Explicit Parallel Programming: System Description

By Jim Gamble and Calvin J. Ribbens

TR 91-20

Explicit Parallel Programming: System Description*

Jim Gamble† and Calvin J. Ribbens
Department of Computer Science
Virginia Polytechnic Institute & State University

July 23, 1991

Abstract

The implementation of the Explicit Parallel Programming (EPP) system is described. EPP is a prototype implementation of a language for writing parallel programs for shared memory multiprocessors. EPP may be viewed as a coordination language, since it is used to define the sequencing or ordering of various tasks, while the tasks themselves are defined in some other compilable language. The two main components of the EPP system—a compiler and an executive—are described in this report. An appendix is included which contains the grammar defining the EPP language as well as templates used by the compiler in code generation.

1 Introduction

The purpose of this report is to describe the Explicit Parallel Programming (EPP) language and system from the point of view of the system designer and implementor. A companion report [Gamb 91] describes EPP from a user's point of view. See also [Gamb 90] for a complete discussion of the goals and findings of the EPP project. The remainder of this report is organized into two main sections, the first describing the EPP compiler and the second the EPP executive. Regarding the compiler, we devote a separate subsection to each of the three typical components of translation: lexical analysis, parsing, and code generation. The section on the executive summarizes the basic tasks of the executive and then discusses in more detail the special issue of memory management.

2 EPP compiler

2.1 Lexical analysis

EPP performs lexical analysis with a combination of finite state machines (yes, plural), internal program states, and through interfacing with parser data structures. Because EPP usually has more than one language to process, it has one finite state machine used to process EPP code, and a second finite state machine which processes generic source code. Both finite state machines convert an input stream of characters into tokens and perform an initial typing of the token. The finite state machine which parses EPP code has only 10 states. The small number of states is possible because EPP restricts itself to logic sequence—the most structured and least variable part of any

*This work was supported in part by Department of Energy grant DE-FG05-88ER25068.
†First author's current address: IBM FSD, 9500 Godwin Drive, Manassas, VA, 22110.

language. The second, or source code, finite state machine has only 4 states. It is even smaller than the EPP finite state machine because no analysis is performed on the tokens returned. Thus, many of the tokens returned have the type **ascii** for ascii characters which might be illegal in EPP, or **white** for white space. The second finite state machine has no error states; its job is simply to pass all input on for later processing.

Certain lexemes in EPP can be defined to have a special type in the program. This is not unusual in compilers, since languages like C and Pascal, among others, allow the user to define variable types which may later be used to declare data items exactly as if the defined types were standard, predefined types. It is perhaps a subtle point that the lexeme thereafter is typed differently than it would be earlier in the program. This feature gives the EPP language added flexibility.

The finite state machine gives a token its initial type. Four types of tokens are redefined by a subsequent typing. As with many languages, a sequence of letters and/or digits which begins with a letter is typed as an identifier. Keywords are initially identified as identifiers, and must be recognized as a keyword during retyping. Keywords are determined by a table look-up, which makes the compiler more modular. Changes and additions to the list of keywords are done by changing a table instead of reprogramming the finite state machine.

The EPP compiler may also redefine an identifier to be a source-identifier or an EPP-identifier. These tokens represent names which were declared in the references section, and therefore may be called from the current program. Source-identifiers call procedures written in source language. EPP-identifiers call procedures written in EPP. The distinction is important since parsing proceeds differently for each case. In the former case, the parser will be processing the argument list as source code; in the latter case, the parser continues in EPP mode.

As with the languages mentioned before, EPP allows types to be defined. Unlike the languages mentioned before, defined types in EPP are not defined by combinations of other types. One reason for this is that EPP recognizes no predefined types. In order to be as flexible as possible, types are defined simply by size and bounding (i.e., full word, double word). No other information is needed, since further information only concerns semantic issues. The size and bounding are needed, however, since EPP does allocate memory space.

Source code, which is an indispensable part of EPP programs, may appear in a few well delimited contexts. At one point in the development of EPP, source code was allowed to appear without delimiters, believing that the context would be sufficient for distinguishing between the two. In practice, it was a simple matter to identify the start of source code; however, it was difficult to reliably find the end of the source code. While the parser processes EPP code, it knows where to expect source code. While parsing source code, the parser is really lost since it knows nothing of the structure of the source language, nor can it rely on EPP keywords being unique—many are not. So there was a choice. Either prepend every EPP statement with some unique keyword, or delimit the source code. The latter option was chosen as a less intrusive and less cumbersome option.

Source code may appear as a type declaration string, in which case it is delimited by quotation marks. The delimiters allow types identified by unusual combinations of characters to be used. Examples are "CHARACTER*12", or "DOUBLE PRECISION". Complicated types, such as might define a record structure, can be handled neatly with constants.

Another context for source code is as controls and conditions for dynamic control structures. Structures in this category are the **if** statement, the **loop** statement, the **until** statement and the **clone** statement. The delimiters for the source code in these structures are parentheses, chosen because many languages use parentheses to delimit logical conditions. Source code which is parsed in this context is given the lexical type **expressn** meaning "expression".

Another place which is a context for source code is as the argument list for a source identifier. As with dynamic structures, source code in an argument list follows the tradition of being delimited by parentheses. Identifiers are passed in the manner used by the source language. In FORTRAN, this means that identifiers are passed by reference. The entire argument list is returned as a single token and given the type **arglist**.

Finally, source code may appear as an invocation—a program unto itself. In such cases, the source code is delimited by EPP keywords **source** and **epp**. The **epp** keyword is not necessary if the source code immediately precedes an **eppend**. Within source programs, the keywords **notord**, **serial**, **concur**, and **eppend** may be used to group source code statements into subtasks. The entire source code, including line breaks and white space, is returned as a token having the type **source**.

In all instances in which source code appears, it is given a special type and is passed in its entirety to the parser, from which it will be included in an output source file. EPP has made provision to handle tokens of any arbitrary length, so that long occurrences of source code are not a problem. The programmer must be responsible, however, for the syntactic correctness of all source code. In the case of FORTRAN, this will mean that statements must appear in columns 7 to 72, and continuation lines must have a continuation character in column 6.

2.2 SLR parser

The EPP compiler uses a simple LR (or SLR) parser because the EPP language is described by a context free grammar, and because the mechanism for parsing is a simple push-down automaton—fast and easy to program. The context-free grammar for EPP has approximately 90 productions, and approximately 80 symbols. None of the productions pertain to the parsing of expressions, which permits the grammar to be as comparatively small as it is.

In truth, the language is not context-free because it has the classic if-else ambiguity, in which an **else** following nested **if** statements could logically be paired with more than one **if**. Following the lead of this classic problem, EPP resolves the conflict by pairing **else** with the nearest **if**. The grammar used by EPP may be found listed in the Appendix.

The EPP grammar evolved over the period of time in which the EPP system programs were written. The grammar has changed several times to accommodate changes in the EPP design. The process of generating parse tables from the grammar was accomplished by a program named **lranal**. The output from **lranal** is in a Pascal format, but the format is regular enough that a few UNIX tools (**awk** and **sed**) can produce the tables in C code automatically.

The parser is also responsible for reduction operations—operations which process information found either on the stack or in global variables. Reduction operations build data structures, track scope, build source code files, and build the logical sequence output file. The reduction operations are partly responsible for the "unnatural" grammar of EPP. For example, the EPP grammar has the following productions.

declare $\longrightarrow$ **gentype symlist**
gentype $\longrightarrow$ **usertype**

The token **usertype** is a terminal symbol in the grammar, so it would seem more logical to combine the two productions into one.

declare $\longrightarrow$ **usertype symlist**

However, this method would mean that the type is not known until a list of symbols has been reduced to a **symlist**. At that point, all the symbols represented by **symlist** would need to be reprocessesd to include the type information. It seemed simpler to have two productions, save the type information upon reducing a **usertype** to a **gentype** (an action which occurs before encountering any symbols in the list), and process each symbol fully when the symbol is encountered.

2.3 Code generation

The internal representation of the EPP program, as constructed by the parser, is a hierarchical data structure dominated by a task list. Each task may optionally have a list of data areas, a tree structure of type definitions, and a tree structure of external references. The data areas each have a tree structure of data items (variables, arrays). The structure of the internal representation is shown in Figure 1. As the compiler parses tasks, new task structures are added to and deleted from the task list. Current scope is defined by the task list so that the first encounter of an identifier in following the task list from the current task to the highest level task is taken as the intended reference.

The EPP compiler actually produces two types of output code. The first type is text program statements of FORTRAN (or other languages, to be implemented later). This output represents the semantic actions of the program. The second type of output represents the logic sequence of the program. In form, this part of the output is a file of data structures, linked hierarchically by pointers.

FORTRAN text is generated with several fill-in-the-blank templates. A separate template exists for handling the different types of source code. The **loop** has its own template for testing and incrementing loop counters. Similarly, the **if**, the **clone**, and the **until** have their own templates. All the templates include code which enact data associations before giving program control to user code. The templates and examples of their use appear in the Appendix.

The data association is a necessary part of each template, and the details of how it works will be given later in this section. The data association is only one-half of the trick, however. With a data association, the program has access to an amorphous piece of physical memory. In order to function in the program, it must have a structure imposed upon it. The compiler does this in a two-step process.

In the first step, the compiler reads through the list of data areas which are a part of the task environment. The data structure which identifies the appropriate data areas is built by reduction operations during compilation. As the program processes the list, it writes variable and array declarations to the text file. In this way, the semantics of type declarations are passed to another compiler.

In the second step, the compiler rereads the same list, but this time generates a series of **equivalence** statements, one per data item. The equivalences are written so that each data item is correctly offset into the data area. An interesting aside is that the same identifier can be assigned a different offset in two distinct programs, yet still reference the same piece of physical memory. The feature responsible for this flexibility also allows that same data area to have different sizes in two distinct programs.

The logic sequence in the data file is described by two major data structures—the process and the invocation. These two structures relate somewhat like a direct object and verb, respectively. A process is composed of zero or more invocations which are ordered according to the type of the process. For example, a **loop** process has one invocation which is executed zero or more times. A **notord** process has one or more invocations which may execute in any order. Each invocation is

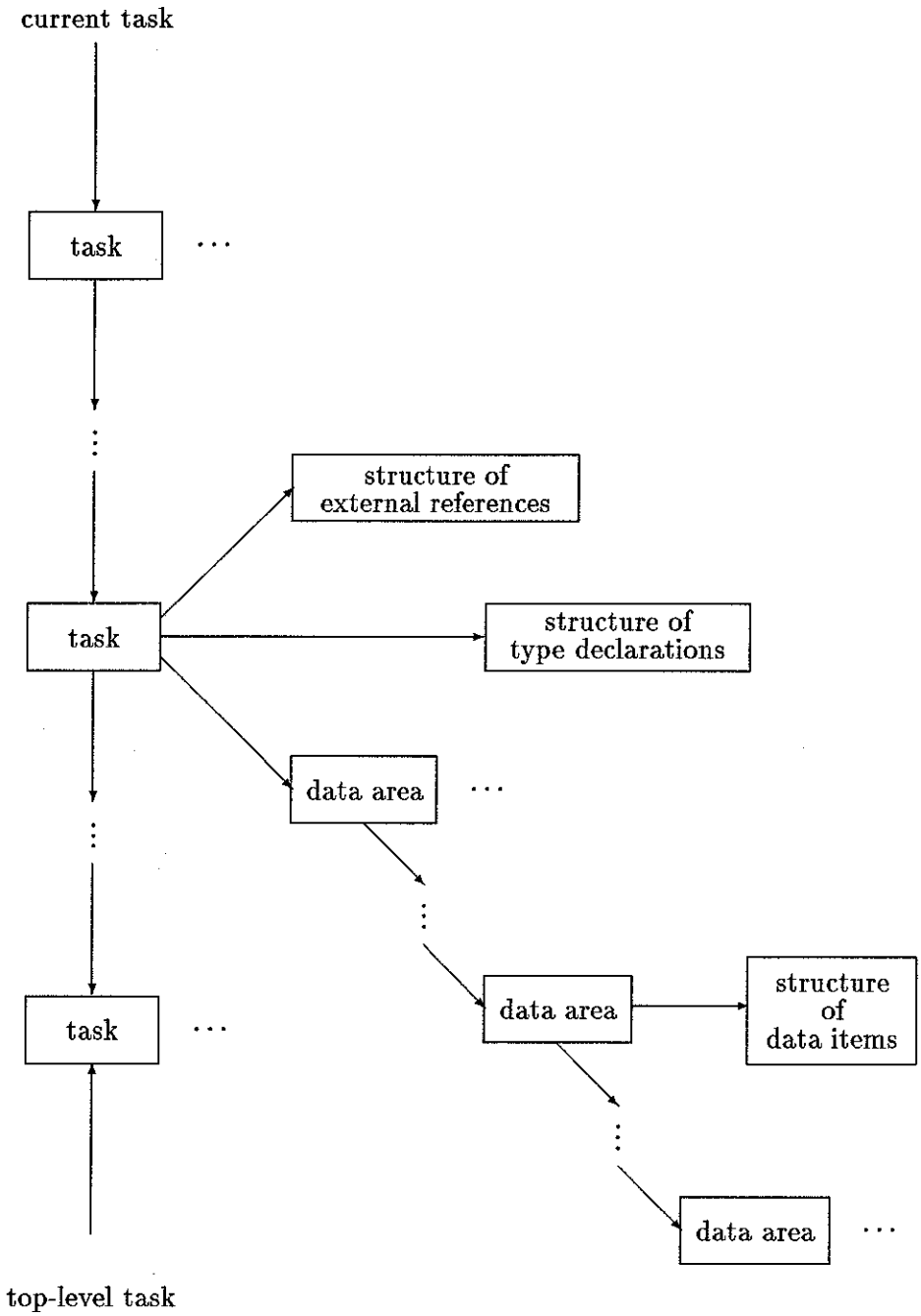

Figure 1: Internal representation of an EPP program.

linked to a single process and exists to store any arguments which are being passed to the process. A diagram of the logic sequence structure is shown in Figure 2.

The data file may also include two minor structures, which are the data area list, and the string which is used to store file names. Both structures are linked to the process structure. Certain processes have semantic actions associated with them, in which case the process links to a filename containing executable code to perform the desired actions. The data area list stores data areas which were defined in the task corresponding to that process. The executive tracks the data areas in order to give each process the proper scope at run time.

3 EPP executive

This section focuses on work done in the EPP executive. As most of the work for this project concerns language design, compiler issues will still arise in this section, however. Curiously, the executive required more time to implement than the compiler. A related distinction between the two programs is that the compiler implements theoretical designs in fairly straightforward ways, while the executive implements the solutions to difficult technical problems which are often system dependent.

3.1 Basic tasks

A discussion of the workings of the executive follows after the definitions of two relevant terms. An *environment* is a set of logical data areas and the structures imposed upon them. The *run time process tree* is a data structure used by the executive to manage processes.

The executive executes four procedures repeatedly on the run time process tree. The *queuing task* examines all tasks and queues any which may run. Queueing proceeds top-down, so that the state of a task may prevent subtasks from being queued. The *activation task* finds all queued tasks, provides each with its proper data association, and starts the task. In a full implementation, this task would also determine whether a task runs immediately, or whether it waits for a more optimal execution time—a decision which will be made by the load balancing logic. At present, all tasks are scheduled in a first come, first served manner. The *wake task* determines when tasks are officially "finished". It follows the rule that no task may be marked finished unless all subtasks are finished. The *janitor task* walks through the run time process tree and prunes any finished tasks. As tasks finish, other tasks may become ready to run. The program terminates when the run time process tree is empty.

As the executive walks the process tree, it must create the appropriate environments for each node. This is harder than one might think, since the tree is a dynamic structure. Each node has its own list of environments, since lists are deleted and augmented during the course of the program. In addition, some environments must be created locally, while others need to be accessible to many tasks. Some environments depend upon task parameters.

The executive handles all these problems by uniquely identifying each data area with an integer. Every local data area requires a new integer. Global data areas are created once, and then identified by that integer thereafter. A particular point concerns passing data areas to an EPP procedure. Upon encountering a procedure, the executive must match the argument to an actual data area, and then associate the identifying integer with the parameter. When entering a procedure, the executive also must eliminate all other environments from the scope of the task, since procedures do not inherit the scope of the calling procedure.

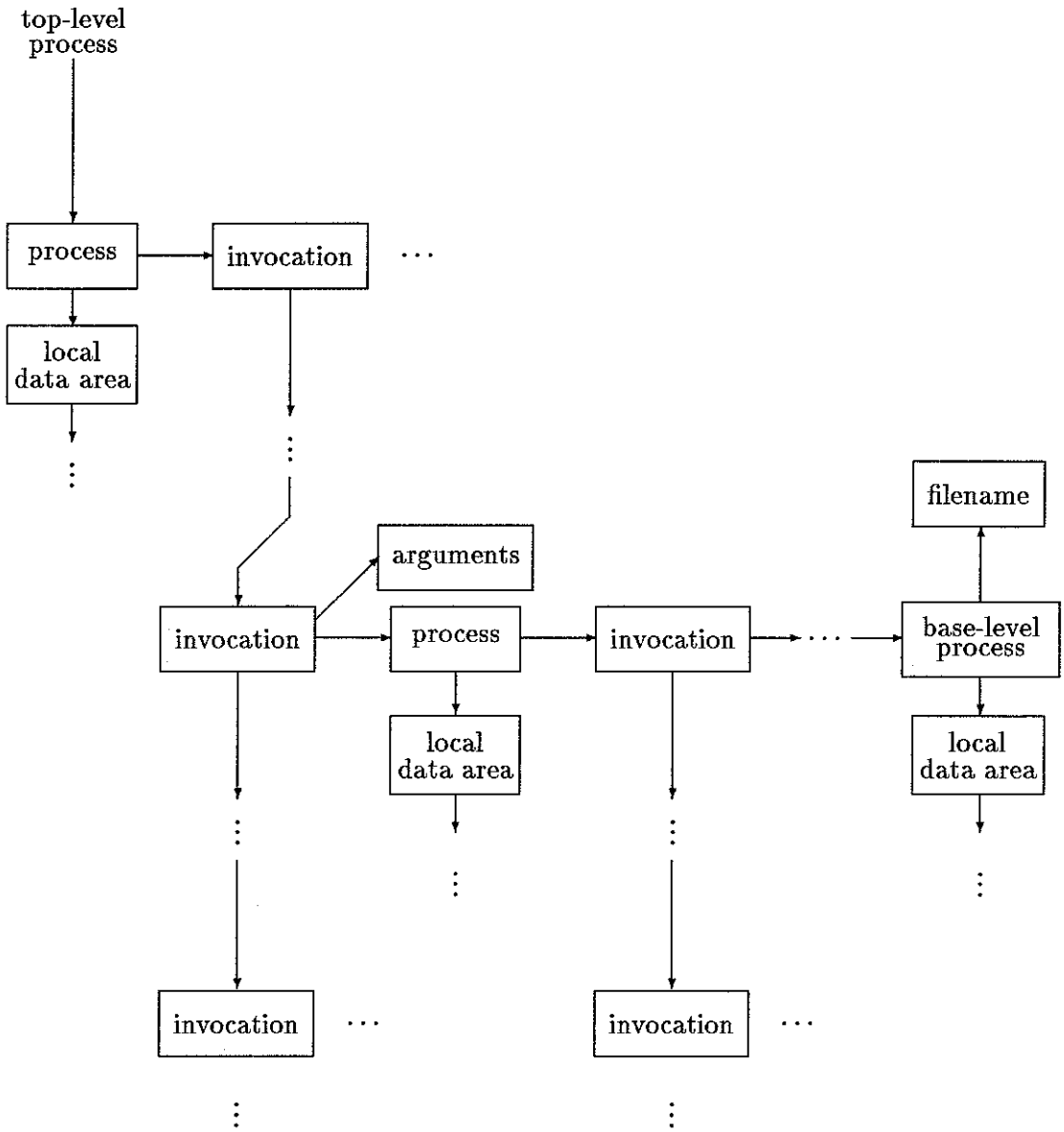

Figure 2: Structure of the logic sequence file.

The current implementation of the EPP executive requires that the current directory have a subdirectory named "epp". The EPP compiler places all task images in this directory from which they will be called by the executive when it decides to run that task. The directory also is used to store the files needed for data associations. These will be recognized by the name "AREA" followed by 4 digits. The names are generated automatically by EPP, and have no relation to any identifiers in the EPP program.

3.2 Memory management

There are two memory management issues which require special attention: managing space for the process run time tree, and the implementation of data associations. The process run time tree data structure is the basis of the executive, and contains one node for each active task. As tasks complete, the executive prunes nodes from the tree and returns them to a heap space. The heap space is necessarily under explicit control of the executive (i.e., it allocates and tracks the heap space on its own) because it was discovered that **malloc**[1] was using the same part of memory that was set aside for data associations. This was not an avoidable coincidence. Whatever address is selected as the start of data association space was automatically used by the linker as the start of **malloc** space.

This section on memory management concludes with an explanation of how EPP handles data associations—how a program includes that amorphous piece of physical memory into its task space. Data areas not only accommodate data flow between cooperating processes, they facilitate communication between the executive and the independent processes. The data association requires that EPP obtain a free page of memory from the system, and then learn how to access that memory. Part of the executive's responsibility is that it be able to pass the knowledge of how to access the new page of memory to certain processes that it creates.

On the Sequent S81, where EPP was developed, we took advantage of the addressing procedure, and a system routine known as MMAP to enact data associations. (Gould/SEL computers have a similar system routine, and similar functionality would be required on other machines in order to implement EPP.) Addressing—the process of translating a logical address into a physical address—is a complicated process which involves a table known as the page table. In a simplified translation procedure, a logical address becomes a physical address by a look-up procedure in the page table. MMAP is a system routine which works by altering a task's page table to point to some free page(s) of memory. The value which MMAP enters into the page table is logically tied to a filename. Any process calling MMAP with that filename will access the same page(s) of physical memory. The file identified by the filename additionally serves as the swap space for that piece of memory. MMAP takes as arguments a filename, an offset into the file, the size of the memory to include into the task, and a start address in logical memory.

A data area can have different sizes if different structures are imposed upon it, but this is no problem for MMAP. In this case, one program would change more page table entries to accommodate the extra memory pages. The additional entries, however, would refer to new physical memory, causing no overlap with adjacent data areas in a different program. The other program has its own page table, which is not affected by changes to the first program.

Data areas of different sizes could be responsible for one identifier having different offsets in distinct files. In this case, the identifier would have to "shift up" into memory. While the offset would be different, the arguments sent to MMAP would change to accommodate the shift in

[1] **malloc** is a UNIX system routine responsible for dynamic memory allocation.

memory. MMAP would change a different page table entry in the two programs, so that the appropriate logical address in each program would reference the same piece of physical memory.

References

[Gamb 90] Jim Gamble, *Explicit Parallel Programming*, MS Thesis, Department of Computer Science, Virginia Polytechnic Institute and State University, 1990.

[Gamb 91] Jim Gamble and Calvin J. Ribbens. "Explicit Parallel Programming: user's guide", Research Report TR 91-19 Department of Computer Science, Virginia Polytechnic Institute and State University, July, 1991.

A SLR grammar

This section contains the symbols and productions of the SLR grammar used by the parser in the EPP compiler.

Terminal symbols:

eppend	notord	serial	concur	fortran
epp	source	eppconst	epptype	eppass
eppstruct	epprefs	lexuntil	lexloop	lexclone
lexinit	lexwhile	lexinc	lexindex	lexif
lexelse	lexid	usertype	lexstring	lexicon
arglist	expressn	colon	lparen	rparen
comma	equal	$		

Nonterminals symbols:

units	unit	nametype	namedarea	eppcode
constant	typespec	id_area	decl_vars	header
task	extheader	value	declare	langid
dataspex	xrefs	actions	params	gentype
symlist	assoc	beginref	refers	sequencer
invox	until	loop	clone	if
ifelse	param	var	array	beginarea
refer	invo	beguntil	loopctl	clonectl
beginif	dims	areas	initloop	continue
increment	index	area		

SLR grammar productions:

S	$\longrightarrow$	units
units	$\longrightarrow$	units unit
units	$\longrightarrow$	unit
unit	$\longrightarrow$	nametype
unit	$\longrightarrow$	namedarea
unit	$\longrightarrow$	eppcode
unit	$\longrightarrow$	constant

nametype ⟶ epptype typespec
namedarea ⟶ id_area decl_vars eppend
eppcode ⟶ header task
eppcode ⟶ extheader task
constant ⟶ eppconst lexid equal value
typespec ⟶ lexid lexicon lparen lexicon rparen lexstring
typespec ⟶ lexid lexicon lexstring
id_area ⟶ eppstruct lexid
decl_vars ⟶ decl_vars declare
decl_vars ⟶ declare
header ⟶ langid lexid colon
task ⟶ dataspex xrefs actions
task ⟶ dataspex actions
task ⟶ xrefs actions
task ⟶ actions
extheader ⟶ langid lexid lparen params rparen colon
value ⟶ lexicon
value ⟶ lexstring
declare ⟶ gentype symlist
langid ⟶ fortran
langid ⟶ epp
dataspex ⟶ dataspex nametype
dataspex ⟶ dataspex assoc
dataspex ⟶ nametype
dataspex ⟶ assoc
xrefs ⟶ beginref refers eppend
actions ⟶ sequencer invox eppend
actions ⟶ until
actions ⟶ loop
actions ⟶ clone
actions ⟶ if
actions ⟶ ifelse
params ⟶ params comma param
params ⟶ param
gentype ⟶ usertype
symlist ⟶ symlist comma var
symlist ⟶ symlist comma array
symlist ⟶ var
symlist ⟶ array
assoc ⟶ eppass lexid
assoc ⟶ beginarea decl_vars eppend
assoc ⟶ id_area decl_vars eppend
beginref ⟶ epprefs
refers ⟶ refers refer
refers ⟶ refer
sequencer ⟶ notord

sequencer ⟶ serial
sequencer ⟶ concur
invox ⟶ invox invo
invox ⟶ invo
until ⟶ beguntil invo lexuntil lparen expressn rparen
loop ⟶ loopctl invo
clone ⟶ clonectl invo
if ⟶ beginif invo
ifelse ⟶ beginif invo lexelse invo
param ⟶ lexid
var ⟶ lexid
array ⟶ lexid lparen dims rparen
beginarea ⟶ eppstruct
refer ⟶ langid lexid lexstring
refer ⟶ langid lexid
invo ⟶ lexid lparen arglist rparen
invo ⟶ lexid lparen areas rparen
invo ⟶ lexid lparen rparen
invo ⟶ task
invo ⟶ source
beguntil ⟶ lexloop
loopctl ⟶ initloop continue increment lexloop
loopctl ⟶ initloop continue lexloop
loopctl ⟶ continue increment lexloop
loopctl ⟶ continue lexloop
clonectl ⟶ initloop continue increment index lexclone
beginif ⟶ lexif lparen expressn rparen
dims ⟶ dims comma lexicon
dims ⟶ lexicon
areas ⟶ areas comma area
areas ⟶ area
initloop ⟶ lexinit lparen expressn rparen
continue ⟶ lexwhile lparen expressn rparen
increment ⟶ lexinc lparen expressn rparen
index ⟶ lexindex lparen usertype lexid rparen
area ⟶ lexid

B Source code templates

This appendix contains the source code templates mentioned in Section 2.3. All templates are included, with boxed pseudo-code to indicate where the compiler includes portions of user code. The templates are presented in the order of data association code, **loop** code, **until** code, **clone** code, and **if** code.

The data association code (see Figure 3) permits no user code. Its parameters are read at run time. The function of this particular template is to call the procedures which in turn call MMAP to enact the data associations.

```
character*80 epp1
integer epp2
integer epp3
character*1 epp0(0:1)
common /eppblock/ epp0
integer iargc

if (iargc() .eq.  3) then
    call getarg(2, epp1)
    read(epp1, *) epp2
    call getarg(3, epp1)
    read(epp1, *) epp3
    call getarg(1, epp1)
    call make_cstring(epp1, 80)
    call environment(epp1, epp2, epp3)
else
    stop
endif
```

Figure 3: Data association template.

All dynamic control templates invoke special interface routines which ensure synchronous exchanges of information between the control process and the executive. Five routines are responsible for the synchronization: `proc_putjunk`, `proc_get_sig1`, `proc_get_sig2`, `proc_set_sig1`, `proc_set_sig2`. The procedure `proc_putjunk` writes a value to the junk space of the calling process. The procedures `proc_get_sig1`, and `proc_get_sig2` retrieve the value of the two flags `sig1` and `sig2`, respectively, for the calling process. `Proc_set_sig1`, and `proc_set_sig2` set values for their respective flags.

The **loop** template is found in Figure 4. The **clone** template follows in Figure 5. The template for **until** is in Figure 6. Finally, the template for **if** and **if-else** appears in Figure 7.

```
logical epp4
integer epp5
integer epp6

source code which is executed once before the loop starts
epp4 = continuation condition
if (epp4) then
    call proc_putjunk(1)
else
    call proc_putjunk(0)
endif
call proc_set_sig1()
call proc_set_sig2()
do while (epp4)
    call proc_get_sig1(epp5)
    call proc_get_sig2(epp6)
    do while ((epp5 .eq.  0) .or.  (epp6 .eq.  1))
        call sleep(0)
        call proc_get_sig1(epp5)
        call proc_get_sig2(epp6)
    enddo
    source code which executes after every iteration
    epp4 = continuation condition
    if (epp4) then
        call proc_putjunk(1)
    else
        call proc_putjunk(0)
    endif
    call proc_set_sig1()
    call proc_set_sig2()
enddo
```

Figure 4: **Loop** template.

```
logical epp4
integer epp5
integer epp6

source code which is executed once before the loop starts
epp4 = continuation condition
if (epp4) then
    call proc_putjunk(index variable)
    call proc_set_sig1()
    call proc_set_sig2()
endif
do while (epp4)
    call proc_get_sig1(epp5)
    call proc_get_sig2(epp6)
    do while ((epp5 .eq.  0) .or.  (epp6 .eq.  1))
        call sleep(0)
        call proc_get_sig1(epp5)
        call proc_get_sig2(epp6)
    enddo
    source code which executes after every iteration
    epp4 = continuation condition
    if (epp4) then
        call proc_putjunk(index variable)
        call proc_set_sig1()
        call proc_set_sig2()
    endif
enddo
```

Figure 5: **Clone** template.

```
logical epp4
integer epp5
integer epp6

call proc_putjunk(1)
epp4 = .true.
call proc_set_sig1()
call proc_set_sig2()
do while (epp4)
    call proc_get_sig1(epp5)
    call proc_get_sig2(epp6)
    do while ((epp5 .eq.  0) .or.  (epp6 .eq.  1))
        call sleep(0)
        call proc_get_sig1(epp5)
        call proc_get_sig2(epp6)
    enddo
    epp4 = continuation condition
    epp4 = .not.  epp4
    if (epp4) then
        call proc_putjunk(1)
    else
        call proc_putjunk(0)
    endif
    call proc_set_sig1()
    call proc_set_sig2()
enddo
```

Figure 6: **Until** template

```
logical epp4
integer epp5
integer epp6

epp4 = logical condition
if (epp4) then
    call proc_putjunk(1)
else
    call proc_putjunk(0)
endif
```

Figure 7: **If, If-Else** template