NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

PROJECT SCHEDULING TOOL

by

John Evans

September 1997

Advisor: Valdis Berzins
Second Reader: Luqi

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, Va 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE September, 1997	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE PROJECT SCHEDULING TOOL			5. FUNDING NUMBERS
6. AUTHORS Evans, John			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE

13. ABSTRACT(*maximum 200 words*)

Optimally scheduling a team of developers on a large software project is an NP-complete problem. The scheduling algorithm employed by the Evolutionary Control System (ECS) portion of the Computer-Aided Prototyping System (CAPS) does near-optimal scheduling using an algorithm that runs in Order N^2 space and time. The problem addressed by this thesis is to improve the performance of the algorithm and make it more useful for scheduling software developers. The thesis accomplished three things: (1) Modified the algorithm to run in order N time and space, preserving its near-optimal behavior; (2) implemented a calendaring package that computes federal holidays for any year after 1970 and schedules tasks only on non-holiday workdays; and (3) incorporated a more realistic capability model to better match programming tasks with each developer's abilities.

14. SUBJECT TERMS Scheduling, CAPS, ECS, Project Management			15. NUMBER OF PAGES 266
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited

PROJECT SCHEDULING TOOL

John Evans
B.S., Mathematics, New York State University, 1982

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 1997

ABSTRACT

Optimally scheduling a team of developers on a large software project is an NP-complete problem. The scheduling algorithm employed by the Evolutionary Control System (ECS) portion of the Computer-Aided Prototyping System (CAPS) does near-optimal scheduling using an algorithm that runs in Order N^2 space and time. The problem addressed by this thesis is to improve the performance of the algorithm and make it more useful for scheduling software developers. The thesis accomplished three things: (1) Modified the algorithm to run in order N time and space, preserving its near-optimal behavior; (2) implemented a calendaring package that computes federal holidays for any year after 1970 and schedules tasks only on non-holiday workdays; and (3) incorporated a more realistic capability model to better match programming tasks with each developer's abilities.

DISCLAIMER

The computer programs in the Appendices are supplied on an "as is" basis, with no warrantees of any kind. The author bears no responsibility for any consequences of using these programs.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Thanks to Professors Berzins and Luqi for their time, and kind solicitude. It is an honor to have been a student of theirs. Thanks to the enlightened management of NRaD for setting up the distance-learning program, thus giving my classmates and I the opportunity to earn an advanced degree from the Naval Postgraduate School. And last, thanks to my family for allowing me to take from them the hundreds of evenings and countless weekends necessary to complete this program. My debt to them is immeasurable.

I. BACKGROUND

Much research into the formalization and automation of software development is underway. The need for such tools is obvious. It is fundamentally driven by Moore's Law, which states that the power of computer systems will double every 18 months—a maxim which has held for the past twenty years, and is expected to continue for at least the next ten. As computer systems grow inexorably faster and more powerful, new software to take advantage of this increased power is needed. The new software, however, is larger, and more complicated, and now requires larger teams of developers to produce in a timely manner. Software tools to manage the complexity of developing these larger programs are needed.

One such tool is the Evolutionary Control System (ECS) being developed at the Naval Postgraduate School (NPS). The basis of the ECS is Salah Badr's Phd. Thesis, *A Model and Algorithms for a Software Evolution Control System*[Ref. 1], which itself was based on work by Luqi[Ref. 4] of NPS.

Salah's thesis delved into a broad array of issues related to managing large projects and their concomitant complexity. One aspect of his thesis, which is the subject of this report, was the development and implementation of an on-line scheduling algorithm that did three specific tasks:

1. Supported teamwork by concurrently assigning ready steps to available designers.
2. Supported incremental replanning as additional information became available.
3. Minimized wasted design effort due to reorganization of the schedule by efficiently scheduling workers to assigned sub-tasks.

Over time, however, certain limitations have become evident. The implementation of the scheduling algorithm was found to be $O(N^2)$ in space. This led to a rapid exhaustion of memory resources on relatively small problem sets. Also, the model of time used to schedule the developers was not realistic. It assumed that the developers were available always, and did not take into account weekends, holidays, or other commitments on a developer's time. Also, the capabilities of the developers was split into just three broad categories: low, medium, and high. This too proved unrealistic, as certain developers bring their own strengths and weaknesses to the task at hand. It would be nice to take note, for instance, of a special ability such as database expertise, and assign a programmer with this capability to a task that require this knowledge. The changes made to Salah Badr's codes do exactly this.

II. THE SCHEDULER

The problem of optimally scheduling tasks for both the preemptive and nonpreemptive cases is NP-complete[Ref. 6]. Scheduling nonpreemptive tasks with arbitrary ready times is also NP-complete in both multiprocessor and uniprocessor systems[Ref. 3]. For dynamic systems with more than one task, and mutual exclusion constraints between tasks, Mok and Dertouzos[Ref. 5] showed that an optimal scheduling algorithm does not exist.

Shiah, et al.[Ref. 2] came up with an heuristic scheduling algorithm that ran in order kN time. Salah Badr extended the algorithm to consider arbitrary precedence constraints between pairs of tasks. His scheduler forms the basis of the current ECS scheduling algorithm.

The scheduling algorithm, as implemented by Badr, was recursive. It consumed order N^2 memory for a set of N tasks. It attempted to improve performance by limiting backtracking, but was still at least order N^2 in time. It was based on an algorithm described in the paper by Stankovic, et al.[Ref. 3] The requirement for order N^2 space limited the size of the problem domain. This thesis describes the algorithm and the steps taken to make the algorithm run using only order N space. It is based on the "myopic" algorithm[Ref. 2] and a radical restructuring of the data structures in the Ada code.

A. THE SCHEDULING MODEL

The task set in the ECS scheduling problem is a variable set of evolution steps $S = \{S_1, S_2, \ldots, S_N\}$, where N varies with time. This set of tasks needs to be scheduled to a set of M designers $D = \{D_1, D_2, \ldots, D_M\}$. The designers are of L different expertise levels.

Tasks as used in the ECS are independent, nonperiodic and non-preemptive. They can be characterized by the following:

1. Task arrival Time T_A;
2. Task deadline T_D;
3. Task worst-case computation time T_C;
4. Task expertise level T_L;
5. Task priority T_P

Each task also has associated with it a precedence constraint given in the form of a directed acyclic graph $G = \{S, E\}$ such that $(S_i, S_j) \in E$ implies that S_j cannot start until S_i has completed.

The priority, T_P, is a small positive integer that is assigned to each task to reflect the criticality of its deadline. The priorites of different tasks should be compatible with the precedence constraints between the steps, i.e. no lower priority step can precede a higher priority step:

$$\text{if } (S_2, S_1) \in E \Rightarrow T_P(2) >= T_P(1)$$

$$\text{if } (S_2, S_1) \in E \wedge T_P(1) >= T_P(3) \Rightarrow T_P(2) >= T_P(3)$$

B. THE SCHEDULING ALGORITHM

The goal of the scheduling algorithm is to determine if there exists a schedule for executing the tasks that satisfies the timing , precedence, and resource constraints, and to calculate such a schedule if it exists. A schedule that meets these constraints is termed *feasible*. It is not guaranteed to be optimal.

Scheduling a set of tasks to find a full feasible schedule is actually a search problem. The search space is a tree. The scheduling algorithm starts at the root of the tree, and using a predetermined heuristic, selects a candidate task to schedule. If the remaining tasks can be added to the schedule, in the order given by the heuristic, without violating the constraints, then the partial schedule is termed *strongly-feasible*, and the task is added to the search tree as a vertex node, and the process is repeated, recursively, till a full, feasible schedule is found. If instead, after the candidate task is selected, and any one of the remaining tasks added to the schedule violates the constraints, the candidate task is rejected, and the next elgible, candidate task (ordered by the ranking function $H(T)$) is selected. The search process continues untill all the tasks are scheduled, or no feasible schedule is found.

Instead of using all of the remaining tasks to determine if a partial schedule is *strongly-feasible*, Stankovic, et al.[Ref. 2], limited the candidate tasks to check to some number k. So, insteady of checking $N, N-1, \ldots, 1$ remaining tasks, or $N(N-1)/2$ total tasks, they limited the search to k or at most kN tasks to check. (This is where the term "myopic" comes in. Instead of looking at all the remaining tasks, we "near-sightedly" examine the next k tasks.)

The set of tasks ready to be scheduled are ordered by the heuristic $H(T)$. The candidate heuristics are

1. Minimum deadline first (Min_D): $H(T) = T_D$;
2. Minimum processing time first (Min_P): $H(T) = T_P$;
3. Minimum earliest start time first (Min_S): $H(T) = T_{est}$;
4. Minimum laxity first (Min_L): $H(T) = T_D - (T_{est} + T_P)$;
5. Min_D + Min_P: $H(T) = T_D + W \times T_P$;
6. Min_D + Min_S: $H(T) = T_D + W \times T_{est}$;

According to Shiah et al.[Ref. 3], The Min_D + Min_S heuristic is superior in all cases. It is supposedly used in Salah Badr's dissertation, but since his simulation studies apparently used tasks with an earliest start time of 0 it defaults to Min_D. Min_D is used in the new implementation of the scheduling algorithm.

C. ANALYSIS

The scheduler as implemented by Salah Badr in Ada was Order N-squared in space. The heart of the code was a call on a search function performing a recursive search in tree-like fashion of potential schedules. In order to make the routine

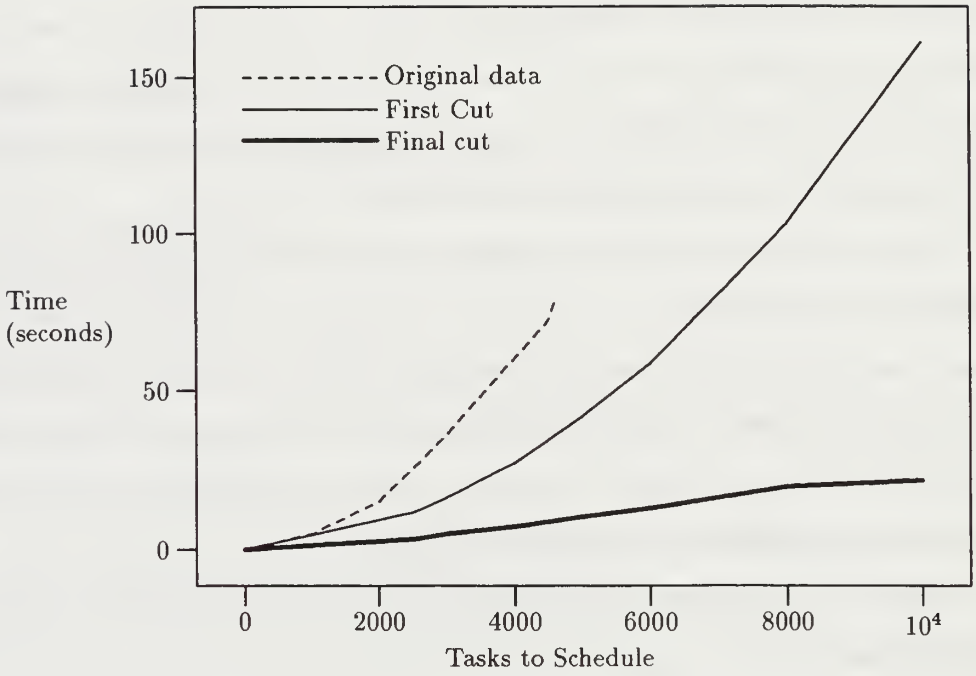


Figure 1. Plot of scheduler run-time vs. number of tasks to schedule

$O(N)$ in space it was necessary to pull many of the large data structures out of the recursive routine, make them global, and manage changes with other global data structures. This necessarily complicated the code to a degree, but the result was an $O(N)$ algorithm in space.

Once the space problem was corrected, it became evident that the routine was also $O(N^2)$ in time. But this was easily rectified by using the "myopic" algorithm. Figure 1 shows the speed-up in processing speed vs. number of tasks to be scheduled for different versions of the code. The original data came with the original code. After the N^2 space problem was resolved, and before the myopic version of the code was added (first cut) we see that the code still runs in order N^2 time. The final cut shows the run-time for the final version of the code.

The original data collected goes upto only 4600 tasks because the storage required was $O(N^2)$ in the number of tasks to be scheduled. A number larger than 4600 tasks would cause the program to raise a storage-error exception.

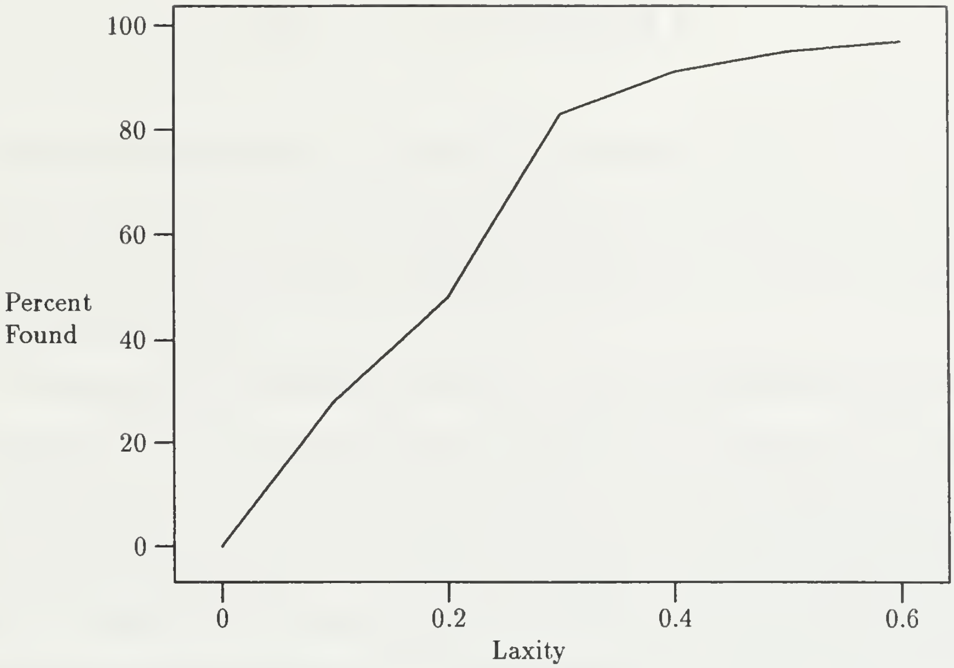


Figure 2. Plot of Laxity vs. percent schedules found

D. SIMULATION

To test the new scheduler routine, a routine to generate tasks that always have a feasible schedule was written. (Actually Badr had a routine to generate tasks, but it generated lists of tasks that were "easy" to schedule—that is the alogorithm never failed to find a schedule.) This routine varies the number of tasks, the number of programmers to use, and the "laxity" of the schedule generated. (Laxity is defined to be $T_D - (T_{est} + T_P)$.) It also uses the Ada '95 random number generators to generate uniform distributions of random variables. The graph in Figure 2 shows the performance of the algorithm when 500 tasks per test case were generated, and the laxity was varied between zero and 0.7. As you can see, the algorithm failed miserably when there was zero laxity, and got progressively better as this constraint was "relaxed."

III. CALENDAR

The scheduling algorithm as originally implemented treated time continuously. Mapping this "continuous" time to calendar working time is a tedious task, especially as the number of tasks to schedule increases. Also, real dates give a better idea of the time-frames involved.

The algorithm to translate a "continous" time to calendar time works as follows: Consider the output of the scheduler in Table I for a simple set of 10 tasks.

The first column is the task id, the second column is the expertise level required for the task (more on expertise levels, later), and the third column is the developer assigned to the task. (In this case we have three developers: L1, M1, H1.) The second to last column is the start time and the last column is the end time in units of hours.

After translating the start times and end times to calendar times we get the output in Table II For this data set the start date was set to July 3rd, 1997. The translator also assumed that the work day is eight hours. At NRaD the the work weeks are 5/4, i.e., 9 hours a day on Monday thru Thursday and 8 hours on Friday, with every other Friday off. Using **-nrad** as an input switch to the program, we get the new output shown in Table III.

The dates in Table III start on the seventh of July because July 4th is a federal

3	HIGH	H1	0	3
2	MEDIUM	M1	0	4
1	LOW	L1	0	6
4	HIGH	H1	3	13
5	MEDIUM	M1	4	12
6	LOW	L1	6	10
8	MEDIUM	M1	12	14
7	LOW	L1	10	15
9	HIGH	H1	13	19
10	MEDIUM	M1	14	24

Table I. Raw output of Scheduler

3	HIGH	H1	07/03/1997+00	07/03/1997+03
2	MEDIUM	M1	07/03/1997+00	07/03/1997+04
1	LOW	L1	07/03/1997+00	07/03/1997+06
4	HIGH	H1	07/03/1997+03	07/07/1997+05
5	MEDIUM	M1	07/03/1997+04	07/07/1997+04
6	LOW	L1	07/03/1997+06	07/07/1997+02
8	MEDIUM	M1	07/07/1997+04	07/07/1997+06
7	LOW	L1	07/07/1997+02	07/07/1997+07
9	HIGH	H1	07/07/1997+05	07/08/1997+03
10	MEDIUM	M1	07/07/1997+06	07/08/1997+08

Table II. Standard Work Day

3	HIGH	H1	07/07/1997+00	07/07/1997+03
2	MEDIUM	M1	07/07/1997+00	07/07/1997+04
1	LOW	L1	07/07/1997+00	07/07/1997+06
4	HIGH	H1	07/07/1997+03	07/08/1997+05
5	MEDIUM	M1	07/07/1997+04	07/08/1997+04
6	LOW	L1	07/07/1997+06	07/08/1997+02
8	MEDIUM	M1	07/08/1997+04	07/08/1997+06
7	LOW	L1	07/08/1997+02	07/08/1997+07
9	HIGH	H1	07/08/1997+05	07/09/1997+03
10	MEDIUM	M1	07/08/1997+06	07/09/1997+08

Table III. NRaD Schedule

holiday, and an NRaD off-Friday, this moves the off-Friday to the 3rd, so the first work-day is actually the seventh. It appears complicated, but the Ada implementation handles it quite easily. The format of `MM/DD/YYYY+HR` is used because daily schedules are idiosyncratic. The notation "`+HH`" means start or finish at that many hours into the workday. It should be easy to map this time format to any person's particular schedule, but in the interest of time was not done here.

The calendar package will also compute non-federal holidays such as Easter, election-day, and other useful dates. The present version runs in order N^2 time. It should be easy to convert to order N, but due to time constraints, this was not done during the course of this thesis. The calendar package was originally added to the scheduler, but it didn't make sense to take an order N^2 algorithm, turn it into an order N one, then turn it back to an order N^2 one with the addition of the calendar package. Besides, the scheduler is used to come up with feasible schedules. Once one is obtained, it can then be easily mapped to calendar dates. This separation of tasks also preserves the modularity of the codes. The conversion routine to convert from "continuous-time" to calendar dates (**contocal**) is in one of the appendices, as part of the scheduler package.

IV. EXPERTISE LEVELS

Every programmer brings certain competencies to the tasks at hand. Some are experts in Ada, others in Java, etc. So, the scheduler has been modified to handle this.

In the Shiah, et al. paper[Ref. 3] on scheduling multiple tasks, resources are represented by a vector data structure as follows:

$$EAT = (EAT_1, EAT_2, \ldots, EAT_r)$$

(EAT stands for earliest available time.) If a task is ready to be scheduled, and it requires resource N, the earliest it can be scheduled is at time EAT_N. If there are multiple instances of a resource then the resources are represented as a matrix, and the earliest time a task can be scheduled is the earliest time any one of the multiple instances of that resource is available. In Salah Badr's thesis, he represented developers as the resources, and since he classified them as $(low, medium, high)$ he could have multiple instances of developers. So the data structure to represent the available resources (developers) was a matrix.

In this latest revision of the code, each developer is unique, there are no multiple instances of a developer, so resources (developers) are representeted as a vector. Each developer, though, has a capability attribute, which is a map of skills to $(low, medium, high)$. For example, one of the inputs to the new scheduler program is a file of developers, as shown in Table IV.

Each developer has an implicit attribute which is their name. Also, if a capability is not given, it is assumed to be *low*. For example developer "Scott McNealy"

Bill Gates	{ActiveX : High, Java : Low}
Scott McNealy	{Java : High, Unix : Medium}
Bill Joy	{Java : High, Unix : High}

Table IV. Sample developer file

Bill Gates	{ActiveX : High, Java : Low, Unix : Low, Bill Gates : High, Scott McNealy : Low, Bill Joy : Low}
Scott McNealy	{ActiveX : Low, Java : High, Unix : Medium, Bill Gates : Low, Scott McNealy : High, Bill Joy : Low}
Bill Joy	{ActiveX : Low, Java : High, Unix : High, Bill Gates : Low, Scott McNealy : Low, Bill Joy : High}

Table V. Sample developer file with implicit capabilities

is assumed to have *low* ActiveX skills, while developer "Bill Gates" is assumed to have *low* Unix skills. If a task is to be scheduled that requires *medium* Unix skills and low ActiveX skills then either developer "Scott McNealy" or "Bill Joy" could be assigned. On the other hand, if a task requires *high* ActiveX skills, then only "Bill Gates" would fit the bill. If a task came in that required *high* skills in both ActiveX and Java, no developer would fit the bill, and the scheduler code would through an Ada (`noqualifieddevelopers`) exception. If a job came in that required *high* or *medium* skills in attribute "Scott McNealy" then only he could possibly be assigned this job. Table V shows what the capabilities of each developer are with the implicit capabilities added.

V. CONCLUSIONS

A. SUMMARY OF DESIGN AND IMPLEMENTATION

The scheduler as implemented can now handle large problems in a reasonable time, i.e., ten thousand or more tasks. The scheduled tasks can now be mapped to a realistic calendar, and the tasks are now associated with problem-solving skills

B. FUTURE WORK

The calendar implementation needs to be optimized. It currently runs in order N^2 time, but could easily be modified to run in order N time. At present the calendar model does not consider individual variations in schedules. If a developer were to take a day off, the model cannot handle that, as it is only aware of work days and holidays for the general work-force. To allow individual schedules into the model a group planning program of some kind would be needed. A kludge to get around this in the present implementaton, is to create pseudo-tasks lasting the period of time off, and requiring only that particular developer perform it. This causes some inaccuracies because the current scheduler in non-preemptive, but in real life time off could be scheduled in the middle of a task. This weakens the algorithm because it can fail to find feasible schedules in which tasks are interrupted by time off.

Another enhancement that would be useful is the identification of critical paths. All schedules have critical paths, that is a sequence of tasks with the least laxity. It would be nice to enhance the scheduler to identify these critical paths. The project manager could then can focus his attention on those tasks in the critical path, as these would be the jobs that puts his schedule most at risk.

Schedule Tools

[Ada '95—Version 1.0]
September 18, 1997

This page intentionally left blank

1. Introduction. Here is the Ada code for utilites used in Salah Badr's scheduler program. His program was written by him May 25, 1993. It was translated by myself, John Evans of NRaD, into Donald Knuth's `WEB` format for literate programming. To compile and link the code in its present format you will need the Ada version of the `WEB` tool.

It is available on-line via the world-wide-web at URL:

http://white.nosc.mil/~evansjr/literate/

.

2. `WEB` is a literate programming paradigm for C, Pascal or Ada, and other languages. This style of programming is called "Literate Programming." For Further information get the book *Literate Programming*, by Donald Knuth, published by the Center for the Study of Language and Information, Stanford University, 1992. Another good source of information is the Usenet group *comp.programming.literate*. It has information on tools and answers to Frequently Asked Questions (FAQs).

3. Who should use the `WEB` paradigm for programming? Well, not everybody. Here are a few paragraphs from Donald Knuth's book that explains it best.

4. **Retrospect and Prospects.** Enthusiastic reports about new computer languages, by the authors of those languages, are commonplace. Hence I'm well aware of the fact that my own experiences cannot be extrapolated too far. I also realize that, whenever I have encountered a problem with `WEB`, I've simply changed the system; other users of `WEB` cannot operate under the same ground rules.

5. However, I believe that I have stumbled on a way of programming that produces better programs that are more portable and more easily understood and maintained than ever before; furthermore, the system seems to work with large programs as well as with small ones. I'm pleased that my work on typography, which began as an application of computers to another field, has come full circle and become an application of typography to the heart of computer science; I like to think of `WEB` as a neat "spinoff" of my research on TeX. However, all of my experiences with this system have been highly colored by my own tastes, and only time will tell if a large number of other people will find `WEB` to be equally attractive and useful.

6. I made a conscious decision not to design a language that would be suitable for everybody. My goal was to provide a tool for system programmers, not for high school students or for hobbyists. I don't have anything against high school students and hobbyists, but I don't believe every computer language should attempt to offer all things to all people. A user of WEB needs to be good enough at computer science that he or she is comfortable dealing with several languates simultaneously. Since WEB combines TEX and Pascal with a few rules of its own, WEB programs can contain WEB syntax errors. TEX syntax errors, Pascal syntax errors, and algorithmic errors; in practice, all four types of errors occur, and a bit of sophistication is needed to sort out which is which. Computer specialists tend to be better at such things than other people. I have found that WEB programs can be debugged rapidly in spite of the profusion of languages, but I'm sure that many other intelligent people will find such a task difficult.

7. In other words, WEB seems to be specifically for the peculiar breed of people who are called computer scientists. And I'm pretty sure that there are also a lot of computer scientists who will not enjoy using WEB; some of us are glad that traditional programming languages have comparatively primitive capabilities for inserted comments, because such difficulties provide a good excuse for not documenting programs well. Thus, WEB may be only for the subset of computer scientists who like to write and to explain what they are doing. My hope is that the ability to make explanations more natural will cause more programmers to discover the joys of literate programming, because I believe it's quite a pleasure to combine verbal and mathematical skills; but perhaps I'm hoping for too much. The fact that a least one paper has been written that is a syntactically correct ALGOL 68 program encourages me to perservere in my hopes for the future. Perhaps we will even one day find Pulitzer prizes awarded to computer programs.

8. Donald Knuth goes on to write about his hopes for the future of WEB programming. In an interview with Donald Knuth by Amazon Books on the release of a new edition of Volume 1 of *The Art of Computer Programming* (July 1, 1997) he was asked:
Amazon.com: What do you see as the most interesting advance in programming since you published the first edition?
Donald Knuth: It's what I call literate programming, a technique for writing, documenting, and maintaining programs using a high-level language combined with a written language like English. This is discussed in my book Literate Programming.

9. In the same book, *Literate Programming*, there is a chapter called *How to read a* WEB. But it is actually quite straightforward.

10. Very briefly, each "Module" within angle brackets (< >) is expanded somewhere further down in the document. The trailing number you see within the brackets is where you can find this expansion. This provides a type of PDL (program descriptor language) for your program and greatly aids modularity and readability. It is also a highly effective method of top-down programming. The first module here is expanded further down, and contains most of the structure in standard Ada packages.

⟨Package boiler-plate 12⟩

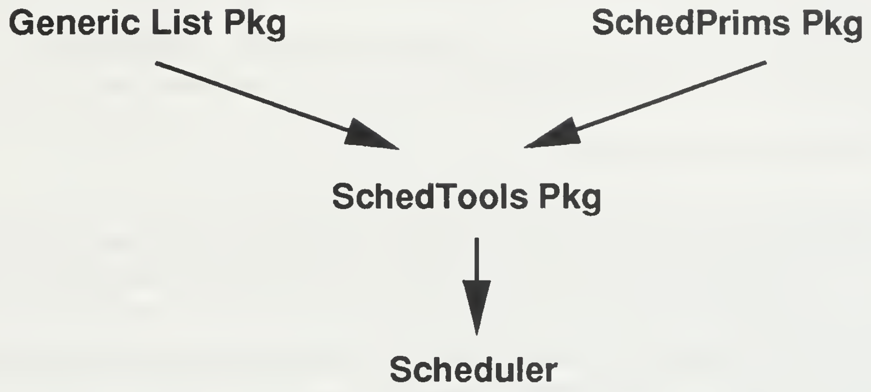
11. Schedule Tools.

12. Here, finally, is the boilerplate. The Ada WEB tool **atangle** reads this and knows to write out two separate files, the specification and the body. (The Ada WEB tool **aweave** will write out just one documentation file.)

```
⟨ Package boiler-plate 12 ⟩ ≡
  output to file schedtools.ads
  with Text_IO;
  use Text_IO;
  with generic_set_pkg;
  with generic_map_pkg;
  with Generic_List;
  with SchedPrims;
  use SchedPrims;
  with capability;
  use capability;
  with ustrings;
  use ustrings;
  package schedtools is
    ⟨ Instantiate generics 16 ⟩
    ⟨ Specification of types and variables visible from schedtools 23 ⟩
    ⟨ Specification of procedures visible from schedtools 26 ⟩
  end schedtools;
  output to file schedtools.adb
  with test_io_pkg;
  use test_io_pkg;
  with Ustrings; Use Ustrings; with Ada.calendar;
  use Ada.calendar;
  with calyr;
  use calyr;
  with capability;
  use capability;
  package body schedtools is
    ⟨ Variables local to schedtools 41 ⟩
    ⟨ Procedures and Tasks in schedtools 42 ⟩
  end schedtools;
```

This code is used in section 10.

13. The scheduling tools in this package rely on some other packages. Here is how they relate to each other.



Library Dependence Structure.

14. The schedules are kept in in linked-lists. Salah Badr's original code had separate routines for each linked list. In this version of the algorithm, I created a generic list type, and make multiple instantiations of it for different record types. Details of the differing records, comparisons, and display routines can be found in the `schedprims` package.

15. Since the main purpose of rewriting the code was to eliminate the order N^2 space requirement, I use linked lists to keep track of additions and deletions to the lists as the search space is traversed. What follows are all the instantiations of new linked-lists.

16. Here I instantiate a list type to manipulate *StepRecord* types.

⟨ Instantiate generics 16 ⟩ ≡
  package *InputList1* **is new** *Generic_list*(*ElementType* ⇒ *StepRecord*,
      DisplayElement ⇒ *DisplayStepRecord*, "<" ⇒ *CompareID*);
  use *InputList1*;
  subtype *InputList* **is** *InputList1.List*;

See also sections 17, 18, 19, 20, 21, and 22.

This code is used in section 12.

17. Here I instantiate a list type to manipulate *StepRecord* types, but to restore deletions, in case the recursive procedure *BranchAndBound* needs to back out changes.

⟨ Instantiate generics 16 ⟩ +≡
  package *DeletedInputList1* **is new** *Generic_list*(*ElementType* ⇒ *StepRecord*,
      DisplayElement ⇒ *DisplayStepRecord*, "<" ⇒ *CompareRecursionLevel*);
  use *DeletedInputList1*;
  subtype *DeletedInputList* **is** *DeletedInputList1.List*;

18. Here I instantiate a list type to manipulate *StepRecord* types for the *ReadyQueue*, which requires that the records be sorted in *Deadline* first order.

⟨Instantiate generics 16⟩ +≡

```
package ReadyList1 is new Generic_list(ElementType ⇒ StepRecord,
        DisplayElement ⇒ DisplayStepRecord, "<" ⇒ CompareDeadline, "=" ⇒ IsEqual);
use ReadyList1;
subtype ReadyList is ReadyList1.List;
```

19. Here I instantiate a list type to manipulate *StepRecord* types for deletions to the *ReadyQueue*, which requires that the records be sorted in *RecursionLevel* first order.

⟨Instantiate generics 16⟩ +≡

```
package DeletedReadyList1 is new Generic_list(ElementType ⇒ StepRecord,
        DisplayElement ⇒ DisplayStepRecord, "<" ⇒ CompareRecursionLevel);
use DeletedReadyList1;
subtype DeletedReadyList is DeletedReadyList1.List;
```

20. Here I instantiate a list type to manipulate *StepRecord* types for additions to the *ReadyQueue*, which requires that the records be sorted in *RecursionLevel* first order.

⟨Instantiate generics 16⟩ +≡

```
package AddedReadyList1 is new Generic_list(ElementType ⇒ StepRecord,
        DisplayElement ⇒ DisplayStepRecord, "<" ⇒ CompareRecursionLevel);
use AddedReadyList1;
subtype AddedReadyList is AddedReadyList1.List;
```

21. Here I instantiate a list type to manipulate *StepRecord* types for the *ReadyQueue*, which requires that the records be sorted in *Deadline* first order.

⟨Instantiate generics 16⟩ +≡

```
package ScheduleList1 is new Generic_list(ElementType ⇒ ScheduleRecord,
        DisplayElement ⇒ DisplayScheduleRecord, "<" ⇒ CompareStartTime);
use ScheduleList1;
subtype ScheduleList is ScheduleList1.List;
```

22. Here I instantiate a list type to manipulate *StepRecord* types for the *ReadyQueue*, which requires that the records be sorted in *Deadline* first order.

⟨Instantiate generics 16⟩ +≡

```
package CalendarList1 is new Generic_list(ElementType ⇒ CalendarRecord,
        DisplayElement ⇒ DisplayCalendarRecord, "<" ⇒ CompareStartTime);
use CalendarList1;
subtype CalendarList is CalendarList1.List;
```

23. Made global and visible.

⟨ Specification of types and variables visible from *schedtools* 23 ⟩ ≡
  max_recursion : *natural* ← 0;
  recursion_level : *natural* ← 0;

See also sections 24, 25, 33, and 59.

This code is used in section 12.

24. When the laxity of the input schedule is "tight," it may be impossible to find a schedule. (Finding a schedule is, after all, an NP-Complete problem.) In this case the routine will give up after some amount of effort. In this implementation, I give up if the number of "backtracks" is *FeasFactor* times the total of number of tasks to be scheduled. If this number is exceeded then the exception *NoFeasibleScheduleFound* is thrown.

⟨ Specification of types and variables visible from *schedtools* 23 ⟩ +≡
  NoFeasibleScheduleFound : *Exception*;
  FeasFactor : *natural* ← 10;

25. Made global and visible.

⟨ Specification of types and variables visible from *schedtools* 23 ⟩ +≡
  StepList : *InputList*;
  ReadyQueue : *ReadyList*;
  DeletedReadyQueue : *DeletedReadyList*;
  DeletedInputQueue : *DeletedInputList*;
  AddedReadyQueue : *AddedReadyList*;
  Schedule : *ScheduleList*;
  Calendar : *CalendarList*;
  FinalSchedule : *ScheduleList*;

26. Print all the records in the Step list.

⟨ Specification of procedures visible from *schedtools* 26 ⟩ ≡
  procedure *PrintAllStepRecords* (*L* : **in** *InputList*);

See also sections 27, 28, 29, 30, 31, 32, 34, 35, 36, 37, 38, and 39.

This code is used in section 12.

27. Print all the records in the Step list.

⟨ Specification of procedures visible from *schedtools* 26 ⟩ +≡
  procedure *PrintAllStepRecords* (*L* : **in** *ReadyList*);

28. Print all the records in the Schedule list.

⟨ Specification of procedures visible from *schedtools* 26 ⟩ +≡
  procedure *PrintAllScheduleRecords* (*L* : **in** *ScheduleList*);

29. Print all the records in the Schedule list.

⟨ Specification of procedures visible from *schedtools* 26 ⟩ +≡
procedure *PrintAllCalendarRecords*(*L* : **in out** *ScheduleList*);

30. Print all the records in the Schedule list.

⟨ Specification of procedures visible from *schedtools* 26 ⟩ +≡
procedure *SaveAllScheduleRecords*(*L* : **in out** *ScheduleList*);

31. Creating new step from a file and linking it to the step list.

⟨ Specification of procedures visible from *schedtools* 26 ⟩ +≡
procedure *CreateNewStepList*(*L* : **in out** *InputList*);

32.

⟨ Specification of procedures visible from *schedtools* 26 ⟩ +≡
Procedure CreateDeadlineFirstSchedule(*mr* : **in out** *natural*; *num_developers* : *natural*);

33.

⟨ Specification of types and variables visible from *schedtools* 23 ⟩ +≡
type *DesignerMatrix* **is array** (POSITIVE **range** <>) **of** *natural*;

34. Creating a new schedule record

⟨ Specification of procedures visible from *schedtools* 26 ⟩ +≡
procedure *CreateScheduleRecord*(*Rec* : **out** *ScheduleRecord*; *S_ID* : **in** *natural*; *TIME1* : **in** *natural*; *TIME2* : **in** *natural*; *S_LEVEL* : **in** *cap_map.map*; *Developer* : **in** *ustring*);

35.

⟨ Specification of procedures visible from *schedtools* 26 ⟩ +≡
procedure *LevelMinmum*(*MATRIX* : **in** *DesignerMatrix*; *LEVEL* : **in** *cap_map.map*; *J* : **in out** *natural*);

36. checking the *in_degree* of the successors of the assigned step. This works with deadline heuristic

⟨ Specification of procedures visible from *schedtools* 26 ⟩ +≡
procedure *CheckInDegree*(*Rec* : **in** *StepRecord*; *Queue* : **in out** *ReadyList*; *InList* : **in out** *InputList*; *finish_t* : **in** *natural*);

37.

⟨ Specification of procedures visible from *schedtools* 26 ⟩ +≡
procedure *StronglyFeasible*(*Queue* : **in out** *ReadyList*; *MATRIX* : **in** *DesignerMatrix*; *FEASIBLE* : **in out** *boolean*);

38. Assign a step to a designer according to its deadline and its expertise level

⟨ Specification of procedures visible from *schedtools* 26 ⟩ +≡
procedure *AssignStep*(*Current* : *StepRecord*; *MATRIX* : **in out** *DesignerMatrix*;
Sch : **in out** *ScheduleList*; *Finish* : **in out** *natural*; *FEAS* : out *boolean*);

39.

⟨ Specification of procedures visible from *schedtools* 26 ⟩ +≡
procedure *BranchAndBound*(*S_List* : **in out** *InputList*; *R_Queue* : **in out** *ReadyList*;
F_Sched : **in out** *ScheduleList*; *MATRIX* : **in** *DesignerMatrix*; *Found* : **in out**
BOOLEAN);

40. Schedule Tools Body.

41. Global variable used to identify different tasks.

⟨ Variables local to *schedtools* 41 ⟩ ≡
StepID : *natural* ← 1;
data_file, *data2_file* : *file_type*;
FOUND : *boolean* ← FALSE;
FEASIBLE : *boolean* ← TRUE;
debug : *boolean* ← *false*;
debug2 : *boolean* ← *false*;
StartTime : *Time*;
dailyhours : *WorkHours* ← (*ConvertHoursToDuration*(8), *ConvertHoursToDuration*(8), *ConvertHoursToDuration*(8), *ConvertHoursToDuration*(8), *ConvertHoursToDuration*(8));
NRaD : *boolean* ← *false*;

See also sections 55 and 56.

This code is used in section 12.

42. Print all the records in the STEP list.

⟨ Procedures and Tasks in *schedtools* 42 ⟩ ≡
procedure *PrintAllStepRecords*(*L* : **in** *InputList*) **is**
begin
StepRecordHeading; *Display*(*L*);
end *PrintAllStepRecords*;

See also sections 43, 44, 45, 47, 49, 52, 53, 57, 58, 62, 66, 70, and 71.

This code is used in section 12.

43. Print all the records in the STEP list.

⟨ Procedures and Tasks in *schedtools* 42 ⟩ +≡
procedure *PrintAllStepRecords*(*L* : **in** *ReadyList*) **is**
begin
StepRecordHeading; *Display*(*L*);
end *PrintAllStepRecords*;

44. Print all the records in the STEP list.

⟨ Procedures and Tasks in *schedtools* 42 ⟩ +≡
procedure *PrintAllScheduleRecords*(*L* : **in** *ScheduleList*) **is**
begin
ScheduleRecordHeading; *Display*(*L*);
end *PrintAllScheduleRecords*;

45. Print all the records in the STEP list.

```
⟨ Procedures and Tasks in schedtools 42 ⟩ +≡
  procedure SaveAllScheduleRecords(L : in out ScheduleList) is
    input : Ustring;
    size : natural;
    cur : ScheduleRecord;
  begin
    ⟨ Get output file name 46 ⟩
    put_line("Opening␣your␣output␣file."); create(data2_file, out_file, S(input));
    size ← ListSize(L); rewind(L);
    for i ∈ 1 .. size loop
      if i = 1 then
        getCurrent(L, cur);
      else
        getNext(L, cur);
      end if;
      SaveScheduleRecord(cur, data2_file);
    end loop;
  end SaveAllScheduleRecords;
```

46.

```
⟨ Get output file name 46 ⟩ ≡
  put_line("Please␣Enter␣Output␣File␣Name:␣"); get_line(input);
```

This code is used in section 45.

47. Print all the records in the STEP list.

```
⟨ Procedures and Tasks in schedtools 42 ⟩ +≡
  procedure PrintAllCalendarRecords(L : in out ScheduleList) is
    size : natural;
    cur : ScheduleRecord;
    cal : CalendarRecord;
    dur : Duration;
  begin
    CalendarRecordHeading;
    ⟨ Convert ScheduleList to CalendarList 48 ⟩Display(Calendar);
  end PrintAllCalendarRecords;
```

48.

⟨Convert *ScheduleList* to *CalendarList* 48⟩ ≡

```
MakeEmpty(Calendar); size ← ListSize(L); Rewind(L);
for i ∈ 1 .. size loop
  if i = 1 then
    GetCurrent(L, cur);
  else
    GetNext(L, cur);
  end if;
  dur ← ConvertHoursToDuration(cur.StartTime);
  cal.StartTime ← DurationToCalendarTime(StartTime, dailyhours, dur, NRaD);
  dur ← ConvertHoursToDuration(cur.FinishTime);
  cal.Finishtime ← DurationtoCalendarTime(StartTime, dailyhours, dur, NRaD);
  cal.StepId ← cur.StepId; cal.Designer ← cur.Designer;
  cap_map.assign(cal.StepLevel, cur.StepLevel); InsertInOrder(Calendar, cal);
end loop;
```

This code is used in section 47.

49. Creating new step from a file.

```
⟨ Procedures and Tasks in schedtools 42 ⟩ +≡
  procedure CreateNewStepList(L : in out InputList) is
    sr : StepRecord;
    input : Ustring;
    do_alternate : boolean ← false;
    ⟨ Variables local to CreateNewStepList 51 ⟩
  begin
    MakeEmpty(L);
    StepId ← 1;
    put_line("Please␣Enter␣␣INPUT␣FILE␣NAME␣");
    get_line(input);
    put_line("Opening␣your␣data␣file␣");
    open(data_file, in_file, S(input));
    while ¬end_of_file(data_file) loop
      sr.StepId ← StepID;
      if do_alternate then
        DeadTime ← get_date(data_file);
      else
        nat_io.get(data_file, sr.Deadline);
      end if;
      nat_io.get(data_file, sr.Priority);
      nat_io.get(data_file, sr.EstimatedDuration);
      if do_alternate then
        Earlytime ← get_date(data_file);
      else
        nat_io.get(data_file, sr.EarliestStartTime);
      end if;
      getf_set(data_file, sr.Predecessors);
      getf_set(data_file, sr.Successors);
      declare
        yrcap : cap_map.map;
      begin
        get_capability(data_file, yrcap); cap_map.assign(sr.ExpLevel, yrcap);
      end;
      sr.InDegree ← nat_set.size(sr.Predecessors);
      if do_alternate then
        ⟨ Convert calendar times to absolute times 50 ⟩
      else
        StartTime ← Time_Of(1997, 7, 3, 0.0);
      end if;
      AddToEnd(L, sr); StepID ← StepID + 1;
    end loop;
```

```
    CLOSE(data_file);
  end CreateNewStepList;
```

50.

⟨ Convert calendar times to absolute times 50 ⟩ ≡

```
  if StepID = 1 then
    StartTime ← Earlytime;
  end if;
  dur ← CalendarTimeToDuration(StartTime, dailyhours, Deadtime, NRaD);
  sr.Deadline ← ConvertDurationToHours(dur);
  dur ← CalendarTimeToDuration(StartTime, dailyhours, EarlyTime, NRaD);
  sr.EarliestStartTime ← ConvertDurationToHours(dur);
```

This code is used in section 49.

51.

⟨ Variables local to *CreateNewStepList* 51 ⟩ ≡

```
  dur : Duration;
  EarlyTime, DeadTime : Time;
```

This code is used in section 49.

52.

⟨ Procedures and Tasks in *schedtools* 42 ⟩ +≡

```
  procedure ReInitializeMatrix(MATRIX : in out DesignerMatrix) is
  begin
    for i ∈ 1 .. matrix'length loop
      matrix(i) ← 0;
    end loop;
  end ReInitializeMatrix;
```

53. Creating new step.

⟨ Procedures and Tasks in *schedtools* 42 ⟩ +≡

```
  Procedure CreateDeadlineFirstSchedule(mr : in out natural; num_developers : natural)
      is Current : StepRecord;
  Feasible : boolean ← True;
  eat : designermatrix(1 .. num_developers);
begin
  Kntr ← ListSize(StepList); ⟨ Initialize the lists for intensive list-processing 54 ⟩
  Rewind(StepList); GetCurrent(StepList, Current);
  for i ∈ 1 .. Kntr loop
    if Current.InDegree = 0 then
      DeleteCurrent(StepList); InsertInOrder(ReadyQueue, Current);
      if i < Kntr then
        GetCurrent(StepList, Current);
      end if;
    else
      if i < Kntr then
        GetNext(StepList, Current);
      end if;
    end if;
  end loop;
  Feasible ← True; Found ← False; ReInitializeMatrix(EAT);
  StronglyFeasible(ReadyQueue, EAT, Feasible);
  if Feasible then
    put_line("Calling␣BranchAndBound␣Routine.");
    BranchAndBound(StepList, ReadyQueue, Schedule, EAT, FOUND);
    put_line("Returned␣from␣BranchAndBound␣Routine.");
  end if;
  if ¬FOUND then
    put_line("SORRY␣THERE␣IS␣NO␣FEASIBLE␣SCHEDULE");
  end if;
  mr ← max_recursion;
end CreateDeadlineFirstSchedule;
```

54. If this is not the first time this routine is called then it behooves us to clean up the old lists from previous processing. If this is the first time, no harm done.

⟨ Initialize the lists for intensive list-processing 54 ⟩ ≡

```
  MakeEmpty(ReadyQueue); MakeEmpty(Schedule); MakeEmpty(DeletedReadyQueue);
  MakeEmpty(DeletedInputQueue); MakeEmpty(AddedReadyQueue);
```

This code is used in section 53.

55.

⟨ Variables local to *schedtools* 41 ⟩ +≡
```
kntr : integer ← 0;
```

56.

⟨ Variables local to *schedtools* 41 ⟩ +≡
```
counter : natural ← 0; @{Used for tracking backtracking@}
```

57. Creating a new schedule record

⟨ Procedures and Tasks in *schedtools* 42 ⟩ +≡
```
procedure CreateScheduleRecord(Rec : out ScheduleRecord; S_ID : in
        natural; TIME1 : in natural; TIME2 : in natural; S_LEVEL : in
        cap_map.map; Developer : in ustring) is
begin
  Rec.StepID ← S_ID; Rec.StartTime ← TIME1; Rec.FinishTime ← TIME2;
  Rec.Designer ← Developer; cap_map.assign(Rec.StepLevel, S_LEVEL);
end CreateScheduleRecord;
```

58.

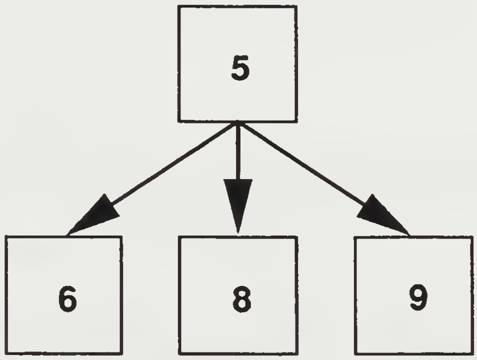
⟨ Procedures and Tasks in *schedtools* 42 ⟩ +≡
```
procedure LevelMinmum(MATRIX : in DesignerMatrix; LEVEL : in
        cap_map.map; J : in out natural) is
  min : natural;
  n : natural;
begin
  j ← 0; min ← natural'last; n ← 1;
  if is_qualified(level, n) then
    j ← 1; min ← matrix(1);
  end if;
  for m ∈ 2 .. matrix'length loop
    if matrix(m) < min then
      if is_qualified(level, m) then
        min ← matrix(m); j ← m;
      end if;
    end if;
  end loop;
  if j = 0 then
    raise noqualifieddevelopers;
  end if;
end levelminmum;
```

59.

⟨ Specification of types and variables visible from *schedtools* 23 ⟩ +≡
noqualifieddevelopers : **exception**;

60. Check In Degree. Checking the *in_degree* of the successors of the assigned step. This works with deadline heuristic

61. Presently changes the start-time of any successors. Will need to modify when I convert the updates from a recursive local variable to a global one. Also deletes a scheduled task from the *INPUT_LIST*. Then it updates the queue of "ready" tasks.



Precedence Graph

62. This procedure loops through the entire *InputList* finding the successors of *Rec*. Once found it updates the *EarliestStartTime*. Also, if the *InDegree* reaches zero this means it no longer is waiting on a predessor to be scheduled, it is "ready" to be sceduled—that is, moved from the *InputList* to the *ReadyQueue*.
Note: It appears that the *Predecessor* field of the *StepRecord* is ignored. Only the successor field is used.

```
⟨ Procedures and Tasks in schedtools 42 ⟩ +≡
  procedure CheckInDegree(Rec : in StepRecord; Queue : in out ReadyList; InList : in
        out InputList; finish_t : in natural) is
    Current : StepRecord;
    t : nat_set.set ← Rec.Successors;
    k, kntr : natural;
    FOUND : boolean ← FALSE;
    deleted : boolean ← false;
  begin
    if nat_set.size(t) ≠ 0 then
      Rewind(InList); kntr ← ListSize(InList); GetCurrent(InList, Current);
      for i ∈ 1 .. kntr loop
        k ← Current.StepId;
        if nat_set.member(k, t) then
          if Current.EarliestStartTime < finish_t then
            Current.EarliestStarttime ← finish_t;
          end if;
          Current.InDegree ← Current.InDegree − 1;
          if Current.InDegree = 0 then
            ⟨ Move record from input list to ready list 64 ⟩
          else
            UpdateCurrent(InList, Current);
          end if;
        end if;
        ⟨ Get next record 63 ⟩
      end loop;
    end if;
  end CheckInDegree;
```

63.

⟨Get next record 63⟩ ≡

```
if i < kntr then
   if deleted then
      GetCurrent(InList, Current); deleted ← false;
   else
      GetNext(InList, Current);
   end if;
end if;
```

This code is used in section 62.

64.

⟨Move record from input list to ready list 64⟩ ≡

```
DeleteCurrent(InList); Current.recursionlevel ← recursion_level;
InsertInOrder(Queue, Current); InsertInOrder(AddedReadyQueue, Current);
Current.InDegree ← Current.Indegree + 1;
InsertInOrder(DeletedInputQueue, Current); deleted ← true;
if debug then
   put_line("Moving␣Record␣to␣DeletedInputQueue."); Display(DeletedInputQueue);
end if;
```

This code is used in section 62.

65. StrongFeasible. Checking the feasibility of the schedule with each step in the ready queue.

66. Definition: A partial feasible schedule is said to be *strongly-feasible* if *all* the schedules obtained by extending the current schedule with any one of the remaining tasks are also feasible. Thus, if a partial feasible schedule is found not to be *strongle-feasible* because, say, task *T* misses its deadline when the current schedule is extended by T, then it is appropriate to stop the search since none of the future extensions involving task *T* will meet its deadline. In this case, a set of tasks can not be scheduled given the current partial schedule. (In the terminology of branch-and-bound techniques, the search path represented by the current partial schedule is *bound* since it will not lead to a feasible complete schedule.)

```
⟨ Procedures and Tasks in schedtools 42 ⟩ +≡
  procedure StronglyFeasible(Queue : in out ReadyList; MATRIX : in
        DesignerMatrix; feasible : in out boolean) is
    temp : natural;
    J : natural ← 1;
    L : natural ← 1;
    min : natural ← 0;
    kntr : natural ← 0;
    myonum : natural ← 0;
    Current : StepRecord;
    Myopic_Num : constant natural ← 7;
  begin
    if debug then
      put_Line("StronglyFeasible>␣Start␣");
    end if;
    feasible ← True; kntr ← ListSize(Queue); ⟨ Compute myopic number 67 ⟩
    Rewind(Queue);
    for i ∈ 1 .. myonum loop
      if ¬feasible then
        exit;
      end if;
      if i = 1 then
        GetCurrent(Queue, Current);
      else
        GetNext(Queue, Current);
      end if;
      LevelMinmum(MATRIX, Current.ExpLevel, J); min ← MATRIX(J);
      ⟨ Debug code set 1 68 ⟩
      if min ≥ Current.EarliestStartTime then
        temp ← min;
      else
        temp ← Current.EarliestStarttime;
      end if;
      temp ← temp + Current.EstimatedDuration; ⟨ Debug code set 2 69 ⟩
```

```
    if temp > Current.Deadline then
      feasible ← False;
    end if;
  end loop;
end StronglyFeasible;
```

67. Without this tidbit of code, the algorithm goes from order n to order n^2.

```
⟨ Compute myopic number 67 ⟩ ≡
  if kntr > Myopic_Num then
    myonum ← Myopic_Num;
  else
    myonum ← kntr;
  end if;
```

This code is used in section 66.

68.

```
⟨ Debug code set 1 68 ⟩ ≡
  if debug then
    put("StronglyFeasible>␣Id␣=␣"); nat_io.put(Current.StepId, 1);
    put("␣␣min␣=␣"); nat_io.put(min, 2); put(".␣Current.EarliestStartTime␣␣=␣");
    nat_io.put(Current.EarliestStartTime, 2); put_line(".␣");
  end if;
```

This code is used in section 66.

69.

```
⟨ Debug code set 2 69 ⟩ ≡
  if debug then
    put("StronglyFeasible>␣"); nat_io.put(i, 2); put(".␣temp␣=␣");
    nat_io.put(temp, 2); put(".␣␣Current.Deadline␣=␣");
    nat_io.put(Current.Deadline, 2); put_line(".␣");
  end if;
```

This code is used in section 66.

70. AssignStep. Assign a step to a designer according to its deadline and its expertise level: BRANCH AND BOUND CASE

⟨ Procedures and Tasks in *schedtools* 42 ⟩ +≡

```
procedure AssignStep(Current : in StepRecord; MATRIX : in out
      DesignerMatrix; Sch : in out ScheduleList; Finish : in out natural; FEAS : out
      boolean) is
  J : natural;
  MIN : natural;
  temp : natural ← 0;
  temp1 : StepRecord ← Current;
  Dummy : ScheduleRecord;
begin
  LevelMinmum(MATRIX, Current.ExpLevel, J); MIN ← MATRIX(J);
  if MIN ≤ Current.EarliestStartTime then
    temp ← Current.EarliestStartTime; finish ← temp + Current.EstimatedDuration;
    if finish > Current.DEADLINE then
      FEAS ← FALSE;
    else
      FEAS ← TRUE; MATRIX(J) ← finish; CreateScheduleRecord(Dummy,
          temp1.StepID, temp, finish, temp1.ExpLevel, get_developer_name(j));
      AddToEnd(Sch, Dummy);
    end if;
  else
    temp ← MIN; finish ← temp + Current.EstimatedDuration;
    if finish > Current.Deadline then
      FEAS ← FALSE;
    else
      FEAS ← TRUE; MATRIX(J) ← finish; CreateScheduleRecord(Dummy,
          temp1.StepID, temp, finish, temp1.ExpLevel, get_developer_name(j));
      AddToEnd(Sch, Dummy);
    end if;
  end if;
end AssignStep;
```

71. Branch And Bound.

```
⟨ Procedures and Tasks in schedtools 42 ⟩ +≡
  procedure BranchAndBound(S_List : in out InputList; R_Queue : in out ReadyList;
        F_Sched : in out ScheduleList; MATRIX : in DesignerMatrix; Found : in out
        BOOLEAN) is
    ⟨ Variables local to BranchAndBound 73 ⟩
  begin
    ⟨ Update some recursion stuff 72 ⟩
    if IsEmpty(R_Queue) then
      if do_verbose then
        ScheduleRecordHeading; PrintAllScheduleRecords(F_Sched); new_line;
      end if;
      put("Backtracking␣:=␣"); test_io_pkg.put(counter); new_line;
      @{Copy(F_Sched, FinalSchedule); @}Found ← True;
      if debug then
        put_line("Found␣a␣valid␣schedule.");
      end if;
    elsif ¬found then
      OrigSize ← ListSize(R_Queue);
      for i IN 1 .. OrigSize loop
        ⟨ Update backtrack counter 74 ⟩
        ⟨ Copy linked lists and the designer matrix onto the stack 80 ⟩
        ⟨ Get appropriate R_Queue record 76 ⟩
        if debug then
          put("BranchAndBound>␣Current␣=␣"); DisplayStepRecord(Current);
          put("BranchAndBound>␣ListSize(R_Queue)␣is␣");
          nat_io.put(ListSize(R_Queue)); put_line(".␣");
        end if;
        AssignStep(Current, MAT, F_Sched, FinishTime, Feasible);
        CheckInDegree(Current, R_Queue, S_List, FinishTime);
        ⟨ Delete appropriate R_Queue record 78 ⟩
        if debug then
          put_line("After␣assigning␣step,␣but␣before␣testing␣for␣Feasibility:");
          PrintAllStepRecords(R_Queue); PrintAllScheduleRecords(F_Sched);
        end if;
        StronglyFeasible(R_Queue, MAT, Feasible1);
        if Feasible1 then
          BranchAndBound(S_List, R_Queue, F_Sched, MAT, Found);
          ⟨ Update recursion stuff again 79 ⟩
        end if;
        ⟨ Free up local linked lists 83 ⟩
        if Found then
          exit;
```

```
      end if;
    end loop;
    if recursion_level ≤ 1 then
      if debug then
        put_line("BranchAndBound>␣Finished␣unwinding␣the␣stack.");
      end if;
    end if;
  end if;
end BranchAndBound;
```

72.

⟨ Update some recursion stuff 72 ⟩ ≡

```
if (diag_sched ∨ diag_step ∨ diag_ready_queue) then
  do_verbose ← true;
end if;
recursion_level ← recursion_level + 1;
if recursion_level > max_recursion then
  max_recursion ← recursion_level;
end if;
```

This code is used in section 71.

73.

⟨ Variables local to *BranchAndBound* 73 ⟩ ≡

```
do_verbose : boolean ← false;
OrigSize : natural;
```

See also sections 75, 77, 82, 85, 88, and 90.

This code is used in section 71.

74.

⟨ Update backtrack counter 74 ⟩ ≡

```
if i ≠ 1 then
  counter ← counter + 1;
end if;
TotSize ← ListSize(R_Queue) + ListSize(S_List) + ListSize(F_Sched);
if counter > (FeasFactor * TotSize) then
  raise NoFeasibleScheduleFound;
end if;
```

This code is used in section 71.

75.

⟨ Variables local to *BranchAndBound* 73 ⟩ +≡

```
TotSize : natural;
```

76.

⟨ Get appropriate *R_Queue* record 76 ⟩ ≡

```
appropriate ← i − (OrigSize − ListSize(R_Queue));
if debug then
    put("BranchAndBound>␣Getting␣number␣"); nat_io.put(Appropriate, 1);
    put("␣record␣in␣Ready␣Queue."); put("(i␣=␣"); nat_io.put(i, 1);
    put(",␣Origsize␣=␣"); nat_io.put(Origsize, 1); put_line(").");
end if;
GetNth(R_Queue, appropriate, Current);
```

This code is used in section 71.

77.

⟨ Variables local to *BranchAndBound* 73 ⟩ +≡

```
appropriate : natural;
```

78.

⟨ Delete appropriate *R_Queue* record 78 ⟩ ≡

```
if debug then
    put_line("Deleting␣appropriate␣R_Queue␣record.");
end if;
GetNth(R_Queue, appropriate, Current); DeleteCurrent(R_Queue);
Current.RecursionLevel ← Recursion_Level;
InsertInOrder(DeletedReadyQueue, Current);
if debug then
    put_line("Finished␣deleting␣appropriate␣R_Queue␣record.");
end if;
```

This code is used in section 71.

79.

⟨ Update recursion stuff again 79 ⟩ ≡

```
recursion_level ← recursion_level − 1;
```

This code is used in section 71.

80. As far as I can see the step list is never modified, so why is it copied? Aha! It is modified in procedure *check_in_degree*.

⟨ Copy linked lists and the designer matrix onto the stack 80 ⟩ ≡

```
⟨ Do diagnostics 81 ⟩
@{Copy(S_List, InList); Copy(R_Queue, Queue); Copy(F_Sched, Sched);
@}MAT ← MATRIX;
```

This code is used in section 71.

81.

⟨ Do diagnostics 81 ⟩ ≡

```
if do_verbose then
  put_line("==========================================================");
  put("Recursion␣level␣is␣"); nat_io.put(recursion_level); put_line(".␣");
end if;
if diag_step then
  PrintAllStepRecords(S_List);
end if;
if diag_ready_queue then
  PrintAllStepRecords(R_QUEUE);
end if;
if diag_sched then
  PrintAllScheduleRecords(F_sched);
end if;
```

This code is used in section 80.

82.

⟨ Variables local to *BranchAndBound* 73 ⟩ +≡

```
diag_step : boolean ← false;
diag_ready_queue : boolean ← false;
diag_sched : boolean ← false;
```

83.

⟨ Free up local linked lists 83 ⟩ ≡

```
@{MakeEmpty(InList); MakeEmpty(Queue); MakeEmpty(Sched);
@}⟨ Restore R_Queue 84 ⟩
⟨ Restore S_List 86 ⟩
⟨ Restore F_Sched 89 ⟩
```

This code is used in section 71.

84.

```
⟨ Restore R_Queue 84 ⟩ ≡
  if ¬Found then
    if debug then
      put_line("Restoring␣R_Queue.");
    end if;
    Dsize ← ListSize(AddedReadyQueue);
    if Dsize ≠ 0 then
      GetNth(AddedReadyQueue, Dsize, Current);
      while Current.recursionlevel = recursion_level loop
        DeleteCurrent(AddedReadyQueue);
        DeleteMatching(R_Queue, Current, Success);
        if debug then
          put("Deleting␣record␣"); put_line("From␣ReadyQueue.");
          DisplayStepRecord(Current);
        end if;
        if ¬Success then
          put_Line("Did␣not␣find␣matching␣record!");
        end if;
        Dsize ← ListSize(AddedReadyQueue);
        if Dsize = 0 then
          exit;
        else
          GetNth(AddedReadyQueue, Dsize, Current);
        end if;
      end loop;
    end if;
    Dsize ← ListSize(DeletedReadyQueue);
    GetNth(DeletedReadyQueue, Dsize, Current); DeleteCurrent(DeletedReadyQueue);
    InsertInOrder(R_Queue, Current); ⟨ Reset InDegree 87 ⟩
    if debug then
      put_line("Finished␣restoring␣R_Queue.");
    end if;
  end if;
```

This code is used in section 83.

85.

```
⟨ Variables local to BranchAndBound 73 ⟩ +≡
  Success : boolean;
```

86.

```
⟨Restore S_List 86⟩ ≡
  if ¬Found then
    Dsize ← ListSize(DeletedInputQueue);
    if Dsize ≠ 0 then
      GetNth(DeletedInputQueue, Dsize, Current);
      while Current.recursionlevel = recursion_level loop
        DeleteCurrent(DeletedInputQueue); InsertInOrder(S_List, Current);
        ⟨Reset InDegree 87⟩
        Dsize ← ListSize(DeletedInputQueue);
        if Dsize ≠ 0 then
          GetNth(DeletedInputQueue, Dsize, Current);
        else
          exit;
        end if;
      end loop;
    end if;
  end if;
```

This code is used in section 83.

87.

```
⟨ Reset InDegree 87 ⟩ ≡
  if debug then
    put("Resetting␣InDegree␣for␣successors␣of␣:␣"); DisplayStepRecord(Current);
  end if;
  Dsize ← ListSize(S_List); t ← Current.Successors; Rewind(S_List);
  for i ∈ 1 .. Dsize loop
    if i = 1 then
      GetCurrent(S_List, Current);
    else
      GetNext(S_List, Current);
    end if;
    k ← Current.StepId;
    if debug then
      put("StepId␣=␣"); put(k); put(".␣␣Now␣checking␣for␣membership.");
    end if;
    if nat_set.member(k, t) then
      if debug then
        put_line("(Member)"); DisplayStepRecord(Current);
      end if;
      Current.InDegree ← Current.InDegree + 1; UpdateCurrent(S_List, Current);
      if debug then
        DisplayStepRecord(Current);
      end if;
    else
      if debug then
        put_line("(Not␣Member)");
      end if;
    end if;
  end loop;
```

This code is used in sections 84 and 86.

88.

```
⟨ Variables local to BranchAndBound 73 ⟩ +≡
  t : nat_set.set;
  k : natural;
```

89.

```
⟨ Restore F_Sched 89 ⟩ ≡
  if ¬Found then
    if debug then
      put_line("Restoring␣F_Sched.");
    end if;
    Dsize ← ListSize(F_Sched); GetNth(F_Sched, Dsize, DCurrent);
    DeleteCurrent(F_Sched);
    if debug then
      put_line("Finished␣restoring␣F_Sched.");
    end if;
  end if;
```

This code is used in section 83.

90.

```
⟨ Variables local to BranchAndBound 73 ⟩ +≡
  InList : InputList;
  DCurrent : ScheduleRecord;
  @{Queue : ReadyList;
  Sched : ScheduleList;
  @}Dsize : natural;
  Current : StepRecord;
  MAT : DesignerMatrix(1 .. matrix'length);
  Feasible : BOOLEAN ← TRUE;
  Feasible1 : BOOLEAN ← TRUE;
  FinishTime : natural ← 0;
```

91. System-dependent changes. This module should be replaced, if necessary, by changes to the program that are necessary to make `MAIN` work at a particular installation. It is usually best to design your change file so that all changes to previous modules preserve the module numbering; then everybody's version will be consistent with the printed program. More extensive changes, which introduce new modules, can be inserted here; then only the index itself will get a new module number.

92. I enclose the RCS Keywords here as well, since that is how I keep track of versions.

$RCSfile: schedtools.aweb,v
$Revision: 1.5
$Date: 1997/08/24 22:27:29
$Author: evansjr
$Id: schedtools.aweb,v 1.5 1997/08/24 22:27:29 evansjr Exp evansjr
$Locker: evansjr
$State: Exp

93. Index. Here is a cross-reference table for the `schedtools` package. All modules in which an identifier is used are listed with that identifier, except that reserved words are indexed only when they appear in format definitions, and the appearances of identifiers in module names are not indexed. Underlined entries of subprograms and packages correspond to sections where this entity is specified, whereas entries in italic type correspond to the section where the entity's body is stated. For any other identifier underlined entries correspond to where the identifier was declared. Error messages and a few other things like "ASCII code" are indexed here too.

⟨ Compute myopic number 67 ⟩ Used in section 66.
⟨ Convert calendar times to absolute times 50 ⟩ Used in section 49.
⟨ Convert *ScheduleList* to *CalendarList* 48 ⟩ Used in section 47.
⟨ Copy linked lists and the designer matrix onto the stack 80 ⟩ Used in section 71.
⟨ Debug code set 1 68 ⟩ Used in section 66.
⟨ Debug code set 2 69 ⟩ Used in section 66.
⟨ Delete appropriate *R_Queue* record 78 ⟩ Used in section 71.
⟨ Do diagnostics 81 ⟩ Used in section 80.
⟨ Free up local linked lists 83 ⟩ Used in section 71.
⟨ Get appropriate *R_Queue* record 76 ⟩ Used in section 71.
⟨ Get next record 63 ⟩ Used in section 62.
⟨ Get output file name 46 ⟩ Used in section 45.
⟨ Initialize the lists for intensive list-processing 54 ⟩ Used in section 53.
⟨ Instantiate generics 16, 17, 18, 19, 20, 21, 22 ⟩ Used in section 12.
⟨ Move record from input list to ready list 64 ⟩ Used in section 62.
⟨ Package boiler-plate 12 ⟩ Used in section 10.
⟨ Procedures and Tasks in *schedtools* 42, 43, 44, 45, 47, 49, 52, 53, 57, 58, 62, 66, 70, 71 ⟩ Used in section 12.
⟨ Reset *InDegree* 87 ⟩ Used in sections 84 and 86.
⟨ Restore *F_Sched* 89 ⟩ Used in section 83.
⟨ Restore *R_Queue* 84 ⟩ Used in section 83.
⟨ Restore *S_List* 86 ⟩ Used in section 83.
⟨ Specification of procedures visible from *schedtools* 26, 27, 28, 29, 30, 31, 32, 34, 35, 36, 37, 38, 39 ⟩ Used in section 12.
⟨ Specification of types and variables visible from *schedtools* 23, 24, 25, 33, 59 ⟩ Used in section 12.
⟨ Update backtrack counter 74 ⟩ Used in section 71.
⟨ Update recursion stuff again 79 ⟩ Used in section 71.
⟨ Update some recursion stuff 72 ⟩ Used in section 71.
⟨ Variables local to *BranchAndBound* 73, 75, 77, 82, 85, 88, 90 ⟩ Used in section 71.
⟨ Variables local to *CreateNewStepList* 51 ⟩ Used in section 49.
⟨ Variables local to *schedtools* 41, 55, 56 ⟩ Used in section 12.

Schedule Primitives

[Ada '95—Version 1.0]
(Printed September 6, 1997)

This page intentionally left blank

1. Introduction. Here is the Ada code for utilites used in Salah Badr's scheduler program. His program was written by him May 25, 1993. It was translated by John Evans of NRaD into Donald Knuth's WEB format for literate programming. To compile and link the code in its present format you will need the Ada version of the WEB tool.

It is available on-line via the world-wide-web at URL:

http://white.nosc.mil/~evansjr/literate/

.

2. WEB is a literate programming paradigm for C, Pascal or Ada, and other languages. This style of programming is called "Literate Programming." For Further information see the paper *Literate Programming*, by Donald Knuth in *The Computer Journal*, Vol 27, No. 2, 1984; or the book *Weaving a Program: Literate Programming in* WEB by Wayne Sewell, Van Nostrand Reinhold, 1989. Another good source of information is the Usenet group *comp.programming.literate*. It has information on new tools and Frequently Asked Questions (FAQs).

3. Since the original AWEB package was written for Ada '83, it does not properly format new Ada '95 keywords **protected** and **private** . We remedy using the web format commands below.

format *protected* ≡ *procedure*
format *private* ≡ *procedure*

4. As a way of explanation, each "Module" withing angle brackets (< >) is expanded somewhere further down in the document. The trailing number you see within the brackets is where you can find this expansion. This provides a type of PDL (program descriptor language) for your program and greatly aids modularity and readability. It is also a highly effective method of top-down programming. The first module here is expanded further down, and contains most of the structure in standard Ada packages.

⟨ Package boiler-plate 5 ⟩

5. Schedule Primitives.

⟨ Package boiler-plate 5 ⟩ ≡

```
output to file schedprims.ads
with generic_set_pkg;
with generic_map_pkg;
with text_io;
use text_io;
with test_io_pkg;
use test_io_pkg;
with Ada.Calendar;
use Ada.calendar;
with capability;
use capability;
with ustrings;
use ustrings;
package schedprims is
   ⟨ Instantiate generics 9 ⟩
   ⟨ Specification of types and variables visible from schedprims 6 ⟩
   ⟨ Specification of procedures visible from schedprims 11 ⟩
end schedprims;
output to file schedprims.adb
with test_io_pkg;
with calyr;
use calyr;
package body schedprims is
   ⟨ Variables local to schedprims 25 ⟩
   ⟨ Procedures and Tasks in schedprims 26 ⟩
end schedprims;
```

This code is used in section 4.

6. I make this a *tagged* record so that I can extend it in other packages that inherit this one.

⟨ Specification of types and variables visible from *schedprims* 6 ⟩ ≡

```
type StepRecord is tagged record StepID : natural;
Deadline : natural ← 0;
Priority : natural;
EstimatedDuration : natural ← 0;
EarliestStartTime : natural ← 0;
ExpLevel : cap_map.map;
Successors : nat_set.set;
Predecessors : nat_set.set;
InDegree : natural ← 0;
RecursionLevel : natural ← 0;
end record;
```

See also sections 7 and 8.

This code is used in section 5.

7.

⟨ Specification of types and variables visible from *schedprims* 6 ⟩ +≡

```
type ScheduleRecord is
  record
    StepID : natural;
    StartTime : natural;
    FinishTime : natural;
    Designer : ustring;
    StepLevel : cap_map.map;
    RecursionLevel : natural ← 0;
  end record;
```

8.

⟨ Specification of types and variables visible from *schedprims* 6 ⟩ +≡

```
type CalendarRecord is
  record
    StepID : natural;
    StartTime : Time;
    FinishTime : Time;
    Designer : ustring;
    StepLevel : cap_map.map;
  end record;
```

9. Here is the specification for generics.

⟨ Instantiate generics 9 ⟩ ≡
  package *nat_set* **is new** *generic_set_pkg*(*natural*, 5);
    { Instantiate instances of the generic map package. }
  package *nat_map* **is new** *generic_map_pkg*(*key* ⇒ *natural*, *result* ⇒ *natural*);
  package *set_map* **is new** *generic_map_pkg*(*key* ⇒ *natural*, *result* ⇒ *nat_set*.*set*);
  package *exp_map* **is new** *generic_map_pkg*(*key* ⇒ *natural*, *result* ⇒ *ExpertiseLevel*);

See also section 10.

This code is used in section 5.

10. Here is the specification for generics.

⟨ Instantiate generics 9 ⟩ +≡
  package *nat_io* **is new** *integer_io*(*natural*);
  procedure *put_set* **is new** *nat_set*.*generic_put*;
  procedure *get_set* **is new** *nat_set*.*generic_input*;
  procedure *getf_set* **is new** *nat_set*.*generic_file_input*;
  package *enu_io* **is new** *text_io*.ENUMERATION_IO(*ExpertiseLevel*);

11. This function is used to compare the *ID* of *StepRecords*

⟨ Specification of procedures visible from *schedprims* 11 ⟩ ≡
  function *CompareID*(*L1*, *L2* : *StepRecord*)**return** *Boolean*;

See also sections 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, and 23.

This code is used in section 5.

12. This function is used to compare the *ID* of *StepRecords*

⟨ Specification of procedures visible from *schedprims* 11 ⟩ +≡
  function *IsEqual*(*L1*, *L2* : *StepRecord*)**return** *Boolean*;

13. This function is used to compare the *Deadline* of *StepRecords*

⟨ Specification of procedures visible from *schedprims* 11 ⟩ +≡
  function *CompareDeadLine*(*L1*, *L2* : *StepRecord*)**return** *Boolean*;

14. This function is used to compare the *Recursion* of *StepRecords*

⟨ Specification of procedures visible from *schedprims* 11 ⟩ +≡
  function *CompareRecursionLevel*(*L1*, *L2* : *StepRecord*)**return** *Boolean*;

15. This function is used to compare the *StartTime* of *StepRecords*

⟨ Specification of procedures visible from *schedprims* 11 ⟩ +≡
  function *CompareStartTime*(*L1*, *L2* : *ScheduleRecord*)**return** *Boolean*;

16. This function is used to compare the *StartTime* of *StepRecords*

⟨ Specification of procedures visible from *schedprims* 11 ⟩ +≡
  function *CompareStartTime*(*L1*, *L2* : *CalendarRecord*)**return** *Boolean*;

17. Printing a atep heading line before printing any records.

⟨ Specification of procedures visible from *schedprims* 11 ⟩ +≡
  procedure *StepRecordHeading*;

18. Display a record given its LOCATION in the list.

⟨ Specification of procedures visible from *schedprims* 11 ⟩ +≡
  procedure *DisplayStepRecord*(*rec* : **in** *StepRecord*);

19. Printing a schedule heading line before printing any record.

⟨ Specification of procedures visible from *schedprims* 11 ⟩ +≡
  procedure *ScheduleRecordHeading*;

20. Printing a schedule heading line before printing any record.

⟨ Specification of procedures visible from *schedprims* 11 ⟩ +≡
  procedure *CalendarRecordHeading*;

21. display a record given its LOCATION in the list.

⟨ Specification of procedures visible from *schedprims* 11 ⟩ +≡
  procedure *DisplayScheduleRecord*(*Current* : **in** *ScheduleRecord*);

22.

⟨ Specification of procedures visible from *schedprims* 11 ⟩ +≡
  procedure *SaveScheduleRecord*(*Current* : **in** *ScheduleRecord*; *fd* : *file_type*);

23. display a record given its LOCATION in the list.

⟨ Specification of procedures visible from *schedprims* 11 ⟩ +≡
  procedure *DisplayCalendarRecord*(*Current* : **in** *CalendarRecord*);

24. Schedule Primitives Body.

25.

⟨ Variables local to *schedprims* 25 ⟩ ≡
```
debug : boolean ← false;
debug2 : boolean ← false;
```
This code is used in section 5.

26.

⟨ Procedures and Tasks in *schedprims* 26 ⟩ ≡
```
function CompareID(L1, L2 : StepRecord)return Boolean is
begin
  if L1.StepId < L2.StepId then
    return True;
  else
    return False;
  end if;
end CompareID;
```
See also sections 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, and 38.

This code is used in section 5.

27. StepId's are suppose to be unique.

⟨ Procedures and Tasks in *schedprims* 26 ⟩ +≡
```
function IsEqual(L1, L2 : StepRecord)return Boolean is
begin
  if debug2 then
    put("L1.StepId␣=␣"); nat_io.put(L1.StepId, 1); put(".␣");
    put("L2.StepId␣=␣"); nat_io.put(L2.StepId, 1); put_line(".␣");
  end if;
  if L1.StepId = L2.StepId then
    return True;
  else
    return False;
  end if;
end IsEqual;
```

28.

⟨ Procedures and Tasks in *schedprims* 26 ⟩ +≡

```
function CompareDeadline(L1, L2 : StepRecord)return Boolean is
   answer : boolean;
   A, B : natural;
begin
   A ← L1.Deadline; B ← L2.Deadline;
   if debug then
      put("L1.Deadline␣=␣"); nat_io.put(A); put_line(".␣");
      put("L2.Deadline␣=␣"); nat_io.put(B); put_line(".␣");
   end if;
   if A < B then
      answer ← True;
   else
      answer ← false;
   end if;
   if debug then
      put("CompareDeadline>␣");
      if answer then
         nat_io.put(A); put("␣is␣LESS␣then␣"); nat_io.put(B); put_line(".␣");
      else
         nat_io.put(A); put("␣is␣NOT␣LESS␣then␣"); nat_io.put(B); put_line(".␣");
      end if;
   end if;
   return answer;
end CompareDeadline;
```

29.

⟨Procedures and Tasks in *schedprims* 26⟩ +≡

```
function CompareRecursionLevel(L1, L2 : StepRecord)return Boolean is
   answer : boolean;
   A, B : natural;
begin
   A ← L1.RecursionLevel; B ← L2.RecursionLevel;
   if A < B then
      answer ← True;
   else
      answer ← false;
   end if;
   if debug then
      put("CompareRecursionLevel>␣");
      if answer then
         nat_io.put(A); put("␣is␣LESS␣then␣"); nat_io.put(B); put_line(".␣");
      else
         nat_io.put(A); put("␣is␣NOT␣LESS␣then␣"); nat_io.put(B); put_line(".␣");
      end if;
   end if;
   return answer;
end CompareRecursionLevel;
```

30.

⟨Procedures and Tasks in *schedprims* 26⟩ +≡

```
function CompareStartTime(L1, L2 : ScheduleRecord)return Boolean is
begin
   if L1.StartTime < L2.StartTime then
      return True;
   else
      return False;
   end if;
end CompareStartTime;
```

31.

⟨ Procedures and Tasks in *schedprims* 26 ⟩ +≡

```
function CompareStartTime(L1, L2 : CalendarRecord)return Boolean is
begin
  if L1.StartTime < L2.StartTime then
    return True;
  else
    return False;
  end if;
end CompareStartTime;
```

32. Printing a step record heading line before printing any records.

⟨ Procedures and Tasks in *schedprims* 26 ⟩ +≡

```
procedure StepRecordHeading is
begin
  text_io.put("STEP_ID␣␣DEADLINE␣␣PRIORITY␣␣PREDECE␣␣␣SUCCESS␣␣E_LEVEL␣␣IN_DEGREE");
  text_io.put("␣␣RECURSION"); text_io.new_line;
  text_io.put("------␣␣␣-------␣␣␣--------␣␣------␣␣␣␣-------␣␣-------␣␣---------");
  text_io.put("␣␣---------"); text_io.new_line;
end StepRecordHeading;
```

33. Display a record given its LOCATION in the list.

⟨ Procedures and Tasks in *schedprims* 26 ⟩ +≡

```
procedure DisplayStepRecord(rec : in StepRecord) is
begin
  text_io.set_col(4); test_io_pkg.put(rec.StepId); text_io.set_col(12);
  test_io_pkg.put(rec.Deadline); text_io.set_col(23); test_io_pkg.put(rec.Priority);
  text_io.set_col(31); put_set(rec.Predecessors); text_io.set_col(41);
  put_set(rec.Successors); text_io.set_col(49); print_capabilities(rec.ExpLevel);
  text_io.set_col(61); test_io_pkg.put(rec.InDegree); text_io.set_col(72);
  test_io_pkg.put(rec.RecursionLevel); text_io.new_line;
end DisplayStepRecord;
```

34. Printing a schedule heading line before printing any record.

⟨ Procedures and Tasks in *schedprims* 26 ⟩ +≡

```
procedure ScheduleRecordHeading is
begin
  text_io.put("ID␣START_TIME␣FINISH_TIME␣␣S_LEVEL␣␣␣␣␣␣␣␣␣␣␣DEVEOPER");
  text_io.new_line;
  text_io.put("--␣----------␣-----------␣␣-------␣␣␣␣␣␣␣␣␣␣␣--------");
  text_io.new_line;
end ScheduleRecordHeading;
```

35. Printing a schedule heading line before printing any record.

⟨ Procedures and Tasks in *schedprims* 26 ⟩ +≡

```
procedure CalendarRecordHeading is
begin
  text_io.put("ID␣START_TIME␣FINISH_TIME␣␣S_LEVEL␣␣␣␣␣␣␣␣␣␣␣DEVEOPER");
  text_io.new_line;
  text_io.put("--␣----------␣-----------␣␣-------␣␣␣␣␣␣␣␣␣␣␣--------");
  text_io.new_line;
end CalendarRecordHeading;
```

36. Display a record given its LOCATION in the list.

⟨ Procedures and Tasks in *schedprims* 26 ⟩ +≡

```
procedure DisplayScheduleRecord(Current : in ScheduleRecord) is
begin
  text_io.set_col(1); nat_io.put(Current.StepID, 1); text_io.set_col(10);
  nat_io.put(Current.StartTime, 1); text_io.set_col(20);
  nat_io.put(Current.FinishTime, 1); text_io.set_col(35);
  print_capabilities(Current.StepLevel); text_io.put("␣");
  text_io.put(S(Current.Designer)); text_io.new_line;
end DisplayScheduleRecord;
```

37. Display a record given its LOCATION in the list.

⟨ Procedures and Tasks in *schedprims* 26 ⟩ +≡

```
procedure SaveScheduleRecord(Current : in ScheduleRecord; fd : file_type) is
  package Nat_Io is new Integer_Io(Natural);
  use Nat_Io;
begin
  text_io.set_col(fd, 1); put(fd, Current.StepID, 1); text_io.set_col(fd, 10);
  put(fd, Current.StartTime, 1); text_io.set_col(fd, 20);
  put(fd, Current.FinishTime, 1); text_io.set_col(fd, 35);
  print_capabilities(fd, Current.StepLevel); put(fd, "␣"); put(fd, Current.Designer);
  text_io.new_line(fd);
end SaveScheduleRecord;
```

38. Display a record given its LOCATION in the list.

⟨ Procedures and Tasks in *schedprims* 26 ⟩ +≡

```
procedure DisplayCalendarRecord(Current : in CalendarRecord) is
begin
   text_io.set_col(2); test_io_pkg.put(Current.StepID); text_io.set_col(10);
   calyr.print_date(Current.StartTime); text_io.set_col(25);
   calyr.print_date(Current.FinishTime); text_io.set_col(40);
   print_capabilities(Current.StepLevel); text_io.put("␣␣");
   text_io.put(S(Current.Designer)); text_io.new_line;
end DisplayCalendarRecord;
```

39. System-dependent changes. This module should be replaced, if necessary, by changes to the program that are necessary to make `MAIN` work at a particular installation. It is usually best to design your change file so that all changes to previous modules preserve the module numbering; then everybody's version will be consistent with the printed program. More extensive changes, which introduce new modules, can be inserted here; then only the index itself will get a new module number.

40. RCS Keywords.

$RCSfile: schedprims.aweb,v
$Revision: 1.4
$Date: 1997/08/22 23:14:45
$Author: evansjr
$Id: schedprims.aweb,v 1.4 1997/08/22 23:14:45 evansjr Exp evansjr
$Locker: evansjr
$State: Exp

41. Index. Here is a cross-reference table for the `MAIN` program. All modules in which an identifier is used are listed with that identifier, except that reserved words are indexed only when they appear in format definitions, and the appearances of identifiers in module names are not indexed. Underlined entries of subprograms and packages correspond to sections where this entity is specified, whereas entries in italic type correspond to the section where the entity's body is stated. For any other identifier underlined entries correspond to where the identifier was declared. Error messages and a few other things like "ASCII code" are indexed here too.

The Project Scheduler

[Ada '95—Version 1.0]
September 18, 1997

This page intentionally left blank

1. Introduction. Here is the Ada code for Salah Badr's scheduler program. It was written by him May 25, 1993. Here it has been translated to Donald Knuth's `WEB` format for literate programming. To compile and link the code in its present format you will need the Ada version of the `WEB` tool.

It is available on-line via the world-wide-web at URL:

http://white.nosc.mil/~evansjr/literate/

.

2. `WEB` is a literate programming paradigm for C, Pascal or Ada, and other languages. This style of programming is called "Literate Programming." For Further information see the paper *Literate Programming*, by Donald Knuth in *The Computer Journal*, Vol 27, No. 2, 1984; or the book *Weaving a Program: Literate Programming in* `WEB` by Wayne Sewell, Van Nostrand Reinhold, 1989. Another good source of information is the Usenet group *comp.programming.literate*. It has information on new tools and Frequently Asked Questions (FAQs).

3. The program consists of several packages that are declared right now; each of these packages and either the specification and the body of the packages are sent to a separate file. The main program itself is declared later. (Since the original AWEB package was written for Ada '83, it does not properly format new Ada '95 keywords **protected** and **private** . We remedy using the web format commands below.

format *protected* ≡ *procedure*
format *private* ≡ *procedure*

4. As a way of explanation, each "Module" within angle brackets (< >) is expanded somewhere further down in the document. The trailing number you see within the brackets is where you can find this expansion. This provides a type of PDL (program descriptor language) for your program and greatly aids modularity and readability. It is also a highly effective method of top-down programming.

5. Main driver. This is the main routine that starts everything.

6. (Note: The following format is used by all the packages. We write the top-level code, in macro-level descriptions, and it gets expanded into code further down. This way you can write small, easily understood modules. It also lets you declare and describe variables and types where you need them.)

output to file `main.adb`

```
pragma suppress(all_checks);
with SchedTools;
with scheduler;
use scheduler;
with text_io;
use text_io;
with capability;
use capability;
with ustrings;
use ustrings;
procedure main is ⟨ Instantiate generic packages 8 ⟩⟨ Variables local to main 9 ⟩
  begin
    loop
      begin
        SCHEDULER_MENU; get(SELECTOR); skip_line;
        case SELECTOR is
          when 1 ⇒
            ⟨ Create new step list 10 ⟩
          when 2 ⇒
            ⟨ Read in developer list 12 ⟩
          when 3 ⇒
            ⟨ Schedule steps according to their deadlines 14 ⟩
          when 4 ⇒
            ⟨ Print all steps in the ready queue 15 ⟩
          when 5 ⇒
            ⟨ Print all step records 19 ⟩
          when 6 ⇒
            ⟨ Print final schedule 16 ⟩
          when 7 ⇒
            ⟨ Save final schedule 17 ⟩
          when 8 ⇒
            ⟨ Print calendar schedule 18 ⟩
          when 9 ⇒
            ⟨ Exit the program to the system 20 ⟩
          when others ⇒
            ⟨ Exception handling for selector case 21 ⟩
        end case;
      exception
```

```
      when storage_error ⇒
        put_line("You␣have␣a␣storage␣error.");
        put("Your␣level␣of␣recursion␣is␣"); nat_io.put(recursion_level);
        put_line(".␣");
      when Data_Error ⇒
        put_line("Value␣entered␣not␣in␣proper␣range.␣␣Please␣try␣again.");
        New_line; Skip_Line;
      when SchedTools.NoFeasibleScheduleFound ⇒
        put_line("Unable␣to␣find␣feasible␣schedule.␣␣Need␣to␣increase␣laxity."
        New_line;
      when NoDevelopers ⇒
        put_line("No␣developers␣to␣schedule␣tasks␣with.␣␣Please␣try␣again.");
        New_line;
      end;
    end loop;
  end main;
```

7. As a way of explanation, each "Module" withing angle brackets (<   >) is expanded somewhere further down in the document. The trailing number you see within the brackets is where you can find this expansion. This provides a type of PDL (program descriptor language) for your program and greatly aids modularity and readability. It is also a highly effective method of top-down programming. The first module here is expanded further down, and contains most of the structure in standard Ada packages.

⟨ Package boiler-plate 22 ⟩

8.

```
⟨ Instantiate generic packages 8 ⟩ ≡
  package nat_io is new integer_io(natural);
  use nat_io;
```

This code is used in section 6.

9.

```
⟨ Variables local to main 9 ⟩ ≡
  type selector_type is new natural range 1 .. 9;
  selector : selector_type ← 1;
  package sel_io is new integer_io(selector_type);
  use sel_io;
```

See also sections 11 and 13.

This code is used in section 6.

10. This routine has been modified to read in a file and build up the linked list of "steps."

⟨ Create new step list 10 ⟩ ≡
  if *num_developers* > 0 **then**
    MakeNewStepList(*num_developers*);
  else
    raise *NoDevelopers*;
  end if;

This code is used in section 6.

11.

⟨ Variables local to main 9 ⟩ +≡
  NoDevelopers : **exception**;

12.

⟨ Read in developer list 12 ⟩ ≡
  put_line("Please␣enter␣developer␣file␣name:␣");
  get_line(*infile*);
  get_developers(*S*(*infile*)); *num_developers* ← *get_num_developers*;

This code is used in section 6.

13.

⟨ Variables local to main 9 ⟩ +≡
  infile : *ustring*;
  num_developers : *natural* ← 0;

14.

⟨ Schedule steps according to their deadlines 14 ⟩ ≡
  Put_line("Scheduling␣steps␣according␣to␣their␣deadlines.");
  MakeDeadlineFirstSchedule(*max_recursion*, *num_developers*);

This code is used in section 6.

15.

⟨ Print all steps in the ready queue 15 ⟩ ≡
  PrintReadyQueue;

This code is used in section 6.

16.

⟨ Print final schedule 16 ⟩ ≡
  PrintFinalSchedule;

This code is used in section 6.

17.

⟨ Save final schedule 17 ⟩ ≡
SaveFinalSchedule;

This code is used in section 6.

18.

⟨ Print calendar schedule 18 ⟩ ≡
PrintCalendarSchedule;

This code is used in section 6.

19.

⟨ Print all step records 19 ⟩ ≡
new_line; *PrintStepList*;

This code is used in section 6.

20.

⟨ Exit the program to the system 20 ⟩ ≡
```
put("Maximum␣recursion␣level␣is␣"); nat_io.put(max_recursion); put_line(".␣");
put("Current␣recursion␣level␣is␣"); nat_io.put(recursion_level); put_line(".␣");
put("thank␣you␣....␣Bye␣...Bye"); new_line; exit;
```

This code is used in section 6.

21.

⟨ Exception handling for selector case 21 ⟩ ≡
```
put("␣BAD␣CHOICE.␣PLEASE␣TRY␣AGAIN"); new_line;
```

This code is used in section 6.

22. Scheduler specification.

⟨ Package boiler-plate 22 ⟩ ≡

```
output to file scheduler.ads
with generic_set_pkg;
with generic_map_pkg;
with generic_list;
with schedprims;
use schedprims;
with schedtools;
use schedtools;
with TEXT_IO;
use TEXT_IO;
with test_io_pkg;
use test_io_pkg;
package scheduler is
   ⟨ Specification of types and variables visible from scheduler 23 ⟩
   ⟨ Specification of procedures visible from scheduler 24 ⟩
end scheduler;
output to file scheduler.adb
with unchecked_deallocation;
package body scheduler is
   ⟨ Procedures and Tasks in scheduler 33 ⟩
end scheduler;
```

This code is used in section 7.

23. Here are variables global to the recursion.

⟨ Specification of types and variables visible from scheduler 23 ⟩ ≡

```
recursion_level : natural ← 0;
max_recursion : natural ← 0;
```

This code is used in section 22.

24. Creating new step.

⟨ Specification of procedures visible from scheduler 24 ⟩ ≡

```
Procedure MakeNewStepList(num_developers : natural);
```

See also sections 25, 26, 27, 28, 29, 30, and 31.

This code is used in section 22.

25. Creating new step.

⟨ Specification of procedures visible from scheduler 24 ⟩ +≡

```
Procedure MakeDeadlineFirstSchedule(max_recursion : in out natural;
      num_developers : natural);
```

26.

⟨ Specification of procedures visible from *scheduler* 24 ⟩ +≡
procedure *SCHEDULER_MENU*;

27.

⟨ Specification of procedures visible from *scheduler* 24 ⟩ +≡
procedure *PrintReadyQueue*;

28.

⟨ Specification of procedures visible from *scheduler* 24 ⟩ +≡
procedure *PrintFinalSchedule*;

29.

⟨ Specification of procedures visible from *scheduler* 24 ⟩ +≡
procedure *SaveFinalSchedule*;

30.

⟨ Specification of procedures visible from *scheduler* 24 ⟩ +≡
procedure *PrintCalendarSchedule*;

31.

⟨ Specification of procedures visible from *scheduler* 24 ⟩ +≡
procedure *PrintStepList*;

32. Scheduler Body.

33. Creating new step.

⟨ Procedures and Tasks in *scheduler* 33 ⟩ ≡
```
Procedure MakeNewStepList(num_developers : natural) is
begin
  CreateNewStepList(StepList);
end MakeNewStepList;
```
See also sections 34, 35, 36, 37, 38, 39, and 40.

This code is used in section 22.

34.

⟨ Procedures and Tasks in *scheduler* 33 ⟩ +≡
```
Procedure MakeDeadLineFirstSchedule(max_recursion : in out natural;
    num_developers : natural) is
begin
  put_line("Start␣of␣CreateDeadlineFirstSchedule.");
  CreateDeadlineFirstSchedule(max_recursion, num_developers);
  put_line("End␣of␣CreateDeadlineFirstSchedule.");
end MakeDeadLineFirstSchedule;
```

35. DISPLAY THE MAIN MENU.

⟨ Procedures and Tasks in *scheduler* 33 ⟩ +≡

```
procedure SCHEDULER_MENU is
begin
  new_line; set_col(25); put("MAIN␣MENU"); new_line; set_col(25);
  put("----------"); new_line(2);
  set_col(5); put("[1]␣Read␣in␣step␣list");
  new_line;
  set_col(5); put("[2]␣Read␣in␣developer␣list");
  new_line;
  set_col(5); put("[3]␣schedule␣steps␣using␣BranchAndBound");
  new_line;
  set_col(5); put("[4]␣Print␣ready␣queue");
  new_line;
  set_col(5); put("[5]␣Print␣step␣list");
  new_line;
  set_col(5); put("[6]␣Print␣final␣schedule");
  new_line;
  set_col(5); put("[7]␣Save␣final␣schedule");
  new_line;
  set_col(5); put("[8]␣Print␣Calendar␣schedule");
  new_line;
  set_col(5); put("[9]␣Quit"); new_line(3); set_col(5);
  put("Enter␣the␣number␣of␣your␣choice␣:␣␣");
end SCHEDULER_MENU;
```

36.

⟨ Procedures and Tasks in *scheduler* 33 ⟩ +≡

```
procedure PrintFinalSchedule is
begin
  PrintAllScheduleRecords(Schedule);
end PrintFinalSchedule;
```

37.

⟨ Procedures and Tasks in *scheduler* 33 ⟩ +≡

```
procedure SaveFinalSchedule is
begin
  SaveAllScheduleRecords(Schedule);
end SaveFinalSchedule;
```

38.

⟨Procedures and Tasks in *scheduler* 33⟩ +≡

```
procedure PrintCalendarSchedule is
begin
   PrintAllCalendarRecords(Schedule);
end PrintCalendarSchedule;
```

39.

⟨Procedures and Tasks in *scheduler* 33⟩ +≡

```
procedure PrintReadyQueue is
begin
   PrintAllStepRecords(ReadyQueue);
end PrintReadyQueue;
```

40.

⟨Procedures and Tasks in *scheduler* 33⟩ +≡

```
procedure PrintStepList is
begin
   PrintAllStepRecords(StepList);
end PrintStepList;
```

41. Continuous Time to Calendar Time Translator. The purpose of this routine is to take the output of the scheduler and translate the continuous time fields (*StartTime* and *FinishTime*) to calendar dates.

output to file contocal.adb

```
pragma suppress(all_checks);
with text_io;
use text_io;
with getopt;
use getopt;
with Ustrings;
use Ustrings;
with Ada.Calendar;
use Ada.Calendar;
with calyr;
use calyr;
with capability;
use capability;
procedure ConToCal is
   ⟨ Variables local to ConToCal 45 ⟩
   package bool_io is new enumeration_io(boolean);
   use bool_io;
begin
   ⟨ Get parameters to ConToCal 43 ⟩
   ⟨ Open files 51 ⟩
   ⟨ Iterate through input file 53 ⟩ end ConToCal;
```

42. The command syntax is as follows:

contocal [-nrad < *boolean* >] [-start < *startdate* >] infile outfile

43. The **-nrad** option is by default false, but when set to *true* will create a schedule that respects NRaD off-fridays. An example invocation coule be:

contocal -nrad true -start 07/03/97+00 infile outfile

If no start is given then the default is the same as the example.

```
⟨ Get parameters to ConToCal 43 ⟩ ≡
   ⟨ Get nrad 44 ⟩
   ⟨ Get start date 46 ⟩
   ⟨ Get input file 48 ⟩
   ⟨ Get output file 50 ⟩
```

This code is used in section 41.

44.

⟨ Get nrad 44 ⟩ ≡

```
if option_present(U("-nrad")) then
   get_option(U("-nrad"), param); get(S(param), nrad, Last);
else
   nrad ← false;
end if;
```

This code is used in section 43.

45.

⟨ Variables local to ConToCal 45 ⟩ ≡

```
param : Ustring;
Last : positive;
nrad : boolean;
```

See also sections 47, 49, 52, 55, 56, and 58.

This code is used in section 41.

46.

⟨ Get start date 46 ⟩ ≡

```
if option_present(U("-start")) then
   get_option(U("-start"), param); StartDate ← get_date(param);
else
   StartDate ← Time_Of(1997, 7, 3, 0.0);
end if;
```

This code is used in section 43.

47.

⟨ Variables local to ConToCal 45 ⟩ +≡

```
StartDate : Time;
```

48.

⟨ Get input file 48 ⟩ ≡

```
if name_present(1) then
   get_name(infile, 1);
else
   raise nofilename;
end if;
```

This code is used in section 43.

49.

⟨ Variables local to ConToCal 45 ⟩ +≡
  nofilename : **exception**;
  infile, *outfile* : *Ustring*;

50.

⟨ Get output file 50 ⟩ ≡
  if *name_present*(2) **then**
    get_name(*outfile*, 2);
  else
    raise *nofilename*;
  end if;

This code is used in section 43.

51.

⟨ Open files 51 ⟩ ≡
  open(*data_file*, *in_file*, *S*(*infile*)); *create*(*data2_file*, *out_file*, *S*(*outfile*));

This code is used in section 41.

52.

⟨ Variables local to ConToCal 45 ⟩ +≡
  data_file, *data2_file* : *file_type*;

53.

⟨ Iterate through input file 53 ⟩ ≡
  while ¬*End_Of_File*(*data_file*) **loop**
    ⟨ Read in record 54 ⟩
    ⟨ Do time translations 57 ⟩
    ⟨ Write out new record 59 ⟩
  end loop;

This code is used in section 41.

54. A typical input file would look like the following:

3	HIGH	H1	0	3
2	MEDIUM	M1	0	4
1	LOW	L1	0	6
4	HIGH	H1	3	13
5	MEDIUM	M1	4	12
6	LOW	L1	6	10
8	MEDIUM	M1	12	14
7	LOW	L1	10	15
9	HIGH	H1	13	19
10	MEDIUM	M1	14	24

The second to last column is the start time and the last column is the end time.

⟨ Read in record 54 ⟩ ≡
kntr ← *kntr* + 1; *put*("Reading␣in␣record␣"); *put*(*kntr*); *put_line*(".␣");
get(*data_file*, *stepid*); *get*(*data_file*, *Start*); *get*(*data_file*, *Finish*);
get_capability(*data_file*, *ExpLevel*); *get_line*(*data_file*, *Developer*);

This code is used in section 53.

55.

⟨ Variables local to ConToCal 45 ⟩ +≡
type *ExpertiseLevel* **is** (*low*, *medium*, *high*);
stepid : *natural*;
ExpLevel : *cap_map.map*;
Developer : *ustring*;
start, *finish* : *natural*;
kntr : *natural* ← 0;

56.

⟨ Variables local to ConToCal 45 ⟩ +≡
package *exp_io* **is new** *enumeration_io*(*ExpertiseLevel*);
use *exp_io*;
package *nat_io* **is new** *integer_io*(*natural*);
use *nat_io*;

57.

⟨ Do time translations 57 ⟩ ≡
dur ← *ConvertHoursToDuration*(*Start*);
StartTime ← *DurationToCalendarTime*(*StartDate*, *dailyhours*, *dur*, *NRaD*);
dur ← *ConvertHoursToDuration*(*Finish*);
FinishTime ← *DurationToCalendarTime*(*StartDate*, *dailyhours*, *dur*, *NRaD*);

This code is used in section 53.

58.

⟨ Variables local to ConToCal 45 ⟩ +≡

```
dur : Duration;
StartTime, FinishTime : Time;
dailyhours : WorkHours ← (ConvertHoursToDuration(8), ConvertHoursToDuration(8),
    ConvertHoursToDuration(8), ConvertHoursToDuration(8),
    ConvertHoursToDuration(8));
```

59.

⟨ Write out new record 59 ⟩ ≡

```
set_col(data2_file, 1); put(data2_file, stepid, 1); set_col(data2_file, 10);
print_date(data2_file, StartTime); set_col(data2_file, 25);
print_date(data2_file, FinishTime); set_col(data2_file, 40);
print_capabilities(data2_file, ExpLevel); put(data2_file, "␣");
put(data2_file, Developer); new_line(data2_file);
```

This code is used in section 53.

60. System-dependent changes. This module should be replaced, if necessary, by changes to the program that are necessary to make `MAIN` work at a particular installation. It is usually best to design your change file so that all changes to previous modules preserve the module numbering; then everybody's version will be consistent with the printed program. More extensive changes, which introduce new modules, can be inserted here; then only the index itself will get a new module number.

61. RCS Keywords.

$RCSfile: main.aweb,v
$Revision: 1.5
$Date: 1997/08/22 23:14:45
$Author: evansjr
$Id: main.aweb,v 1.5 1997/08/22 23:14:45 evansjr Exp evansjr
$Locker: evansjr
$State: Exp

62. Index. Here is a cross-reference table for the MAIN program. All modules in which an identifier is used are listed with that identifier, except that reserved words are indexed only when they appear in format definitions, and the appearances of identifiers in module names are not indexed. Underlined entries of subprograms and packages correspond to sections where this entity is specified, whereas entries in italic type correspond to the section where the entity's body is stated. For any other identifier underlined entries correspond to where the identifier was declared. Error messages and a few other things like "ASCII code" are indexed here too.

⟨ Create new step list 10 ⟩ Used in section 6.
⟨ Do time translations 57 ⟩ Used in section 53.
⟨ Exception handling for selector case 21 ⟩ Used in section 6.
⟨ Exit the program to the system 20 ⟩ Used in section 6.
⟨ Get input file 48 ⟩ Used in section 43.
⟨ Get nrad 44 ⟩ Used in section 43.
⟨ Get output file 50 ⟩ Used in section 43.
⟨ Get parameters to *ConToCal* 43 ⟩ Used in section 41.
⟨ Get start date 46 ⟩ Used in section 43.
⟨ Instantiate generic packages 8 ⟩ Used in section 6.
⟨ Iterate through input file 53 ⟩ Used in section 41.
⟨ Open files 51 ⟩ Used in section 41.
⟨ Package boiler-plate 22 ⟩ Used in section 7.
⟨ Print all step records 19 ⟩ Used in section 6.
⟨ Print all steps in the ready queue 15 ⟩ Used in section 6.
⟨ Print calendar schedule 18 ⟩ Used in section 6.
⟨ Print final schedule 16 ⟩ Used in section 6.
⟨ Procedures and Tasks in *scheduler* 33, 34, 35, 36, 37, 38, 39, 40 ⟩ Used in section 22.
⟨ Read in developer list 12 ⟩ Used in section 6.
⟨ Read in record 54 ⟩ Used in section 53.
⟨ Save final schedule 17 ⟩ Used in section 6.
⟨ Schedule steps according to their deadlines 14 ⟩ Used in section 6.
⟨ Specification of procedures visible from *scheduler* 24, 25, 26, 27, 28, 29, 30, 31 ⟩ Used in section 22.
⟨ Specification of types and variables visible from *scheduler* 23 ⟩ Used in section 22.
⟨ Variables local to ConToCal 45, 47, 49, 52, 55, 56, 58 ⟩ Used in section 41.
⟨ Variables local to main 9, 11, 13 ⟩ Used in section 6.
⟨ Write out new record 59 ⟩ Used in section 53.

Generic List processing routines

[Ada '95—Version 1.0]

This page intentionally left blank

1. Introduction. The scheduler designed and implemented by Salah Badr uses lists extensively. However, it has specific routines for each list used by the scheduler. This is redundant, as well as error prone. In my design to eliminate some large data structures that are slightly modified and duplicated in a very recursive and space consuming manner, I have decided to use additional linked lists to keep track of additions and deletions at each level of recursion. By keeping track of just the "changes" This will turn an N squared space problem into one that is linear. (This is shown to be true, later.)

2. So since linked lists are used extensively, it pays to have a single **generic** routine. The implementation is thus hidden from the user. This allows "information-hiding," and increased modularity,

3. This code is written using Donald Knuth's **WEB** paradigm for literate programming. To compile and link the code in its present format you will need the Ada version of the **WEB** tool.

It is available on-line via the world-wide-web at URL:

http://white.nosc.mil/~evansjr/literate/

.

4. **WEB** is a literate programming paradigm for C, Pascal or Ada, and other languages. This style of programming is called "Literate Programming." For Further information see the paper *Literate Programming*, by Donald Knuth in *The Computer Journal*, Vol 27, No. 2, 1984; or the book *Weaving a Program: Literate Programming in* **WEB** by Wayne Sewell, Van Nostrand Reinhold, 1989. Another good source of information is the Usenet group *comp.programming.literate*. It has information on new tools and Frequently Asked Questions (FAQs).

5. The program consists of several packages that are declared right now; each of these packages and either the specification and the body of the packages are sent to a separate file. The main program itself is declared later. (Since the original AWEB package was written for Ada '83, it does not properly format new Ada '95 keywords **protected** and **private** . We remedy using the web format commands below.

format *protected* ≡ *procedure*
format *private* ≡ *procedure*

6. As a way of explanation, each "Module" withing angle brackets (< >) is expanded somewhere further down in the document. The trailing number you see within the brackets is where you can find this expansion. It is top-down in appearance, and in actual fact.

7. All the modules follow the same, top-down format. I will group all the boiler-plate into one module, for the compiler, but you will see it with the packages, as they are described.

⟨ Package boiler-plate 8 ⟩

8. List Specification. This specification is a modification of the one presented in the book *Ada 95 Problem Solving and Program Design*, by Michael Feldman and Elliot B. Koffman. The implementation was left as an exercise for the student.

```
⟨ Package boiler-plate 8 ⟩ ≡
  output to file generic_list.ads
  with TEXT_IO;
  use TEXT_IO;
  generic
  type ElementType is private ; { Any nonlimited type will do }
  with procedure DisplayElement(Item : IN ElementType);
  with function "<"(L1 , L2 : ElementType)return Boolean;
  with function "="(L1 , L2 : ElementType)return Boolean is <>;
  package generic_list is
    ⟨ Specification of types and variables visible from generic_list 9 ⟩⟨ Specification of
        procedures visible from generic_list 12 ⟩ private
    ⟨ Specification of private types and variables in generic_list 10 ⟩
  end generic_list;
  output to file generic_list.adb
  ⟨ Packages needed by generic_list body 32 ⟩
  package body generic_list is
    ⟨ Variables local to generic_list 30 ⟩
    ⟨ Procedures and Tasks in generic_list 33 ⟩
  end generic_list;
```

This code is used in section 7.

9.

```
⟨ Specification of types and variables visible from generic_list 9 ⟩ ≡
  type list is limited private ;
  ListEmpty : exception;
```

This code is used in section 8.

10.

```
⟨ Specification of private types and variables in generic_list 10 ⟩ ≡
  type ListNode; type ListPtr is access ListNode;
  type ListNode is
    record
      Element : ElementType;
      Next : ListPtr;
    end record;
```

See also section 11.

This code is used in section 8.

11. Added *Size* field to original code.

⟨ Specification of private types and variables in *generic_list* 10 ⟩ +≡
type *List* **is**
  record
    Size : *Natural*;
    Head : *ListPtr*;
    Tail : *ListPtr*;
    Current : *ListPtr*;
    Previous : *ListPtr*;
  end record;

12.

⟨ Specification of procedures visible from *generic_list* 12 ⟩ ≡
function *ListSize*(*L* : **in** *List*)**return** *natural*;

See also sections 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, and 28.

This code is used in section 8.

13. Returns *True* if *L* is empty, *False* otherwise.

⟨ Specification of procedures visible from *generic_list* 12 ⟩ +≡
function *IsEmpty*(*L* : *IN List*)*RETURN Boolean*;

14.
Pre: *Element* is defined; *L* may be empty.
Post: *Element* is inserted at the beginning of *L*.

⟨ Specification of procedures visible from *generic_list* 12 ⟩ +≡
procedure *AddToFront*(*L* : **in out** *List*; *Element* : **in** *ElementType*);

15.
Pre: *L* is defined; *L* may be empty.
Post: returns a complete copy of the list *L*.
Raises: *ListEmpty* if the list is empty before the retrieval.

⟨ Specification of procedures visible from *generic_list* 12 ⟩ +≡
function *RetrieveFront*(*L* : **in** *List*)**return** *ElementType*;

16.
Pre: *L* is defined; *L* may be empty.
Post: The first node of *L* is removed.
Raises: *ListEmpty* if the list is empty before the removal.

⟨ Specification of procedures visible from *generic_list* 12 ⟩ +≡
procedure *RemoveFront*(*L* : **in out** *List*);

17.
Pre: L is defined.
Post: L is empty.

⟨ Specification of procedures visible from *generic_list* 12 ⟩ +≡
procedure *MakeEmpty*(*L* : **in out** *List*);

18.
Pre: *Element* is defined; *L* may be empty.
Post: *Element* is appended to the end of *L*.

⟨ Specification of procedures visible from *generic_list* 12 ⟩ +≡
procedure *AddToEnd*(*L* : **in out** *List*; *Element* : **in** *ElementType*);

19.
Pre: Source may be empty.
Post: Returns a complete copy of Source in Target.

⟨ Specification of procedures visible from *generic_list* 12 ⟩ +≡
procedure *Copy*(*Source* : **in** *List*; *Target* : **out** *List*);

20.
Pre: *L* may be empty.
Post: displays the contents of *L*'s *Element* fields, in the order in which they appear in *L*.

⟨ Specification of procedures visible from *generic_list* 12 ⟩ +≡
procedure *Display*(*L* : *IN List*);

21.

⟨ Specification of procedures visible from *generic_list* 12 ⟩ +≡
procedure *InsertInOrder*(*L* : **in out** *List*; *Element* : *ElementType*);

22.

⟨ Specification of procedures visible from *generic_list* 12 ⟩ +≡
procedure *GetNext*(*L* : **in out** *List*; *Element* : **out** *ElementType*);

23.

⟨ Specification of procedures visible from *generic_list* 12 ⟩ +≡
procedure *DeleteCurrent*(*L* : **in out** *List*);

24.

⟨ Specification of procedures visible from *generic_list* 12 ⟩ +≡
procedure *DeleteMatching*(*L* : **in out** *List*; *Element* : **in** *ElementType*; *success* : **out** *boolean*);

25.

⟨ Specification of procedures visible from *generic_list* 12 ⟩ +≡
procedure *GetCurrent*(*L* : **in** *List*; *Element* : **out** *ElementType*);

26.

⟨ Specification of procedures visible from *generic_list* 12 ⟩ +≡
procedure *UpdateCurrent*(*L* : **in** *List*; *Element* : **in** *ElementType*);

27.

⟨ Specification of procedures visible from *generic_list* 12 ⟩ +≡
procedure *GetNth*(*L* : **in out** *List*; *N* : **in** *natural*; *Element* : **out** *ElementType*);

28.

⟨ Specification of procedures visible from *generic_list* 12 ⟩ +≡
procedure *Rewind*(*L* : **in out** *List*);

29. List Body.

30.

⟨ Variables local to *generic_list* 30 ⟩ ≡
   debug : *boolean* ← *false*;

See also section 31.

This code is used in section 8.

31.

⟨ Variables local to *generic_list* 30 ⟩ +≡
   procedure *Dispose* **is new** *unchecked_deallocation*(*Object* ⇒ *ListNode*,
         Name ⇒ *ListPtr*);
   package *nat_io* **is new** *integer_io*(*natural*);

32.

⟨ Packages needed by *generic_list* body 32 ⟩ ≡
   with *unchecked_deallocation*;
   with *Ustrings*;
   use *Ustrings*;

This code is used in section 8.

33.

⟨ Procedures and Tasks in *generic_list* 33 ⟩ ≡
   function *ListSize*(*L* : **in** *List*)**return** *Natural* **is**
   begin
      return *L.Size*;
   end *ListSize*;

See also sections 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, and 49.

This code is used in section 8.

34. Returns *True* if *L* is empty, *False* otherwise.

⟨ Procedures and Tasks in *generic_list* 33 ⟩ +≡
   function *IsEmpty*(*L* : *IN List*)*RETURN Boolean* **is**
   begin
      if *ListSize*(*L*) = 0 **then**
         return *True*;
      else
         return *False*;
      end if;
   end *IsEmpty*;

35.

Pre: *Element* is defined; *L* may be empty.
Post: *Element* is inserted at the beginning of *L*.

⟨ Procedures and Tasks in *generic_list* 33 ⟩ +≡

```
procedure AddToFront(L : in out List; Element : in ElementType) is
   Temp : ListPtr;
begin
   Temp ← new ListNode; Temp.all.Element ← Element; Temp.all.Next ← L.Head;
   L.Head ← Temp; L.Size ← L.Size + 1;
   if L.Size = 1 then
      L.Tail ← L.Head;
   end if;
end AddToFront;
```

36.

Pre: *L* is defined; *L* may be empty.
Post: returns a complete copy of the list *L*.
Raises: *ListEmpty* if the list is empty before the retrieval.

⟨ Procedures and Tasks in *generic_list* 33 ⟩ +≡

```
function RetrieveFront(L : in List)return ElementType is
   Temp : ListPtr;
begin
   if L.Head = null then
      raise ListEmpty;
   else   { L.Head points to a node; remove it }
      Temp ← L.Head; return Temp.Element;
   end if;
end RetrieveFront;
```

37.

Pre: L is defined; L may be empty.
Post: The first node of L is removed.
Raises: *ListEmpty* if the list is empty before the removal.

```
⟨Procedures and Tasks in generic_list 33⟩ +≡
  procedure RemoveFront(L : in out List) is
    Temp : ListPtr;
  begin
    if L.Head = null then
      raise ListEmpty;
    else  {L.Head points to a node; remove it}
      Temp ← L.Head; L.Head ← L.Head.all.Next;  {jump around first node}
      Dispose(X ⇒ Temp); L.Size ← L.Size − 1;
    end if;
  end RemoveFront;
```

38.

```
⟨Procedures and Tasks in generic_list 33⟩ +≡
  procedure MakeEmpty(L : IN OUT List) is
    ptr : ListPtr;
  begin
    While L.Head ≠ null loop
      RemoveFront(L);
    end loop;
    L.Size ← 0; L.Tail ← null;
  end MakeEmpty;
```

39.
Pre: *Element* is defined; *L* may be empty.
Post: *Element* is appended to the end of *L*.

⟨ Procedures and Tasks in *generic_list* 33 ⟩ +≡

```
procedure AddToEnd(L : IN OUT List; Element : IN ElementType) is
   ptr : ListPtr;
begin
   if debug then
      put("AddToEnd>␣Adding␣to␣end␣of␣list:␣"); DisplayElement(Element);
   end if;
   if L.Head = null then
      L.Tail ← new ListNode'(Element, null); L.Head ← L.Tail;
   else
      ptr ← new ListNode'(Element, null); L.Tail.all.next ← ptr; L.Tail ← ptr;
   end if;
   L.Size ← L.Size + 1;
end AddToEnd;
```

40.
Pre: Source may be empty.
Post: Returns a complete copy of Source in Target.

⟨ Procedures and Tasks in *generic_list* 33 ⟩ +≡

```
procedure Copy(Source : in List; Target : out List) is
   ptr : listptr;
begin
   ptr ← Source.head; MakeEmpty(Target);
   while ptr ≠ null loop
      AddToEnd(Target, ptr.all.element); ptr ← ptr.all.next;
   end loop;
end Copy;
```

41.

Pre: *L* may be empty.
Post: displays the contents of *L*'s *Element* fields, in the order in which they appear in *L*.

```
⟨ Procedures and Tasks in generic_list 33 ⟩ +≡
  procedure Display(L : IN List) is
    ptr : ListPtr;
  begin
    if debug then
      put_Line("Display>");
    end if;
    ptr ← L.Head;
    while ptr ≠ null loop
      DisplayElement(ptr.all.Element); ptr ← ptr.all.Next;
    end loop;
  end Display;
```

42.

```
⟨ Procedures and Tasks in generic_list 33 ⟩ +≡
  procedure InsertInOrder(L : in out List; Element : ElementType) is
    Current : ListPtr;
    Previous : ListPtr;
    Temp : ListPtr;
  begin
    if debug then
      put("InsertInOrder>"); put_Line("Your␣input␣list␣is:␣"); display(L);
      put("InsertInOrder>"); put_Line("Your␣input␣element␣is:␣");
      displayelement(Element);
    end if;
    if L.Head = null then
      AddToFront(L, Element);
    elsif Element < L.Head.all.Element then
      AddToFront(L, Element);
    elsif (L.Tail.all.Element < Element) ∨ (L.Size = 1) then
      AddToEnd(L, Element);
    else
      if L.size = 1 then
        put_line("InsertInOrder>␣Should␣not␣be␣here!"); raise ListEmpty;
      end if;
      Temp ← new ListNode'(Element, null); Previous ← L.Head;
      Current ← Previous.all.next;
      while Current.all.element < Element loop
        Previous ← Current; Current ← Current.all.next;
      end loop;
      Temp.all.next ← Current; Previous.all.next ← Temp; L.Size ← L.Size + 1;
    end if;
    if debug then
      put("InsertInOrder>"); put_Line("Your␣input␣list␣is␣now:␣"); display(L);
    end if;
  end InsertInOrder;
```

43. Must be used with *ListSize* and *Rewind* or you will never know when you are at the end of the list.

```
⟨ Procedures and Tasks in generic_list 33 ⟩ +≡
  procedure GetNth(L : in out List; N : in natural; Element : out ElementType) is
  begin
    if debug then
      put("GetNth>␣Getting␣"); nat_io.put(N,1); put("´th␣record␣=>");
    end if;
    if L.Head = null then
      raise ListEmpty;
    elsif N > L.Size then
      raise ListEmpty;
    elsif N = 1 then
      L.Current ← L.Head; L.Previous ← L.Tail;
    else
      Rewind(L);
      for i ∈ 2 .. N loop
        L.Previous ← L.Current; L.Current ← L.Current.all.next;
      end loop;
    end if;
    Element ← L.Current.all.element;
    if debug then
      DisplayElement(Element);
    end if;
  end GetNth;
```

44. Must be used with *ListSize* and *Rewind* or you will never know when you are at the end of the list.

```
⟨ Procedures and Tasks in generic_list 33 ⟩ +≡
  procedure GetCurrent(L : in List; Element : out ElementType) is
  begin
    if L.Head = null then
      raise ListEmpty;
    end if;
    Element ← L.Current.all.element;
  end GetCurrent;
```

45. Must be used with *ListSize* and *Rewind* or you will never know when you are at the end of the list.

⟨ Procedures and Tasks in *generic_list* 33 ⟩ +≡

```
procedure UpdateCurrent(L : in List; Element : in ElementType) is
begin
  if L.Head = null then
    raise ListEmpty;
  end if;
  L.Current.all.element ← Element;
end UpdateCurrent;
```

46.

⟨ Procedures and Tasks in *generic_list* 33 ⟩ +≡

```
procedure DeleteCurrent(L : in out List) is
  Temp : ListPtr;
begin
  if L.Head = null then
    raise ListEmpty;
  elsif L.Size = 1 then
    Temp ← L.Current; L.Current ← null; L.Previous ← null; L.Head ← null;
    L.Tail ← null;
  else
    Temp ← L.Current;
    if L.Current = L.Tail then
      L.Previous.all.next ← L.Current.all.next; L.Current ← L.Head;
      L.Tail ← L.Previous;
    elsif L.Current = L.Head then
      L.Current ← L.Current.all.Next;   { jump around current node }
      L.Head ← L.Current;
    else
      L.Previous.all.next ← L.Current.all.next; L.Current ← L.Current.all.Next;
          { jump around current node }
    end if;
  end if;
  if debug then
    put("DeleteCurrent>Deleting=>"); DisplayElement(Temp.all.Element);
  end if;
  Dispose(X ⇒ Temp); L.Size ← L.Size − 1;
end DeleteCurrent;
```

47.

⟨ Procedures and Tasks in *generic_list* 33 ⟩ +≡

```
procedure DeleteMatching(L : in out List; Element : in ElementType; success : out
        boolean ) is
  kntr : natural;
  Current : ElementType;
begin
  success ← false;
  if L.Head = null then
    raise ListEmpty;
  end if;
  Rewind(L); kntr ← L.Size;
  for i ∈ 1 .. kntr loop
    GetNext(L, Current);
    if Current = Element then
      DeleteCurrent(L); success ← true; exit;
    end if;
  end loop;
end DeleteMatching;
```

48.

⟨ Procedures and Tasks in *generic_list* 33 ⟩ +≡

```
procedure Rewind(L : in out List) is
begin
  L.Current ← L.Head; L.Previous ← L.Tail;
end Rewind;
```

49. Must be used with *ListSize* and *Rewind* or you will never know when you are at the end of the list.

⟨ Procedures and Tasks in *generic_list* 33 ⟩ +≡

```
procedure GetNext(L : in out List; Element : out ElementType) is
begin
  if L.Head = null then
    raise ListEmpty;
  elsif L.Current = L.Tail then
    L.Current ← L.Head; L.Previous ← L.Tail;
  else
    L.Previous ← L.Current; L.Current ← L.Current.all.next;
  end if;
  Element ← L.Current.all.element;
end GetNext;
```

50. System-dependent changes. This module should be replaced, if necessary, by changes to the program that are necessary to make `MAIN` work at a particular installation. It is usually best to design your change file so that all changes to previous modules preserve the module numbering; then everybody's version will be consistent with the printed program. More extensive changes, which introduce new modules, can be inserted here; then only the index itself will get a new module number.

51. RCS Keywords.

$RCSfile: list.aweb,v
$Revision: 1.4
$Date: 1997/08/06 16:54:30
$Author: evansjr
$Id: list.aweb,v 1.4 1997/08/06 16:54:30 evansjr Exp evansjr
$Locker: evansjr
$State: Exp

52. Index. Here is a cross-reference table for the `MAIN` program. All modules in which an identifier is used are listed with that identifier, except that reserved words are indexed only when they appear in format definitions, and the appearances of identifiers in module names are not indexed. Underlined entries of subprograms and packages correspond to sections where this entity is specified, whereas entries in italic type correspond to the section where the entity's body is stated. For any other identifier underlined entries correspond to where the identifier was declared. Error messages and a few other things like "ASCII code" are indexed here too.

calyr

[Ada '95—Version 1.0]

This page intentionally left blank

1. Introduction. This package computes federal holidays, and off-fridays for NRaD. (We work five days one week, four the next—nine hours a day, except for Fridays.) The input is just the year. If you do not work the 5/4 weeks then there is a switch (`-nps true`) that you can use to turn it off.

2. This is based on a C program *calyr*, written by Bob Hall of Nrad in the eighties. Bob was a brilliant, and prolific programmer at NRaD who retired in the early nineties. One of his programs *msgs*, formed the basis of *Eudora*, a popular mail tool for PC's and Macintoshes, and now owned by Qualcomm.

3. This program was written to work for dates after 1970. It should work till the year 2099. (A year 3000 problem!) To test it out compile the driver program and run it with the following command line:

```
main [-year <year>] [-nps <boolean>]
```

For example:

```
main -year 1993 -nps false
```

4. This code is written using Donald Knuth's **WEB** paradigm for literate programming. To compile and link the code in its present format you will need the Ada version of the **WEB** tool.

It is available on-line via the world-wide-web at URL:

http://white.nosc.mil/~evansjr/literate/

. p

5. **WEB** is a literate programming paradigm for C, Pascal or Ada, and other languages. This style of programming is called "Literate Programming." For Further information see the paper *Literate Programming*, by Donald Knuth in *The Computer Journal*, Vol 27, No. 2, 1984; or the book *Weaving a Program: Literate Programming in* **WEB** by Wayne Sewell, Van Nostrand Reinhold, 1989. Another good source of information is the Usenet group *comp.programming.literate*. It has information on new tools and Frequently Asked Questions (FAQs).

6. The program consists of several packages that are declared right now; each of these packages and either the specification and the body of the packages are sent to a separate file. The main program itself is declared later. (Since the original AWEB package was written for Ada '83, it does not properly format new Ada '95 keywords **protected** and **private** . We remedy using the web format commands below.

format *protected* ≡ *procedure*
format *private* ≡ *procedure*

7. As a way of explanation, each "Module" withing angle brackets (< >) is expanded somewhere further down in the document. Consider it a high-level PDL (Program Descriptor Language). The trailing number you see within the brackets is where you can find this expansion. It is top-down in appearance, and in actual fact.

8. All the modules follow the same, top-down format. I will group all the boiler-plate into one module, for the compiler, but you will see it with the packages, as they are described.

⟨ Package boiler-plate 9 ⟩

9. Calyr Specification.

⟨ Package boiler-plate 9 ⟩ ≡

```
output to file calyr.ads
with Ustrings;
use Ustrings;
with TEXT_IO;
use TEXT_IO;
with Ada.Command_Line;
use Ada.Command_Line;
with Ada.Calendar;
use Ada.Calendar;
package calyr is
   ⟨ Specification of types and variables visible from calyr 11 ⟩
   ⟨ Specification of procedures visible from calyr 16 ⟩
end calyr;
output to file calyr.adb
⟨ Packages needed by calyr body 10 ⟩
package body calyr is
   ⟨ Types local to calyr 57 ⟩
   ⟨ Variables local to calyr 33 ⟩
   ⟨ Local Procedures 59 ⟩
   ⟨ Procedures and Tasks in calyr 39 ⟩
end calyr;
```

This code is used in section 8.

10.

⟨ Packages needed by calyr body 10 ⟩ ≡

```
with text_io;
use text_io;
```

See also section 32.

This code is used in section 9.

11.

⟨ Specification of types and variables visible from calyr 11 ⟩ ≡

```
subtype Hour_Number is integer range 0 .. 23;
subtype Minute_Number is integer range 0 .. 59;
subtype Second_Number is integer range 0 .. 59;
```

See also sections 12, 13, 14, and 15.

This code is used in section 9.

12.

⟨ Specification of types and variables visible from *calyr* 11 ⟩ +≡
BadYear : *Exception*;
BadDay : *Exception*;
type *fourarray* **is array** (0 .. 3) **of** *integer*;
type *threearray* **is array** (0 .. 2) **of** *integer*;

13.

⟨ Specification of types and variables visible from *calyr* 11 ⟩ +≡
type *month* **is** (*Jan*, *Feb*, *Mar*, *Apr*, *May*, *Jun*, *Jul*, *Aug*, *Sep*, *Oct*, *Nov*, *Dec*);
type *DayOfWeek* **is** (*Sun*, *Mon*, *Tue*, *Wed*, *Thu*, *Fri*, *Sat*);

14.

⟨ Specification of types and variables visible from *calyr* 11 ⟩ +≡
subtype *WeekDay* **is** *DayofWeek* **range** *Mon* .. *Fri*;

15.

⟨ Specification of types and variables visible from *calyr* 11 ⟩ +≡
Type WorkHours **is array** (*WeekDay*) **of** *Duration*;

16.

⟨ Specification of procedures visible from *calyr* 16 ⟩ ≡
procedure *print_holidays*(*yr* : **in** *Year_Number*; *do_nps* : **in** *boolean*);

See also sections 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, and 30.

This code is used in section 9.

17. Given a date (in Ada time format), *hol_dy* returns any special info about it.

status return values:	0	not a special day
	1	a non-work holiday
	2	observation of a non-work holiday
	3	other special day (not non-work)
	4	an off-Friday or Thursday
di return values:	di[0]	weekday of holiday (0 to 6)
	di[1]	day identification index
	di[2]	2: off-Friday; 1 : off-Thursday
		0: not an offday

⟨ Specification of procedures visible from *calyr* 16 ⟩ +≡
procedure *hol_dy*(*yrdate* : *time*; *di* : **out** *threearray*; *status* : **out** *integer*);

18. This function was taken from the book *Numerical Recipes.* It actually works on any year, not the artificial limit imposed by Ada type *Year_Number* (1901 .. 2099).

⟨ Specification of procedures visible from *calyr* 16 ⟩ +≡
  function *julian_day*(*Month* : *Month_Number*; *Day* : *Day_Number*; *Year* : *Integer*)**return** *long_integer*;

19. This procedure calculates the Month, Day, and Year when given the *Julian_day*.

⟨ Specification of procedures visible from *calyr* 16 ⟩ +≡
  procedure *caldate*(*Julian* : *Long_integer*; *Month* : **out** *Month_Number*; *Day* : **out** *Day_Number*; *Year* : **out** *Integer*);

20. Computes whether this is an off-day or a work-day.

⟨ Specification of procedures visible from *calyr* 16 ⟩ +≡
  function *IsWorkDay*(*YrDate* : *Time*; *NRaD* : *boolean* ← *false*; *debugit* : *boolean* ← *false*)**return** *boolean*;

21. Function aid computing new dates based on work hours.

⟨ Specification of procedures visible from *calyr* 16 ⟩ +≡
  function *DurationToCalendarTime*(*StartDate* : *Time*; *dailyhours* : *WorkHours*; *hrs* : *Duration*; *NRaD* : *boolean*)**return** *Time*;

22. Inverse of above.

⟨ Specification of procedures visible from *calyr* 16 ⟩ +≡
  function *CalendarTimeToDuration*(*StartDate* : *Time*; *dailyhours* : *WorkHours*; *EndDate* : *Time*; *NRaD* : *boolean*)**return** *Duration*;

23.

⟨ Specification of procedures visible from *calyr* 16 ⟩ +≡
  function *SameDay*(*Time1* , *Time2* : *Time*)**return** *boolean*;

24.

⟨ Specification of procedures visible from *calyr* 16 ⟩ +≡
  function *GetDayOfWeek*(*Today* : *Time*)**return** *DayOfWeek*;

25. Straightforward transformation.

⟨ Specification of procedures visible from *calyr* 16 ⟩ +≡
  function *ConvertHoursToDuration*(*hrs* : *natural*)**return** *Duration*;

26. Inverse of previous.

⟨ Specification of procedures visible from *calyr* 16 ⟩ +≡
  function *ConvertDurationToHours*(*dur* : *Duration*)**return** *natural*;

27.

⟨ Specification of procedures visible from *calyr* 16 ⟩ +≡
Procedure Split(*Seconds* : *Day_Duration*; *Hour* : **out** *Hour_Number*; *Minute* : **out** *Minute_Number*; *Second* : **out** *Second_Number*);

28.

⟨ Specification of procedures visible from *calyr* 16 ⟩ +≡
procedure *print_date*(*date* : *time*);
procedure *print_date*(*outfile* : *file_type*; *date* : *time*);

29.

⟨ Specification of procedures visible from *calyr* 16 ⟩ +≡
function *get_date*(*str* : **in** *Ustring*)**return** *Time*;
function *get_date*(*infile* : *file_type*)**return** *Time*;

30. Adds one day to the input parameter.

⟨ Specification of procedures visible from *calyr* 16 ⟩ +≡
function *IncrementDay*(*YrDate* : *Time*)**return** *Time*;

31. Calyr Body.

32.

⟨ Packages needed by *calyr* body 10 ⟩ +≡
  with *Ada.Strings.Unbounded*; *Use* *Ada.Strings.Unbounded*; **with** *Ustrings*;
  use *Ustrings*;

33. Number days in the month.

⟨ Variables local to *calyr* 33 ⟩ ≡
  ndm : **array** (*Month_Number*) **of** *natural* ← (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);

See also sections 34, 35, 36, 37, 38, 46, 48, 49, 50, 58, and 65.

This code is used in section 9.

34. Last day/previous month.

⟨ Variables local to *calyr* 33 ⟩ +≡
  ldpm : **array** (*Month_Number*) **of** *natural* ← (0, 31, 59, 90, 120, 151, 181, 212, 243, 273,
    304, 334);

35. List of holidays.

⟨ Variables local to *calyr* 33 ⟩ +≡
  NumHolidays : **constant** *natural* ← 20;
  holidays : **constant array** (1 .. *NumHolidays*) **of** *Ustring* ← (*U*("New␣Year´s␣Day"),
    U("ML␣King␣Day"), *U*("Presidents´␣Day"), *U*("Memorial␣Day"),
    U("Independence␣Day"), *U*("Labor␣Day"), *U*("Columbus␣Day"),
    U("Veterans´␣Day"), *U*("Thanksgiving␣Day"), *U*("Christmas␣Day"),
    U("Valentine´s␣Day"), *U*("St␣Patrick´s␣Day"), *U*("Good␣Friday"),
    U("Easter"), *U*("Mothers´␣Day"), *U*("Armed␣Forces␣Day"), *U*("Flag␣Day"),
    U("Fathers´␣Day"), *U*("Halloween"), *U*("Election␣Day"));

36. Index of Holidays.

⟨Variables local to *calyr* 33⟩ +≡
JNYD : **constant** *integer* ← 1;
JMLK : **constant** *integer* ← 2;
JPRS : **constant** *integer* ← 3;
JMEM : **constant** *integer* ← 4;
JIND : **constant** *integer* ← 5;
JLAB : **constant** *integer* ← 6;
JCOL : **constant** *integer* ← 7;
JVET : **constant** *integer* ← 8;
JTHX : **constant** *integer* ← 9;
JCHR : **constant** *integer* ← 10;
JVAL : **constant** *integer* ← 11;
JSPT : **constant** *integer* ← 12;
JGFR : **constant** *integer* ← 13;
JEST : **constant** *integer* ← 14;
JMOT : **constant** *integer* ← 15;
JAFD : **constant** *integer* ← 16;
JFLG : **constant** *integer* ← 17;
JFAT : **constant** *integer* ← 18;
JHAL : **constant** *integer* ← 19;
JELC : **constant** *integer* ← 20;

37. Index of something.

⟨Variables local to *calyr* 33⟩ +≡
INYD : **constant** *integer* ← 0; {index for NEW YEAR'S DAY, etc.}
IMLK : **constant** *integer* ← 1;
IPRS : **constant** *integer* ← 1;
IGFR : **constant** *integer* ← 1;
IEST : **constant** *integer* ← 2;
IMEM : **constant** *integer* ← 2;
ICOL : **constant** *integer* ← 0;
IVET : **constant** *integer* ← 1;
IHAL : **constant** *integer* ← 2;
IELC : **constant** *integer* ← 0;

38.

⟨ Variables local to *calyr* 33 ⟩ +≡

```
debug : boolean ← false;
debug2 : boolean ← false;
verbose : boolean ← true;
nps : boolean ← false;
already_leaped : boolean ← false;
package int_io is new integer_io(integer);
use int_io;
```

39.

⟨ Procedures and Tasks in *calyr* 39 ⟩ ≡

```
procedure hol_dy(yrdate : time; di : out threearray; status : out integer) is
   ⟨ Types and Variables local to hol_dy 41 ⟩
begin
   ⟨ Parse date 40 ⟩
   ⟨ Check if leap year 42 ⟩
   ⟨ Set year 43 ⟩
   ⟨ Set month 63 ⟩
   ⟨ Loop over holidays and Off-Fridays 67 ⟩
end hol_dy;
```

See also sections 81, 88, 93, 101, 109, 118, 129, 131, 132, 133, 134, 136, 137, 139, 141, and 143.

This code is used in section 9.

40.

⟨ Parse date 40 ⟩ ≡

```
Split(yrdate, Year, Month, Day, Seconds); hmn ← Calyr.month'val(Month − 1);
status ← 0; di(0) ← 0; di(1) ← 0; di(2) ← 0;
```

This code is used in section 39.

41.

⟨ Types and Variables local to *hol_dy* 41 ⟩ ≡

```
Year : Year_Number;
Month : Month_Number;
Day : Day_Number;
Seconds : Day_Duration;
hmn : Calyr.Month;
```

See also sections 45, 64, 68, and 71.

This code is used in section 39.

42. Simple-minded check. Must later look up what to do at end of century.

⟨ Check if leap year 42 ⟩ ≡
  if ((*Year* **mod** 4) = 0) ∧ (¬*already_leaped*) **then**
    already_leaped ← *true*; *ndm*(*calyr.month'pos*(*Feb*) + 1) ← 29;
    for *j* ∈ 3 .. 12 **loop**
      ldpm(*j*) ← *ldpm*(*j*) + 1;
    end loop;
  end if;

This code is used in section 39.

43. The datatype *hol* must be modified based on the year. The following code does just that.

⟨ Set year 43 ⟩ ≡
  ⟨ Calculate weekday of Jan 1. 44 ⟩
  ⟨ Calculate beginning date of 1st pay period in year 47 ⟩
  ⟨ Update ML King Day 51 ⟩
  ⟨ Update President's Day 52 ⟩
  ⟨ Update Memorial Day 53 ⟩
  ⟨ Update Columbus Day 54 ⟩
  ⟨ Update Veteran's Day 55 ⟩
  ⟨ Compute Easter 56 ⟩

This code is used in section 39.

44.

⟨ Calculate weekday of Jan 1. 44 ⟩ ≡
  jul := *julian_day*(1, 1, *Year*); *fdy* ← *DayOfWeek'val*((*jul* + 1) **mod** 7);
    jul := *julian_day*(*Month*, *Day*, *Year*); *di*(1) ← *integer*((*jul* + 1) **mod** 7);

This code is used in section 43.

45.

⟨ Types and Variables local to *hol_dy* 41 ⟩ +≡
  jul : *long_integer*;

46. Make global.

⟨ Variables local to *calyr* 33 ⟩ +≡
  fdy : *DayOfWeek*;

47. Funny C logic. Seems to work.

⟨ Calculate beginning date of 1st pay period in year 47 ⟩ ≡

```
tYear ← Year − 1970; tmp ← (Year − 1) rem 4;
if tmp = 0 then
   tmp ← 1;
else
   tmp ← 0;
end if;
bpp ← (11 − tYear − tmp − (tYear/4)) rem 14;
if bpp < 1 then
   bpp ← bpp + 14;
end if;
```

This code is used in section 43.

48. Make global.

⟨ Variables local to *calyr* 33 ⟩ +≡

```
bpp : integer;
tYear : integer;
tmp : integer;
```

49.

⟨ Variables local to *calyr* 33 ⟩ +≡

```
type hol_type is
   record
      dy : fourarray;   { Day of week or date of holiday }
      wn : fourarray;   { Week number (-1 -¿ skip }
      fl : fourarray;   { 1/0 -¿ non-work/work holiday }
      ix : fourarray;   { Index of holiday name }
   end record;
```

50. I know this is ugly, but it comes directly from C code.

⟨ Variables local to *calyr* 33 ⟩ +≡

```
hol : array (Month) of hol_type ← (((1, DayOfWeek'pos(MON), −1, 0), (0, 3, 0, 0), (1,
    1, 0, 0), (JNYD, JMLK, 0, 0)), ((14, DayOfWeek'pos(MON), −1, 0), (0, 3, 0, 0), (0, 1,
    0, 0), (JVAL, JPRS, 0, 0)), ((17, 0, 0, −1), (0, −1, −1, 0), (0, 0, 0, 0), (JSPT, JGFR,
    JEST, 0)), ((0, 0, 0, −1), (−1, −1, −1, 0), (0, 0, 0, 0), (0, JGFR, JEST, 0)),
    ((DayOfWeek'pos(SUN), DayOfWeek'pos(SAT), DayOfWeek'pos(MON), −1), (2,
    3, 5, 0), (0, 0, 1, 0), (JMOT, JAFD, JMEM, 0)), ((14, DayOfWeek'pos(SUN), −1, 0),
    (0, 3, 0, 0), (0, 0, 0, 0), (JFLG, JFAT, 0, 0)), ((4, −1, 0, 0), (0, 0, 0, 0), (1, 0, 0, 0), (JIND,
    0, 0, 0)), ((−1, 0, 0, 0), (0, 0, 0, 0), (0, 0, 0, 0), (0, 0, 0, 0)), ((DayOfWeek'pos(MON),
    −1, 0, 0), (1, 0, 0, 0), (1, 0, 0, 0), (JLAB, 0, 0, 0)), ((DayOfWeek'pos(MON),
    DayOfWeek'pos(MON), 31, −1), (2, −1, 0, 0), (1, 1, 0, 0), (JCOL, JVET, JHAL, 0)),
    ((DayOfWeek'pos(TUE), 11, DayOfWeek'pos(THU), −1), (−1, 0, 4, 0), (0, 1, 1, 0),
    (JELC, JVET, JTHX, 0)), ((25, −1, 0, 0), (0, 0, 0, 0), (1, 0, 0, 0), (JCHR, 0, 0, 0)));
```

51. ML King Day became federal holiday in 1986.

⟨ Update ML King Day 51 ⟩ ≡

```
if Year > 1985 then
   hol(JAN).wn(IMLK) ← 3;
else
   hol(JAN).wn(IMLK) ← −1;
end if;
```

This code is used in section 43.

52. President's day is third Monday (after 1971).

⟨ Update President's Day 52 ⟩ ≡

```
hol(Feb).dy(IPRS) ← DayOfWeek'pos(Mon); hol(Feb).wn(IPRS) ← 3;
if (Year < 1971) then
   hol(Feb).dy(IPRS) ← 22; hol(Feb).wn(IPRS) ← 0;
end if;
```

This code is used in section 43.

53. Memorial Day is last Monday in May.

⟨ Update Memorial Day 53 ⟩ ≡

```
hol(May).dy(IMEM) ← DayOfWeek'pos(Mon); hol(May).wn(IMEM) ← 5;
```

This code is used in section 43.

54. Columbus Day is second Monday in October. Did not exist before 1971, I guess?

⟨ Update Columbus Day 54 ⟩ ≡
  hol(*Oct*).*wn*(*ICOL*) ← 2;
  if *Year* < 1971 **then**
    hol(*Oct*).*wn*(*ICOL*) ← −1;
  end if;

This code is used in section 43.

55.

⟨ Update Veteran's Day 55 ⟩ ≡
  hol(*Oct*).*wn*(*IVET*) ← −1; *hol*(*Nov*).*wn*(*IVET*) ← 0;
  if *Year* < 1978 **then**
    hol(*Oct*).*wn*(*IVET*) ← 4; *hol*(*Nov*).*wn*(*IVET*) ← −1;
  end if;

This code is used in section 43.

56. Calls the function Easter. Also computes Good Friday.

⟨ Compute Easter 56 ⟩ ≡
  edt ← *easter*(*Year*); *hol*(*edt*.*mn*).*dy*(*IEST*) ← *edt*.*dt*; *hol*(*edt*.*mn*).*wn*(*IEST*) ← 0;
  edt.*dt* ← *edt*.*dt* − 2;
  if *edt*.*dt* < 1 **then**
    edt.*dt* ← *edt*.*dt* + *ndm*(3); *edt*.*mn* ← *Mar*;
  end if;
  hol(*edt*.*mn*).*dy*(*IGFR*) ← *edt*.*dt*; *hol*(*edt*.*mn*).*wn*(*IGFR*) ← 0;

This code is used in section 43.

57.

⟨ Types local to *calyr* 57 ⟩ ≡
  type *caldat* **is**
    record
      mn : *Month*;
      dt : *integer*;
    end record;

This code is used in section 9.

58.

⟨ Variables local to *calyr* 33 ⟩ +≡
  edt : *caldat*;

59. Here is the function easter that returns the day and month Easter occurs for a given year.

```
⟨ Local Procedures 59 ⟩ ≡
   function easter(Year : in Year_Number)return caldat is
      ⟨ Types and variables local to easter 60 ⟩
   begin
      fde ← ndm(1) + ndm(2); dt.dt ← pfm((Year − 1900) mod 19);
      if dt.dt < 0 then
         dt.mn ← Mar; dt.dt ← −dt.dt;
      else
         dt.mn ← Apr; fde ← fde + ndm(3);
      end if;
      ⟨ Compute weekday for Paschal Full Moon 62 ⟩
      return dt;
   end easter;
```

This code is used in section 9.

60. Here is the Paschal Full Moon table used to find Easter.

```
⟨ Types and variables local to easter 60 ⟩ ≡
   pfm : constant array (0 .. 18) of integer ← (14, 3, −23, 11, −31, 18, 8, −28, 16, 5, −25,
         13, 2, −22, 10, −30, 17, 7, −27);
```

See also section 61.

This code is used in section 59.

61.

```
⟨ Types and variables local to easter 60 ⟩ +≡
   fde : integer;
   dt : caldat;
```

62. Easter is the next Sunday following the Paschal Full Moon.

```
⟨ Compute weekday for Paschal Full Moon 62 ⟩ ≡
   fde ← (dt.dt + fde − (8 − DayOfWeek'pos(fdy))) rem 7;
   if fde < 0 then
      fde ← (7 + fde) rem 7;
   end if;
   dt.dt ← dt.dt + 7 − fde;
   if dt.dt > ndm(month'pos(dt.mn) + 1) then
      dt.dt ← dt.dt − ndm(month'pos(dt.mn) + 1); dt.mn ← month'succ(dt.mn);
   end if;
```

This code is used in section 59.

63. Used to determine off-fridays of month. Also, if November, figure out election day.

```
⟨ Set month 63 ⟩ ≡
  declare
    ldm, ofr, ii, jj : integer;
  begin
    ofr ← bpp − 2;
    if ofr ≤ 1 then
      ofr ← ofr + 14;
    end if;
    ldm ← ldpm(Month); ofr ← ofr + (ldm/14) * (14) − ldm;
    if ofr < 0 then
      ofr ← ofr + 14;
    elsif ((Month > 1) ∧ (ofr > 14)) then
      ofr ← ofr − 14;
    end if;
    ofrdy(0) ← UNST; ofrdy(1) ← UNST; ofrdy(2) ← UNST; ofrdy(3) ← UNST;
    if (Year > 1979) then
      jj ← 0;
      loop
        if ((Year ≠ 1982) ∨ (Month ≠ 4) ∨ (ofr ≠ 2)) ∧ ((Year ≠ 1980) ∨ (Month ≠ 1))
              then
          ofrdy(jj) ← ofr; jj ← jj + 1;
        end if;
        ofr ← ofr + 14; exit when ofr > ndm(Month);
      end loop;
    end if;
    fdm ← (ldm − (7 − DayOfWeek'pos(fdy))) rem 7;
    if fdm < 0 then
      fdm ← (7 + fdm) rem 7;
    end if;
    ⟨ Figure out election day 66 ⟩
  end;
```

This code is used in section 39.

64.

```
⟨ Types and Variables local to hol_dy 41 ⟩ +≡
  UNST : constant integer ← 64;
  jj : integer;
  ofrdy : fourarray;
```

65. Make global.

```
⟨ Variables local to calyr 33 ⟩ +≡
  fdm : integer;
```

66.

```
⟨ Figure out election day 66 ⟩ ≡
  if (hmn = Nov) ∧ ((Year rem 2) = 0) then
    ii ← hol(Nov).dy(IELC) − fdm + 1;
    if ii < 1 then
      ii ← ii + 7;
    end if;
    if ii < 2 then
      hol(Nov).wn(IELC) ← 2;
    else
      hol(Nov).wn(IELC) ← 1;
    end if;
  end if;
```

This code is used in section 63.

67. Main part of *hol_dy*.

```
⟨ Loop over holidays and Off-Fridays 67 ⟩ ≡
  ii ← 0; jj ← 0;
  loop
    ⟨ Check for no more holidays 69 ⟩
    if (hol(hmn).wn(ii) ≥ 0) then
      ho ← 0; ⟨ Holiday with fixed week day or fixed date 70 ⟩
      ⟨ Exhaust any earlier off-Fridays 72 ⟩
      ⟨ Check if off-Friday moved back to Thursday 73 ⟩
      ⟨ Work, and normal and Sunday non-work, holiday 74 ⟩
      ⟨ Monday/Friday extra day 75 ⟩
      ⟨ Saturday non-work holiday 76 ⟩
    end if;
    ii ← ii + 1;
  ⟨⟨ugly⟩⟩ exit when (ii > 3);
    exit when ((hol(hmn).dy(ii) < 0) ∧ (ofrdy(jj) = UNST));
  end loop;
  ⟨ December processing 77 ⟩
```

This code is used in section 39.

68.

```
⟨ Types and Variables local to hol_dy 41 ⟩ +≡
  ii, ho : integer;
```

69.

⟨ Check for no more holidays 69 ⟩ ≡

```
if hol(hmn).dy(ii) < 0 then
  if (integer(Month) < 12) ∨ (ofrdy(jj) < ndm(12)) then
    if ofrdy(jj) = Day then
      di(2) ← 2;
      if status = 0 then
        status ← 4;
      end if;
    end if;
    jj ← jj + 1; goto ugly;
  else
    exit;
  end if;
end if;
```

This code is used in section 67.

70.

⟨ Holiday with fixed week day or fixed date 70 ⟩ ≡

```
if hol(hmn).wn(ii) > 0 then
  dw ← hol(hmn).dy(ii); date ← dw − fdm + 1;
  if date < 1 then
    date ← date + 7;
  end if;
  date ← date + (7 * (hol(hmn).wn(ii) − 1));
  if date > ndm(Month) then   {Takes care of Memorial Day}
    date ← date − 7;
  end if;
else   {Holiday with fixed date}
  date ← hol(hmn).dy(ii); dw ← (date − (8 − fdm)) rem 7;
  if dw < 0 then
    dw ← (7 + dw) rem 7;
  end if;
  if hol(hmn).fl(ii) > 0 then   {Take care of weekend holidays}
    if dw = DayOfWeek'pos(Sun) then
      ho ← 1;
    elsif dw = DayOfWeek'pos(Sat) then
      ho ← −1;
    else
      ho ← 0;
    end if;
  end if;
end if;
```

This code is used in section 67.

71.

⟨ Types and Variables local to hol_dy 41 ⟩ +≡

```
date, dw : integer;
```

72.

⟨ Exhaust any earlier off-Fridays 72 ⟩ ≡

while $((hol(hmn).fl(ii) > 0) \wedge (((ho \geq 0) \wedge ofrdy(jj) < date) \vee ((ho < 0) \wedge (ofrdy(jj) < (date - 1))))) \vee ((hol(hmn).fl(ii) = 0) \wedge (ofrdy(jj) \leq date))$ **loop**
  if $ofrdy(jj) = Day$ **then**
    $di(2) \leftarrow 2$;
    if $status = 0$ **then**
      $status \leftarrow 4$;
    end if;
  end if;
  $jj \leftarrow jj + 1$;
end loop;

This code is used in section 67.

73.

⟨ Check if off-Friday moved back to Thursday 73 ⟩ ≡

if $(ofrdy(jj) > 1) \wedge (hol(hmn).fl(ii) > 0) \wedge ((ofrdy(jj) = date) \vee (ofrdy(jj) = (date + ho)))$ **then**
  if $(ofrdy(jj) - 1) = Day$ **then**
    $di(2) \leftarrow 1$;
    if $status = 0$ **then**
      $status \leftarrow 4$;
    end if;
  end if;
  $jj \leftarrow jj + 1$;
end if;

This code is used in section 67.

74.

⟨ Work, and normal and Sunday non-work, holiday 74 ⟩ ≡

if $(ho \geq 0) \wedge (date = Day)$ **then**
  $di(0) \leftarrow dw$; $di(1) \leftarrow hol(hmn).ix(ii)$; $status \leftarrow 1 + 2 * (di(1)/JVAL)$;
end if;

This code is used in section 67.

75.

⟨ Monday/Friday extra day 75 ⟩ ≡

if $(ho \neq 0) \wedge ((date + ho) > 0) \wedge ((date + ho) = Day)$ **then**
  $di(0) \leftarrow dw + ho$; $di(1) \leftarrow hol(hmn).ix(ii)$; $status \leftarrow 2$;
end if;

This code is used in section 67.

76.

⟨ Saturday non-work holiday 76 ⟩ ≡

```
if (ho < 0) ∧ (date = Day) then
   di(0) ← dw; di(1) ← hol(hmn).ix(ii); status ← 1;
end if;
```

This code is used in section 67.

77.

⟨ December processing 77 ⟩ ≡

```
if hmn = Dec then
   ⟨ Is first of next year a Friday or Saturday and this is an off-Friday? 78 ⟩
   ⟨ Weekday of December 31 79 ⟩
   ⟨ December 31 a Friday the observe Saturday, January 1st 80 ⟩ end if;
```

This code is used in section 67.

78.

⟨ Is first of next year a Friday or Saturday and this is an off-Friday? 78 ⟩ ≡

```
if jj ≠ 0 then
   if (ofrdy(jj) = ndm(12)) then
      tmp ← 1;
   else
      tmp ← 0;
   end if;
   if ((ofrdy(jj−1) = (ndm(12)−13))∨(ofrdy(jj) = ndm(12)))∧((ndm(12)−tmp) = Day)
         then
      di(2) ← 1;
      if status = 0 then
         status ← 4;
      end if;
   end if;
end if;
```

This code is used in section 77.

79.

⟨ Weekday of December 31 79 ⟩ ≡

```
dw ← (ndm(12) − (8 − fdm)) rem 7;
if dw < 0 then
   dw ← (7 + dw) rem 7;
end if;
```

This code is used in section 77.

80.

⟨ December 31 a Friday the observe Saturday, January 1st 80 ⟩ ≡
```
  if (dw = DayOfWeek'pos(Fri)) ∧ ndm(12) = Day then
    di(0) ← DayofWeek'pos(Fri); di(1) ← hol(Jan).ix(INYD); status ← 2;
  end if;
```
This code is used in section 77.

81.

⟨ Procedures and Tasks in *calyr* 39 ⟩ +≡
```
  procedure print_holidays(yr : in Year_Number; do_nps : in boolean) is
    ⟨ Variables local to print_holidays 84 ⟩
  begin
    nps ← do_nps; ⟨ Loop through months 82 ⟩
  end print_holidays;
```

82. Straightforward.

⟨ Loop through months 82 ⟩ ≡
```
  for mon ∈ Jan .. Dec loop
    if ¬verbose then
      put(month'image(mon)); put(">");
    end if;
    ⟨ Loop through days of month 83 ⟩
    if ¬verbose then
      put_line("␣");
    end if;
  end loop;
```
This code is used in section 81.

83.

⟨ Loop through days of month 83 ⟩ ≡

```
for ii ∈ 1 .. (ndm(month'pos(mon) + 1)) loop
  hol_dy(Time_of(yr, month'pos(mon) + 1, ii, 0.0), di, status);
  if ¬verbose then
    if (status > 0) then
      put("day␣=␣"); put(ii, 1); put(".␣␣status␣=␣"); put(status, 1);
      put("␣␣di␣=␣{");
      for i ∈ 0 .. 2 loop
        put(di(i), i);
        if i < 2 then
          put(",");
        end if;
      end loop;
      put_line("}␣");
    end if;
  else
    ⟨ Print out first day of month 85 ⟩
    ⟨ Print out holidays, as necessarily 87 ⟩
  end if;
end loop;
```

This code is used in section 82.

84.

⟨ Variables local to *print_holidays* 84 ⟩ ≡

```
status : integer;
di : threearray;
```

See also section 86.

This code is used in section 81.

85.

⟨ Print out first day of month 85 ⟩ ≡

```
if ii = 1 then
  put(month'image(mon)); put(ii, 3); put("␣"); hfdm ← DayOfWeek'val(fdm);
  put(DayOfWeek'image(hfdm)); put_line("");
end if;
```

This code is used in section 83.

86.

⟨ Variables local to *print_holidays* 84 ⟩ +≡

```
hfdm : DayOfWeek;
```

87.

```
⟨ Print out holidays, as necessarily 87 ⟩ ≡
  if status > 0 then
    if ¬nps then
      if (di(2) > 0) then
        hfdm ← DayOfWeek'val(di(2) + 3); put(DayOfWeek'image(hfdm)); put(ii, 3);
        put("␣"); put_line("Offday");
      end if;
    end if;
    if status ≤ 3 then
      hfdm ← DayOfWeek'val(di(0)); put(DayOfWeek'image(hfdm)); put(ii, 3);
      put("␣"); put(S(holidays(di(1))));
      if status ≠ 2 then
        put_line("");
      else
        put_line("␣(Observed)");
      end if;
    end if;
  end if;
```

This code is used in section 83.

88.

```
⟨ Procedures and Tasks in calyr 39 ⟩ +≡
  procedure caldate(Julian : Long_integer; Month : out Month_Number; Day : out
        Day_Number; Year : out Integer) is
    ⟨ Variables local to caldat 90 ⟩
  begin
    if (julian ≥ IGREG) then
      ⟨ Correct for to Gregorian Calendar 89 ⟩
    else
      ja ← julian;
    end if;
    ⟨ Now finish computation 91 ⟩
  end caldate;
```

89.

```
⟨ Correct for to Gregorian Calendar 89 ⟩ ≡
  jalpha ← long_integer(((float(julian − 1867216) − 0.25)/36524.25) − 0.5);
  ja ← julian + 1 + jalpha − long_integer(0.25 * float(jalpha) − 0.5);
```

This code is used in section 88.

90.

⟨ Variables local to *caldat* 90 ⟩ ≡

```
IGREG : constant long_integer ← (15 + 31 * (10 + 12 * 1582));
ja, jalpha : long_integer;
```

See also section 92.

This code is used in section 88.

91.

⟨ Now finish computation 91 ⟩ ≡

```
jb ← ja + 1524;
jc ← long_integer((6680.0 + (float(jb − 2439870) − 122.1)/365.25) − 0.5);
jd ← (365 * jc) + long_integer(0.25 * float(jc) − 0.5);
je ← long_integer(float(jb − jd)/30.6001 − 0.5);
Day ← Integer(jb − jd − long_integer(30.6001 * float(je) − 0.5));
TMonth ← Integer(je − 1);
if (TMonth > 12) then
   Month ← Tmonth − 12;
else
   Month ← Tmonth;
end if;
Year ← integer(jc − 4715);
if (Month > 2) then
   Year ← Year − 1;
end if;
if Year ≤ 0 then
   Year ← Year − 1;
end if;
```

This code is used in section 88.

92.

⟨ Variables local to *caldat* 90 ⟩ +≡

```
jb, jc, jd, je : long_integer;
Tmonth : integer;
```

93.

⟨ Procedures and Tasks in *calyr* 39 ⟩ +≡

```
function julian_day(Month : Month_Number; Day : Day_Number; Year : Integer)return
      long_integer is
   ⟨ Variables local to Julian_Day 94 ⟩
begin
   ⟨ Check for bad year 95 ⟩
   ⟨ Twiddle some variables before computing 96 ⟩
   ⟨ Compute julian number 98 ⟩
   ⟨ Test whether to change to Gregorian Calendar 99 ⟩
   return jul;
end julian_day;
```

94.

⟨ Variables local to *Julian_Day* 94 ⟩ ≡

```
jul : long_integer;
```

See also sections 97 and 100.

This code is used in section 93.

95. There is no year zero!

⟨ Check for bad year 95 ⟩ ≡

```
if (Year = 0) then
   raise BadYear;
end if;
```

This code is used in section 93.

96. I translated this from C. I don't pretend to understand it.

⟨ Twiddle some variables before computing 96 ⟩ ≡

```
if Year < 0 then
   TYear ← Year + 1;
else
   TYear ← Year;
end if;
if Month > 2 then
   jy ← TYear; jm ← Month + 1;
else
   jy ← TYear − 1; jm ← Month + 13;
end if;
```

This code is used in section 93.

97.

⟨ Variables local to *Julian_Day* 94 ⟩ +≡
  TYear, *jy*, *jm* : *integer*;

98. Probably taken from the book *Astronomical Formulae for Calculators.*

⟨ Compute julian number 98 ⟩ ≡
  jul ← *long_integer*(365.25 ∗ *float*(*jy*) − 0.5) + *long_integer*(30.6001 ∗ *float*(*jm*) − 0.5) +
    long_integer(*Day* + 1720995);

This code is used in section 93.

99. Gregorian Calendar was adopted on October 15, 1582.

⟨ Test whether to change to Gregorian Calendar 99 ⟩ ≡
  if *long_integer*(*integer*(*Day*) + 31 ∗ (*integer*(*Month*) + 12 ∗ *Year*)) ≥ *IGREG* **then**
    ja ← *integer*(0.01 ∗ *float*(*jy*) − 0.5);
    jul ← *jul* + *long_integer*(2 − *ja* + *integer*(0.25 ∗ *float*(*ja*) − 0.5));
  end if;

This code is used in section 93.

100.

⟨ Variables local to *Julian_Day* 94 ⟩ +≡
  IGREG : **constant** *long_integer* ← (15 + 31 ∗ (10 + 12 ∗ 1582));
  ja : *integer*;

101.

```
⟨ Procedures and Tasks in calyr 39 ⟩ +≡
  function IsWorkDay(YrDate : Time; NRaD : boolean ← false;
        debugit : boolean ← false)return boolean is
    ⟨ Variables local to IsWorKDay 103 ⟩
  begin
    status ← 1; workday ← false; hol_dy(Current_Time, di, status);
    if debugit then
      ⟨ Display hol_dy output 102 ⟩
    end if;
    dow ← GetDayOfWeek(YrDate);
    if status = 0 then
      ⟨ Make sure not a Saturday or Sunday 106 ⟩
    elsif nrad then
      ⟨ Look if NraD off-Friday (or off-Thursday if Friday a holiday) 107 ⟩
    else
      ⟨ See if federal holiday 108 ⟩
    end if;
    if debugit then
      ⟨ Print if workday 104 ⟩
    end if;
    return workday;
  end IsWorkDay;
```

102.

```
⟨ Display hol_dy output 102 ⟩ ≡
  Split(Yrdate, Year, Month, Day, Seconds); put("Status␣=␣"); put(status, 1);
  put("␣␣di␣=␣{");
  for i ∈ 0 .. 2 loop
    put(di(i), i);
    if i < 2 then
      put(",");
    end if;
  end loop;
  put_line("}␣");
```

This code is used in section 101.

103.

⟨ Variables local to *IsWorKDay* 103 ⟩ ≡

```
Year : Year_Number;
Month : Month_Number;
Day : Day_Number;
Seconds : Day_Duration;
dow : DayOfWeek;
```

See also section 105.

This code is used in section 101.

104.

⟨ Print if workday 104 ⟩ ≡

```
print_date(Yrdate);
if workday then
  put_line("␣is␣a␣workday.");
else
  put_line("␣is␣NOT␣a␣workday.");
end if;
```

This code is used in section 101.

105.

⟨ Variables local to *IsWorKDay* 103 ⟩ +≡

```
status : integer;
workday : boolean;
di : threearray;
Current_Time : Time ← YrDate;
```

106.

⟨ Make sure not a Saturday or Sunday 106 ⟩ ≡

```
if (dow ≠ Sun) ∧ (dow ≠ Sat) then
  workday ← true;
end if;
```

This code is used in sections 101, 107, and 108(2).

107. Make allowances for people (NRaD) working 5/4 weekly schedule.

⟨ Look if NraD off-Friday (or off-Thursday if Friday a holiday) 107 ⟩ ≡

```
if status = 3 then
  ⟨ Make sure not a Saturday or Sunday 106 ⟩
end if;
```

This code is used in section 101.

108. If *status* > 2 could be Arbor Day, or other work holiday.

⟨ See if federal holiday 108 ⟩ ≡
```
if status = 3 then
  ⟨ Make sure not a Saturday or Sunday 106 ⟩
end if;
if status = 4 then
  ⟨ Make sure not a Saturday or Sunday 106 ⟩
end if;
```
This code is used in section 101.

109.

⟨ Procedures and Tasks in *calyr* 39 ⟩ +≡
```
function DurationToCalendarTime(StartDate : Time; dailyhours : WorkHours;
      hrs : Duration; NRaD : boolean)return Time is
  ⟨ Variables local to DurationToCalendarTime 111 ⟩
begin
  ⟨ Find next work-day 110 ⟩
  ⟨ Remove slop 112 ⟩
  ⟨ Find next work-day 110 ⟩
  ⟨ If partial day, account for it 114 ⟩⟨ Find next work-day 110 ⟩
  ⟨ Find last work-day 116 ⟩
  ⟨ Figure out partial day 117 ⟩
  return Current_Time;
end DurationToCalendarTime;
```

110.

⟨ Find next work-day 110 ⟩ ≡
```
while (¬IsWorkDay(Current_Time, NRaD)) loop
  Current_Time ← IncrementDay(Current_Time);
end loop;
```
This code is used in sections 109(3) and 116.

111.

⟨ Variables local to *DurationToCalendarTime* 111 ⟩ ≡
```
Current_Time : Time ← StartDate;
```
See also sections 113 and 115.

This code is used in section 109.

112. If the start date was not a work day, and the the number of hours in Start Date is greater then zero, remove it. (Maybe this should be an error.)

⟨ Remove slop 112 ⟩ ≡
  Split(*Current_Time*, *Year*, *Month*, *Day*, *Seconds*);
  if *Current_Time* ≠ *StartDate* **then**
    Seconds ← 0.0; *Current_Time* ← *Time_of*(*Year*, *Month*, *Day*, *Seconds*);
  end if;

This code is used in section 109.

113.

⟨ Variables local to *DurationToCalendarTime* 111 ⟩ +≡
  Year : *Year_Number*;
  Month : *Month_Number*;
  Day : *Day_Number*;
  Seconds : *Day_Duration*;

114. If the StartDate has seconds ¿ zero then this means we are starting a new task in the middle of the day.

⟨ If partial day, account for it 114 ⟩ ≡
  yhrs ← *hrs*; *yrday* ← *GetDayOfWeek*(*Current_Time*);
  if (*dailyhours*(*yrday*) − *seconds*) > *yhrs* **then**
    Current_Time ← *Current_Time* + *yhrs*; *yhrs* ← 0.0;
  else
    Current_Time ← *Current_Time* − *Seconds*;
    Current_Time ← *IncrementDay*(*Current_Time*);
    yhrs ← *yhrs* − (*dailyhours*(*yrday*) − *seconds*);
  end if;

This code is used in section 109.

115.

⟨ Variables local to *DurationToCalendarTime* 111 ⟩ +≡
  yhrs : *Duration*;
  yrday : *DayOfWeek*;

116.

```
⟨Find last work-day 116⟩ ≡
  yrday ← GetDayOfWeek(Current_Time);
  while yhrs > dailyhours(yrday) loop
    yhrs ← yhrs − dailyhours(yrday); Current_Time ← IncrementDay(Current_Time);
    ⟨Find next work-day 110⟩
    yrday ← GetDayOfWeek(Current_Time);
    if (yrday = Sat) ∨ (yrday = Sun) then
      put("ERROR!␣ERROR!␣ERROR!"); new_line;
      put("For␣some␣reason␣failed␣to␣find␣next␣work-day␣for␣date␣=␣");
      print_date(Current_Time);
      if (¬IsWorkDay(Current_Time, NRaD, True)) then
        put("␣(NOT␣a␣work-day.)");
      else
        put("␣(IS␣␣a␣work-day.)");
      end if;
      new_line; raise BadDay;
    end if;
  end loop;
```

This code is used in section 109.

117.

```
⟨Figure out partial day 117⟩ ≡
  if yhrs > 0.0 then
    Current_Time ← Current_Time + yhrs; yhrs ← 0.0;
  end if;
```

This code is used in section 109.

118.

```
⟨Procedures and Tasks in calyr 39⟩ +≡
  function CalendarTimeToDuration(StartDate : Time; dailyhours : WorkHours;
        EndDate : Time; NRaD : boolean)return Duration is
    ⟨Variables local to CalendarTimeToDuration 121⟩
  begin
    ⟨Assert that input dates are correct 119⟩
    ⟨Count work hours over total span of days 122⟩
  end CalendarTimeToDuration;
```

119. The StartDate and EndDate must be valid work days and must have hours less then or equal to the total number of hours worked in a day. If this is not true, raise the *BadDay* exception.

⟨ Assert that input dates are correct 119 ⟩ ≡

```
if ¬IsWorkDay(StartDate, NRaD) ∨ ¬IsWorkDay(EndDate, NRaD) then
   raise BadDay;
end if;
Split(StartDate, StartYear, StartMonth, StartDay, StartSeconds);
dow ← GetDayOfWeek(StartDate);
if StartSeconds > dailyhours(dow) then
   raise BadDay;
end if;
Split(EndDate, EndYear, EndMonth, EndDay, EndSeconds);
dow ← GetDayOfWeek(EndDate);
if EndSeconds > dailyhours(dow) then
   raise BadDay;
end if;
```

See also section 120.

This code is used in section 118.

120. Also check that EndDate ¿ StartDate.

⟨ Assert that input dates are correct 119 ⟩ +≡

```
if StartDate > EndDate then
   raise BadDay;
end if;
```

121.

⟨ Variables local to *CalendarTimeToDuration* 121 ⟩ ≡

```
StartYear, EndYear : Year_Number;
StartMonth, EndMonth : Month_Number;
StartDay, EndDay : Day_Number;
StartSeconds, EndSeconds : Day_Duration;
dow : DayOfWeek;
```

See also sections 124 and 127.

This code is used in section 118.

122.

```
⟨ Count work hours over total span of days 122 ⟩ ≡
  if SameDay(StartDate, EndDate) then
    ⟨ Figure out duration for same day 123 ⟩
  else
    ⟨ Count work hours for first day 125 ⟩
    ⟨ Count work hours for intermediate days 126 ⟩
    ⟨ Count work hours for last day 128 ⟩
  end if;
  return hrs;
```

This code is used in section 118.

123. Easy. Just Subtract.

```
⟨ Figure out duration for same day 123 ⟩ ≡
  hrs ← EndDate − StartDate;
```

This code is used in section 122.

124.

```
⟨ Variables local to CalendarTimeToDuration 121 ⟩ +≡
  hrs : duration;
```

125.

```
⟨ Count work hours for first day 125 ⟩ ≡
  dow ← GetDayOfWeek(StartDate); hrs ← dailyhours(dow) − StartSeconds;
```

This code is used in section 122.

126.

```
⟨ Count work hours for intermediate days 126 ⟩ ≡
  Current_Time ← Time_Of(StartYear, StartMonth, StartDay, 0.0);
  Current_Time ← IncrementDay(Current_Time);
  while ¬SameDay(Current_Time, EndDate) loop
    if IsWorkDay(Current_Time, NraD) then
      dow ← GetDayOfWeek(Current_Time); hrs ← hrs + dailyhours(dow);
    end if;
    Current_Time ← IncrementDay(Current_Time);
  end loop;
```

This code is used in section 122.

127.

```
⟨ Variables local to CalendarTimeToDuration 121 ⟩ +≡
  Current_Time : Time;
```

128.

⟨ Count work hours for last day 128 ⟩ ≡

```
hrs ← hrs + EndSeconds;
```

This code is used in section 122.

129.

⟨ Procedures and Tasks in *calyr* 39 ⟩ +≡

```
function SameDay(Time1, Time2 : Time)return boolean is
   ⟨ Variables local to SameDay 130 ⟩
begin
   Split(Time1, Year1, Month1, Day1, Seconds);
   Split(Time2, Year2, MOnth2, Day2, Seconds);
   if (Year1 = Year2) ∧ (Month1 = Month2) ∧ (Day1 = Day2) then
      return true;
   else
      return false;
   end if;
end SameDay;
```

130.

⟨ Variables local to *SameDay* 130 ⟩ ≡

```
Year1, Year2 : Year_Number;
Month1, Month2 : Month_Number;
Day1, Day2 : Day_Number;
Seconds : Day_Duration;
```

This code is used in section 129.

131.

⟨ Procedures and Tasks in *calyr* 39 ⟩ +≡

```
function GetDayOfWeek(Today : Time)return DayOfWeek is
   jul : long_integer;
   Month : Month_Number;
   Day : Day_Number;
   Year : Year_Number;
   Seconds : Day_Duration;
   fdy : DayOfWeek;
begin
   Split(Today, Year, Month, Day, Seconds); jul := julian_day(Month, Day, Year);
        fdy ← DayOfWeek'val((jul + 1) mod 7); return fdy;
end GetDayOfWeek;
```

132. Essentially converts hours to seconds.

```
⟨ Procedures and Tasks in calyr 39 ⟩ +≡
  function ConvertHoursToDuration(hrs : natural)return Duration is
    dur : duration;
  begin
    dur ← duration(hrs) * 3600.0; return dur;
  end ConvertHoursToDuration;
```

133. Essentially converts seconds to hours.

```
⟨ Procedures and Tasks in calyr 39 ⟩ +≡
  function ConvertDurationToHours(dur : Duration)return natural is
    hrs : natural;
  begin
    hrs ← natural(float(dur)/3600.0); return hrs;
  end ConvertDurationToHours;
```

134.

```
⟨ Procedures and Tasks in calyr 39 ⟩ +≡
  Procedure Split(Seconds : Day_Duration; Hour : out Hour_Number; Minute : out
      Minute_Number;
      Second : out Second_Number) is yrsecs : Day_Duration ← Seconds;
begin
  Hour ← integer(yrsecs)/3600; yrsecs ← yrsecs − Duration(Hour * 3600);
  Minute ← integer(yrsecs)/60; yrsecs ← yrsecs − Duration(Minute * 60);
  Second ← integer(yrsecs);
end Split;
```

135. Prints out the date.

mm	Month number
dd	Day number in the month
HH	Hour number (24 hour system)
MM	Minute number
SS	Second number
cc	Century minus one
yy	Last 2 digits of the year number

The month, day, year, and century may be omitted; the current values are applied as defaults. For example:

date 10080045

sets the date to Oct 8, 12:45 a.m. The current year is the default because no year is supplied.

136. This was written because there seemed to be an error in adding 86,400.0 seconds to a day and then expecting the answer to come out right. Errors occured around April 7, 1997 and October 26, 1997. I believe it is a GNAT bug for version 3.09.

```
⟨ Procedures and Tasks in calyr 39 ⟩ +≡
  function IncrementDay(YrDate : Time)return Time is
    jul : long_integer;
    Year : Year_Number;
    Day : Day_Number;
    Month : Month_Number;
    Seconds : Day_Duration;
  begin
    Split(Yrdate, Year, Month, Day, Seconds); jul ← julian_day(Month, Day, Year);
    jul ← jul + 1; caldate(jul, Month, Day, Year);
    return Time_Of(Year, Month, Day, Seconds);
  end IncrementDay;
```

137.

```
⟨ Procedures and Tasks in calyr 39 ⟩ +≡
  procedure print_date(date : time) is
    ⟨ Variables local to print_date 138 ⟩
    do_alternate : boolean ← true;
  begin
    Split(date, Year, Month, Day, Seconds);
    if Month < 10 then
      put("0");
    end if;
    put(natural(Month), 1); put("/");
    if day < 10 then
      put("0");
    end if;
    put(natural(Day), 1); put("/"); put(natural(Year), 4);
    Split(Seconds, Hour, Minute, Second);
    if do_alternate then
      put("+");
      if Hour < 10 then
        put("0");
      end if;
      put(natural(Hour), 1);
    else
      put("␣");
      if Hour < 10 then
        put("0");
      end if;
      put(natural(Hour), 1); put(":");
      if Minute < 10 then
        put("0");
      end if;
      put(natural(Minute), 1); put(":");
      if Second < 10 then
        put("0");
      end if;
      put(natural(Second), 1);
    end if;
  end print_date;
```

138.

⟨ Variables local to *print_date* 138 ⟩ ≡
Year : *Year_Number*;
Month : *Month_Number*;
Day : *Day_Number*;
Seconds : *Day_Duration*;
Hour : *Hour_Number*;
Minute : *Minute_Number*;
Second : *Second_Number*;

This code is used in section 137.

139.

```
⟨ Procedures and Tasks in calyr 39 ⟩ +≡
  procedure print_date(outfile : file_type; date : time) is
    ⟨ Variables local to fprint_date 140 ⟩
    do_alternate : boolean ← true;
  begin
    Split(date, Year, Month, Day, Seconds);
    if Month < 10 then
      put(outfile, "0");
    end if;
    put(outfile, natural(Month), 1); put(outfile, "/");
    if day < 10 then
      put(outfile, "0");
    end if;
    put(outfile, natural(Day), 1); put(outfile, "/"); put(outfile, natural(Year), 4);
    Split(Seconds, Hour, Minute, Second);
    if do_alternate then
      put(outfile, "+");
      if Hour < 10 then
        put(outfile, "0");
      end if;
      put(outfile, natural(Hour), 1);
    else
      put(outfile, "␣");
      if Hour < 10 then
        put(outfile, "0");
      end if;
      put(outfile, natural(Hour), 1); put(outfile, ":");
      if Minute < 10 then
        put(outfile, "0");
      end if;
      put(outfile, natural(Minute), 1); put(outfile, ":");
      if Second < 10 then
        put(outfile, "0");
      end if;
      put(outfile, natural(Second), 1);
    end if;
  end print_date;
```

140.

```
⟨ Variables local to fprint_date 140 ⟩ ≡
  Year : Year_Number;
  Month : Month_Number;
  Day : Day_Number;
  Seconds : Day_Duration;
  Hour : Hour_Number;
  Minute : Minute_Number;
  Second : Second_Number;
```

This code is used in section 139.

141.

```
⟨ Procedures and Tasks in calyr 39 ⟩ +≡
  function get_date(infile : file_type)return Time is
    ⟨ Variables local to fget_date 142 ⟩
  begin
    get(infile, ndum); Month ← ndum;
    if debug2 then
      put("Month␣=␣"); put(Month, 1); put_line(".");
    end if;
    get_immediate(infile, chr); get(infile, ndum); Day ← ndum;
    if debug2 then
      put("Day␣=␣"); put(Day, 1); put_line(".");
    end if;
    get_immediate(infile, chr); get(infile, ndum);
    if ndum < 100 then
      if ndum < 50 then
        Year ← ndum + 2000;
      else
        Year ← ndum + 1900;
      end if;
    else
      Year ← ndum;
    end if;
    if debug2 then
      put("Year␣=␣"); put(Year, 1); put_line(".");
    end if;
    get_immediate(infile, chr); get(infile, ndum); Hour ← ndum;
    return Time_Of(Year, Month, Day, ConvertHoursToDuration(Hour));
  end get_date;
```

142.

⟨ Variables local to *fget_date* 142 ⟩ ≡
  ndum : *natural*;
  chr : *character*;
  Year : *Year_Number*;
  Month : *Month_Number*;
  Day : *Day_Number*;
  Hour : *natural*;

This code is used in section 141.

143.

```
⟨ Procedures and Tasks in calyr 39 ⟩ +≡
  function get_date(str : in Ustring)return Time is ⟨ Variables local to get_date 144 ⟩
    begin
      if debug2 then
        put("Parsing␣string␣`"); put(S(str)); put_line("´.");
      end if;
      tstr ← str; get(S(tstr), ndum, Last); Month ← ndum;
      if debug2 then
        put("Month␣=␣"); put(Month, 1); put_line(".");
      end if;
      ind ← index(tstr, "/"); tstr ← tail(tstr, length(tstr) − ind);
      get(S(tstr), ndum, Last); Day ← ndum;
      if debug2 then
        put("Day␣=␣"); put(Day, 1); put_line(".");
      end if;
      ind ← index(tstr, "/"); tstr ← tail(tstr, length(tstr) − ind);
      get(S(tstr), ndum, Last);
      if debug2 then
        put("Parsing␣string␣`"); put(S(tstr)); put_line("´."); put("ndum␣=␣");
        put(ndum, 1); put_line(".");
      end if;
      if ndum < 100 then
        if ndum < 50 then
          Year ← ndum + 2000;
        else
          Year ← ndum + 1900;
        end if;
      else
        Year ← ndum;
      end if;
      if debug2 then
        put("Year␣=␣"); put(Year, 1); put_line(".");
      end if;
      ind ← index(tstr, "+"); tstr ← tail(tstr, length(tstr) − ind);
      get(S(tstr), ndum, Last); Hour ← ndum;
      return Time_Of(Year, Month, Day, ConvertHoursToDuration(Hour));
    end get_date;
```

144.

⟨ Variables local to *get_date* 144 ⟩ ≡
  ndum : *natural*;
  Year : *Year_Number*;
  Month : *Month_Number*;
  Day : *Day_Number*;
  Hour : *natural*;
  Last : *positive*;
  tstr : *ustring*;
  ind : *natural*;

This code is used in section 143.

145. Test Driver. This is the main routine that starts everything.

146.

```
output to file main.adb
with Text_IO;
use Text_IO;
with Ada.Calendar;
use Ada.Calendar;
with calyr;
use calyr;
with ustrings;
use ustrings;
with getopt;
use getopt;
procedure main is
   ⟨ Variables local to main 150 ⟩
   package yr_io is new integer_io(Year_Number);
   use yr_io;
   package bool_io is new enumeration_io(boolean);
   use bool_io;
begin
   ⟨ Get options 147 ⟩
   print_holidays(yr, nps);
end main;
```

147.

```
⟨ Get options 147 ⟩ ≡
  ⟨ Get year 148 ⟩
  ⟨ Get nps 149 ⟩
```

This code is used in section 146.

148.

```
⟨ Get year 148 ⟩ ≡
  if option_present(U("-year")) then
    get_option(U("-year"), param);  get(S(param), yr, Last);
  else
    yr ← 1997;
  end if;
```

This code is used in section 147.

149.

⟨ Get nps 149 ⟩ ≡

```
if option_present(U("-nps")) then
   get_option(U("-nps"), param); get(S(param), nps, Last);
else
   nps ← false;
end if;
```

This code is used in section 147.

150.

⟨ Variables local to main 150 ⟩ ≡

```
yr : Year_number;
param : Ustring;
Last : positive;
nps : boolean;
```

This code is used in section 146.

151. System-dependent changes. This module should be replaced, if necessary, by changes to the program that are necessary to make `MAIN` work at a particular installation. It is usually best to design your change file so that all changes to previous modules preserve the module numbering; then everybody's version will be consistent with the printed program. More extensive changes, which introduce new modules, can be inserted here; then only the index itself will get a new module number.

152. RCS Keywords.

$RCSfile: calyr.aweb,v
$Revision: 1.1
$Date: 1997/08/18 22:43:35
$Author: evansjr
$Id: calyr.aweb,v 1.1 1997/08/18 22:43:35 evansjr Exp evansjr
$Locker: evansjr
$State: Exp

153. Index. Here is a cross-reference table for the MAIN program. All modules in which an identifier is used are listed with that identifier, except that reserved words are indexed only when they appear in format definitions, and the appearances of identifiers in module names are not indexed. Underlined entries of subprograms and packages correspond to sections where this entity is specified, whereas entries in italic type correspond to the section where the entity's body is stated. For any other identifier underlined entries correspond to where the identifier was declared. Error messages and a few other things like "ASCII code" are indexed here too.

⟨ Print out holidays, as necessarily 87 ⟩ Used in section 83.
⟨ Procedures and Tasks in *calyr* 39, 81, 88, 93, 101, 109, 118, 129, 131, 132, 133, 134, 136, 137, 139, 141, 143 ⟩ Used in section 9.
⟨ Remove slop 112 ⟩ Used in section 109.
⟨ Saturday non-work holiday 76 ⟩ Used in section 67.
⟨ See if federal holiday 108 ⟩ Used in section 101.
⟨ Set month 63 ⟩ Used in section 39.
⟨ Set year 43 ⟩ Used in section 39.
⟨ Specification of procedures visible from *calyr* 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30 ⟩ Used in section 9.
⟨ Specification of types and variables visible from *calyr* 11, 12, 13, 14, 15 ⟩ Used in section 9.
⟨ Test whether to change to Gregorian Calendar 99 ⟩ Used in section 93.
⟨ Twiddle some variables before computing 96 ⟩ Used in section 93.
⟨ Types and Variables local to *hol_dy* 41, 45, 64, 68, 71 ⟩ Used in section 39.
⟨ Types and variables local to *easter* 60, 61 ⟩ Used in section 59.
⟨ Types local to *calyr* 57 ⟩ Used in section 9.
⟨ Update Columbus Day 54 ⟩ Used in section 43.
⟨ Update ML King Day 51 ⟩ Used in section 43.
⟨ Update Memorial Day 53 ⟩ Used in section 43.
⟨ Update President's Day 52 ⟩ Used in section 43.
⟨ Update Veteran's Day 55 ⟩ Used in section 43.
⟨ Variables local to main 150 ⟩ Used in section 146.
⟨ Variables local to *CalendarTimeToDuration* 121, 124, 127 ⟩ Used in section 118.
⟨ Variables local to *DurationToCalendarTime* 111, 113, 115 ⟩ Used in section 109.
⟨ Variables local to *IsWorKDay* 103, 105 ⟩ Used in section 101.
⟨ Variables local to *Julian_Day* 94, 97, 100 ⟩ Used in section 93.
⟨ Variables local to *SameDay* 130 ⟩ Used in section 129.
⟨ Variables local to *caldat* 90, 92 ⟩ Used in section 88.
⟨ Variables local to *calyr* 33, 34, 35, 36, 37, 38, 46, 48, 49, 50, 58, 65 ⟩ Used in section 9.
⟨ Variables local to *fget_date* 142 ⟩ Used in section 141.
⟨ Variables local to *fprint_date* 140 ⟩ Used in section 139.
⟨ Variables local to *get_date* 144 ⟩ Used in section 143.
⟨ Variables local to *print_date* 138 ⟩ Used in section 137.
⟨ Variables local to *print_holidays* 84, 86 ⟩ Used in section 81.
⟨ Weekday of December 31 79 ⟩ Used in section 77.
⟨ Work, and normal and Sunday non-work, holiday 74 ⟩ Used in section 67.

Probability Functions

[Ada '95—Version 1.0]
September 4, 1997

This page intentionally left blank

1. Introduction. Here is the Ada code for routines used in calculating probaility distributions. This code uses Donald Knuth's **WEB** format for literate programming. To compile and link the code in its present format you will need the Ada version of the **WEB** tool.

It is available on-line via the world-wide-web at URL:

http://white.nosc.mil/~evansjr/literate/

.

2. **WEB** is a literate programming paradigm for C, Pascal or Ada, and other languages. This style of programming is called "Literate Programming." For Further information see the paper *Literate Programming*, by Donald Knuth in *The Computer Journal*, Vol 27, No. 2, 1984; or the book *Weaving a Program: Literate Programming in* **WEB** by Wayne Sewell, Van Nostrand Reinhold, 1989. Another good source of information is the Usenet group *comp.programming.literate*. It has information on new tools and Frequently Asked Questions (FAQs).

3. The program consists of several packages that are declared right now; each of these packages and either the specification and the body of the packages are sent to a separate file. The main program itself is declared later. (Since the original AWEB package was written for Ada '83, it does not properly format new Ada '95 keywords **protected** and **private** . We remedy using the web format commands below.

format *protected* ≡ *procedure*
format *private* ≡ *procedure*

4. As a way of explanation, each "Module" withing angle brackets (< >) is expanded somewhere further down in the document. The trailing number you see within the brackets is where you can find this expansion. You can treat the modules names as a PDL (Program Descriptor Language), a highly recommened way of writing and documenting code.

⟨ Package boiler-plate 5 ⟩

5. Probability Primitives.

```
⟨ Package boiler-plate 5 ⟩ ≡
  output to file probability.ads
  ⟨ Needed packages 6 ⟩
  package probability is
    ⟨ Specification of types and variables visible from probability 7 ⟩
    ⟨ Specification of procedures visible from probability 8 ⟩
  end probability;
  output to file probability.adb
  package body probability is
    ⟨ Variables local to probability 10 ⟩
    ⟨ Procedures and Tasks in probability 11 ⟩
  end probability;
```

This code is used in section 4.

6. Here is the specification for generics.

```
⟨ Needed packages 6 ⟩ ≡
  with Ada.Numerics.Float_Random;
```

See also section 12.

This code is used in section 5.

7.

```
⟨ Specification of types and variables visible from probability 7 ⟩ ≡
  type bool_array is array (integer range <>) of boolean;
```

This code is used in section 5.

8.

```
⟨ Specification of procedures visible from probability 8 ⟩ ≡
  function Uniform(Low, High : Float)return float;
  function Uniform(Low, High : Natural)return Natural;
  procedure sample(M, N : in natural; yrsample : out bool_array);
```

This code is used in section 5.

9. Probability functions Body.

10.

⟨ Variables local to *probability* 10 ⟩ ≡
debug : *boolean* ← *false*;
FirstTime : *boolean* ← *true*;

This code is used in section 5.

11.

⟨ Procedures and Tasks in *probability* 11 ⟩ ≡

```
function Uniform(Low, High : Float)return float is
   use Ada.Numerics.Float_Random;
   P1 : Uniformly_Distributed;
   G : Generator;
   answer : float;
   tmp : float;
begin
   Reset(G); P1 ← Random(G); tmp ← (High − Low); answer ← tmp * (P1) + Low;
   return answer;
end Uniform;
```

See also sections 13 and 14.

This code is used in section 5.

12.

⟨ Needed packages 6 ⟩ +≡

```
with Text_IO;
use Text_IO;
```

13.

```
⟨ Procedures and Tasks in probability 11 ⟩ +≡
  function Uniform(Low, High : natural)return natural is
    use Ada.Numerics.Float_Random;
    P1 : Uniformly_Distributed;
    G : Generator;
    tmp, tmp2 : float;
    answer : natural;
    package flt_io is new float_io(float);
    use flt_io;
  begin
    if Low = High then
      answer ← Low;
    else
      if FirstTime then
        Reset(G, 68069); FirstTime ← false;
      else
        Reset(G);
      end if;
      P1 ← Random(G); tmp ← float(High − Low + 1); tmp2 ← (tmp * P1) − 0.5;
      if (debug) then
        put("Random␣generated␣"); put(P1); put_line(".␣");
        put("(high-low+1)␣tmp␣=␣"); put(tmp); put_line(".␣");
        put("(tmp*p1)␣tmp2␣=␣"); put(tmp2); put_line(".␣");
      end if;
      answer ← natural(tmp2) + Low;
    end if;
    return answer;
  end Uniform;
```

14. Based on a routine from the September, 1987 *Communications of the ACM.*

⟨ Procedures and Tasks in *probability* 11 ⟩ +≡

```
procedure sample(M, N : in natural; yrsample : out bool_array) is
  t : natural;
  k : natural;
begin
  for j ∈ 1 .. N loop
    yrsample(j) ← false;
  end loop;
  k ← N − M + 1;
  for j ∈ k .. N loop
    t ← uniform(1, j);
    if yrsample(t) then
      yrsample(j) ← true;
    else
      yrsample(t) ← true;
    end if;
  end loop;
end sample;
```

15. System-dependent changes. This module should be replaced, if necessary, by changes to the program that are necessary to make `MAIN` work at a particular installation. It is usually best to design your change file so that all changes to previous modules preserve the module numbering; then everybody's version will be consistent with the printed program. More extensive changes, which introduce new modules, can be inserted here; then only the index itself will get a new module number.

16. RCS Keywords.

$RCSfile: probability.aweb,v
$Revision: 1.1
$Date: 1997/08/03 21:35:14
$Author: evansjr
$Id: probability.aweb,v 1.1 1997/08/03 21:35:14 evansjr Exp evansjr
$Locker: evansjr
$State: Exp

17. Index. Here is a cross-reference table for the MAIN program. All modules in which an identifier is used are listed with that identifier, except that reserved words are indexed only when they appear in format definitions, and the appearances of identifiers in module names are not indexed. Underlined entries of subprograms and packages correspond to sections where this entity is specified, whereas entries in italic type correspond to the section where the entity's body is stated. For any other identifier underlined entries correspond to where the identifier was declared. Error messages and a few other things like "ASCII code" are indexed here too.

⟨ Needed packages 6, 12 ⟩ Used in section 5.
⟨ Package boiler-plate 5 ⟩ Used in section 4.
⟨ Procedures and Tasks in *probability* 11, 13, 14 ⟩ Used in section 5.
⟨ Specification of procedures visible from *probability* 8 ⟩ Used in section 5.
⟨ Specification of types and variables visible from *probability* 7 ⟩ Used in section 5.
⟨ Variables local to *probability* 10 ⟩ Used in section 5.

getopt

[Ada '95—Version 1.0]

This page intentionally left blank

1. Introduction. This package provides some primitive command-line processing typical of Unix commands.

2. This code is written using Donald Knuth's **WEB** paradigm for literate programming. To compile and link the code in its present format you will need the Ada version of the **WEB** tool.

It is available on-line via the world-wide-web at URL:

http://white.nosc.mil/~evansjr/literate/

. p

3. **WEB** is a literate programming paradigm for C, Pascal or Ada, and other languages. This style of programming is called "Literate Programming." For Further information see the paper *Literate Programming*, by Donald Knuth in *The Computer Journal*, Vol 27, No. 2, 1984; or the book *Weaving a Program: Literate Programming in* **WEB** by Wayne Sewell, Van Nostrand Reinhold, 1989. Another good source of information is the Usenet group *comp.programming.literate*. It has information on new tools and Frequently Asked Questions (FAQs).

4. The program consists of several packages that are declared right now; each of these packages and either the specification and the body of the packages are sent to a separate file. The main program itself is declared later. (Since the original AWEB package was written for Ada '83, it does not properly format new Ada '95 keywords **protected** and **private** . We remedy using the web format commands below.

format *protected* ≡ *procedure*
format *private* ≡ *procedure*

5. As a way of explanation, each "Module" withing angle brackets (< >) is expanded somewhere further down in the document. Consider it a high-level PDL (Program Descriptor Language). The trailing number you see within the brackets is where you can find this expansion. It is top-down in appearance, and in actual fact.

6. All the modules follow the same, top-down format. I will group all the boiler-plate into one module, for the compiler, but you will see it with the packages, as they are described.

⟨ Package boiler-plate 7 ⟩

7. Getopt Specification.

```
⟨ Package boiler-plate 7 ⟩ ≡
  output to file getopt.ads
  with Ustrings;
  use Ustrings;
  with TEXT_IO;
  use TEXT_IO;
  with Ada.Command_Line;
  use Ada.Command_Line;
  package getopt is
    ⟨ Specification of types and variables visible from getopt 8 ⟩
    ⟨ Specification of procedures visible from getopt 9 ⟩
  end getopt;
  output to file getopt.adb
  ⟨ Packages needed by getopt body 11 ⟩
  package body getopt is
    ⟨ Variables local to getopt 12 ⟩
    ⟨ Procedures and Tasks in getopt 13 ⟩
  end getopt;
```

This code is used in section 6.

8.

⟨ Specification of types and variables visible from *getopt* 8 ⟩ ≡

This code is used in section 7.

9.

```
⟨ Specification of procedures visible from getopt 9 ⟩ ≡
  function option_present(option : in Ustring)return boolean;
  function name_present(Num : natural)return boolean;
  procedure get_option(option : in Ustring;param : out Ustring);
  procedure get_name(name : out Ustring;Num : in natural);
```

This code is used in section 7.

10. GetOpt Body.

11.

⟨ Packages needed by *getopt* body 11 ⟩ ≡

```
with Ada.Strings.Unbounded; Use Ada.Strings.Unbounded; with Ustrings;
use Ustrings;
```

This code is used in section 7.

12.

⟨ Variables local to *getopt* 12 ⟩ ≡

```
debug : boolean ← false;
```

This code is used in section 7.

13.

⟨ Procedures and Tasks in *getopt* 13 ⟩ ≡

```
package natio is new integer_io(natural);
```

See also sections 14, 15, 16, and 19.

This code is used in section 7.

14.

⟨ Procedures and Tasks in *getopt* 13 ⟩ +≡

```
function option_present(option : in Ustring)return boolean is
   knt : natural;
   ispresent : boolean;
begin
   knt ← Argument_Count; ispresent ← false;
   for i ∈ 1 .. knt loop
      if S(option) = Argument(i) then
         ispresent ← true; exit;
      end if;
   end loop;
   return ispresent;
end option_present;
```

15.

```
⟨ Procedures and Tasks in getopt 13 ⟩ +≡
  procedure get_option(option : in Ustring; param : out Ustring) is
    knt : natural;
  begin
    knt ← Argument_Count;
    for i ∈ 1 .. knt loop
      if S(option) = Argument(i) then
        param ← U(Argument(i + 1));
      end if;
    end loop;
  end get_option;
```

16.

```
⟨ Procedures and Tasks in getopt 13 ⟩ +≡
  function name_present(Num : natural)return boolean is
    knt, ic : natural;
    i : natural ← 1;
    fknt : natural ← 0;
    ispresent : boolean;
  begin
    ispresent ← false;
    if debug then
      put_line("name_present>");
    end if;
    knt ← Argument_Count;
    while (i ≤ knt) loop
      ⟨ If found option, skip it and its parameter 17 ⟩
      ⟨ if not option, must be name, return true if right number 18 ⟩
    end loop;
    if debug then
      put("Argument␣"); natio.put(Num, 1);
      if ispresent then
        put_line("␣is␣present.");
      else
        put_line("␣is␣NOT␣present.");
      end if;
    end if;
    return ispresent;
  end name_present;
```

17.

⟨ If found option, skip it and its parameter 17 ⟩ ≡

```
ic ← Index(U(Argument(i)), "-");
if ic > 0 then
  i ← i + 2;
end if;
if debug then
  put_line("Skipping␣first␣option.");
end if;
```

This code is used in sections 16 and 19.

18.

⟨ if not option, must be name, return *true* if right number 18 ⟩ ≡

```
if ic = 0 then
  fknt ← fknt + 1;
  if fknt = num then
    if debug then
      put_line("Found␣your␣input␣file␣name!");
    end if;
    ispresent ← true; exit;
  end if;
  i ← i + 1;
end if;
```

This code is used in section 16.

19.

⟨ Procedures and Tasks in *getopt* 13 ⟩ +≡

```
procedure get_name(name : out Ustring; Num : natural) is
  knt, ic : natural;
  i : natural ← 1;
  fknt : natural ← 0;
begin
  if debug then
    put_line("get_name>");
  end if;
  knt ← Argument_Count;
  while (i ≤ knt) loop
    ⟨ If found option, skip it and its parameter 17 ⟩
    ⟨ if not option, must be name, return if right number 20 ⟩
  end loop;
end get_name;
```

20.

⟨if not option, must be name, return if right number 20⟩ ≡

```
if ic = 0 then
   fknt ← fknt + 1;
   if fknt = num then
      if debug then
         put_line("Found␣your␣input␣file␣name!");
      end if;
      name ← U(Argument(i)); exit;
   end if;
   i ← i + 1;
end if;
```

This code is used in section 19.

21. System-dependent changes. This module should be replaced, if necessary, by changes to the program that are necessary to make `MAIN` work at a particular installation. It is usually best to design your change file so that all changes to previous modules preserve the module numbering; then everybody's version will be consistent with the printed program. More extensive changes, which introduce new modules, can be inserted here; then only the index itself will get a new module number.

22. RCS Keywords.

$RCSfile: getopt.aweb,v
$Revision: 1.1
$Date: 1997/09/05 00:28:36
$Author: evansjr
$Id: getopt.aweb,v 1.1 1997/09/05 00:28:36 evansjr Exp evansjr
$Locker: evansjr
$State: Exp

23. Index. Here is a cross-reference table for the `MAIN` program. All modules in which an identifier is used are listed with that identifier, except that reserved words are indexed only when they appear in format definitions, and the appearances of identifiers in module names are not indexed. Underlined entries of subprograms and packages correspond to sections where this entity is specified, whereas entries in italic type correspond to the section where the entity's body is stated. For any other identifier underlined entries correspond to where the identifier was declared. Error messages and a few other things like "ASCII code" are indexed here too.

⟨ If found option, skip it and its parameter 17 ⟩ Used in sections 16 and 19.
⟨ Package boiler-plate 7 ⟩ Used in section 6.
⟨ Packages needed by *getopt* body 11 ⟩ Used in section 7.
⟨ Procedures and Tasks in *getopt* 13, 14, 15, 16, 19 ⟩ Used in section 7.
⟨ Specification of procedures visible from *getopt* 9 ⟩ Used in section 7.
⟨ Specification of types and variables visible from *getopt* 8 ⟩ Used in section 7.
⟨ Variables local to *getopt* 12 ⟩ Used in section 7.
⟨ if not option, must be name, return if right number 20 ⟩ Used in section 19.
⟨ if not option, must be name, return *true* if right number 18 ⟩ Used in section 16.

Capabilities Package

[Ada '95—Version 1.0]
September 18, 1997

This page intentionally left blank

1. Introduction. Here is some code to test capabilities. It is written using Donald Knuth's WEB format for literate programming. To compile and link the code in its present format you will need the Ada version of the WEB tool.

It is available on-line via the world-wide-web at URL:

http://white.nosc.mil/~evansjr/literate/

.

2. WEB is a literate programming paradigm for C, Pascal or Ada, and other languages. This style of programming is called "Literate Programming." For Further information get the book *Literate Programming*, by Donald Knuth, published by the Center for the Study of Language and Information, Stanford University, 1992. Another good source of information is the Usenet group *comp.programming.literate*. It has information on tools and answers to Frequently Asked Questions (FAQs).

3. Who should use the WEB paradigm for programming? Well, not everybody. Here are a few paragraphs from Donald Knuth's book that explains it best.

4. **Retrospect and Prospects.** Enthusiastic reports about new computer languages, by the authors of those languages, are commonplace. Hence I'm well aware of the fact that my own experiences cannot be extrapolated too far. I also realize that, whenever I have encountered a problem with WEB, I've simply changed the system; other users of WEB cannot operate under the same ground rules.

5. However, I believe that I have stumbled on a way of programming that produces better programs that are more portable and more easily understood and maintained than ever before; furthermore, the system seems to work with large programs as well as with small ones. I'm pleased that my work on typography, which began as an application of computers to another field, has come full circle and become an application of typography to the heart of computer science; I like to think of WEB as a neat "spinoff" of my research on TEX. However, all of my experiences with this system have been highly colored by my own tastes, and only time will tell if a large number of other people will find WEB to be equally attractive and useful.

6. I made a conscious decision not to design a language that would be suitable for everybody. My goal was to provide a tool for system programmers, not for high school students or for hobbyists. I don't have anything against high school students and hobbyists, but I don't believe every computer language should attempt to offer all things to all people. A user of WEB needs to be good enough at computer science that he or she is comfortable dealing with several languates simultaneously. Since WEB combines TEX and Pascal with a few rules of its own, WEB programs can contain WEB syntax errors. TEX syntax errors, Pascal syntax errors, and algorithmic errors; in practice, all four types of errors occur, and a bit of sophistication is needed to sort out which is which. Computer specialists tend to be better at such things than other people. I have found that WEB programs can be debugged rapidly in spite of the profusion of languages, but I'm sure that many other intelligent people will find such a task difficult.

7. In other words, WEB seems to be specifically for the peculiar breed of people who are called computer scientists. And I'm pretty sure that there are also a lot of computer scientists who will not enjoy using WEB; some of us are glad that traditional programming languages have comparatively primitive capabilities for inserted comments, because such difficulties provide a good excuse for not documenting programs well. Thus, WEB may be only for the subset of computer scientists who like to write and to explain what they are doing. My hope is that the ability to make explanations more natural will cause more programmers to discover the joys of literate programming, because I believe it's quite a pleasure to combine verbal and mathematical skills; but perhaps I'm hoping for too much. The fact that a least one paper has been written that is a syntactically correct ALGOL 68 program encourages me to perservere in my hopes for the future. Perhaps we will even one day find Pulitzer prizes awarded to computer programs.

8. Donald Knuth goes on to write about his hopes for the future of WEB programming. In an interview with Donald Knuth by Amazon Books on the release of a new edition of Volume 1 of *The Art of Computer Programming* (July 1, 1997) he was asked:

> **Amazon.com**: What do you see as the most interesting advance in programming since you published the first edition?
>
> **Donald Knuth**: It's what I call literate programming, a technique for writing, documenting, and maintaining programs using a high-level language combined with a written language like English. This is discussed in my book Literate Programming.

9. In the same book, *Literate Programming*, there is a chapter called *How to read a* WEB. But it is actually quite straightforward.

10. Very briefly, each "Module" within angle brackets (< >) is expanded somewhere further down in the document. The trailing number you see within the brackets is where you can find this expansion. This provides a type of PDL (program descriptor language) for your program and greatly aids modularity and readability. It is also a highly effective method of top-down programming. The first module here is expanded further down, and contains most of the structure in standard Ada packages.

⟨ Package boiler-plate 11 ⟩

11. Capabilities specification.

⟨ Package boiler-plate 11 ⟩ ≡

```
output to file capability.ads
with TEXT_IO;
use TEXT_IO;
with test_io_pkg;
use test_io_pkg;
with generic_set_pkg;
with generic_map_pkg;
with ustrings;
use ustrings;
package capability is
   ⟨ Specification of types and variables visible from capability 12 ⟩
   ⟨ Specification of procedures visible from capability 14 ⟩
private
   ⟨ Specification of private types in capability 21 ⟩
end capability;
output to file capability.adb
with unchecked_deallocation;
with generic_map_pkg;
with Ada.Strings.Unbounded; Use Ada.Strings.Unbounded; with Ustrings;
use ustrings;
with Ada.Strings;
use Ada.strings;
with Ada.Characters.handling;
use Ada.Characters.handling;
package body capability is
   ⟨ Variables and types local to capability 23 ⟩
   ⟨ Procedures and Tasks in capability 28 ⟩
begin
   ⟨ Initialize capabilities 42 ⟩
end capability;
```

This code is used in section 10.

12.

⟨ Specification of types and variables visible from capability 12 ⟩ ≡

```
type devel_num is private;
type AString is access String;
type ExpertiseLevel is (low, medium, high);
package cap_map is new generic_map_pkg(key ⇒ Astring, result ⇒ ExpertiseLevel);
```

See also section 13.

This code is used in section 11.

13.

⟨ Specification of types and variables visible from *capability* 12 ⟩ +≡
  badid : **exception**;
  parsecapabilityerror : **exception**;

14.

⟨ Specification of procedures visible from *capability* 14 ⟩ ≡
  procedure *create_developer*(*developer* : **in** *String*; *yrid* : **out** *natural*);

See also sections 15, 16, 17, 18, 19, and 20.

This code is used in section 11.

15.

⟨ Specification of procedures visible from *capability* 14 ⟩ +≡
  procedure *add_capability*(*id* : **in** *natural*; *yrcap* : *String*; *exp* : *ExpertiseLevel*);
  procedure *add_capability*(*yrid* : **in** *devel_num*; *yrcap* : *cap_map.map*);
  procedure *add_capability*(*yrtask* : **in out** *cap_map.map*; *yrcap* : *String*;
    exp : *ExpertiseLevel*);

16.

⟨ Specification of procedures visible from *capability* 14 ⟩ +≡
  procedure *copy_capability*(*yrid* : **in** *natural*; *yrcap* : **out** *cap_map.map*);

17.

⟨ Specification of procedures visible from *capability* 14 ⟩ +≡
  procedure *print_capabilities*(*id* : *natural*);
  procedure *print_capabilities*(*yrtask* : *cap_map.map*);
  procedure *print_capabilities*(*fd* : *file_type*; *yrtask* : *cap_map.map*);
  procedure *print_developers*;
  function *get_developer_name*(*id* : *natural*)**return** *ustring*;

18.

⟨ Specification of procedures visible from *capability* 14 ⟩ +≡
  function *is_qualified*(*yrtask* : *cap_map.map*; *id* : *natural*)**return** *boolean*;

19.

⟨ Specification of procedures visible from *capability* 14 ⟩ +≡
  procedure *get_capability*(*str* : **in** *String*; *yrcap* : **out** *cap_map.map*);
  procedure *get_capability*(*fd* : *file_type*; *yrcap* : **out** *cap_map.map*);

20.

⟨ Specification of procedures visible from *capability* 14 ⟩ +≡
  procedure *get_developers*(*infile* : *string*);
  function *get_num_developers* **return** *natural*;

21.

⟨ Specification of private types in *capability* 21 ⟩ ≡
  package *cap_set* **is new** *generic_set_pkg*(*Astring*);
  type *capability* **is new** *cap_set.set*;
  max_developers : **constant** *natural* ← 20;
  type *devel_num* **is new** *natural* **range** 1 .. *max_developers*;

This code is used in section 11.

22. Capability Body.

23.

⟨ Variables and types local to *capability* 23 ⟩ ≡
```
debug : boolean ← false;
debug2 : boolean ← false;
gstring : Ustring;
```
See also sections 24, 25, 26, and 27.

This code is used in section 11.

24. Maintain a global set of capabilities;

⟨ Variables and types local to *capability* 23 ⟩ +≡
```
globalcaps : cap_set.set;
total_developers : natural ← 0;
```

25. Creating new step.

⟨ Variables and types local to *capability* 23 ⟩ +≡
```
function "+"(str : string)return Astring is
begin
  return new string'(str);
end "+";
```

26.

⟨ Variables and types local to *capability* 23 ⟩ +≡
```
MAXCAPS : constant natural ← 30;
type cap_num is new natural range 1 .. MAXCAPS;
type cap_array is array (cap_num) of Astring;
capabilities : cap_array ← (+"Ada", +"Database", +"XWindows", +"Graphics",
    +"Unix", others ⇒ null);
mycaps : cap_set.set;
total_caps : cap_num ← 5;
```

27.

⟨ Variables and types local to *capability* 23 ⟩ +≡
```
type cap_rec is
  record
    inuse : boolean ← false;
    name : Astring;
    cmap : cap_map.map;
  end record;
type developer_array is array (devel_num) of cap_rec;
developers : developer_array;
```

28.

```
⟨ Procedures and Tasks in capability 28 ⟩ ≡
  procedure create_developer(developer : in String; yrid : out natural) is
    knt : devel_num;
    tmpcap : cap_map.map;
  begin
    ⟨ Fetch an unused developer 29 ⟩
    ⟨ Assign capabilities to him 30 ⟩
    ⟨ Create capability out of his name 31 ⟩
  end create_developer;
```

See also sections 33, 34, 35, 41, 44, 45, 46, 47, 48, 49, 50, 52, 62, 63, and 69.

This code is used in section 11.

29.

```
⟨ Fetch an unused developer 29 ⟩ ≡
  knt ← 1;
  while developers(knt).inuse loop
    knt ← knt + 1;
  end loop;
  developers(knt).inuse ← true; total_developers ← total_developers + 1;
  yrid ← natural(knt);
```

This code is used in section 28.

30.

```
⟨ Assign capabilities to him 30 ⟩ ≡
  @{for i ∈ 1 .. total_caps loop
    cap_map.bind(capabilities(i), low, developers(knt).cmap);
  end loop;
  @}
```

This code is used in section 28.

31.

```
⟨ Create capability out of his name 31 ⟩ ≡
  total_caps ← total_caps + 1; capabilities(total_caps) ← +developer;
  cap_map.bind(capabilities(total_caps), high, developers(knt).cmap);
  cap_set.add(capabilities(total_caps), globalcaps);
  developers(knt).name ← capabilities(total_caps);
  ⟨ Add this capability to all the other developers 32 ⟩
  if debug then
    print_developers;
  end if;
```

This code is used in section 28.

32.

⟨ Add this capability to all the other developers 32 ⟩ ≡

```
@{for i ∈ devel_num loop
  if (developers(i).inuse) ∧ (i ≠ id) then
    if debug2 then
      put("Adding␣capability␣"); put(developer); put("␣to␣developer␣");
      put(developers(i).name.all); put_line(".␣");
    end if;
    cap_map.bind(capabilities(total_caps), low, developers(i).cmap);
  end if;
end loop;
@}
```

This code is used in section 31.

33.

⟨ Procedures and Tasks in *capability* 28 ⟩ +≡

```
procedure add_capability(yrtask : in out cap_map.map; yrcap : String;
        exp : ExpertiseLevel) is
  acap : Astring;
  is_member : boolean;
  knt : cap_num;
begin
  ⟨ First convert to upper-case 37 ⟩⟨ See if already in capabilities array 38 ⟩
  if ¬is_member then
    total_caps ← total_caps + 1; capabilities(total_caps) ← acap;
    cap_set.add(capabilities(total_caps), globalcaps); knt ← total_caps;
  end if;
  cap_map.bind(capabilities(knt), exp, yrtask);
end add_capability;
```

34.

```
⟨ Procedures and Tasks in capability 28 ⟩ +≡
  procedure add_capability(yrid : in devel_num; yrcap : cap_map.map) is
    exp1 : ExpertiseLevel;
    id : natural;
  begin
    id ← natural(yrid);
    for i ∈ 1 .. total_caps loop
      if cap_map.member(capabilities(i), yrcap) then
        exp1 ← cap_map.fetch(yrcap, capabilities(i));
        add_capability(id, capabilities(i).all, exp1);
      end if;
    end loop;
  end add_capability;
```

35.

```
⟨ Procedures and Tasks in capability 28 ⟩ +≡
  procedure add_capability(id : in natural; yrcap : String; exp : ExpertiseLevel) is
    acap : Astring;
    is_member : boolean;
    knt : cap_num;
    yrid : devel_num;
    package enum_io is new enumeration_io(ExpertiseLevel);
  begin
    yrid ← devel_num(id);
    if ¬developers(yrid).inuse then
      raise badid;
    end if;
    ⟨ First convert to upper-case 37 ⟩⟨ See if already in capabilities array 38 ⟩
    if ¬is_member then
      total_caps ← total_caps + 1; capabilities(total_caps) ← acap;
      cap_set.add(capabilities(total_caps), globalcaps); ⟨ Add to all developers 40 ⟩
    else
      ⟨ Update capabilities of this developer 39 ⟩
    end if;
  end add_capability;
```

36.

⟨ Convert to upper-case 36 ⟩ ≡

```
declare
   tstr : string ← yrcap;
   name : ustring;
begin
   name ← get_developer_name(natural(yrid));
   if tstr ≠ S(name) then
      for j ∈ 1 .. yrcap'length loop
         tstr(j) ← to_upper(tstr(j));
      end loop;
   end if;
   acap ← +tstr;
end;
```

37.

⟨ First convert to upper-case 37 ⟩ ≡

```
declare
   tstr : string ← yrcap;
begin
   for j ∈ 1 .. yrcap'length loop
      tstr(j) ← to_upper(tstr(j));
   end loop;
   acap ← +tstr;
end;
```

This code is used in sections 33 and 35.

38.

⟨ See if already in capabilities array 38 ⟩ ≡

```
is_member ← false;
for i ∈ 1 .. total_caps loop
   if (capabilities(i).all = acap.all) then
      acap ← capabilities(i); knt ← i; is_member ← true; exit;
   end if;
end loop;
```

This code is used in sections 33 and 35.

39.

⟨ Update capabilities of this developer 39 ⟩ ≡

```
if debug then
   put("Updating␣capabilities␣of␣developer:␣");
   put(S(get_developer_name(natural(yrid)))); put("␣␣"); put(capabilities(knt).all);
   put("␣=>␣"); enum_io.put(exp); new_line;
end if;
cap_map.bind(capabilities(knt), exp, developers(yrid).cmap);
```

This code is used in section 35.

40.

⟨ Add to all developers 40 ⟩ ≡

```
for i ∈ devel_num loop
   if (i ≠ yrid) then
      if (developers(i).inuse) then
         cap_map.bind(capabilities(total_caps), low, developers(i).cmap);
      end if;
   else
      cap_map.bind(capabilities(total_caps), exp, developers(i).cmap);
   end if;
end loop;
```

This code is used in section 35.

41. Copy everything but developer's name.

⟨ Procedures and Tasks in *capability* 28 ⟩ +≡

```
procedure copy_capability(yrid : in natural; yrcap : out cap_map.map) is
   exp1 : ExpertiseLevel;
   yr : devel_num;
   name1, name2 : ustring;
begin
   yr ← devel_num(yrid);
   for i ∈ 1 .. total_caps loop
      if cap_map.member(capabilities(i), developers(yr).cmap) then
         name1 ← U(capabilities(i).all); name2 ← get_developer_name(yrid);
         if name1 ≠ name2 then
            exp1 ← cap_map.fetch(developers(yr).cmap, capabilities(i));
            add_capability(yrcap, capabilities(i).all, exp1);
         end if;
      end if;
   end loop;
end copy_capability;
```

42.

⟨ Initialize capabilities 42 ⟩ ≡

```
cap_set.empty(globalcaps);
for i ∈ 1 .. total_caps loop
   ⟨ Convert to uppercase 43 ⟩
   capabilities(i) ← +S(gstring); cap_set.add(capabilities(i), globalcaps);
end loop;
```

This code is used in section 11.

43.

⟨ Convert to uppercase 43 ⟩ ≡

```
declare
   tstr : String ← capabilities(i).all;
   chr : Character;
begin
   for j ∈ 1 .. tstr'length loop
      chr ← tstr(j); tstr(j) ← to_upper(chr);
   end loop;
   gstring ← U(tstr);
end;
```

This code is used in section 42.

44.

```
⟨Procedures and Tasks in capability 28⟩ +≡
  function is_qualified(yrtask : cap_map.map; id : natural)return boolean is
    exp1, exp2 : ExpertiseLevel;
    answer : boolean ← true;
    yrid : devel_num;
  begin
    yrid ← devel_num(id);
    for i ∈ 1 .. total_caps loop
      if cap_map.member(capabilities(i), yrtask) then
        exp1 ← cap_map.fetch(yrtask, capabilities(i));
        if cap_map.member(capabilities(i), developers(yrid).cmap) then
          exp2 ← cap_map.fetch(developers(yrid).cmap, capabilities(i));
        else
          exp2 ← low;
        end if;
        if exp2 < exp1 then
          answer ← false; exit;
        end if;
      end if;
    end loop;
    return answer;
  end is_qualified;
```

45.

⟨ Procedures and Tasks in *capability* 28 ⟩ +≡

```
procedure print_capabilities(yrtask : cap_map.map) is
   exp : Expertiselevel;
   package exp_io is new enumeration_io(Expertiselevel);
   use exp_io;
   knt1, knt2 : cap_num;
begin
   knt1 ← 1; knt2 ← 1;
   for i ∈ 1 .. total_caps loop
      if cap_map.member(capabilities(i), yrtask) then
         knt1 ← knt1 + 1;
      end if;
   end loop;
   put("{");
   for i ∈ 1 .. total_caps loop
      if cap_map.member(capabilities(i), yrtask) then
         put(capabilities(i).all); put(":␣"); exp ← cap_map.fetch(yrtask, capabilities(i));
         put(exp); knt2 ← knt2 + 1;
         if knt2 < knt1 then
            put(",␣");
         end if;
      end if;
   end loop;
   put("}");
end print_capabilities;
```

46.

```
⟨ Procedures and Tasks in capability 28 ⟩ +≡
  procedure print_capabilities(fd : file_type; yrtask : cap_map.map) is
    exp : Expertiselevel;
    package exp_io is new enumeration_io(Expertiselevel);
    use exp_io;
    knt1, knt2 : cap_num;
  begin
    knt1 ← 1; knt2 ← 1;
    for i ∈ 1 .. total_caps loop
      if cap_map.member(capabilities(i), yrtask) then
        knt1 ← knt1 + 1;
      end if;
    end loop;
    put(fd, "{");
    for i ∈ 1 .. total_caps loop
      if cap_map.member(capabilities(i), yrtask) then
        put(fd, capabilities(i).all); put(fd, ":␣");
        exp ← cap_map.fetch(yrtask, capabilities(i)); put(fd, exp); knt2 ← knt2 + 1;
        if knt2 < knt1 then
          put(fd, ",␣");
        end if;
      end if;
    end loop;
    put(fd, "}");
  end print_capabilities;
```

47.

```
⟨ Procedures and Tasks in capability 28 ⟩ +≡
  procedure print_capabilities(id : natural) is
    exp : Expertiselevel;
    package exp_io is new enumeration_io(Expertiselevel);
    use exp_io;
    knt1, knt2 : cap_num;
    yrid : devel_num;
  begin
    yrid ← devel_num(id); knt1 ← 1; knt2 ← 1;
    for i ∈ 1 .. total_caps loop
      if cap_map.member(capabilities(i), developers(yrid).cmap) then
        knt1 ← knt1 + 1;
      end if;
    end loop;
    put("{");
    for i ∈ 1 .. total_caps loop
      if cap_map.member(capabilities(i), developers(yrid).cmap) then
        put(capabilities(i).all); put(":␣");
        exp ← cap_map.fetch(developers(yrid).cmap, capabilities(i)); put(exp);
        knt2 ← knt2 + 1;
        if knt2 < knt1 then
          put(",␣");
        end if;
      end if;
    end loop;
    put("}");
  end print_capabilities;
```

48.

```
⟨ Procedures and Tasks in capability 28 ⟩ +≡
  procedure print_developers is
    name : ustring;
  begin
    for i ∈ 1 .. total_developers loop
      name ← get_developer_name(i); put(S(name)); put(">␣"); print_capabilities(i);
      new_line;
    end loop;
  end print_developers;
```

49.

⟨ Procedures and Tasks in *capability* 28 ⟩ +≡

```
function get_developer_name(id : natural)return ustring is
  yrid : devel_num;
begin
  yrid ← devel_num(id); return U(developers(yrid).name.all);
end get_developer_name;
```

50.

⟨ Procedures and Tasks in *capability* 28 ⟩ +≡

```
procedure get_capability(fd : file_type; yrcap : out cap_map.map) is
  ⟨ Variables local to fget_capability 51 ⟩
begin
  chr ← ';';
  while chr ≠ '{' loop
    get_immediate(fd, chr);
  end loop;
  j ← 1; newstr(j) ← '{';
  while chr ≠ '}' loop
    j ← j + 1; get_immediate(fd, chr); newstr(j) ← chr;
  end loop;
  declare
    newstr2 : String(1 .. j);
  begin
    for k ∈ 1 .. j loop
      newstr2(k) ← newstr(k);
    end loop;
    tstr ← U(newstr2);
  end;
  if debug then
    put("get_capabilities␣(file)>␣calling␣get_capabilities␣(string)␣with");
    put("␣string␣=␣"); put(S(tstr)); new_line;
  end if;
  get_capability(S(tstr), yrcap);
end get_capability;
```

51.

⟨ Variables local to *fget_capability* 51 ⟩ ≡

```
j : positive;
chr : character;
newstr : String(1 .. 80);
tstr : ustring;
```

This code is used in section 50.

52.

⟨ Procedures and Tasks in *capability* 28 ⟩ +≡
  procedure *get_capability*(*str* : **in** *String*; *yrcap* : **out** *cap_map.map*) **is** ⟨ Variables local to *get_capability* 53 ⟩
    begin
      tstr ← *U*(*str*); *ind1* ← *index*(*tstr*, "{"); *ind2* ← *index*(*tstr*, "}");
      tstr ← *U*(*slice*(*tstr*, *ind1*, *ind2*));
      if *debug2* **then**
        put("Parsing␣string␣´"); *put*(*S*(*tstr*)); *put_line*("´.");
      end if;
      tstr ← *tail*(*tstr*, *length*(*tstr*) − *ind1*); *finished* ← *false*; **while** ¬*finished* **loop**
        ⟨ Get *capability* name pairs 54 ⟩ **end loop**; **end** *get_capability*;

53.

⟨ Variables local to *get_capability* 53 ⟩ ≡
  tstr : *ustring*;
  ind1 : *natural*;
  finished : *boolean*;

See also sections 56, 58, and 60.

This code is used in section 52.

54.

⟨ Get *capability* name pairs 54 ⟩ ≡
  ⟨ Check if finished 55 ⟩
  if ¬*finished* **then**
    ⟨ Get capability 57 ⟩⟨ Get *ExpertiseLevel* 59 ⟩⟨ Add new capability to map 61 ⟩
  end if;

This code is used in section 52.

55. Each name pair is separated by a colon ':'. It it is not there, then we are finished. (Provided we didn't look past the brace '}'.

⟨ Check if finished 55 ⟩ ≡
  ind2 ← *index*(*tstr*, ":"); *ind3* ← *index*(*tstr*, "}");
  if (*ind2* = 0) ∨ (*ind2* > *ind3*) **then**
    finished ← *true*;
  end if;
  if *ind3* = 0 **then**
    raise *parsecapabilityerror*;
  end if;

This code is used in section 54.

56.

⟨ Variables local to *get_capability* 53 ⟩ +≡
```
ind2, ind3 : natural;
```

57.

⟨ Get capability 57 ⟩ ≡
```
ind1 ← index_non_blank(tstr); tstr2 ← U(slice(tstr, ind1, ind2 − 1));
if debug2 then
  put("tstr2␣=␣"); put(S(tstr2)); new_line;
end if;
tstr ← tail(tstr, length(tstr) − ind2);
if debug2 then
  put("tstr␣=␣"); put(S(tstr)); new_line;
end if;
```
This code is used in section 54.

58.

⟨ Variables local to *get_capability* 53 ⟩ +≡
```
tstr2 : ustring;
```

59.

⟨ Get *ExpertiseLevel* 59 ⟩ ≡
```
ind1 ← index(tstr, ","); ind2 ← index(tstr, "}");
if ind1 = 0 then
  ind1 ← ind2; finished ← true;
end if;
tstr3 ← U(slice(tstr, 1, ind1 − 1));
if debug2 then
  put("tstr3␣=␣"); put(S(tstr3)); new_line;
end if;
enum_io.get(S(tstr3), exp, Last); tstr ← tail(tstr, length(tstr) − ind1);
if debug2 then
  put("tstr␣=␣"); put(S(tstr)); new_line;
end if;
```
This code is used in section 54.

60.

⟨ Variables local to *get_capability* 53 ⟩ +≡
```
tstr3 : ustring;
exp : ExpertiseLevel;
package enum_io is new enumeration_io(ExpertiseLevel);
Last : positive;
```

61.

⟨ Add new capability to map 61 ⟩ ≡
  add_capability (*yrcap* , *S*(*tstr2*), *exp*);

This code is used in section 54.

62.

⟨ Procedures and Tasks in *capability* 28 ⟩ +≡
  function *get_num_developers* **return** *natural* **is**
  begin
    return *total_developers* ;
  end *get_num_developers* ;

63.

⟨ Procedures and Tasks in *capability* 28 ⟩ +≡
  procedure *get_developers* (*infile* : *string*) **is**
    ⟨ Variables local to *get_developer* 65 ⟩
  begin
    ⟨ Open file 64 ⟩⟨ Read in developers 66 ⟩
  end *get_developers* ;

64.

⟨ Open file 64 ⟩ ≡
  open (*data_file* , *in_file* , *infile*);

This code is used in section 63.

65.

⟨ Variables local to *get_developer* 65 ⟩ ≡
  data_file : *file_type* ;

See also section 68.

This code is used in section 63.

66.

⟨ Read in developers 66 ⟩ ≡
  while ¬*end_of_file* (*data_file*) **loop**
    ⟨ Get developer's name and capabilities 67 ⟩
  end loop;

This code is used in section 63.

67.

⟨ Get developer's name and capabilities 67 ⟩ ≡

```
get_line(data_file, new_str, Last); tstr ← U(new_str); ind2 ← index(tstr, "{");
ind1 ← index_non_blank(tstr); name ← U(slice(tstr, ind1, ind2 − 1));
tstr ← tail(tstr, length(tstr) − ind2 + 1);
declare
  yrcap : cap_map.map;
begin
  get_capability(S(tstr), yrcap); create_developer(S(name), dummy);
  add_capability(devel_num(dummy), yrcap);
end;
```

This code is used in section 66.

68.

⟨ Variables local to *get_developer* 65 ⟩ +≡

```
Last : natural;
new_str : String(1 .. 132);
ind1, ind2 : natural;
name, tstr : ustring;
dummy : natural;
```

69.

⟨ Procedures and Tasks in *capability* 28 ⟩ +≡

```
procedure put_developers(outfile : string) is
  data_file : file_type;
begin
  create(data_file, out_file, outfile);
end put_developers;
```

70. Test capabilities driver. Here, finally, is the boilerplate. The Ada `WEB` tool **atangle** reads this and knows to write out two separate files, the specification and the body. (The Ada `WEB` tool **aweave** will write out just one documentation file.)

```
output to file testcap.adb
pragma suppress(all_checks);
with ustrings;
use ustrings;
with text_io;
use text_io;
with capability;
use capability;
procedure testcap is ⟨ Instantiate generic packages 71 ⟩⟨ Variables local to testcap 73 ⟩
  begin
    ⟨ Test if items are in set 72 ⟩
    ⟨ Create a task map and see if any developers qualify 76 ⟩
    ⟨ Print out items in set 74 ⟩
    ⟨ Check qualifications 75 ⟩
    ⟨ Try reading in some capabilities 78 ⟩
  end testcap;
```

71.

```
⟨ Instantiate generic packages 71 ⟩ ≡
  package nat_io is new integer_io(natural);
  use nat_io;
```

This code is used in section 70.

72.

```
⟨ Test if items are in set 72 ⟩ ≡
  create_developer("Bill␣Gates", myid); add_capability(myid, "Breathing", High);
  create_developer("Scott␣McNealy", myid2); add_capability(myid2, "Java", high);
  create_developer("Bill␣Joy", myid3); add_capability(myid3, "Unix", high);
  add_capability(myid3, "Systems␣programming", high);
```

This code is used in section 70.

73.

```
⟨ Variables local to testcap 73 ⟩ ≡
  myid, myid2, myid3 : natural;
```

See also sections 77 and 79.

This code is used in section 70.

74.

⟨ Print out items in set 74 ⟩ ≡

```
new_line; print_capabilities (myid); new_line; print_capabilities (myid2); new_line;
print_capabilities (myid3); new_line; print_capabilities (task1); new_line;
```

This code is used in section 70.

75.

⟨ Check qualifications 75 ⟩ ≡

```
if is_qualified (task1, myid) then
  put_line("Bill␣Gates␣is␣qualified.");
end if;
if is_qualified (task1, myid2) then
  put_line("Scott␣McNeally␣is␣qualified.");
end if;
if is_qualified (task1, myid3) then
  put_line("Bill␣Joy␣is␣qualified.");
end if;
```

This code is used in section 70.

76.

⟨ Create a task map and see if any developers qualify 76 ⟩ ≡

```
add_capability (task1, "Unix", medium);
```

This code is used in section 70.

77.

⟨ Variables local to testcap 73 ⟩ +≡

```
task1 : cap_map.map;
```

78.

⟨ Try reading in some capabilities 78 ⟩ ≡

```
create_developer ("John␣Evans", myid4); get_capability (testcapstr, task2);
print_capabilities (task2); new_line;
put_line ("Here␣is␣Bill␣Joy´s␣capabilities␣again>"); print_capabilities (myid3);
put_line ("Here␣are␣all␣the␣developer´s␣capabilities␣again."); 
print_developers; get_developers ("developers.txt"); print_developers;
```

This code is used in section 70.

79.

⟨ Variables local to testcap 73 ⟩ +≡

```
testcapstr : String ←
    "{Unix:high,Ada:high,Xwindows:medium,Systems␣Programming:medium}";
task2 : cap_map.map;
myid4 : natural;
```

80. System-dependent changes. This module should be replaced, if necessary, by changes to the program that are necessary to make TESTCAP work at a particular installation. It is usually best to design your change file so that all changes to previous modules preserve the module numbering; then everybody's version will be consistent with the printed program. More extensive changes, which introduce new modules, can be inserted here; then only the index itself will get a new module number.

81. RCS Keywords.

$RCSfile: capability.aweb,v
$Revision: 1.1
$Date: 1997/09/05 00:31:42
$Author: evansjr
$Id: capability.aweb,v 1.1 1997/09/05 00:31:42 evansjr Exp evansjr
$Locker: evansjr
$State: Exp

82. Index. Here is a cross-reference table for the TESTCAP program. All modules in which an identifier is used are listed with that identifier, except that reserved words are indexed only when they appear in format definitions, and the appearances of identifiers in module names are not indexed. Underlined entries of subprograms and packages correspond to sections where this entity is specified, whereas entries in italic type correspond to the section where the entity's body is stated. For any other identifier underlined entries correspond to where the identifier was declared. Error messages and a few other things like "ASCII code" are indexed here too.

⟨ Add new capability to map 61 ⟩ Used in section 54.
⟨ Add this capability to all the other developers 32 ⟩ Used in section 31.
⟨ Add to all developers 40 ⟩ Used in section 35.
⟨ Assign capabilities to him 30 ⟩ Used in section 28.
⟨ Check if finished 55 ⟩ Used in section 54.
⟨ Check qualifications 75 ⟩ Used in section 70.
⟨ Convert to upper-case 36 ⟩
⟨ Convert to uppercase 43 ⟩ Used in section 42.
⟨ Create a task map and see if any developers qualify 76 ⟩ Used in section 70.
⟨ Create capability out of his name 31 ⟩ Used in section 28.
⟨ Fetch an unused developer 29 ⟩ Used in section 28.
⟨ First convert to upper-case 37 ⟩ Used in sections 33 and 35.
⟨ Get capability 57 ⟩ Used in section 54.
⟨ Get developer's name and capabilities 67 ⟩ Used in section 66.
⟨ Get *ExpertiseLevel* 59 ⟩ Used in section 54.
⟨ Get *capability* name pairs 54 ⟩ Used in section 52.
⟨ Initialize capabilities 42 ⟩ Used in section 11.
⟨ Instantiate generic packages 71 ⟩ Used in section 70.
⟨ Open file 64 ⟩ Used in section 63.
⟨ Package boiler-plate 11 ⟩ Used in section 10.
⟨ Print out items in set 74 ⟩ Used in section 70.
⟨ Procedures and Tasks in *capability* 28, 33, 34, 35, 41, 44, 45, 46, 47, 48, 49, 50, 52, 62, 63, 69 ⟩ Used in section 11.
⟨ Read in developers 66 ⟩ Used in section 63.
⟨ See if already in capabilities array 38 ⟩ Used in sections 33 and 35.
⟨ Specification of private types in *capability* 21 ⟩ Used in section 11.
⟨ Specification of procedures visible from *capability* 14, 15, 16, 17, 18, 19, 20 ⟩ Used in section 11.
⟨ Specification of types and variables visible from *capability* 12, 13 ⟩ Used in section 11.
⟨ Test if items are in set 72 ⟩ Used in section 70.
⟨ Try reading in some capabilities 78 ⟩ Used in section 70.
⟨ Update capabilities of this developer 39 ⟩ Used in section 35.
⟨ Variables and types local to *capability* 23, 24, 25, 26, 27 ⟩ Used in section 11.
⟨ Variables local to testcap 73, 77, 79 ⟩ Used in section 70.
⟨ Variables local to *fget_capability* 51 ⟩ Used in section 50.
⟨ Variables local to *get_capability* 53, 56, 58, 60 ⟩ Used in section 52.
⟨ Variables local to *get_developer* 65, 68 ⟩ Used in section 63.

Task Generator

[Ada '95—Version 2.0]
September 4, 1997

This page intentionally left blank

1. Introduction. This routine generates a number of tasks for which a valid schedule exists. The output of this routine is fed into the scheduling algorithm to test its performance. This particular version uses the capability model described in my thesis.

2. This is the main routine that starts everything.

```
output to file task_generator.adb
pragma Unsuppress(all_checks);
with CALENDAR;
use CALENDAR;
with text_io;
use text_io;
⟨ Needed packages 10 ⟩
procedure task_generator is
  package nat_io is new integer_io(natural);
  use nat_io;
  package flt_io is new float_io(float);
  use flt_io;
  package bool_io is new enumeration_io(boolean);
  use bool_io;
  ⟨ Variables local to task_generator 6 ⟩
  ⟨ Functions local to task_generator 33 ⟩
begin
  ⟨ Get input parameters 4 ⟩
  declare
    ⟨ Allocate a static array to hold tasks for schedule 12 ⟩
  begin
    ⟨ Compute earliest available time (EAT) in resource matrix 15 ⟩
    R ← laxity;
    for i ∈ 1 .. tasks loop
      ⟨ Generate another task 16 ⟩
    end loop;
    if do_alternate then
      ⟨ Convert to calendar time 35 ⟩
    end if;
    ⟨ Print out results 34 ⟩
  end;
end task_generator;
```

3. This routine takes two input parameters. (1) "**-tasks**" the number of tasks to generate; and (2) "**-laxity**" the laxity, or tightness, parameter. This is formally defined as

$$T_D - T_{est} + T_P$$

where T_D is the deadline, T_{est} is the earliest start-time, and T_P is the processing time. It is computed *apriori* by the *task_generator*.

$$T_D = (1 + R) * SC$$

where R is an input parameter, and SC is the shortest completion time.

4. The input values are read in using the routines in package *getopt*. I read in the number of tasks to compute, the "laxity" of the schedule, and a "seed" for the random number generator.

```
⟨ Get input parameters 4 ⟩ ≡
  tasks ← 10;
  if option_present(U("-tasks")) then
    get_option(U("-tasks"), param); get(S(param), tasks, Last);
  end if;
  laxity ← 0.0;
  if option_present(U("-laxity")) then
    get_option(U("-laxity"), param); get(S(param), laxity, Last);
  end if;
  seed ← 68069;
  if option_present(U("-seed")) then
    get_option(U("-seed"), param); get(S(param), seed, Last);
  end if;
  ⟨ Get NRaD option 5 ⟩
  ⟨ Get developer file 7 ⟩
  ⟨ Get developers 8 ⟩
```

This code is used in section 2.

5.

```
⟨ Get NRaD option 5 ⟩ ≡
  if option_present(U("-nrad")) then
    get_option(U("-nrad"), param); get(S(param), nrad, Last);
  else
    nrad ← true;
  end if;
```

See also section 45.

This code is used in section 4.

6.

⟨ Variables local to *task_generator* 6 ⟩ ≡
  tasks : *natural*;
  laxity : *float*;
  Last : *positive*;
  param : *Ustring*;
  seed : *natural*;
  nrad : *boolean*;

See also sections 9, 13, 18, 19, 22, 26, 29, 30, 32, 37, 41, 44, and 46.

This code is used in section 2.

7.

⟨ Get developer file 7 ⟩ ≡
  if *name_present*(1) **then**
    get_name(*devfile*, 1);
  else
    raise *nofilename*;
  end if;

This code is used in section 4.

8.

⟨ Get developers 8 ⟩ ≡
  get_developers(*S*(*devfile*)); *num_developers* ← *get_num_developers*;

This code is used in section 4.

9.

⟨ Variables local to *task_generator* 6 ⟩ +≡
  nofilename : **exception**;
  devfile : *ustring*;
  num_developers : *natural*;

10. We need some more packages to read in the parameters. Specifically the package *getopt* written by this student; and the package *Ustrings*—used for manipulating "unbounded" strings.

⟨ Needed packages 10 ⟩ ≡
  with *Ustrings*;
  use *Ustrings*;
  with *GetOpt*;
  use *GetOpt*;

See also sections 11, 14, 24, and 39.

This code is used in section 2.

11. We also add the following package to enhance the capability model the scheduler (and *task_generator*) can use.

⟨ Needed packages 10 ⟩ +≡
```
with capability;
use capability;
```

12.

⟨ Allocate a static array to hold tasks for schedule 12 ⟩ ≡
```
sched : array (1 .. tasks) of StepRecord;
newsched : array (1 .. tasks) of NewStepRecord;
mysample : bool_array(1 .. tasks);
```
See also section 31.

This code is used in section 2.

13.

⟨ Variables local to *task_generator* 6 ⟩ +≡
```
type NewStepRecord is
   record
      CalDuration : Duration;
      CalStartTime : Time;
      CalDeadLine : Time;
   end record;
```

14.

⟨ Needed packages 10 ⟩ +≡
```
with generic_set_pkg;
with SchedPrims;
use SchedPrims;
```

15.

⟨ Compute earliest available time (EAT) in resource matrix 15 ⟩ ≡
```
MATRIX_MIN(EAT, Min, COL);
```
This code is used in sections 2 and 28.

16.

⟨ Generate another task 16 ⟩ ≡
  ⟨ Compute duration of task *T_p* 17 ⟩
  ⟨ Compute predecessors 25 ⟩
  ⟨ Compute earliest start time 20 ⟩
  ⟨ Compute deadline *T_D* 21 ⟩
  ⟨ Compute priority *P* 23 ⟩
  sched(*i*).*StepID* ← *i*; *sched*(*i*).*Deadline* ← *T_D*; *sched*(*i*).*Priority* ← *P*;
  sched(*i*).*EstimatedDuration* ← *T_p*; ⟨ Assign expertise level 27 ⟩
  ⟨ Update resource matrix 28 ⟩

This code is used in section 2.

17. The duration varies in length between *MIN_D* and *MAX_D*. The duration will not go over the maximum task deadline (*MTD*).

⟨ Compute duration of task *T_p* 17 ⟩ ≡
  T_p ← *uniform*(*MIN_D*, *MAX_D*); { duration }

This code is used in section 16.

18. Minimum task duration.

⟨ Variables local to *task_generator* 6 ⟩ +≡
  Min_D : *natural* ← 2;

19. Maximum task duration.

⟨ Variables local to *task_generator* 6 ⟩ +≡
  Max_D : *natural* ← 10;

20.

⟨ Compute earliest start time 20 ⟩ ≡

```
for j ∈ 1 .. (i − 1) loop
  if nat_set.member(j, sched(i).predecessors) then
    if Sched(j).deadline > sched(i).EarliestStartTime then
      sched(i).EarliestStartTime ← Sched(j).Deadline;
      if debug then
        put("Modified␣Sched.("); put(i,1); put(")␣to␣be␣");
        put(sched(i).EarliestStartTime,1); put_line(".␣");
      end if;
    end if;
  end if;
end loop;
```

This code is used in section 16.

21. The deadline (T_D) is a function of the duration and the least value of a resource in the resource matrix.

```
⟨ Compute deadline T_D 21 ⟩ ≡
  TT ← integer(float(T_p) * (1.0 + laxity));
  if debug then
    put("Old␣deadline␣is␣"); put(sched(i).Deadline, 1); put(".␣");
  end if;
  if sched(i).EarliestStartTime > EAT(COL) then
    T_D ← TT + sched(i).EarliestStartTime;
  else
    T_D ← TT + EAT(COL);
  end if;
  if debug then
    put("New␣deadline␣is␣"); put(T_D, 1); put_line(".␣");
  end if;
```

This code is used in section 16.

22.

```
⟨ Variables local to task_generator 6 ⟩ +≡
  debug : boolean ← false;
  debug2 : boolean ← false;
```

23. A random value.

```
⟨ Compute priority P 23 ⟩ ≡
  P ← uniform(4, 10);
```

This code is used in section 16.

24.

```
⟨ Needed packages 10 ⟩ +≡
  with Probability;
  use Probability;
```

25. I choose to select M out of N tasks as predecessors. M has an upper limit of *Max_Predecessors* and N is the number of previous tasks assigned. If the number of previous tasks scheduler is less than *Max_Predecessors* then the minimum is selected then the upper limit is the number of previous tasks scheduled. M is selected randomly.

```
⟨ Compute predecessors 25 ⟩ ≡
  if do_predecessor then
    if i ≤ Max_Predecessors then
      ptasks ← (i − 1);
    else
      ptasks ← Max_Predecessors;
    end if;
    nsamp ← uniform(0, ptasks);
    if i > 1 then
      sample(nsamp, i − 1, mysample);
      for j ∈ 1 .. (i − 1) loop
        if mysample(j) then
          t1 ← nat_set.size(Sched(i).Predecessors);
          t2 ← nat_set.size(Sched(j).Successors);
          if (t1 < Max_Predecessors) ∧ (t2 < Max_Predecessors) then
            nat_set.add(j, Sched(i).Predecessors); nat_set.add(i, Sched(j).Successors);
          end if;
        end if;
      end loop;
    end if;
  end if;
```

This code is used in section 16.

26.

```
⟨ Variables local to task_generator 6 ⟩ +≡
  Max_Predecessors : constant natural ← 0;
  ptasks, nsamp : natural;
  t1, t2 : natural;
```

27.

```
⟨ Assign expertise level 27 ⟩ ≡
  declare
    tmpcap : cap_map.map;
  begin
    copy_capability(COL, sched(i).ExpLevel);
  end;
```

This code is used in section 16.

28.

⟨ Update resource matrix 28 ⟩ ≡
```
if debug then
  put("Before␣Update:␣"); put("EAT("); put(COL,1); put(")␣=␣");
  put(EAT(COL),1); put_line(".␣");
end if;
EAT(COL) ← T_D;
if debug then
  put("After␣Update:␣"); put("EAT("); put(COL,1); put(")␣=␣");
  put(EAT(COL),1); put_line(".␣");
end if;
⟨ Compute earliest available time (EAT) in resource matrix 15 ⟩
```
This code is used in section 16.

29.

⟨ Variables local to *task_generator* 6 ⟩ +≡
```
R : float ← 0.7;
R3 : natural ← 3;   { laxity }
UU : natural ← 1;
U1 : natural ← 3;   { seed }
U2 : natural ← 1;
type RESOURCE_MATRIX is array (POSITIVE range <>) of natural;
do_predecessor : BOOLEAN ← true;
```

30. Max task deadline.

⟨ Variables local to *task_generator* 6 ⟩ +≡
```
MTD : natural ← 70000;
```

31. The way this is defined, it "hard-codes" the maximum number of designers per leve to '**2**.' (Must be in concordance with the maximum number of designers defined above.)

⟨ Allocate a static array to hold tasks for schedule 12 ⟩ +≡
```
EAT : RESOURCE_MATRIX(1 .. num_developers) ← (others ⇒ 0);
```

32.

⟨ Variables local to *task_generator* 6 ⟩ +≡
```
P, T_D, T_p, R1, R2, C : natural;
Min : natural ← 0;
COL : natural ← 1;
COUNT : natural ← 0;
TT : integer;
```

33. This finds the smallest value in the resource matrix and returns the index of the minimum value.

```
⟨ Functions local to task_generator 33 ⟩ ≡
  procedure MATRIX_MIN (MATRIX : in RESOURCE_MATRIX; MIN : out
        natural; K1 : out natural) is
    Min1 : natural ← MATRIX(1);
  begin
    K1 ← 1;
    for j ∈ 2 .. MATRIX'Length loop
      if Min1 > MATRIX(j) then
        Min1 ← MATRIX(j); K1 ← j;
      end if;
    end loop;
    MIN ← Min1;
  end MATRIX_MIN;
```

This code is used in section 2.

34. Procedure *Put_set* is declared in package *schedprims*.

```
⟨ Print out results 34 ⟩ ≡
  for i ∈ 1 .. tasks loop
    if ¬do_alternate then
      put(sched(i).Deadline, 4); put(sched(i).Priority, 4);
      put(sched(i).EstimatedDuration, 5); put(sched(i).EarliestStartTime, 5);
          { earliest start time }
      put("␣"); put_set(Sched(i).Predecessors); put("␣"); put_set(Sched(i).Successors);
      put("␣"); print_capabilities(Sched(i).ExpLevel); new_line;
    else
      print_date(newsched(i).CalDeadline); put(sched(i).Priority, 5);
      put(sched(i).EstimatedDuration, 5); put("␣");
      print_date(newsched(i).CalStartTime); put("␣"); put_set(Sched(i).Predecessors);
      put("␣"); put_set(Sched(i).Successors); put("␣");
      print_capabilities(Sched(i).ExpLevel); new_line;
    end if;
  end loop;
```

This code is used in section 2.

35.

```
⟨ Convert to calendar time 35 ⟩ ≡
  ⟨ Get start date 36 ⟩
  Start_Time ← Current_Time;
  for i ∈ 1 .. tasks loop
    ⟨ Convert Start Time to Calendar Time 43 ⟩
    ⟨ Convert Task Duration to Duration type 42 ⟩
    ⟨ Convert Deadline to Calendar Time 47 ⟩
  end loop;
```

This code is used in section 2.

36. For now "hard-code" a date (July 1st, 1997).

```
⟨ Get start date 36 ⟩ ≡
  Current_Time ← Time_of(1997, 7, 3); ⟨ Find first work-day 38 ⟩ if debug2 then ⟨ Print
        out first work day 40 ⟩ end if;
```

This code is used in section 35.

37.

```
⟨ Variables local to task_generator 6 ⟩ +≡
  Current_Time, Start_Time : Time;
  do_alternate : boolean ← false;
```

38.

```
⟨ Find first work-day 38 ⟩ ≡
  while (¬IsWorkDay(Current_Time, nrad)) loop
    Current_Time ← Current_time + Day_Duration'Last;
  end loop;
```

This code is used in section 36.

39. Package to find federal off-days till year 2099 (barring acts of God, or Congress).

```
⟨ Needed packages 10 ⟩ +≡
  with calyr;
  use calyr;
```

40.

```
⟨ Print out first work day 40 ⟩ ≡
  Split(Current_Time, Year, Month, Day, Seconds); put("The␣first␣work␣day␣is␣");
  put(Month, 3); put("/"); put(Day, 3); put("/"); put(Year, 4); put_line(".");
```

This code is used in section 36.

41.

⟨ Variables local to *task_generator* 6 ⟩ +≡

```
Year : Year_number;
Month : Month_number;
Day : Day_Number;
Seconds : Day_Duration;
```

42.

⟨ Convert Task Duration to Duration type 42 ⟩ ≡

```
newsched(i).CalDuration ← ConvertHourstoDuration(sched(i).EstimatedDuration);
```

This code is used in section 35.

43.

⟨ Convert Start Time to Calendar Time 43 ⟩ ≡

```
TotalTime ← ConvertHoursToDuration(Sched(i).EarliestStartTime);
newsched(i).CalStartTime ← DurationToCalendarTime(Start_Time, dailyhours,
     TotalTime, NRad);
if debug2 then
   testduration ← CalendarTimetoDuration(Start_Time, dailyhours,
        newsched(i).CalStartTime, NRaD);
   testhours ← ConvertDurationToHours(testduration);
   if sched(i).EarliestStartTime ≠ testhours then
      put("ERROR␣in␣CalendarTimetoWorkHours"); new_line;
      put("CalendarTime␣returned␣"); put(testhours);
      put("␣and␣it␣should␣have␣returned␣"); put(sched(i).EarliestStartTime);
      put("."); put("(NRaD)␣=␣"); put(NRaD); put(")."); new_line;
      put("The␣Start␣Time␣is␣"); print_date(Start_Time);
      put(".␣The␣TotalTime␣is␣"); put(float(TotalTime));
      put("In␣hours␣that␣is␣"); put(ConvertDurationtoHours(TotalTime));
      put(")"); put(".␣␣"); new_line;
   end if;
end if;
```

This code is used in section 35.

44.

⟨ Variables local to *task_generator* 6 ⟩ +≡

```
testduration : Duration;
testhours : natural;
```

45.

```
⟨Get NRaD option 5⟩ +≡
  for day ∈ Mon .. Thu loop
    if nrad then
      dailyhours(Day) ← 9.0 * SecondsPerHour;
    else
      dailyhours(Day) ← 8.0 * SecondsPerHour;
    end if;
  end loop;
  dailyhours(Fri) ← 8.0 * SecondsPerHour;
```

46.

```
⟨Variables local to task_generator 6⟩ +≡
  dailyhours : Workhours;
  SecondsPerHour : constant Duration ← 3600.0;
  TotalTime : duration;
```

47.

```
⟨Convert Deadline to Calendar Time 47⟩ ≡
  TotalTime ← ConvertHoursToDuration(Sched(i).Deadline);
  newsched(i).CalDeadline ← DurationToCalendarTime(Start_Time, dailyhours,
      TotalTime, NRad);
```

This code is used in section 35.

48. System-dependent changes. This module should be replaced, if necessary, by changes to the program that are necessary to make MAIN work at a particular installation. It is usually best to design your change file so that all changes to previous modules preserve the module numbering; then everybody's version will be consistent with the printed program. More extensive changes, which introduce new modules, can be inserted here; then only the index itself will get a new module number.

49. RCS Keywords.

$RCSfile: task_generator.aweb,v
$Revision: 1.3
$Date: 1997/09/05 00:35:25
$Author: evansjr
$Id: task_generator.aweb,v 1.3 1997/09/05 00:35:25 evansjr Exp evansjr
$Locker: evansjr
$State: Exp

50. Index. Here is a cross-reference table for the MAIN program. All modules in which an identifier is used are listed with that identifier, except that reserved words are indexed only when they appear in format definitions, and the appearances of identifiers in module names are not indexed. Underlined entries of subprograms and packages correspond to sections where this entity is specified, whereas entries in italic type correspond to the section where the entity's body is stated. For any other identifier underlined entries correspond to where the identifier was declared. Error messages and a few other things like "ASCII code" are indexed here too.

⟨ Allocate a static array to hold tasks for schedule 12, 31 ⟩ Used in section 2.
⟨ Assign expertise level 27 ⟩ Used in section 16.
⟨ Compute deadline *T_D* 21 ⟩ Used in section 16.
⟨ Compute duration of task *T_p* 17 ⟩ Used in section 16.
⟨ Compute earliest available time (EAT) in resource matrix 15 ⟩ Used in sections 2 and 28.
⟨ Compute earliest start time 20 ⟩ Used in section 16.
⟨ Compute predecessors 25 ⟩ Used in section 16.
⟨ Compute priority P 23 ⟩ Used in section 16.
⟨ Convert Deadline to Calendar Time 47 ⟩ Used in section 35.
⟨ Convert Start Time to Calendar Time 43 ⟩ Used in section 35.
⟨ Convert Task Duration to Duration type 42 ⟩ Used in section 35.
⟨ Convert to calendar time 35 ⟩ Used in section 2.
⟨ Find first work-day 38 ⟩ Used in section 36.
⟨ Functions local to *task_generator* 33 ⟩ Used in section 2.
⟨ Generate another task 16 ⟩ Used in section 2.
⟨ Get NRaD option 5, 45 ⟩ Used in section 4.
⟨ Get developer file 7 ⟩ Used in section 4.
⟨ Get developers 8 ⟩ Used in section 4.
⟨ Get input parameters 4 ⟩ Used in section 2.
⟨ Get start date 36 ⟩ Used in section 35.
⟨ Needed packages 10, 11, 14, 24, 39 ⟩ Used in section 2.
⟨ Print out first work day 40 ⟩ Used in section 36.
⟨ Print out results 34 ⟩ Used in section 2.
⟨ Update resource matrix 28 ⟩ Used in section 16.
⟨ Variables local to *task_generator* 6, 9, 13, 18, 19, 22, 26, 29, 30, 32, 37, 41, 44, 46 ⟩ Used in section 2.

LIST OF REFERENCES

[1] Salah El-Din Mohammed Badr. *A Model and Algorighms For A Software Evolution Control System.* PhD thesis, Naval Postgraduate School, Monterey, CA 93943, December 1993.

[2] Ramamritham K., Stankovic J. A., and P. Shiah. Efficient scheduling algorithm for real-time multiprocessor systems. Technical Report 89-37, University of Massachusetts, Amherst, 1989. Dept. of Computer and Information Science.

[3] Ramamritham K., Stankovic J. A., P. Shiah, and Zhao W. Real-time scheduling algorithms for multiprocessors. Technical Report 89-47, University of Massachusetts, Amherst, 1989. Dept. of Computer and Information Science.

[4] Luqi. A graph model for sofware evolution. *IEEE Transactions on Software Engineering*, 16(8), August 1990.

[5] A.K. Mok. and M.L. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *Proceedings of the IEEE Real-Time Systems Symposium*, November 1978.

[6] J. D. Ullman. NP-complete scheduling problems. *J. Comput. System Sci.*, 10:384–393, 1975.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road., Ste 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   Monterey, CA 93943

3. Center for Naval Analysis 1
   4401 Ford Ave.
   Alexandria, VA 22302

4. Dr. Ted lewis, Chairman, Code CS/Lt 1
   Computer Science Dept.
   Naval Postgraduate School
   Monterey, CA 93943

5. Chief of Naval Research 1
   800 North Quincy St.
   Arlington, VA 22217

6. Dr. Luqi, Code CS/Lq 1
   Computer Science Dept.
   Naval Postgraduate School
   Monterey, CA 93943

7. Dr. Marvin Langston 1
   1225 Jefferson Davis Highway
   Crystal Gateway 2 / Suite 1500
   Arlington, VA 22202-4311

8. David Hislop 1
   U.S. Army Research Office
   PO Box 12211
   Research Triangle Park,NC 27709-2211

9. Capt. Talbot Manvel 1
   Naval Sea Systems Command
   2531 Jefferson Davis Hwy.
   Attn: TMS 378 Capt. Manvel
   Arlington ,VA 22240-5150

10. CDR Michael McMahon 1
Naval Sea System Command
2531 Jefferson Davis Hwy.
Arlington, VA 22242-5160

11. Elizabeth Wald 1
Office Of Naval Research
800 N. Quincy St.
ONR CODE 311
Arlington , VA 22132-5660

12. Dr. Ralph Wachter 1
Office of Naval Research
800 N. Quincy St.
CODE 311
Arlington, VA 22217-5660

13. Army Research Lab 1
115 O'Keefe Building
Attn: Mark Kendall
Atlanta, GA 30332-0862

14. National Science Foundation 1
Attn: Bruce Barnes
Div. Computer & Computation Research
1800 G St. NW
Washington, DC 20550

15. National Science Foundation 1
Attn: Bill Agresty
4201 Wilson Blvd.
Arlington, VA 22230

16. Hon. John W. Douglas 1
Assistant Secretary of the Navy
(Research, Devlopment and Aquisition)
Room E741
1000 Navy Pentagon
Washington, DC 20350-1000

17. Technical Library Branch 1
Naval Command, Control, and Ocean Surveillance Center
RDT&E Division, Code D0274
San Diego, CA 92152-5001

18. Head, Intelligence, Surveillance, & Reconnaissance Dept. 1
Naval Command, Control and Ocean Surveillance Center
RDT&E Division, Code D70
San Diego, CA 92152-5001

19. Head, Joint & National Systems Division 1
Naval Command, Control and Ocean Surveillance Center
RDT&E Division, Code D73
San Diego, CA 92152-5001