instance, can be accepted only if the source of the data transfer is an 11-bit signed integer and the destination is an auxiliary register, because the test associated with the `set` macro in Figure 5 rejects calls of other kinds. In Figure 7, in contrast, all the tests are associated with some version of the macro, not directly with the macro itself.

Let us look closely at each one of the `my_move` versions:

- `same` guarantees that the expansion result is an empty code sequence when the source and the destination are the same cell. The empty `my_null` call serves two purposes: it verifies that parameters `s` and `d` do represent the same cell, and it "writes" the output parameter `d` (see Section 6.8 for the constraint concerned here).

- `as_set`, `as_load`, `as_store`, and `as_move` may only be converted into the respective corresponding system macros.

```
my_move(s > d) {
  same: { my_null(s > d); }
  as_set: { set(s > d); }
  INSIST Not(And2(Int(s), ?R(d)));
  as_load: { load(s > d); }
  as_store: { store(s > d); }
  as_move: { move(s > d); }
  clear_acc: TEST Zero(s); USE R[r]; {
    INIT( > r); move(r > d); sub(d,r > d);
  }
  temp_is_needed: USE R[r]; {
    my_move(s > r); my_move(r > d);
  }
}
```

Figure 7: Definition of the `my_move` application macro.

- `clear_acc` is used for resetting the accumulator without having to overwrite any auxiliary register: the auxiliary register represented by the temporary `r` is only read and not written. The `INIT` pseudomacro is specifically aimed at situations of this kind. (`INIT` is the only pseudomacro of the language.)
- `temp_is_needed` requires a free auxiliary register. This register is used as an intermediate storage when, for instance, the contents of a memory location are to be transferred into the accumulator. The version is recursive, but the recursion depth can be seen to be at most one.

In Figure 7, the `INSIST` clause detects such integers that were too large for the `set` system macro. Removal of this clause would allow non-terminating recursion, instead of the controlled failure naturally preferred by the macro writer.

Like `my_move`, our next example is also a storage-class-independent generalization. The `my_zero` macro, which is shown in Figure 8, is a branch that is taken if the value represented by its argument is equal to zero. (The expansion result of a sample `my_zero` call will be shown in Figure 10 in Section 2.6.)

Note that the empty `next` version of `my_zero`, which employs reserved word `NEXT`, matches the case in which the branch target, i.e. the instruction labeled by `l`, immediately follows the `my_zero` call.

Our fourth and final application macro, `my_mswap` shown in Figure 9, interchanges the contents of two memory locations. This is our first application

```
BRANCH my_zero(x) [l] {
  next: TEST &(l,NEXT); { }
  const_zero: TEST Zero(x); { JUMP goto() [l]; }
  const: TEST Int(x); { }
  acc: TEST ?A(x); { BRANCH eq(x) [l]; }
  default: USE A[a]; { my_move(x > a); BRANCH eq(a) [l]; }
}
```

Figure 8: Definition of the `my_zero` application macro.

macro with which a test (i.e. one that verifies the storage class) is directly associated. (Section 2.6 will also present the expansion result of a sample `my_mswap` call, in Figure 11.)

```
my_mswap(m,n > n,m) {
  TEST ?M(m,n);
  two_aux_free: TEST Gte(#R(),2); USE R[r]; {
    my_move(m > r); my_move(n > m); my_move(r > n);
  }
  acc_and_aux_free: USE A[a]; {
    my_move(m > a); my_move(n > m); my_move(a > n);
  }
}
```

Figure 9: Definition of the `my_mswap` application macro.

To perform the swap, `my_mswap` requires two cells of free storage, because direct transfers between memory locations are not supported by the system macro set. At least one of these two cells must be an auxiliary register, while the other one may alternatively be the accumulator. Accordingly, form `#R`, seen in the `my_mswap` definition, returns the number of free cells in storage class `R`. (The test associated with the `two_aux_free` version is necessary, because in the version code only one auxiliary register, i.e. `r`, is written, but the middle one of the lower-level `my_move` calls certainly needs another one. Without the test, `two_aux_free` might be selected even in such a case that the `acc_and_aux_free` version might be the only fully expandable one of these two. This is because ReFlEx cannot backtrack from choices that only at some later stage prove to be unsuccessful.)

2.6 Generating code for macro calls

Finally, in this last subsection of Section 2 we put our macro definitions into use and produce some code for our target processor. We must start with a single macro call: ReFlEx 1.0 expects that the expansion source can be represented as one top-level macro call.

Suppose that ReFlEx has been installed on our workstation and our rule file is called `simple.m`. We can start ReFlEx by typing the following command (the combination of options `-t` and `-f` provides us with some interesting optional output):

```
reflex -t -f simple.m
```

If ReFlEx starts successfully, we may then type the following expansion source as a response to the ReFlEx prompt, i.e. '>':

```
> BRANCH my_zero(M[5]) [L8]; {A,R[2]}
```

This means that we want to branch to label `L8` if the contents of memory location `M[5]` are zero, and we additionally specify that accumulator `A` and auxiliary register `R[2]` are free at the macro call (an explicit specification like this is needed and allowed only in the expansion source). The expansion result is shown in Figure 10. The intermediate output at the top reveals the expansion-time tree structure, which is flattened in the final output at the bottom. Each macro call is provided with a set containing the cells that are free at it (normally, each cell written by the macro call is included in the set).

```
BRANCH my_zero(M[5]) [L8] {A,R[2]} {
  my_move(M[5] > A) {A,R[2]} {
    my_move(M[5] > R[2]) {A,R[2]} {
      load(M[5] > R[2]); {A,R[2]}
    }
    my_move(R[2] > A) {A,R[2]} {
      move(R[2] > A); {A,R[2]}
    }
  }
  BRANCH eq(A) [L8]; {A,R[2]}
}

BRANCH my_zero(M[5]) [L8] {A,R[2]} {
  load(M[5] > R[2]); {A,R[2]}
  move(R[2] > A); {A,R[2]}
  BRANCH eq(A) [L8]; {A,R[2]}
}
```

Figure 10: Expansion result of a `my_zero` macro call.

Second, we want to expand the following call of the `my_mswap` macro:

```
> my_mswap(M[8],M[6] > M[6],M[8]); {R[1],R[3]}
```

Observe that in this case we have marked two auxiliary registers as free ones. The expansion result is now shown in Figure 11. (You might find it interesting to try to figure out by yourself what would happen if either one of the free auxiliary registers were replaced with the accumulator. As a hint, we give the fact that the new expansion result would consist of six system macro calls.)

```
my_mswap(M[8],M[6] > M[6],M[8]) {M[6],M[8],R[1],R[3]} {
  my_move(M[8] > R[3]) {M[8],R[1],R[3]} {
    load(M[8] > R[3]); {M[8],R[1],R[3]}
  }
  my_move(M[6] > M[8]) {M[6],M[8],R[1]} {
    my_move(M[6] > R[1]) {M[6],M[8],R[1]} {
      load(M[6] > R[1]); {M[6],M[8],R[1]}
    }
    my_move(R[1] > M[8]) {M[6],M[8],R[1]} {
      store(R[1] > M[8]); {M[6],M[8],R[1]}
    }
  }
  my_move(R[3] > M[6]) {M[6],R[1],R[3]} {
    store(R[3] > M[6]); {M[6],R[1],R[3]}
  }
}

my_mswap(M[8],M[6] > M[6],M[8]) {M[6],M[8],R[1],R[3]} {
  load(M[8] > R[3]); {M[8],R[1],R[3]}
  load(M[6] > R[1]); {M[6],M[8],R[1]}
  store(R[1] > M[8]); {M[6],M[8],R[1]}
  store(R[3] > M[6]); {M[6],R[1],R[3]}
}
```

Figure 11: Expansion result of a `my_mswap` macro call.

We would like to stress the crucial point of the expansion result shown in Figure 11. When ReFlEx tries to expand the `my_move(M[6] > M[8])` call at the first level below the initial `my_mswap` call, it recognizes that the originally free auxiliary register `R[3]` is not free any more. Thus, `R[1]` is the only possibility for a temporary storage at the next-lower level. This deduction rests on the flow-sensitivity of ReFlEx.

3 Target architecture model

We claim that even the rudimentary ReFlEx 1.0 clearly favors some important common features of low-cost embedded real-time processors such as the DSPs. Typically, a DSP is a sequential uniprocessor, i.e. a processor with a single instruction stream and a single data stream. Moreover, the DSPs are prominent ReFlEx targets especially because of their heterogeneous register set.

Still, the possible targets must be a great deal simpler than any real-world processor—let alone any of DSPs, which indeed pose exceptionally difficult problems for code generators [2]. The most notable restriction concerns indirect addressing modes, which are completely unsupported by ReFlEx (we hope that in forthcoming releases even this major deficiency can be adequately fixed). However, one should remember that with many processor architectures indirect addressing applies only to memory locations, and not to CPU registers.

3.1 Code storage

We assume that there is only a single instruction stream, which is determined by the contents of the program memory. Furthermore, the program memory has to be at least logically both unsegmented and separate from the data memory. Accordingly, the program counter should hold full program memory addresses.

We adopt the non-restrictive convention that a unique *instruction label* is associated with each program memory location.

3.2 Data storage

We assume that the data storage is divided into distinct *storage classes*, each of which consists of a finite number of distinct *storage cells*; the largest storage class is typically constituted by the data memory. ReFlEx expects that all the cells in a single storage class are treated fully interchangeably by the machine instruction set: if an occurrence of a certain instruction refers to some cell, other occurrences of that instruction may refer to any cell in the same storage class. The size, i.e. the number of bit positions, of the cells in a particular storage class is irrelevant to ReFlEx. Accordingly, different storage classes may have different cell sizes, but ReFlEx remains unaware of this difference.

ReFlEx is capable of automatic intraclass—but not interclass—cell allocation. The division of labour is as follows: the human programmer chooses the optimum storage class for each macro temporary, and ReFlEx selects an appropriate cell from the chosen class. In particular, ReFlEx usually has to verify that the selected cell is free at the point of the macro call.

ReFlEx is not capable of autonomously spilling a non-free register into memory, even if its contents will be needed only in the distant future and a register of that class is immediately needed for other purposes. Because such a spill might be expensive when the target architecture is irregular, leaving the spill decision under the explicit control of the programmer seems to be sound.

As implicitly expressed above, in regard to real-world processor architectures our concept of data storage has two major drawbacks:

- Any two storage cells must be physically distinct, that is, they cannot overlap.
- Neither hardware nor software stack is supported.

3.3 Comparison with real machine instruction sets

ReFlEx 1.0 imposes essential restrictions on the allowed machine instructions of the target architecture. Most of all, the following features are *not* supported:

- Indirect addressing.
- Indirect jumps.
- Function calls.
- Mode control.
- VLIW-type fine-grained parallelism.

Of the diverse addressing modes found in special-purpose processors, only direct and immediate addressing are supported. Thus, instructions may only explicitly refer to certain storage cells or instruction labels, or explicitly specify certain integer values. If *indirect addressing* of storage cells were allowed, it would pose a problem in the data flow analysis, because the macro expander would then have to keep track of the possible values of the address registers. Similarly, enabling *indirect jumps* would complicate the control flow analysis.

In high-level programming languages such as C, *functions* serve two purposes: they provide the source code with the basic structure, and their use reduces the amount of memory consumed by the executable code. Even if at least in principle the ReFlEx macros meet the first one of these two goals just as well as C functions, their careless use may indeed waste memory space.

A representative example of *mode control* [2] would be a case in which the saturation of addition operations is governed by a status register, whose contents the programmer may modify only with dedicated instructions. Because extraneous instructions are seldom cheap, the programmer must carefully keep track of the current value of the status register to avoid redundant modification instructions. This task could clearly be automated by applying program flow analysis, but ReFlEx 1.0 provides no explicit support for mode control. (For any register, of course, such avoidance of redundant loads by register value tracking would be useful—ReFlEx 1.0 can track which registers are in use, but not their contents.)

Very long instruction word (VLIW) processors [8] contain multiple functional units that may all be independently controlled by the different bit fields of a single instruction word. Such *fine-grained parallelism* supports fully static compilation-time instruction scheduling, which is in many cases more attractive than dynamic execution-time instruction scheduling (performed by *superscalar* processors, for instance). In this fashion, individual DSP instructions may synchronously drive as many as six functional units, which include an arithmetic-logic unit and a hardware multiplier. Programmer-controlled pipelining of this kind is perhaps the most characteristic feature of the DSP architectures.

3.4 Our machine instruction model

ReFlEx 1.0 assumes that the machine instructions are, in a sense, independent of each other. More precisely, each instruction instance should explicitly and unambiguously specify

- the set of storage cells it reads;
- the set of storage cells it writes; and
- the set of instruction labels it targets, that is, the labels that mark the program memory locations to one of which the control is transferred after its execution.

These sets may be empty (in practice, even an instruction implementing an empty non-terminating loop may be handy when an external interrupt is waited for). As a more general example, the hypothetical instruction (or rather, system macro call) instance

```
BRANCH probe(M[2],M[3] > R[0],M[7]) [L5,L8];
```

reads cells M[2] and M[3], writes cells R[0] and M[7], and transfers the control either to one of the locations labeled L5 and L8 or to the location immediately following the instruction instance itself (or loops forever).

In effect, the above restriction prevents indirect addressing of any kind. In addition to direct addressing, only immediate addressing is enabled: instruction instances may also "read" integer literals. This same restriction holds for application macro calls, which are identical with the system macro calls with respect to the syntax and semantics of the caller-visible interface.

Our meanings for the words 'read' and 'write' are perhaps not as simple as one might hope. For example, how should an operation described by the C language statement

```
if (x < 0) y = z;                  /* C code */
```

be modeled as (a call of) a ReFlEx system macro named, say, cond_copy? The answer is

```
cond_copy(x,y,z > y);
```

and the explanation is as follows:

- Clearly, this cond_copy instance reads x.
- It may read z. Thus z should not be inadvertently corrupted prior to the operation, which is indicated by including z among the input arguments.
- It may write y, but because the writing may not take place, even y should not be corrupted prior to the operation.

On the other hand, the longer C fragment

```
if (x < 0) y = z; else y = 1;     /* C code */
```

can be modeled simply as

```
select_copy(x,z > y);
```

This distinction may at first seem confusing, but fortunately it concerns only system macro implementors. They have to create a sound interface between ReFlEx and some real target architecture, which is a task of little significance in the case of the purely experimental ReFlEx 1.0. The application macro definitions, in contrast, can be verified in this respect by ReFlEx itself. (You may test this final claim by adding an application macro behaving like `cond_copy` to the rule file presented in Section 2.)

4 On syntactical conventions

Creating a ReFlEx rule file is easier if you are aware of some general syntactical conventions, many of which resemble the ones enforced by the C programming language [12]. However, a notable difference from C is that names of such entities that will be introduced only later in the rule file may freely be used: no advance declarations whatsoever are needed. For instance, the storage classes may be referred to in the compound form definitions in the `HEADER` part, which must precede the `STORAGE` part in the rule file.

The following description applies both to the rule file and to the expansion source.

4.1 Characters

Only the following characters are recognized:

- Alphanumeric characters: the underscore '`_`', the digits '`0`'–'`9`', the uppercase letters '`A`'–'`Z`', and the lowercase letters '`a`'–'`z`'.
- Special characters: `! # ( ) % & + , - . / : ; < = > ? [ ] { }`
- White space characters: *space*, *horizontal tab*, and *newline*.

The two-character sequence '`//`' starts a comment, which ends with the end of the current line. C-style multiline comments (`/*` ... `*/`) are not supported. Here is an example of a comment:

```
my_macro(a > b);     // This is a comment!
```

Other characters than the ones listed above may occur only inside a comment. White space characters and comments are collectively called *white space.*

4.2 Tokens

Each occurrence of such a character that is not white space belongs to exactly one *token*. Actually, the only significance of white space is that it is often (but not always) necessary for separating two consecutive tokens. This function of white space becomes apparent with the following definition.

Each token must be exactly one of the following:

- A *reserved word* (see below).
- An *identifier*, that is, a maximal sequence of alphanumeric characters that does not begin with a digit and that is different from the reserved words.
- A *number*, that is, a maximal sequence of digits that is not immediately preceded by any other alphanumeric character.
- An occurrence of a special character.

The distinction between upper and lower case is significant. All the reserved words consist of uppercase letters only (contrary to the C convention):

```
ASSERT            JUMP              STORAGE
BRANCH            NEXT              SYSTEM
HEADER            SAFE              TEST
INIT              SEED              THIS
INPUT             SLEEP             USE
INSIST            STATE             UTILITY
```

Here are three examples of valid identifiers:

```
assert            A_012             _____
```

And here are three examples of valid numbers:

```
0                 007               12345
```

4.3 Structuring mechanisms

Concerning physical structuring, such file inclusion as performed by the C preprocessor is not supported by ReFlEx. Furthermore, ReFlEx does not allow multiple translation units (as C does). Therefore, the rule file must be a single physical file at the time when it is read by the macro expander.

Concerning logical structuring, the ReFlEx syntax employs heavily certain special character pairs as list delimiters in order to create logical hierarchy in the rule file. Not surprisingly, these list delimiter pairs are `( )`, `[ ]`, and `{ }`. Lists generally use comma as an element separator; as an exception, certain lists delimited by curly braces employ semicolon as an element terminator. Here are simple examples of lists:

```
( a, b, c )
[ a, b, c ]
{ a, b, c }
{ a; b; c; }
```

Nevertheless, similarly to the compound statements of C, the right curly brace is never followed by a semicolon:

```
{ a; b; { c; d; } }
```

Finally, parenthesizing arithmetic expressions is not a problem, since ReFlEx provides no infix operators. Parentheses should be used only for surrounding argument lists of macro and form calls. For instance, a parenthesized integer literal is no more a syntactically valid integer designator:

```
my_macro(5 > b);   // my_macro((5) > b) would be invalid!
```

5 Constant expressions

ReFlEx includes a built-in expansion-time expression interpreter. Using a simple notation, you may define conventional shorthands for complicated arithmetic expressions. Furthermore, the interpreter offers you certain primitives for extracting information about the data and control flow of the program under expansion.

Why do we need such an interpreter? The reason is that ReFlEx lacks two features found in the C language: the built-in preprocessor, and even more importantly, the predefined set of (relatively) machine-independent run-time operators such as '+' for addition and '`*`' for multiplication. (Most macro assemblers provide some means for achieving goals similar to the ones of our interpreter.)

Let us have an example. Suppose that someone asks us to write a program that calculates the value of the formula a^3bxy for any x and y, given that the values of a and b are 29 and 41, respectively. First, we produce a straightforward C implementation:

```
z = 29 * 29 * 29 * 41 * x * y;            /*  C code  */
```

Even if the above statement looks clumsy, it is an efficient one: because the predefined operator '*' is a part of the C definition, the C compiler can be expected to be capable of folding the four constants into one [1, Sec. 10.2] already at compilation time. Furthermore, we can make our implementation look more elegant by utilizing the C preprocessor:

```
#define A_VAL 29
#define B_VAL 41
#define CUBE(a) ((a) * (a) * (a))
#define COEF (CUBE(A_VAL) * B_VAL)

z = COEF * x * y;                         /*  C code  */
```

Thus, in the case of C, having predefined operators enables compile-time constant folding, while the preprocessor makes the code more easily readable and maintainable. With ReFlEx, in contrast, machine-independent run-time operators are ruled out for ultimate efficiency, and therefore the folding of constants must be performed by the expansion-time expression interpreter. It is also our intention that the interpreter should be able to handle most preprocessing-type tasks encountered (the possibly remaining ones of these tasks have to be relegated to a fully independent text preprocessor).

As an instantaneous preview into the present Section 5, we provide here the corresponding ReFlEx definitions for the above C preprocessor definitions:

```
A_val() = 29;
B_val() = 41;
Cube(a) = _Mul(a,_Mul(a,a));
Coef() = _Mul(Cube(A_val()),B_val());
```

However, we cannot construct a similar ReFlEx definition that would match the C preprocessor definition that next comes to one's mind:

```
#define EXPR(x,y) (COEF * (x) * (y))      /*  C code  */
```

The reason is that now, according to the initial assignment given to us, the multiplication operators represent calculations that cannot be performed until execution time. For such calculations, we have to put into use the whole ReFlEx macro definition facility to be described in Section 6.

5.1 Taxonomy of integer, cell, and label designators

An expression that may produce an integer is called an *integer designator*. Integer designators are divided into *integer literals*, *form calls*, and *integer references*; see Sections 5.2, 5.3, and 5.4, respectively.

Correspondingly, there are *cell designators* and *label designators*, which may specify (storage) cells and (instruction) labels, respectively. Cell designators are divided into *cell literals* and *cell references*, and label designators are similarly divided into *label literals* and *label references*; no form call can act as a cell or label designator. Cell and label literals can only be used in the expansion source (see Section 8.1). For cell and label references, see Section 5.4.

Whether a given expression is an integer, cell, or label designator, is actually a property of each particular instance of the expression: if two syntactically identical expression instances appear in different contexts, it may be that only one of them is a valid integer designator. Furthermore, as ReFlEx classifies the designators statically already before the macro expansion, a single expression instance may be, say, both an integer designator and a label designator. Whether such an ambiguous designator really represents an object of the kind that is anticipated by the dynamic expansion environment, cannot generally be determined until expansion time.

5.2 Integer literals

An *integer literal* is any number (see Section 4.2) optionally preceded by arbitrarily many instances of special characters '+' and '-'. The semantics of integer literals is obvious. Here are five examples of valid integer literals:

```
7          + 7          - 7          + - 7          - - 7
```

5.3 Form calls

Form calls can be used in the rule file but not in the expansion source. Each form call is evaluated by the expansion-time expression interpreter according to the definition of the particular form. There is a small set of predefined *primitive forms*, which consists of nine *autonomous forms* and six *context-sensitive forms*. Additionally, the user may define new *compound forms* in the `HEADER` part of the rule file.

We say that a form is *rigid* if each instance of the form call requires the same fixed number of arguments, and *flexible* if its calls accept any number of arguments. Each form is either rigid or flexible.

Virtual form arguments are those shown in the form calls in the rule file. A virtual form argument can be an integer designator, a cell reference, or a label reference (thus, cell and label literals are ruled out). Furthermore, a macro temporary (which is always either a cell reference or a label reference) can never be used as a virtual form argument.

Actual form arguments are those integers, cells, and labels that replace the virtual arguments at expansion time; each actual argument is the expansion-time evaluation result of the corresponding virtual argument. (When we use the plain ‘argument’, we usually mean an actual argument; the exception proving this rule is that we simply say ‘an argument is evaluated’, with the obvious interpretation.)

Next, in Section 5.4, we complete the characterization of the possible virtual form arguments. In the rest of the present Section 5, we investigate all the different form types.

5.4 Integer, cell, and label references

Integer references are identifiers or reserved words that possibly represent integers. There are the following types of integer references:

- Each input parameter of a macro definition.
- The `STATE` pocket. (*Pockets* can only be employed in flexible compound form definitions; see Section 5.8 for the details.)
- The `INPUT` pseudopocket.
- Each auxiliary pocket.

Cell references are identifiers or reserved words that possibly represent cells. There are the following types of cell references:

- Each data parameter of a macro definition.
- Each data temporary of a macro definition.
- The `INPUT` pseudopocket.

Label references are identifiers or reserved words that possibly represent labels. There are the following types of label references:

- Each label parameter of a macro definition.
- Each label temporary of a macro definition.
- The `INPUT` pseudopocket.
- Reserved words `THIS` and `NEXT`. (These can only be employed in macro definitions; see Section 5.6 for the details.)

As specified above, reserved word `INPUT`, for instance, is in an appropriate context valid as an integer reference, as a cell reference, *and* as a label reference. Such conflicts cannot be resolved until expansion time. Obviously, there are three possible cases of reference mismatch: if a supposed reference to an integer actually turns out to represent a non-integer (i.e. a cell or a label), ReFlEx simply takes zero as the value produced by the reference; for the other two cases, see Section 5.6.

5.5 Autonomous primitive forms

All autonomous primitive forms are rigid. Their virtual arguments must be integer designators. If the virtual argument still, at expansion time, turns out not to represent an integer, then ReFlEx uses zero as the actual argument.

The autonomous primitive forms can be divided into three categories:

- `_Abort(x,y,z)`, which raises an exception.
- `_If(x,y,z)`, which implements lazy argument evaluation.
- *Arithmetic* primitive forms (see below).

Instead of returning, each call of `_Abort` aborts the expansion immediately. No expansion result is produced, but the three arguments of the `_Abort` call are passed to the user as an explanation.

The `_If(x,y,z)` call is processed similarly to the `x?y:z` expression of the C language. That is, the following steps are taken:

1. The first argument is evaluated.
2. If the result is nonzero: the second argument is evaluated, and the resulting value is returned as the value of the call.
3. Otherwise: the third argument is evaluated, and the resulting value is returned as the value of the call.

Other autonomous primitive forms always evaluate all their arguments, but either the second or the third argument of each `_If` call always remains unevaluated. Lazy evaluation of this kind makes even recursive compound form definitions feasible.

There are, in all, seven arithmetic primitive forms, which are shown in Table 1. The rightmost column of the table consists of C language expressions. These expressions fix the semantics of the primitives. Still, the precise meaning of each primitive is determined only by the particular C (or rather, C++) compiler that is used for compiling the ReFlEx source code, including in particular the expressions in Table 1 (see also Section 10.1).

`_Add(x,y)`	addition	`x + y`
`_BNand(x,y)`	bitwise NAND	`~(x & y)`
`_BShl(x,y)`	bitwise shift	`(y > 0) ? (x << y) : (x >> -y)`
`_Div(x,y)`	division	`x / y`
`_Lt(x,y)`	less-than	`(x < y) ? 1 : 0`
`_Mul(x,y)`	multiplication	`x * y`
`_Nand(x,y)`	logical NAND	`(x && y) ? 0 : 1`

Table 1: Arithmetic primitive forms.

We also give brief verbal descriptions for the arithmetic primitive forms:

`_Add(x,y)` Addition of the two arguments.

`_BNand(x,y)` Bitwise NAND operation.

`_BShl(x,y)` The bitwise representation of `x` is shifted to the left by `y` positions if `y` is positive, and otherwise to the right by −`y` positions.

`_Div(x,y)` Division of `x` by `y`.

`_Lt(x,y)` If `x` is less than `y`, 1 is returned; otherwise, 0 is returned.

`_Mul(x,y)` Multiplication of the two arguments.

`_Nand(x,y)` Logical NAND operation that results in either 1 or 0.

5.6 Context-sensitive primitive forms

The virtual arguments of context-sensitive primitive forms must be cell or label references. However, if any virtual argument turns out to represent an integer, then the whole form call is taken to return zero. The same happens if an expected storage cell turns out to be an instruction label, or vice versa.

(Thus, one could say that “type checking” is here somewhat stricter than with autonomous primitives.)

Except for `#()`, all the context-sensitive primitives are flexible. There is an implicit conjunction between the arguments of any flexible context-sensitive primitive. For instance, the `?M(x)` form call checks whether `x` represents a cell that belongs to storage class `M`, and `?M(x,y,z)` checks whether all three of `x`, `y`, and `z` belong to `M`.

A form call inside a macro definition produces always the same result that would have been obtained if an identical form call had been evaluated as a macro-specific test of the current macro (see Section 6.10). Remember also that macro temporaries cannot be used as virtual form arguments.

There are six context-sensitive primitive forms, most of which may be further parametrized with a *storage class specifier* (such as ‘`M`’ of the `?M(x)` form call above). The use of the context-sensitive primitives is explained below; the individual form descriptions are followed by a concise summary.

Are the arguments cells (of a particular storage class)?

The `?(...)` form is flexible. It returns 1 if all the arguments are storage cells, and 0 otherwise.

Examples:

`?(x)` Does `x` represent a cell?

`?(x,y,z)` Do `x`, `y`, and `z` all represent cells?

`?()` Trivially, 1 is returned.

`?A(x)` Does `x` represent a cell that belongs to storage class `A`?

Are the arguments similar cells?

The `%(...)` form is flexible. It returns 1 if all the arguments are storage cells belonging to a common storage class, and 0 otherwise.

Examples:

`%(x,y,z)` Do `x`, `y`, and `z` represent cells that belong to some single common storage class?

`%A(x,y)` The same as `?A(x,y)`.

Are the argument cells the same?

The `=(...)` form is flexible. It returns 1 if all the arguments are identical cells, and 0 otherwise. (It cannot be used in *safety declarations*; see Section 6.9.)

Examples:

`=(x,y,z)`	Do `x`, `y`, and `z` all represent the same cell?
`=A(x,y)`	Do `x` and `y` represent a single common cell of storage class `A`?

Are the arguments free cells?

The `!(...)` form is flexible. It returns 1 if all the arguments are free storage cells, and 0 otherwise. (It cannot be used in safety declarations; see Section 6.9.) For the definition of a free cell, see Section 7.3.

Examples:

`!(x,y,z)`	Do `x`, `y`, and `z` all represent free cells?
`!A(x)`	Does `x` represent a free cell of storage class `A`?

How many free cells are there?

The `#()` form is rigid and expects no arguments. It returns the number of distinct free storage cells. (It cannot be used in safety declarations; see Section 6.9.)

Examples:

`#()`	How many free cells are there?
`#A()`	How many free cells of storage class `A` are there?

Are the argument labels the same?

The `&(...)` form is flexible. It returns 1 if all the arguments are equivalent instruction labels, and 0 otherwise. It does not accept a storage class specifier.

The reserved words `THIS` and `NEXT` are label references that can be used as virtual arguments of this form. `THIS` represents the label of the current macro call, whereas `NEXT` represents the label that immediately follows the current macro call. Thus, the code produced from the current macro call will be located between these two labels.

Label references can be fully resolved only at the final stage of the expansion. Consequently, ReFlEx must be conservative in its decisions: even if such a call as `&(x,y)` returns 0, it may still be so that `x` and `y` actually represent equivalent labels. However, most importantly, if the call returns 1, then the arguments *are* guaranteed to be equivalent.

Examples:

`&(x,y,z)` Do `x`, `y`, and `z` all represent equivalent labels?

`&(x)` Does `x` represent a label?

`&(x,THIS)` Does `x` represent the label of the first instruction of the code segment resulting from the current macro call?

Summary of context-sensitive primitive forms

In Table 2, we summarize the use of the context-sensitive primitive forms. From the table, you see that all these forms except `&(...)` accept a storage class specifier. Other forms than `#()` are flexible. Of the flexible forms, `&(...)` expects labels for arguments, whereas the others expect cells. The rigid `#()` takes no arguments and returns a non-negative integer. Other forms than `#()` return either zero or one. Finally, forms `=(...)`, `!(...)`, and `#()` cannot be employed in safety declarations.

Cells?	`?(...)`	`?`*class*`(...)`	*cells*	0 or 1	`SAFE`
Similar cells?	`%(...)`	`%`*class*`(...)`	*cells*	0 or 1	`SAFE`
Same cell?	`=(...)`	`=`*class*`(...)`	*cells*	0 or 1	—
Free cells?	`!(...)`	`!`*class*`(...)`	*cells*	0 or 1	—
How many free cells?	`#()`	`#`*class*`()`	—	n	—
Equivalent labels?	`&(...)`	—	*labels*	0 or 1	`SAFE`

Table 2: Context-sensitive primitive forms.

5.7 Rigid compound forms

Two most simple rigid compound form definitions read as follows:

```
Dozen() = 12;
Gross() = _Mul(Dozen(),Dozen());
```

More complicated definitions are of course possible:

```
Sub(x,y) = _Add(x,_Add(_BNand(y,y),1));
```

The above definition of subtraction captures precisely the details of the two's complement representation for negative integers. (It is often important to be able to faithfully emulate the operation of the *target* processor already at expansion time. Still, the *host* processor, which executes the macro expander program, may support a different representation; in that case, we are likely to need two subtraction variants—one for each processor. Actually, it seems natural that the "host subtraction" would be the one more heavily utilized.)

As the first stage of the evaluation of a call of a rigid compound form, and therefore even before the evaluation of the arguments, the interpreter replaces the call with the definition of the form called. This strategy supports lazy evaluation, because the definition may consist of an `_If` call.

Rigid compound forms may be recursive, again because of the special property of the `_If` form. Here is a recursive definition for the factorial function:

```
Fact(x) = _If(_Lt(x,2), 1, _Mul(x,Fact(Sub(x,1))));
```

It is often convenient to hide the autonomous primitive forms inside a separate "module" whose settings can be easily revised if necessary. (In particular, we might need, say, a "target multiplication" in addition to the predefined "host multiplication"; for an explanation, see the above discussion on subtraction variants.) Accordingly, we "redefine" the autonomous primitives (note especially the definition of `Div`):

```
Abort(x,y,z) = _Abort(x,y,z);
If(x,y,z) = _If(x,y,z);
Add(x,y) = _Add(x,y);
BNand(x,y) = _BNand(x,y);
BShl(x,y) = _BShl(x,y);
Div(x,y) = _If(y,_Div(x,y),_Abort(100,x,y));
Lt(x,y) = _Lt(x,y);
Mul(x,y) = _Mul(x,y);
Nand(x,y) = _Nand(x,y);
```

With a few more definitions, we can extend our collection of basic forms into a fairly useful one:

```
Not(x) = Nand(x,x);
Lte(x,y) = Not(Lt(y,x));
Gt(x,y) = Lt(y,x);
```

```
Gte(x,y) = Lte(y,x);
Eq(x,y) = If(Gte(x,y),Gte(y,x),0);
Neq(x,y) = Not(Eq(x,y));
```

In Figure 12, we define some auxiliary forms that are utilized in the macro definition examples of Appendix C. As you can see, we use as building blocks even some context-sensitive primitive forms, in addition to the autonomous ones. We assume that the available storage classes are `M`, `R`, and `A` (as shown in Figure 4 on page 9). We give some comments concerning these auxiliary forms:

`Int` checks whether its argument is an integer (that is, neither a cell nor a label).

`Type` tells its caller exactly which kind of object its argument is.

`Ordered` can be very useful in "normalizing" the order of macro arguments, as you can see by examining the macro definitions in Appendix C.

```
Int(x) = If(?(x),0,If(&(x),0,1));
Type(x) = If(?A(x),0,
             If(?R(x),1,
                If(?M(x),2,
                   If(Int(x),3,4))));
Ordered(x,y) = If(If(Int(x),Int(y),0),
                  Lte(x,y),
                  Lte(Type(x),Type(y)));
```

Figure 12: Definitions of some auxiliary forms.

5.8 Flexible compound forms

Flexible compound form definitions have to be somewhat more complicated than the ones for rigid compound forms, because of the variable number of arguments. (In contrast, there are no differences in the call syntax.) Our first flexible compound form is the logical disjunction:

```
Or {
  SEED = 0;
  STATE = If(STATE, 1, INPUT);
}
```

This Or definition consists of the subdefinitions of two *pockets*, SEED and STATE; in general, these two subdefinitions, in this order, are obligatory. By a 'pocket', we denote a store capable of holding an integer at expansion time. In addition to these two pockets, in the STATE subdefinition the INPUT *pseudopocket* is referred to. The "volatile" INPUT pseudopocket points to each form argument in turn. Thus, it may represent cells and labels as well as integers.

An Or call is evaluated according to the following procedure (which is applied to each call of each flexible compound form):

1. The integer designator that constitutes the SEED subdefinition is evaluated, and the resulting value is copied into the STATE pocket (as you see, the SEED pocket itself is actually redundant).
2. From left to right, for each argument of the original call, the integer designator that constitutes the STATE subdefinition is evaluated according to the following subprocedure:
    (a) Each reference to the STATE pocket is (temporarily) replaced with the current contents of STATE.
    (b) If there are one or more references to the INPUT pocket that must be evaluated (note that even If, through _If, implements lazy evaluation), then the current argument of the original call is evaluated, and the references to INPUT are (temporarily) replaced with the resulting object (which is an integer, a cell, or a label).
    (c) The integer that results from the evaluation of the whole modified STATE subdefinition is put in the STATE pocket (thus, the old value of the pocket is lost).
3. The evaluation result of the original form call is determined by the final value of the STATE pocket.

It should be noted that the above Or definition would represent logical disjunction even if the references to the STATE pocket and to the INPUT pseudopocket in the STATE subdefinition were interchanged. However, if the change took place, all the arguments of each Or call would always have to be evaluated.

The definition of the logical conjunction, And, is fairly similar to the Or definition:

```
And {
  SEED = 1;
  STATE = If(STATE, INPUT, 0);
}
```

Next we define a flexible form that examines whether its arguments are all integers. The `Const` definition closely resembles the `And` definition above:

```
Const {
  SEED = 1;
  STATE = If(STATE,Int(INPUT),0);
}
```

In addition the `SEED` and `STATE` pockets, the user may define new pockets. Such *auxiliary pockets* are always zero-initialized, and they are updated in parallel and fully synchronously with the `STATE` pocket. Our last form, in Figure 13, checks whether all of its arguments are equal integers. This `Equal` definition uses two auxiliary pockets, `later` and `value` (`later` indicates whether the current argument is a non-first one, and `value` holds the value of the first argument).

```
Equal {
  SEED = 1;
  STATE = If(STATE,
             If(later,
                And(Const(INPUT),Eq(value,INPUT)),
                Const(INPUT)),
             0);
  later = 1;
  value = If(later,value,INPUT);
}
```

Figure 13: Definition of the `Equal` flexible compound form.

Finally, we collect the main points you should remember when defining flexible compound forms:

- The `SEED` and `STATE` subdefinitions, in this order, are obligatory; after them, you may introduce auxiliary pockets. Furthermore, there is also the predefined `INPUT` pseudopocket.
- In the `SEED` subdefinition, no pocket can be referred to. Conversely, the `SEED` pocket itself cannot be referred to in any pocket subdefinition.
- The initial value of the `STATE` pocket is determined by the `SEED` subdefinition, and the initial value of each auxiliary pocket is 0.
- For each form argument, the `STATE` pocket and the possible auxiliary pockets are updated in parallel and fully synchronously.

6 Macro definitions

As depicted already in Figure 2 on page 7, a ReFlEx rule file consists of four main parts:

```
HEADER {      ...      }
STORAGE {      ...      }
SYSTEM {      ...      }
UTILITY {      ...      }
```

In this section, we concentrate on the `UTILITY` part, which contains the application macro definitions. Additionally, the system macro declarations of the `SYSTEM` part can be seen as reduced variants of application macro definitions; the differences are described in Section 6.2.

The compound form definitions of the `HEADER` part were discussed in Sections 5.7 and 5.8. In this and the forthcoming sections, we assume that all the forms of Section 5 are defined in our `HEADER` part. Furthermore, we assume the `STORAGE` part defined in Figure 4 on page 9: we have an accumulator `A`, four auxiliary registers `R[0]`–`R[3]`, and 1024 data memory locations `M[0]`–`M[1023]`. Finally, we also assume the `SYSTEM` part shown in Figure 5 on page 10.

In this section, our examples are short and simple; slightly more elaborate macro definitions can be seen in Appendix C. We adopt the convention that any macro whose name begins with a '`t_`' prefix is only for a momentary use. Often, no definition is provided for such a macro.

6.1 General structure of a macro definition

The `UTILITY` part of a ReFlEx rule file consists of a sequence of macro definitions (whose mutual order is insignificant). The general structure of such a definition is shown in Figure 14. As indicated in the figure, some of the items are optional.

The meaning of the items in Figure 14 is as follows:

deviation Is the macro a branch of any kind? See Section 6.3.

name Name of the macro being defined.

params Macro parameters (which represent integers, storage cells, and instruction labels). See Section 6.4.

```
deviation_opt name params {

    safety_opt
    test_opt
    temp_opt

    version
    version
       ...
    assertion_opt
    version
       ...
    insist_opt
    version
       ...
}
```

Figure 14: Structure of a macro definition.

safety Safety declaration (a promise by the macro writer). See Section 6.9.

test Macro-specific test clause. See Section 6.6.

temp Declaration of macro-specific temporaries. See Section 6.7.

version Definition of a macro version. See Section 6.5.

assertion An assertion clause. See Section 6.6.

insist An insist clause. See Section 6.6.

The macro definition can be divided into the following parts:

- The *macro exterior* contains the *deviation*, *name*, *params*, *safety* and *test* items. It can be seen as the external interface of the macro.
  - The *macro head* is a subset of the macro exterior. It consists of the *deviation*, *name*, and *params* items.
- The *macro interior* contains the *temp*, *version*, *assertion*, and *insist* items. It can be seen as the internal implementation of the macro.

Here is a simple macro definition which contains a single version:

```
JUMP t_switch(a,c) [l1,l2] {
  TEST And(?A(a), Const(c));
  USE R[t1], M[t2,t3];
  implem: {
    t_pre(a,c > t1,t2,t3);
    JUMP t_post(t1,t2,t3) [l1,l2];
  }
}
```

If a macro definition includes several alternative versions, they must be listed in the order of decreasing priority: at expansion time, ReFlEx traverses through the versions in the order specified by the macro writer.

6.2 System macros

The SYSTEM part of the rule file contains the system macro declarations (see Figure 5 on page 10 for an example). A system macro declaration consists of a macro exterior only:

```
BRANCH eq(a) [l] { TEST ?A(a); }
```

However, unlike an application macro exterior, a system macro declaration cannot contain a safety declaration (see Section 6.9). (Here the macro language designer had to choose between simplicity and uniformity, and simplicity was chosen.)

6.3 Control transfer

Concerning the control transfer after execution, ReFlEx macros are, for code readability, divided into four types by three distinct *deviation qualifiers*. All these qualifiers can be found in the macro calls shown in Figure 15.

```
t_trans1(x > y);
BRANCH t_trans2(x > y) [l1,l2];
JUMP t_trans3(x > y) [l3];
l1: SLEEP t_trans4(x > y);
```

Figure 15: The deviation qualifiers.

Macros of the BRANCH and JUMP types are the only ones that can transfer the control to a remote location (specified by a label parameter); macros of the default and BRANCH types are the only ones that can transfer the control to the location immediately following (the code resulting from) the macro call itself. Thus `BRANCH` and `JUMP` macros are conditional and unconditional branches, respectively, whereas `SLEEP` macros can represent non-terminating loops. Supposing that the macro calls in Figure 15 constitute a code fragment, the `t_trans4` call would be unreachable if it were not provided with the `l1` label temporary.

Each macro call instance must be explicitly provided with the same deviation qualifier that is specified already in the definition of the particular macro.

6.4 Macro parameters and arguments

Macro variables are divided into *parameters* and *temporaries* (for the temporaries, see Section 6.7). Because ReFlEx 1.0 does not support global variables, macro parameters are the only means for passing information across macro boundaries at execution time. There are both *data parameters* and *label parameters*. A data parameter may be an *input* one, an *output* one, or an *input-output* one. Label parameters represent branch targets.

The macro head specifies the parameters. We present some examples in Figure 16.

```
t_param1(x,y,x > y,z) {     ...     }
BRANCH t_param2(i,j) [l,l,m] {     ...     }
t_param3( > s) {     ...     }
JUMP t_param4() [t] {     ...     }
```

Figure 16: Declarations of macro parameters.

Note the following concerning the examples in Figure 16:

- Output parameters are separated from input ones by the '>' special character, which divides the data parameter list into two sublists. Understandably, input-output parameters must occur in both these sublists: `y` is an input-output parameter of `t_param1`. The '>' separator is omitted if there are neither output nor input-output parameters.
- If there are no label parameters, the label parameter list is omitted. In contrast, even if there are no data parameters, an empty data parameter list must still be present.
- Parameter `x` appears twice in the input parameter sublist of `t_param1`, and parameter `l` appears twice in the label parameter list of `t_param2`.

No parameter should appear twice in an output parameter sublist; duplicate input or label parameters are seldom useful, although they are fully supported by ReFlEx.

- For each macro definition, the set of data parameters and the set of label parameters must be distinct.

When a macro is called, *macro arguments* stand for the macro parameters. With macro arguments, we do not employ such a special notion as the notion of an input-output parameter with macro parameters—an output argument may simply simultaneously be even an input argument.

When a macro is called inside a definition of another macro, the macro writer provides the call with *virtual* arguments that represent the *actual* arguments, i.e. integers, storage cells, and instruction labels. The virtual arguments must meet the following constraints:

- A label parameter must be matched by a label variable (the reserved words `THIS` and `NEXT` cannot do).
- In general, a data parameter must be matched by a data variable; alternatively, an input (but not an input-output) parameter may be matched by an integer designator.

Not surprisingly, if a parameter appears multiple times in the parameter lists, the corresponding actual arguments should also be identical. The satisfaction of this constraint is not verified until expansion time—the corresponding *virtual* arguments need *not* be identical.

Here we provide sample calls for the macros of Figure 16:

```
t_param1(5,a,Add(2,3) > a,b);
BRANCH t_param2(u,v) [k,k,k];
t_param3( > a);
JUMP t_param4() [k];
```

6.5 Structure of a version definition

The macro definition contains an ordered sequence of version definitions; optionally, there may also be assertion clauses or insist clauses between the version definitions (see Section 6.6). The overall structure of each version definition is shown in Figure 17.

```
name : test_opt temp_opt {
   statement
   statement
     ...
}
```

Figure 17: Structure of a version definition.

The meaning of the items in Figure 17 is as follows:

name Name of the version being defined. (This name is more like a comment; it is never referred to.)

test Version-specific test clause. See Section 6.6.

temp Declaration of version-specific temporaries. See Section 6.7.

stmt A statement. See below.

A *statement* consists of a unique label temporary, a macro or pseudomacro call (for pseudomacros, see Section 6.8), and a terminating semicolon. Both the temporary and the call are optional. A statement is *empty* if it does not contain a macro (or pseudomacro) call. The statement sequence of a version definition is called the *version body.*

Simple examples of version definitions can be found, for instance, in Figure 18 in Section 6.6, and in Figure 19 in Section 6.7.

6.6 Test, assertion, and insist clauses

Sometimes the user needs to tell ReFlEx explicitly whether it should or should not take a particular course of action at expansion time. A directive of this kind is typically a conditional one: the course should be taken if and only if a given constraint is satisfied. Such constraints can be represented by integer designators: by convention, the constraint is taken as being satisfied if the integer designator evaluates to a nonzero value.

The user writes conditional directives by inserting the constraint in an appropriate *clause*. If the constraint of the clause is satisfied, we say that the clause itself is also satisfied. In a macro definition, you may find four different types of clauses: *macro-specific test* clauses (`TEST`), *version-specific test* clauses (`TEST`, again), *assertion* clauses (`ASSERT`), and *insist* clauses (`INSIST`).

In Figure 18 we show all the four clause types. In the `t_amacro` definition, there is a macro-specific test, two version-specific tests, a single assertion, and a single insist.

```
t_amacro(x > y) {
  TEST And(Or(?A(y),?R(y)), Gte(#R(),2)); USE R[t];
  implem1: TEST ?A(y); USE R[u]; {
    t_am0(x > t,u); t_am1(t,u > y);
  }
  INSIST Not(?A(y));
  implem2: TEST ?R(y); {
    t_am0(x > t,y); t_am2(t,y > y);
  }
  ASSERT 0;
}
```

Figure 18: The different clause types.

The meaning of the clauses is as follows:

- The macro-specific test determines whether a call of the current macro is acceptable as a part of the expansion-time realization of an upper-level macro call.
- The version-specific test determines whether the particular macro version can be considered as an expansion-time realization of a call of the current macro. (Even if the clause is satisfied, the version will be rejected if ReFlEx cannot allocate cells for the data variables of the version, or if the macro-specific tests of the macros called in the version body cannot be satisfied.)
- During the version selection, if ReFlEx reaches an assertion that is not satisfied, the whole expansion is immediately aborted and an error message is issued. Furthermore, no expansion result (not even a failing one) is produced.
- During the version selection, if ReFlEx reaches an insist that is not satisfied, the refinement of the current macro call fails, as if there were no more versions (an insist clause is useless if it is not succeeded by at least one more version). Moreover, because at this stage backtracking is no more possible, the whole expansion result of the original expansion source becomes a failure.

Note that there is at most one macro-specific test per macro definition, and at most one version-specific test per version, but there may be arbitrarily many assertions and insists in the version sequence of a macro definition. Still, there is an important property that all the four clause types have in common: a missing (macro-specific or version-specific) test can be considered as a satisfied one, and missing assertions and insists could trivially be “replaced” with ones that are guaranteed to be always satisfied.

6.7 Macro temporaries

The set of macro variables may include *macro temporaries*, in addition to macro parameters. Similarly to the macro parameters, the macro temporaries are divided into *data temporaries* and *label temporaries*. Data and label temporaries are valid as cell and label references, respectively, but invalid as virtual form arguments (see also Section 6.10). A data temporary is introduced in a temporary declaration (`USE`), and a label temporary is introduced by including it as a prefix in some statement. The temporaries must be different from the parameters and unique up to the particular macro version.

The macro writer must select the storage class for each data temporary. When the macro expander links an appropriate realization to a macro call, it binds each data temporary permanently to some fixed cell of the user-selected class. Data temporaries cannot be made “static”: none of them can retain its contents between different execution times of the expansion result.

Data temporaries may be either macro-specific or version-specific, while label temporaries are always version-specific. The only purpose of introducing macro-specific data temporaries is making the macro definition shorter and thus more easily readable. Examples of both macro-specific and version-specific temporaries can be found in Figure 19.

```
t_bmacro(x > y) { TEST ?(x,y); USE A[a], M[t1,t2];
  implem1: TEST ?R(x); USE R[t3]; {
    t_bm1(x > t1,t3);
    t_bm2(t1,t3 > y);
  }
  implem2: USE M[t3]; {
    BRANCH t_bm3(x > a,t2,t3) [l0];
    t_bm4(t3 > t2);
    l0: t_bm5(a,t2 > y);
  }
}
```

Figure 19: Declaration of macro temporaries.

In the example in Figure 19, temporary `l0` is specific to version `implem2`.

Temporaries a, t1, and t2 are macro-specific ones from storage classes A, M, and M, respectively. Version implem1 has a version-specific temporary t3 of class R, and version implem2 happens to have a temporary with the same name of class M.

There remains one important temporary-related feature, whose use may sometimes significantly shorten the text of a macro definition. The storage class for each temporary has to be selected by the macro writer, but this selection can actually be delayed until expansion time. Consider the definition of the my_swap macro shown in Figure 20; the macro is a generalization of the my_mswap macro introduced in Figure 9 on page 13.

```
my_swap(x,y > y,x) {
  body: USE A<And(?M(x,y),Lt(#R(),2))>
           .A<And(?R(x,y),Lt(#R(),3))>
           .R[t];
  {
    my_move(x > t); my_move(y > x); my_move(t > y);
  }
}
```

Figure 20: Delayed data temporary classification.

Even the restricted my_mswap needed two separate versions—why is a single version now enough for my_swap? The answer is that the version-specific temporary t of my_swap is not classified before expansion time. If both the arguments are from class M and there is at most one cell from class R free, or if both the arguments are from class R and there is no additional cell from class R free, t is taken from class A; otherwise, it is taken from class R.

If the macro writer wishes to specify several alternative storage classes for a temporary, as with t of my_swap, all of them but the last one (i.e. the one with the lowest priority) must be provided with an *adjunct clause*. In general, an adjunct clause, as well as a clause, consists of an integer designator that is evaluated at expansion time: if the evaluation result is nonzero, ReFlEx takes the action with which the adjunct clause is associated. In fact, the use of adjunct clauses differs from the use of the "proper" clauses only syntactically: the adjunct clauses are embedded in larger declarations. (See Section 6.9 for another context that embraces adjunct clauses.)

6.8 Initialization of data variables

Usually, a statement consists of a macro call. However, a statement may alternatively consist of a call of the INIT *pseudomacro* (there are no other types of pseudomacros). The use of this pseudomacro is closely related to two constraints on version body structure, which aim to catch some mistakes possibly made by the macro writer:

- No data temporary or output parameter can be read before it is written or initialized.
- Each output parameter must be written or initialized (neither this nor the preceding constraint concerns input-output parameters).

`INIT` is directive-like and never produces any machine instructions. It requires one virtual output argument, which should be the data variable to be initialized. By such an initialization, the macro writer indicates that the current contents of the cell represented by the data variable are fully insignificant from the viewpoint of the present macro definition. (Still, ReFlEx carefully protects the contents if they are significant to some upper-level macro of the expansion-time environment.)

The macro definition in Figure 21 contains a version employing `INIT`. The whole `my_move` macro definition appeared (in a slightly different guise) already in Figure 7 on page 12 (see Figure 6 on page 11 for the definition of the empty `my_null` macro; the version-specific test of the `same` version is actually redundant). It implements a general data transfer from an arbitrary source to an arbitrary destination.

```
my_move(s > d) {
  same: TEST =(s,d); { my_null(s > d); }
  as_set: { set(s > d); }
  INSIST Not(And(Const(s),?R(d)));
  as_load: { load(s > d); }
  as_store: { store(s > d); }
  as_move: { move(s > d); }
  clear_acc: TEST Equal(s,0); USE R[r]; {
    INIT( > r); move(r > d); sub(d,r > d);
  }
  temp_is_needed: USE R[r]; {
    my_move(s > r); my_move(r > d);
  }
}
```

Figure 21: The `my_move` macro.

The `clear_acc` version of `my_move` is used for resetting the accumulator to zero. The version uses an auxiliary register, whose arbitrary contents are first copied into and then subtracted from the accumulator. Because this register is thus not overwritten, `clear_acc` can be used even when there are no free auxiliary registers. However, the fact that the original value of the register is insignificant must be explicitly indicated with an `INIT` call.

6.9 Safety declarations

A *safety declaration* is a macro-specific promise by the author of the macro definition saying that some output is left unwritten: ReFlEx can first utilize and later verify this promise. One might regard safety declarations as a secondary and perhaps the most complicated feature of the ReFlEx macro definition language. On the other hand, their use often shortens macro definitions and is itself fairly "safe".

The safety declaration provides the particular output (or input-output) parameter with an adjunct clause (see the end of Section 6.7) specifying the conditions of the promise. However, there is an essential restriction: in such an adjunct clause, context-sensitive forms `=(...)`, `!(...)`, and `#()` cannot be even indirectly referred to. This is because ReFlEx evaluates the conditions already before the intraclass cell allocation; in contrast, the evaluation result of, say, `%(...)` does not depend on the intraclass allocation.

The rest of this subsection deals with a simple but fundamental example. The `my_snull` macro in Figure 22 contains a safety declaration (`SAFE`). This empty macro unconditionally promises not to write its input-output parameter; this promise is clearly fulfilled by the single version of the macro. (The motivation behind this macro definition will soon become obvious.)

```
my_snull(x > x) {
  SAFE x<1>;
  null: { }
}
```

Figure 22: Macro `my_snull` with a safety declaration.

Then, the `my_smove` macro in Figure 23 both calls `my_snull` and includes a safety declaration of its own. Provided that `s` and `d` represent cells belonging to the same storage class, `my_smove` promises not to write output parameter `d` (but at the same time requires that the two parameters are mapped to a single common cell). By combining the information from the macro-specific test and the `null` version definition, you notice that `my_smove` calls can indeed be successful even under this promise (because the `my_snull` call is known not to write `d`).

```
my_smove(s > d) {
  SAFE d<%(s,d)>;
  TEST Or(=(s,d),Not(%(s,d)));
  null: { my_snull(s > d); }
  move: { my_move(s > d); }
}
```

Figure 23: Macro `my_smove` that calls `my_snull`.

Next, the `my_double` macro in Figure 24 is a multiplier-by-two. The call of `my_smove` in the macro definition saves us from one additional version: as it is, the `default` version works fine regardless of whether input parameter `x` represents a cell of class `R` or `M`.

```
my_double(x > y) {
  USE R[r], A[a];
  const: TEST Const(x); { my_move(Add(x,x) > y); }
  acc: TEST ?A(x); {
    my_move(x > r); add(x,r > x); my_move(x > y);
  }
  default: {
    my_smove(x > r); my_move(r > a);
    add(a,r > a); my_move(a > y);
  }
}
```

Figure 24: Macro `my_double` that calls `my_smove`.

Finally, ReFlEx expands a sample `my_double` call

`my_double(R[2] > A); {R[0],R[1],R[3]}`

in the manner shown in Figure 25. Note that ReFlEx has accepted the `default` version of `my_double`, and moreover, mapped temporary `r` to cell `R[2]`—which is the optimum one—although the `my_smove` call writes `r` and `R[2]` is the only cell of class `R` that is not free. Without the safety declaration of `my_smove`, this optimum mapping could not have been found (though even then the macro-specific test of `my_smove` would have been sufficient for rejecting the non-optimum mappings).

6.10 Form calls in macro definitions

Unlike macro parameters, macro temporaries cannot be used as virtual form arguments. The only allowed virtual form arguments inside a macro definition

```
my_double(R[2] > A) {A,R[0],R[1],R[3]} {
  my_smove(R[2] > R[2]) {A,R[0],R[1],R[3]} {
    my_snull(R[2] > R[2]) {A,R[0],R[1],R[3]} {
    }
  }
  my_move(R[2] > A) {A,R[0],R[1],R[3]} {
    move(R[2] > A); {A,R[0],R[1],R[3]}
  }
  add(A,R[2] > A); {A,R[0],R[1],R[3]}
  my_move(A > A) {A,R[0],R[1],R[3]} {
    my_null(A > A) {A,R[0],R[1],R[3]} {
    }
  }
}

my_double(R[2] > A) {A,R[0],R[1],R[3]} {
  move(R[2] > A); {A,R[0],R[1],R[3]}
  add(A,R[2] > A); {A,R[0],R[1],R[3]}
}
```

Figure 25: The expansion result of a `my_double` call.

are macro parameters, integer literals, form calls, and the two reserved words `THIS` and `NEXT`.

You should remember that even if a form call serves as a virtual macro argument, its value is evaluated as if it constituted the macro-specific test of the current macro definition. For instance, in the macro definition

```
t_cmacro(x > x) {
  impl: { t_cm1(x > x); t_cm2(!(x)); t_cm3(x > x); }
}
```

the `!(x)` form call is guaranteed to return 1, even if at the call of `t_cm2` the parameter `x` is certainly not free. The reason is that because `x` is included in the output parameter list (and there is no safety declaration), the corresponding argument is free at each call of `t_cmacro`.

Finally, we repeat that the context-sensitive forms `?(...)`, `!(...)`, and `#()` cannot be called in safety declarations, not even indirectly.

7 Macro expansion procedure

In this section, we give a concise systematic description of the macro expansion procedure. Despite its technical nature and its total lack of examples, the description may help you to understand the ReFlEx operation as a whole. However, you can use ReFlEx even without reading this section.

The macro expansion begins from an *expansion source* and produces an *expansion result.* The expansion source always consists of a single macro call, and the expansion result is a sequence of macro calls. The macro expansion procedure comprises two main phases: a *linking phase* and a *merging phase.*

First, during the linking phase a tree structure is built upon the source macro call. Each *node* of this *expansion tree* contains a macro call; the node that contains the source macro call is the *root.* The nodes of the tree can be divided into *stock nodes* and *leaf nodes.* To the macro call of each stock node a *realization* has already been linked, while the macro calls of the leaf nodes have not yet been provided with a realization. The realization of a macro call is selected from the versions of the called macro. The linking phase is ended when there is no more room for further refinement, that is, when each leaf of the tree is either a system macro call (which need not be further refined) or such an application macro call that cannot be successfully refined. Second, the merging phase linearizes the tree into a macro call sequence by removing the stock nodes.

The merging phase is fairly trivial. Its details are briefly addressed finally in Section 7.7. The rest of this Section 7 deals with the linking phase.

7.1 Overall linking strategy

The linking phase comprises a number of successive steps: each step links a realization to an application macro call, which thus changes from a leaf node into a stock node. How should the leaf node to be processed next be selected? The answer is that this selection is fully insignificant, as explained in Section 7.3.

Suppose that we have somehow selected a leaf node. Then, a suitable realization version for its macro call must be selected from the versions of the particular macro. ReFlEx traverses the versions in the order specified by the macro writer. The criteria according to which each one of the alternative versions is either accepted or rejected are cumulatively formulated in Sections 7.2–7.5. In particular, the notion of a free cell is crystallized in Section 7.3.

It is possible that the constructed expansion tree contains an application macro call leaf for which no suitable macro version exists. This means that the whole task originally given to ReFlEx, i.e. the expansion of the expansion source, has irreversibly failed: ReFlEx 1.0 does not support backtracking.

The linking phase is guaranteed to end, since infinite recursion is disabled: the user must set a limit on the number of nodes in the expansion tree (and, correspondingly, on the allowed form call recursion depth).

7.2 Preliminary definitions

We aim at generality and brevity in our discussion; in particular, we want that some crucial notions apply to both macro definitions and expansion trees. We begin with some simple generalizing conventions:

- Cells and data variables are collectively called *markers*.
- We assume that ReFlEx quietly associates a unique *implicit label temporary* with each such statement in a macro version body that the user has not explicitly provided with a label temporary.
- The macro head is a syntactically valid macro call; since we want to be able to treat it as a statement, we assume that ReFlEx provides even it with an implicit label temporary.

Because of the implicit label temporaries, no macro version body can contain two identical statements. Next, we give our main umbrella definition:

- A *macro scope*, or more shortly, a *scope* consists of a *scope head*, which is a statement, and a *scope body*, which is a sequence of statements.

Intuitively, the scope body is an implementation of the scope head. We introduce two scope types, *definition scopes* and *expansion scopes*:

- For a macro definition, there are as many definition scopes as there are macro versions. The head of each definition scope is the macro head statement, and the scope body is the body of one of the macro versions.
- For an expansion tree, there are as many expansion scopes as there are stock nodes. The head of each expansion scope is the macro call statement of the stock node, and the scope body is the realization of that macro call. Thus, any stock node statement other than the root statement both is a head of an expansion scope and belongs to the body of another expansion scope.

We also introduce some notions related to execution-time control propagation. First, concerning intrascope control propagation, we adopt the following terminology:

- Each statement in a scope body has a set of *immediate successor* statements associated with it. For instance, a two-way branch typically has two immediate successors.
- A *path* in a scope body is a chain of statements that is glued together by the immediate successorship.

Second, concerning interscope control propagation, we adopt two more terms:

- Each scope body has exactly one *entry point*, which is the first statement in the scope body, i.e. the statement that receives the control from the outside.
- Each scope body has a set of *exit points*: any statement in the scope body that may relinquish the control outside the scope is an exit point. (The last statement in the scope body is an exit point unless the deviation qualifier of its macro call is `JUMP` or `SLEEP`; additionally, any statement whose macro call has a label argument in common with the scope head is an exit point.)

7.3 Free cell analysis

In this subsection, we formulate a precise definition for the notion of a free marker, that is, a free cell or a free data variable. We begin by making three fairly intuitive conventions explicit:

- Each macro call *reads* its input arguments.
- Each macro call *writes* its output arguments.
- Each `INIT` pseudomacro call *initializes* its output argument.

We also say that a statement reads, writes, or initializes a marker if the macro or pseudomacro call of the statement reads, writes, or initializes, respectively, the marker. Now, we are ready to define life paths, which indicate data dependencies that span over statements in the code.

A path is a *life path* of a marker if all the following conditions are met:

- No statement in the path writes or initializes the marker.
- The first statement of the path meets at least one of the following subconditions:
  - It reads the marker.
  - It is an immediate successor of some statement that writes the marker.
  - It is an entry point of the scope body, and the scope head reads the marker.
- The last statement of the path meets at least one of the following subconditions:
  - It has an immediate successor that reads the marker.
  - It is an exit point of the scope body, and the scope head writes the marker.

Finally, we are able to define the set of *free* markers for each statement in any scope. Below, you may choose to neglect the bracketed references to *safeguarding*, which are relevant only when the macro definitions contain safety declarations; the notion of safeguarding is explained in Section 7.6.

We start with the simple case of definition scopes:

1. Every data variable is free at the definition scope head (this part is included only for the sake of completeness).
2. A data variable is free at a statement in a definition scope body if the statement belongs to no life path of the data variable.

The case of expansion scopes is more complicated. We assume that the user explicitly provides the expansion source with a *disposal set*, which lists the cells that can be used as an auxiliary storage. The definition is recursive:

3. A cell is free at the root statement of an expansion tree if the cell [is not safeguarded by the root statement and] either belongs to the disposal set or is written by the root statement.
4. A cell is free at a statement in an expansion scope body if both the following subconditions are met:
   - [The cell is not safeguarded by the statement and] the statement belongs to no life path of the cell.

- The cell is free at the expansion scope head.

Note in particular that the freedom in expansion scopes depends only on the structure of the current scope and the enclosing upper-level scopes, and on the disposal set. Effectively, this means that the order in which the appropriate realizations are linked to the leaves of the expansion tree is indeed insignificant, as we claimed already in Section 7.1. (It does not matter whether safeguarding is taken into consideration or not.)

Often when there is no room for misunderstanding, we simply say that a cell or a data variable is free at some macro call and actually mean freedom at the statement that contains the macro call.

7.4 Variable binding

When a macro version body is about to be linked to a leaf macro call, each macro variable is mapped to a single integer, cell, or label. This *variable binding* must respect a number of constraints. For instance, the parameter lists of the macro definition establish certain requirements on the corresponding argument lists of the leaf macro call. We specify now all the constraints on variable binding (again, you may choose to neglect the references to safeguarding):

1. Each data parameter must be bound to the cell or integer that is the corresponding argument of the leaf macro call.
2. No input parameter that is written by some macro call in the version body can be bound to an integer.
3. Each label parameter must be bound to the label that is the corresponding argument of the leaf macro call.
4. Each data temporary must be bound to a cell of the indicated storage class.
5. Each data variable that is written [without safeguarding] by some macro call in the version body must be bound to a cell that is free at the leaf macro call.
6. If two distinct data variables are bound to a single common cell, and if one of them is written by some macro call in the version body, then the other one must not be written by that macro call.

7. If two distinct data variables are bound to a single common cell, and if one of them is written [without safeguarding] by some macro call in the version body, then the other one must be free at that macro call.

There may be several successful but still essentially different data temporary bindings available; in contrast, the label temporaries can trivially be bound to unique labels that do not previously occur in the expansion tree. We intentionally leave it unspecified how ReFlEx 1.0 chooses among alternative successful data temporary bindings. Still, a wrong choice may later prove to be a fatal mistake: the whole expansion may result in a failure that could have been prevented by a better choice.

If ReFlEx finds a successful variable binding, *variable replacement* can take place: in the version body, each occurrence of each variable is replaced with an occurrence of the integer, cell, or label to which the particular variable is bound. In addition to the variable replacement, some further modification in the macro version body must be simultaneously carried out:

- Each `INIT` pseudomacro call statement is removed if it initializes such a variable that is either bound to an integer or bound to the same cell as some other variable that is not free at that `INIT` call.

7.5 Accepting or rejecting a macro version

The expansion tree is not complete as long as there are leaves consisting of application macro calls. For each such leaf, the versions of the appropriate macro definition are traversed until an acceptable version is found. Once the version selection is made, it cannot ever be canceled, not even in the case that a blind alley is later met. In other words, ReFlEx 1.0 does not support backtracking in version selection. Therefore, the ordering of the macro versions and the content of their version-specific tests should be orchestrated very carefully.

Each macro version is accepted or rejected according to the following criteria:

1. The version-specific test must be successful.

2. A successful variable binding must be found (see Section 7.4 for the details).

3. The macro-specific test of each macro call *in the version body* must be successful. (To evaluate these tests, ReFlEx has to create a provisional link between the leaf node and an appropriately modified copy of the

version body.) An implicit supplement to each macro-specific test is the requirement that if two elements of the macro parameter lists are identical, then the two corresponding (now provisionally fixed) macro arguments must also be identical.

If a particular macro version is accepted, an appropriately modified copy of the version body is linked to the leaf node and thus irreversibly attached to the expansion tree. Because of the new link, the leaf node becomes a stock node and the number of expansion scopes in the tree is incremented by one.

Note that we have not yet specified the time at which the root macro call has to undergo its macro-specific test. Now, we straighten out even this defect: the whole linking phase begins with the evaluation of this test.

7.6 Safeguarding

Above, we have mentioned 'safeguarding'. A macro call may safeguard only its own output arguments. More precisely, we specify the meaning of the term as follows (for safety declarations, see Section 6.9):

- In an expansion tree, a macro call statement *safeguards* a cell written by it if the macro definition contains a safety declaration that provides the corresponding output (or input-output) parameter with an adjunct clause that is satisfied at that macro call statement.

Clearly, the above formulation concerns only expansion trees; however, in Section 7.4 we spoke of safeguarding even in definition scope bodies. We give an intuitive explanation: what we meant is such an expansion tree that could be constituted by linking the particular macro version body to the leaf node in question, according to any such variable binding that meets the first four of the seven constraints listed in Section 7.4. Because the "most powerful" ones of the context-sensitive primitive forms cannot be employed in a safety declaration, the adjunct clauses in the declaration may be accurately evaluated even against such premature information about the expansion-time environment. The reason for this look-ahead is rather obvious: if an instance of safeguarding is detected in the version body, the consequences of the remaining constraints (specifically, 5 and 7) may well become less restrictive.

7.7 Merging phase

The merging phase linearizes the expansion tree into an expansion result, which is a sequence of macro calls. This linearization means that all the following nodes are dropped off from the tree:

- All the stock nodes.
- Each remaining `INIT` pseudomacro call node.
- Each unreachable leaf node. (Such nodes may exist, because macro definitions may well include versions that do not refer to every label parameter.)

Technically, the expansion result does not constitute a macro scope; nevertheless, the macro calls in it are provided with explicit free cell information inherited from the expansion tree (in which the free cell information is implicit, on condition that safeguarding is excluded).

8 Input and output

The input to ReFlEx should be a specification of the expansion source, while the output produced by ReFlEx is a listing of the expansion result. The expansion source is a macro call and the expansion result is a sequence of macro calls. The expansion can be considered as a success if the expansion result contains no application macro calls.

Optionally, you can also view the intermediate tree structure built by ReFlEx during the expansion. This tree shows the links between the individual macro calls and the macro version bodies that are their respective realizations.

In the examples below, we will consistently use the `my_zero` macro defined in Figure 9 on page 13 in Section 2.5.

8.1 Expansion source

An expansion source line should contain a macro call statement:

```
BRANCH my_zero(M[5]) [L7];  // hopeless!
```

Here, ReFlEx is required to produce code that branches to label `L7` if the value in cell `M[5]` is zero. Note that the possible deviation qualifier cannot be dropped out and the macro call must be ended with a semicolon '`;`'. Note also that a comment is allowed (comments may be useful especially when the expansion sources are not given interactively but as a batch).

Our next, slightly extended example

```
BRANCH my_zero(M[5]) [L7]; {A}
```

permits ReFlEx to use accumulator `A` as a temporary data storage: the disposal set is seen to consist of cell `A[0]`. Note that because storage class `A` has only one member, ‘`A`’ can stand for ‘`A[0]`’.

Our third example

```
L3: BRANCH my_zero(M[5]) [L7]; >L7
```

provides the source macro call with label `L3`, and moreover, specifies label `L7` as the *follower label.* The follower label is the label associated with the code location immediately following the source macro call (or rather, its expansion result). Additional information of this kind may be useful when the follower label is also a label argument of the source macro call (as in the example).

Finally, our fourth and last example

```
L3: BRANCH my_zero(M[5]) [L7]; {A,R[2]} >L9
```

presents a “normal” case that produces a successful expansion result (see Section 8.2). (Note that if both the disposal set and the follower label are given, then the disposal set specification must come first.)

More generally, the expansion source is subject to following restrictions:

- The source statement must consist either of an application macro call or of a system macro call (and thus not of an `INIT` call).
- The input arguments must be cell or integer literals. Form calls are not accepted.
- The output arguments must be distinct cell literals.
- Any syntactically valid identifier is a potential label literal and can thus be used as a label argument.

8.2 Expansion result

Here we show the expansion results for the examples presented in Section 8.1. In addition to the expansion result, the ReFlEx output always includes a heading that lists the source statement, the disposal set (augmented with the cells written by the source macro call), and the possible follower label.

The first example, that is,

```
BRANCH ?my_zero(M[5]) [L7] {} {
  BRANCH ?my_zero(M[5]) [L7];
}
```

is an expansion with a failing result. By default, the disposal set is empty, and thus ReFlEx finds no free cells. As shown above, such application macro calls that cannot be provided with a realization are preceded with a question mark '?'. (A system macro call is treated similarly if it constitutes the expansion source but does not satisfy the appropriate macro-specific test.)

The second example, with a non-empty disposal set,

```
BRANCH my_zero(M[5]) [L7] {A} {
  ?my_move(M[5] > A);
  BRANCH eq(A) [L7];
}
```

shows another failing expansion result. In this case, the source can be provided with a realization, but no realization is found for the lower-level `my_move` call: a free auxiliary register is still wanting.

The next example is somewhat anomalous. Because here the target of the conditional branch is the follower label, the expansion result

```
L3: BRANCH my_zero(M[5]) [L7] {} >L7 {
}
```

is empty (but successful).

Finally, the fourth and last example gives us a non-empty successful expansion result:

```
L3: BRANCH my_zero(M[5]) [L7] {A,R[2]} >L9 {
  load(M[5] > R[2]);
  move(R[2] > A);
  BRANCH eq(A) [L7];
}
```

Actually, the expansion result contains also the set of free cells for each macro call. Safe estimates for the free cell sets can be computed even from listings such as the above one, but Section 9.2 tells you how to view the very sets (inherited from the expansion tree) themselves:

```
L3: BRANCH my_zero(M[5]) [L7] {A,R[2]} >L9 {
  load(M[5] > R[2]); {A,R[2]}
  move(R[2] > A); {A,R[2]}
  BRANCH eq(A) [L7]; {A,R[2]}
}
```

8.3 Intermediate tree structure

The intermediate tree structure built by ReFlEx is often very informative. The expansion tree records the expansion history: the realizations for the macro calls that are not present in the expansion result any more. Instructions for tracing it are given in Sections 9.2 and 9.3. For the final expansion source example considered above, that is,

```
L3: BRANCH my_zero(M[5]) [L7]; {A,R[2]} >L9
```

the tree is as follows:

```
L3: BRANCH my_zero(M[5]) [L7] {
  my_move(M[5] > A) {
    my_move(M[5] > R[2]) {
      load(M[5] > R[2]);
    }
    my_move(R[2] > A) {
      move(R[2] > A);
    }
  }
  BRANCH eq(A) [L7];
}
```

You can easily spot the leaves because of the trailing semicolons. Again, you could include the free cell sets in the listing. In this case, however, they are redundant, that is, deducible from the listing above. This also holds for expansion trees in general—on condition that the possibly safety declarations of the application macros are additionally known.

9 Running ReFlEx

Suppose you have successfully installed ReFlEx, along the guidelines given in Section 10. This section, then, tells you how to run ReFlEx.

A ReFlEx session can be outlined as follows. After loading the given rule file containing macro definitions, ReFlEx prompts for the first expansion source. Once you have typed it in, ReFlEx responds by producing the expansion result, and prompts again for the next expansion source.

9.1 Invoking ReFlEx

You start ReFlEx by giving the command

> `reflex` *<rule-file>*

at the operating system shell level. ReFlEx assumes that *<rule-file>* is a reference to the operating system file that should be taken as the rule file. (The reference cannot begin with a minus sign '`-`'.)

Additionally, there are several command-line options, which are described in Section 9.2. Some of the options are useful especially when the standard input or output stream is redirected so that ReFlEx reads the expansion sources from or writes the expansion results into some operating system file.

When ReFlEx has started successfully, it prints its own prompt, '`>`'.

9.2 Command-line options

The rule file must be specified on the ReFlEx command line. Additionally, there are a few possible command-line *options*. Each option begins with a minus sign '`-`' and contains no spaces. The options are:

`-s` Omits printing the '`>`' prompt ('s' is for *silent*). This is useful if the input stream is redirected from an operating system file.

`-e` Inserts a copy of the expansion source at the beginning of the output ('e' is for *echo*). This may be useful in the case of input or output stream redirection.

`-t` Outputs even the intermediate tree structure, in addition to the expansion result ('t' is for *tree*).

`-f` Includes in the output the set of free cells for each macro call ('f' is for *free*).

`-n:`***<int>*** Sets a limit on the total number of recursively generated macro calls for each expansion source, in order to cut off infinite recursion ('n' is for *number*). The default limit is 128, and this option can be used for enabling larger expansion trees. The *<int>* field can be any unsigned decimal number; however, the limit is automatically increased to at least 16.

`-d:`*<int>* Sets a limit on the form call recursion depth, in order to cut off infinite recursion ('d' is for *depth*). The default limit is 1024, and this option can be used for enabling deeper recursion. The *<int>* field can be any unsigned decimal number; however, the limit is automatically increased to at least 16.

`-r:`*<file>* Enables you to select interactively the expansion sources from an operating system file ('r' is for *read*). Furthermore, you can edit this file with you favorite text editor during a single ReFlEx session. See Section 9.3 for more information.

`-w:`*<file>* Starts recording the expansion sources to be processed into an operating system file ('w' is for *write*). The generated file can be fed back into ReFlEx, even during the same interactive session by using the `-r:`*<file>* option.

These options may also be given interactively (see Section 9.3). You may even issue several instances of a single option during a ReFlEx session, either on the command line or interactively. On the command line, the rightmost instance is the effective one. However, the `-s`, `-e`, `-t`, and `-f` options are toggles: for example, a second `-s` cancels the effect of the first one.

9.3 Interactive control

Normally, you respond to the ReFlEx prompt '>' by typing an expansion source on the input line:

```
> my_move(7 > A); {R[2]}
```

However, there are also some possible *control buttons* available for you. To exit ReFlEx, type a full stop:

```
> .
```

To receive a brief help message on the expansion source syntax, type a question mark:

```
> ?
```

You can also update any command-line option (but only one at a time):

```
> -n:1000
```

Having updated an option, you may wish to process the previous expansion source again. To achieve this, type an exclamation mark:

```
> !
```

By using the `-r:`<*file*> option, you may select an operation system file. ReFlEx assumes that this file consists of lines containing expansion sources. A positive integer k such as

```
> 5
```

is a control button that tells ReFlEx to process the kth expansion source from the file (the expansion sources recorded through the `-w:`<*file*> option are provided with comments that explicitly build up the appropriate indexing). You can freely edit the file between issuing instances of this control button.

9.4 Exceptional conditions and exception messages

There are several reasons why the expansion might break up without producing even a failing result. We believe that ReFlEx can detect most of such exceptional conditions and provide you with appropriate error messages. Still, an unguarded run-time stack overflow, for instance, may sometimes arise (see below).

Every type of *exception message* that may be issued by ReFlEx has a unique message number; see Section 10.2 for obtaining more information by means of the message number. The exception messages are divided into three categories:

- *Fatal errors* cause ReFlEx to terminate immediately.
- *Errors* prevent the processing of the current expansion source.
- *Warnings* indicate less severe but still unsatisfactory conditions that should be polished by the user.

Below, we list some exceptional conditions.

Errors in the rule file and in the expansion source

The possible syntactical and semantical errors in the rule file are detected already when ReFlEx is started, and they cause a premature program termination. Similarly, syntactical and semantical errors in the expansion source are detected before the actual expansion.

ReFlEx 1.0 cannot recover from syntactical errors: no more than the first syntactical error can be pointed to the user. Semantical errors, in contrast, can be recovered from.

Assertion failure in version selection

When ReFlEx is selecting a macro version for some macro call, an assertion clause may be reached. If the assertion clause fails, an error message is produced and the current expansion source is discarded. See Section 6.6 for more information on assertion clauses.

Abortion in form call evaluation

Because of the lazy evaluation, the actual set of form calls that have to be evaluated is normally not fully predetermined. If a call of the `_Abort(x,y,z)` primitive form is ever encountered during the macro expansion, an error message containing the evaluated arguments `x`, `y`, and `z` is produced, and the current expansion source is discarded.

Infinite recursion

In principle, both macro and form definitions may suffer from infinite recursion. In practice, however, the recursion depth is always limited. If the limit is reached, an error message is produced and the current expansion source is discarded. The recursion limits can be changed by using the `-n:`<*int*> and `-d:`<*int*> command-line options (see Section 9.2).

Stack overflow

If the limits on recursion depth are too large, the software stack (probably set up by the compiler that created the ReFlEx executable) may overflow. The overflow is not detected by ReFlEx and may therefore have unpredictable consequences. The recursion limits can be changed by using the `-n:`<*int*> and `-d:`<*int*> command-line options (see Section 9.2).

Heap overflow

During its execution, ReFlEx may ask the operating system for more dynamic memory for new run-time data structures. If no more memory can then be allocated, a premature program termination takes place.

Bugs and internal errors

Obviously, there may be bugs in the ReFlEx 1.0 implementation. It is possible that some bugs are detected by ReFlEx itself; these *internal errors* are understandably fatal ones. We would greatly appreciate any information on bugs and internal errors you may encounter; please see Section 10.2.

10 Installing ReFlEx

Any non-commercial use of ReFlEx is free of charge. You may install ReFlEx on your local workstation. The operating system should support ASCII character code and unsegmented virtual memory (as yet, we have not tried to port ReFlEx to MS-DOS). The practical details are referred to in Section 10.2.

10.1 Technical information about ReFlEx 1.0

ReFlEx 1.0 source code is written in the C++ programming language [21]. The original compiler was GNU `g++`. Other programming tools utilized were the `flex` lexical analyzer generator and the GNU `bison` parser generator [15].

The integer range supported by ReFlEx 1.0 is $-32768 \ldots 32767$. Still, the C++ compiler must implement the `long` integers as at least 32-bit entities, as required by the ANSI C definition.

As mentioned in Section 5.5, the semantics of the arithmetic primitive forms is determined by the C++ compiler used for compiling the ReFlEx source code. Indeed, ReFlEx simply converts calls of these primitive forms into the C++ (or equivalently, C) expressions shown in Table 1 on page 27. This means, for instance, that on different platforms calling the `_Div` form with zero as the divider may have different consequences.

10.2 Obtaining a copy of ReFlEx

ReFlEx can be obtained through the Internet by anonymous `ftp`: contact `ftp` server `saturn.hut.fi`, move into directory `pub/reflex`, and look for file `README`. Alternatively, if you have a World Wide Web browser, you may directly view the location

```
ftp://saturn.hut.fi/pub/reflex/README
```

File `README` specifies how you can find the following items:

- Description of the possible changes made to the ReFlEx documentation.
- ReFlEx source code and instructions for compiling it.
- A listing of ReFlEx exception messages.
- A sample ReFlEx rule file (which also constitutes Appendix C) and a companion file with sample expansion sources.
- An off-the-record exercise for prospective ReFlEx programmers (see also Appendix C).

Finally, we would greatly appreciate any comments, questions, bug reports, and suggestions for improvements. The `README` file records some convenient ways to contact us.

A Rule file syntax

In this appendix, we present a definition for the ReFlEx rule file syntax. You can absorb this definition easily if you are familiar with the syntax of `lex` and `yacc` input files. (These two programs are widely used compiler-generation tools: `lex` is a lexical analyzer generator and `yacc` is a parser generator [15].)

Token structure

First, we present the required token structure by using a formalism that closely resembles the `lex` input formalism. (Consult [15] if you are not familiar with `lex`.)

```
letter    [A-Za-z_]
digit     [0-9]
id        {letter}({letter}|{digit})*
int       {digit}+

%%

" "       { }
\t        { }
\n        { }
\r        { }
"//".*    { }
ASSERT    { return(ASSERT); }
```

```
BRANCH     { return(BRANCH); }
HEADER     { return(HEADER); }
INIT       { return(INIT); }
INPUT      { return(INPUT); }
INSIST     { return(INSIST); }
JUMP       { return(JUMP); }
NEXT       { return(NEXT); }
SAFE       { return(SAFE); }
SEED       { return(SEED); }
SLEEP      { return(SLEEP); }
STATE      { return(STATE); }
STORAGE    { return(STORAGE); }
SYSTEM     { return(SYSTEM); }
TEST       { return(TEST); }
THIS       { return(THIS); }
USE        { return(USE); }
UTILITY    { return(UTILITY); }
{id}       { return(ID); }
{int}      { return(INT); }
.          { return yytext[0]; }
```

Overall structure

Second, we present the overall hierarchical structure of the rule file by using a context-free grammar formalism. Some of the terminal symbols (i.e. the uppercase ones) of the grammar appeared already in the token structure definition above.

```
rulebase    ::=   header storage system utility
header      ::=   HEADER '{' form0 '}'
form0       ::=   form form0
            ::=
form        ::=   ID '(' id0 ')' '=' const ';'
            ::=   ID '{' fseed fstate fother0 '}'
item0       ::=   item1
            ::=
item1       ::=   item ',' item1
            ::=
item        ::=   STATE
            ::=   INPUT
            ::=   THIS
            ::=   NEXT
            ::=   ID
```

```
id0         ::=   id1
            ::=
id1         ::=   ID ',' id1
            ::=   ID
fseed       ::=   SEED '=' const ';'
fstate      ::=   STATE '=' const ';'
fother0     ::=   fother fother0
            ::=
fother      ::=   ID '=' const ';'
storage     ::=   STORAGE '{' class0 '}'
class0      ::=   class class0
            ::=
class       ::=   ID '[' const ']' ';'
system      ::=   SYSTEM '{' instr0 '}'
instr0      ::=   instr instr0
            ::=
utility     ::=   UTILITY '{' macrodef0 '}'
macrodef0   ::=   macrodef macrodef0
            ::=
stmt0       ::=   stmt stmt0
            ::=
instr       ::=   ihead '{' test '}'
macrodef    ::=   ihead '{' save test local choice0 '}'
stmt        ::=   ID ':' shead ';'
            ::=   shead ';'
ihead       ::=   ID '(' inlist outlist ')' dstlist
            ::=   dev ID '(' inlist outlist ')' dstlist
shead       ::=   ID '(' clist outlist ')' dstlist
            ::=   dev ID '(' clist outlist ')' dstlist
            ::=   INIT '(' '>' ID ')'
            ::=
dev         ::=   JUMP
            ::=   BRANCH
            ::=   SLEEP
inlist      ::=   id0
clist       ::=   const0
outlist     ::=   '>' id1
            ::=
dstlist     ::=   '[' id1 ']'
            ::=
save        ::=   SAFE savedecl1 ';'
            ::=
savedecl1   ::=   savedecl ',' savedecl1
```

```
             ::=   savedecl
savedecl     ::=   ID '<' const '>'
test         ::=   TEST const ';'
             ::=
local        ::=   USE tempdecl1 ';'
             ::=
tempdecl1    ::=   tempdecl ',' tempdecl1
             ::=   tempdecl
tempdecl     ::=   ID '<' const '>' '.' tempdecl
             ::=   ID '[' id1 ']'
const0       ::=   const1
             ::=
const1       ::=   const ',' const1
             ::=   const
const        ::=   ID '(' const0 ')'
             ::=   item
             ::=   integer
             ::=   dsym classid '(' item0 ')'
             ::=   '&'  '(' item0 ')'
integer      ::=   '+' integer
             ::=   '-' integer
             ::=   INT
classid      ::=   ID
             ::=
dsym         ::=   '='
             ::=   '!'
             ::=   '#'
             ::=   '?'
             ::=   '%'
choice0      ::=   choice choice0
             ::=
choice       ::=   ASSERT const ';'
             ::=   INSIST const ';'
             ::=   ID ':' test local '{' stmt0 '}'
```

B Expansion source syntax

In this appendix, we present a syntax definition for a ReFlEx expansion source, which is to be typed on the input line as a response to the ReFlEx prompt. You can absorb this definition easily if you are familiar with the syntax of `lex` and `yacc` input files. (These two programs are widely used compiler-generation tools: `lex` is a lexical analyzer generator and `yacc` is a parser generator [15].)

Token structure

First, we present the required token structure by using a formalism that closely resembles the `lex` input formalism. (Consult [15] if you are not familiar with `lex`.)

```
letter    [A-Za-z_]
digit     [0-9]
id        {letter}({letter}|{digit})*
int       {digit}({digit})*

%%

" "       { }
\t        { }
\n        { return(END_LINE); }
\r        { }
"//".*    { }
ASSERT    { return(NO_GOOD); }
BRANCH    { return(BRANCH); }
HEADER    { return(NO_GOOD); }
INIT      { return(NO_GOOD); }
INPUT     { return(NO_GOOD); }
INSIST    { return(NO_GOOD); }
JUMP      { return(JUMP); }
NEXT      { return(NO_GOOD); }
SAFE      { return(NO_GOOD); }
SEED      { return(NO_GOOD); }
SLEEP     { return(SLEEP); }
STATE     { return(NO_GOOD); }
STORAGE   { return(NO_GOOD); }
SYSTEM    { return(NO_GOOD); }
TEST      { return(NO_GOOD); }
THIS      { return(NO_GOOD); }
USE       { return(NO_GOOD); }
UTILITY   { return(NO_GOOD); }
{id}      { return(ID); }
{int}     { return(INT); }
.         { return yytext[0]; }
```

Overall structure

Second, we present the overall hierarchical structure of the expansion source by using a context-free grammar formalism. Some of the terminal symbols (i.e. the uppercase ones) of the grammar appeared already in the token structure definition above.

```
inputline  ::=   stmt free next END_LINE
stmt       ::=   ID ':' shead ';'
           ::=   shead ';'
shead      ::=   ID '(' ilist outlist ')' dstlist
           ::=   dev ID '(' ilist outlist ')' dstlist
dev        ::=   SLEEP
           ::=   BRANCH
           ::=   JUMP
ilist      ::=   const0
outlist    ::=   '>' item1
           ::=
dstlist    ::=   '[' id1 ']'
           ::=
free       ::=   '{' item0 '}'
           ::=
next       ::=   '>' ID
           ::=
const0     ::=   const1
           ::=
const1     ::=   const ',' const1
           ::=   const
const      ::=   item
           ::=   integer
integer    ::=   '+' integer
           ::=   '-' integer
           ::=   INT
item0      ::=   item1
           ::=
item1      ::=   item ',' item1
           ::=   item
item       ::=   ID '[' integer ']'
           ::=   ID
id1        ::=   ID ',' id1
           ::=   ID
```

C A sample code generator

This appendix presents a self-contained ReFlEx rule file. It can be seen as a simple code generator for the same simple hypothetical processor architecture that is dealt with in the examples in the text. (See Section 10.2 for obtaining an electronic copy of this rule file.)

At this point, you might be interested in trying to solve a non-trivial macro writing exercise. Therefore, we ask you to extend the rule file as follows:

- Write a macro `left_shift(s,n > d)` that transfers data from an arbitrary source `s` into an arbitrary destination `d` and, most of all, simultaneously shifts the data to the left by `n` bit positions (you can require that `n` represents a non-negative integer—and not a cell). In other words, the macro should implement the C language expression `d = s << n`.

We would be pleased to answer any questions concerning this exercise. (Again, see Section 10.2 for a pointer to our contact information.)

```
// ******************************
// **  A ReFlEx 1.0 rule file  **
// ******************************

HEADER {

Abort(x,y,z) = _Abort(x,y,z);
If(x,y,z) = _If(x,y,z);

Add(x,y) = _Add(x,y);
BNand(x,y) = _BNand(x,y);
BShl(x,y) =  _BShl(x,y);
Div(x,y) =  If(y,_Div(x,y),Abort(100,x,y));
Lt(x,y) = _Lt(x,y);
Mul(x,y) = _Mul(x,y);
Nand(x,y) = _Nand(x,y);

Not(x) = Nand(x,x);
BNot(x) = BNand(x,x);

Lte(x,y) = Not(Lt(y,x));
Gt(x,y) = Lt(y,x);
Gte(x,y) = Lte(y,x);
Eq(x,y) = And(Gte(x,y),Gte(y,x));
```

```
Neq(x,y) = Not(Eq(x,y));

Sub(x,y) = Add(x,Add(BNot(y),1));

And {
  SEED = 1;
  STATE = If(STATE,INPUT,0);
}

Or {
  SEED = 0;
  STATE = If(STATE,1,INPUT);
}

Const {
  SEED = 1;
  STATE = And(STATE,Not(?(INPUT)),Not(&(INPUT)));
}

Equal {
  SEED = 1;
  STATE = If(STATE,
             If(later,
                And(Const(INPUT),Eq(value,INPUT)),
                Const(INPUT)),
             0);
  later = 1;
  value = If(later,value,INPUT);
}

Type(x) = If(?A(x),0,
             If(?R(x),1,
                If(?M(x),2,
                   If(Const(x),3,4))));
Ordered(x,y) = If(Const(x,y),
                  Lte(x,y),
                  Lte(Type(x),Type(y)));

} // HEADER

STORAGE {
  M[1024];
  R[4];
  A[1];
}

SYSTEM {
  set(c > r) {
    TEST And(?R(r), Const(c), Gte(c,-1024), Lte(c,1023));
  }
  load(m > r) { TEST And(?R(r), ?M(m)); }
  store(r > m) { TEST And(?R(r), ?M(m)); }
  move(s > d) {
    TEST And(Or(?A(s), ?R(s)), Or(?A(d),?R(d)));
  }
  add(a,r > a) { TEST And(?A(a), ?R(r)); }
  sub(a,r > a) { TEST And(?A(a), ?R(r)); }
  JUMP goto() [l] { }
  BRANCH eq(a) [l] { TEST ?A(a); }
  BRANCH gt(a) [l] { TEST ?A(a); }
  BRANCH lt(a) [l] { TEST ?A(a); }
}

UTILITY {

my_null(x > x) {
    // do not do anything
  null: { }
}

my_move(s > d) {
    // move data from s into d
  same: TEST =(s,d); { my_null(s > d); }
  as_set: { set(s > d); }
  INSIST Not(And(Const(s),?R(d)));
  as_load: { load(s > d); }
  as_store: { store(s > d); }
  as_move: { move(s > d); }
  clear_acc: TEST Equal(s,0); USE R[r]; {
    INIT( > r); move(r > d); sub(d,r > d);
  }
  temp_is_needed: USE R[r]; {
    my_move(s > r); my_move(r > d);
  }
}
```

```
my_snull(x > x) {
    // do not do anything;
    // safeguard x
  SAFE x<1>;
  null: { }
}

my_smove(s > d) {
    // move data from s into d;
    // if s and d belong to a common class,
    // they are forced to be aliases of each other
  SAFE d<%(s,d)>;
  TEST Or(=(s,d),Not(%(s,d)));
  null: { my_snull(s > d); }
  move: { my_move(s > d); }
}

my_double(x > y) {
    // y is set to the value of x multiplied by 2
  USE R[r], A[a];
  const: TEST Const(x); { my_move(Add(x,x) > y); }
  acc: TEST ?A(x); {
    my_move(x > r); add(x,r > x); my_move(x > y);
  }
  default: {
    my_smove(x > r); my_move(r > a);
    add(a,r > a); my_move(a > y);
  }
}

my_swap(x,y > y,x) {
    // swap the contents of x and y
  body: USE A<And(?M(x,y),Lt(#R(),2))>
            .A<And(?R(x,y),Lt(#R(),3))>
            .R[t];
  {
    my_move(x > t); my_move(y > x); my_move(t > y);
  }
}

BRANCH my_zero(x) [l] {
    // is x zero?
  next: TEST &(l, NEXT); { }
  zero: TEST Equal(x,0); { JUMP goto() [l]; }
  const: TEST Const(x); { }
  default: USE A[a]; { my_smove(x > a); BRANCH eq(a) [l]; }
}

BRANCH my_pos(x) [l] {
    // is x positive?
  next: TEST &(l, NEXT); { }
  pos: TEST Gt(x,0); { JUMP goto() [l]; }
  const: TEST Const(x); { }
  default: USE A[a]; { my_smove(x > a); BRANCH gt(a) [l]; }
}

BRANCH my_neg(x) [l] {
    // is x negative?
  next: TEST &(l, NEXT); { }
  neg: TEST Lt(x,0); { JUMP goto() [l]; }
  const: TEST Const(x); { }
  default: USE A[a]; { my_smove(x > a); BRANCH lt(a) [l]; }
}

BRANCH my_gt(x,y) [l] {
    // is x greater than y?
  TEST Or(#A(),&(l,NEXT),=(x,y),Const(x,y),
          And(?A(x),Equal(y,0)),And(?A(y),Equal(x,0)));
  USE A[a], R[r];
  swap: TEST Not(Ordered(x,y)); { BRANCH my_lt(y,x) [l]; }
  INSIST Ordered(x,y);
  next: TEST &(l,NEXT); { }
  same: TEST =(x,y); { }
  const: TEST Const(x,y); { }
  y_zero: TEST Equal(y,0); {
    my_smove(x > a); BRANCH gt(a) [l];
  }
  x_to_acc: {
    my_smove(x > a); my_smove(y > r);
    sub(a,r > a); BRANCH gt(a) [l];
  }
  y_to_acc: {
    my_smove(y > a); my_smove(x > r);
    sub(a,r > a); BRANCH lt(a) [l];
  }
}
```

```
BRANCH my_lt(x,y) [l] {
    // is x less than y?
  TEST Or(#A(),&(l,NEXT),=(x,y),Const(x,y),
          And(?A(x),Equal(y,0)),And(?A(y),Equal(x,0)));
  USE A[a], R[r];
  swap: TEST Not(Ordered(x,y)); { BRANCH my_gt(y,x) [l]; }
  INSIST Ordered(x,y);
  next: TEST &(l,NEXT); { }
  same: TEST =(x,y); { }
  const_eq: TEST Equal(x,y); { }
  const: TEST Const(x,y); { JUMP goto() [l]; }
  y_zero: TEST Equal(y,0); {
    my_smove(x > a); BRANCH lt(a) [l];
  }
  x_to_acc: {
    my_smove(x > a); my_smove(y > r);
    sub(a,r > a); BRANCH lt(a) [l];
  }
  y_to_acc: {
    my_smove(y > a); my_smove(x > r);
    sub(a,r > a); BRANCH gt(a) [l];
  }
}

my_add(x,y > z) {
    // add x to y, and put the result into z
  TEST Or(#A(),Const(x,y),Equal(y,0),=(x,y));
  USE A[a], R[r];
  swap: TEST Not(Ordered(x,y)); { my_add(y,x > z); }
  INSIST Ordered(x,y);
  both_const: TEST Const(x,y); { my_move(Add(x,y) > z); }
  one_zero: TEST Equal(y,0); { my_smove(x > z); }
  same: TEST =(x,y); { my_double(x > z); }
  default: {
    my_smove(x > a); my_smove(y > r);
    add(a,r > a); my_smove(a > z);
  }
}

my_sub(x,y > z) {
    // subtract y from x, and put the result into z
  USE A[a], R[r];
  both_const: TEST Const(x,y); { my_move(Sub(x,y) > z); }
  same: TEST =(x,y); { my_move(0 > z); }
  y_zero: TEST Equal(y,0); { my_smove(x > z); }
  y_reg_x_zero: TEST And(?R(y),Equal(x,0)); {
    my_move(y > a); sub(a,y > a);
    sub(a,y > a); my_smove(a > z);
  }
  y_acc_x_zero: TEST And(?A(y),Equal(x,0)); {
    my_move(y > r); sub(y,r > y);
    sub(y,r > y); my_smove(y > z);
  }
  y_acc_x_aux_only_free:
    TEST And(?A(y),?R(x),!(x),Equal(#R(),1));
  {
    sub(y,x > y); my_move(y > x);
    sub(y,x > y); sub(y,x > y); my_smove(y > z);
  }
  y_acc_1_aux_free:
    TEST And(?A(y),Or(?M(x),Const(x)),Equal(#R(),1));
  {
    my_move(x > r); sub(y,r > y); my_move(y > r);
    sub(y,r > y); sub(y,r > y); my_smove(y > z);
  }
  default1: {
    my_smove(x > a); my_smove(y > r);
    sub(a,r > a); my_smove(a > z);
  }
  default2: {
    my_smove(y > r); my_smove(x > a);
    sub(a,r > a); my_smove(a > z);
  }
}

BRANCH my_eq(x,y) [l] {
    // are x and y equal?
  TEST Or(#A(),&(l,NEXT),Equal(x,y),Const(x,y),=(x,y));
  next: TEST &(l,NEXT); { }
  eq_const: TEST Equal(x,y); { JUMP goto() [l]; }
  neq_const: TEST Const(x,y); { }
  same: TEST =(x,y); { JUMP goto() [l]; }
  swap: TEST Not(Ordered(x,y)); { BRANCH my_eq(y,x) [l]; }
  INSIST Ordered(x,y);
  default: USE A[a]; {
      my_sub(x,y > a); BRANCH eq(a) [l];
    }
  }

  } // UTILITY
```

Acknowledgements

The author wishes to thank Professor Leo Ojala for his firm interest in the subject and for the arrangements that have made this work possible. During the implementation and documentation phases of the work, Dr. Johan Lilius and Dr. Ilkka Niemelä have provided many valuable suggestions. The financial support from Helsinki Graduate School in Computer Science and Engineering, Academy of Finland, and Foundation of Technology is also gratefully acknowledged.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading (Massachusetts, USA) 1986.

[2] G. Araujo, S. Devadas, K. Keutzer, S. Liao, S. Malik, A. Sundarsanam, S. Tjiang, and A. Wang. Challenges in code generation for embedded processors. In [17], pp. 48–64.

[3] AT&T Microelectronics. *DSP1610 Digital Signal Processor Information Manual.* Allentown (Pennsylvania, USA), December 1992.

[4] P. J. Brown. *Macro Processors and Techniques for Portable Software.* Wiley, London (UK), 1974.

[5] A. J. Cole. *Macro Processors* (second edition). Cambridge University Press, Cambridge (UK), 1981.

[6] A. Dollas and J. D. S. Babcock. Rapid prototyping of microelectronic systems. In M. C. Yovits and M. Zelkowitz (eds.), *Advances in Computers*, vol. 40. Academic Press, San Diego (California, USA), 1995.

[7] D. J. Farber. A survey of the systematic use of macros in systems building. *ACM SIGPLAN Notices*, vol. 6, no. 9, pp. 29–36. October 1971.

[8] J. A. Fisher. The VLIW machine: a multiprocessor for compiling scientific code. *IEEE Computer*, vol. 17, no. 7, pp. 44–53. July 1984.

[9] I. D. Greenwald. A technique for handling macro instructions. *Communications of the ACM*, vol. 2, no. 11, pp. 21–22. November 1959.

[10] S. M. Kafka. An assembly source level global compacter for digital signal processors. *Proceedings of 1990 International Conference on Acoustics, Speech, and Signal Processing*, vol. 2, pp. 1061–1064. IEEE, Piscataway (New Jersey, USA), 1990.

[11] K. Kennedy. A survey of data flow analysis techniques. In S. S. Muchnik and N. D. Jones (eds.), *Program Flow Analysis: Theory and Applications*, pp. 5–54. Prentice-Hall, Englewood Cliffs (New Jersey, USA), 1981.

[12] B. W. Kernighan and D. M. Ritchie. *The C Programming Language* (second edition). Prentice-Hall, Englewood Cliffs (New Jersey, USA) 1988.

[13] J. R. Larus. Assemblers, linkers, and the SPIM simulator. In J. L. Hennessy and D. A. Patterson, *Computer Organization and Design: The Hardware/Software Interface*, Appendix A. Morgan Kaufmann, San Mateo (California, USA) 1994.

[14] E. A. Lee. Programmable DSP architectures. *IEEE ASSP Magazine*, vol. 5, no. 4, pp. 4–19 (Part I) and vol. 6, no. 1, pp. 4–14 (Part II). October 1988 and January 1989.

[15] J. R. Levine, T. Mason, and D. Brown. *Lex & yacc* (second edition). O'Reilly & Associates, Sebastopol (California, USA) 1992.

[16] P. Marwedel. Code generation for embedded processors: an introduction. In [17], pp. 14–31.

[17] P. Marwedel and G. Goossens (eds.). *Code Generation for Embedded Processors*. Kluwer, Boston (Massachusetts, USA) 1995.

[18] M. D. McIlroy. Macro instruction extensions of compiler languages. *Communications of the ACM*, vol. 3, no. 4, pp. 214–220. April 1960.

[19] D. A. Patterson. Reduced instruction set computers. *Communications of the ACM*, vol. 28, no. 1, pp. 8–21. January 1985.

[20] D. Salomon. *Assemblers and Loaders*. Ellis Horwood, Chichester (UK) 1992.

[21] B. Stroustrup. *The C++ Programming Language* (second edition). Addison-Wesley, Reading (Massachusetts, USA) 1991.

[22] T. Swan. *Mastering Turbo Assembler* (second edition). Sams Publishing, Indianapolis (Indiana, USA), 1995.

[23] W. A. Wulf. Compilers and computer architecture. *IEEE Computer*, vol. 14, no. 7, pp. 41–47. July 1981.