AN EXPERIMENTAL IMPLEMENTATION OF ACTION-BASED CONCURRENCY

AN EXPERIMENTAL IMPLEMENTATION OF ACTION-BASED CONCURRENCY

By

XIAO-LEI CUI, B.SC.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree of

Master of Science

McMaster University

MASTER OF SCIENCE (2009)
(Computing and Software)

McMaster University
Hamilton, Ontario

TITLE: An Experimental Implementation of Action-Based Concurrency

AUTHOR: Xiao-Lei Cui, B.Sc. (McMaster University)

SUPERVISOR: Dr. Emil Sekerinski

NUMBER OF PAGES: viii, 198

Abstract

This thesis reports on an implementation of an action-based model for concurrent programming. Concurrency is expressed by allowing objects to have actions with a guard and a body. Each action has its own execution context, and concurrent execution is realized when program execution is happening in more than one context at a time. Two actions of different objects can run concurrently, and they are synchronized whenever a shared object is accessed simultaneously by both actions. The appeal of this model is that it allows a conceptually simple framework for designing and analyzing concurrent programs.

To experiment with action-based concurrency, we present a small language, ABC Pascal, which is an experimental attempt as a proof of feasibility of such a model, and also meant to help identify issues for achieving reasonable efficiency in implementation. It extends a subset of Pascal that supports basic sequential programming constructs, and provides action-based concurrency as the action-based model prescribes.

This work deals with the specification and implementation of ABC Pascal. The one-pass compiler directly generates assembly code, without devoting efforts to optimization. While the code is not optimized, the results that ABC Pascal has achieved in performance testing are so far comparable to mainstream concurrent programming languages.

Acknowledgements

In no particular order:

I am very grateful to many people for their support and help through this process.

First, I would like to thank the subscribers of several google groups (e.g. multi-threading, Ada, Intel x86, etc) who were willing to share their knowledge and opinions and help me clarify various obscurities.

I would also like to make special mention of three reviewers who contributed detailed proofreading and offered insightful suggestions: Eden Burton, David Kelk and Dan Zingaro.

Many thanks to Dan Zingaro and his original work on the stack-based compiler of Pascal0. Two years ago, my work was started by building a derivation of Dan's previous effort.

Many thanks to Dr. Franya Franek and Dr. Ryszard Janicki, the examiners of my work, for providing insightful comments for further improvement of this thesis, especially in light of their tight schedules before the holiday season.

Thanks most of all to my advisor, Dr. Emil Sekerinski, for giving me this opportunity to work on the area that greatly interested me. Without his ideas, guidance and constant support, this project could not have progressed to where it is today.

My final thanks to my family for their long time support for my studies in Canada.

Contents

List of Figures

Listings

Chapter 1

Introduction

With the growing popularity of multiprocessor machines and multicore processors, more and more computers are able to execute programs in parallel. The increasing kernel support for multithreading found on various contemporary operating systems is another factor that boosts the development of concurrent programming languages. Thus, the motivation that leads to the development of concurrent programming tools is based on advances in both computer hardware and operating systems. However, the design of concurrent programs remains difficult, and analyzing concurrent programs remains an elusive task.

In this thesis, we present a small language, called ABC Pascal, that supports action-based concurrency. The major point of departure is the potential to simplify the design and analysis of concurrent programs. Yet, the underlying theory development is incomplete and will not be addressed in this thesis.

The implementation is intended to run on IA-32 platforms, an acronym for Intel 32-bit Architecture. The compiler generates assembly code which is in turn to be processed by an assembler and linker.

1.1 Introduction to Action-Based Concurrency

In the model of action-based concurrency, a program is described by a set of actions of the form $A = p \rightarrow S$, where p is a predicate, the *guard*, and S is the body of the action A. If the guard of A evaluates to true, action A is enabled, meaning that it is eligible to execute its body. In the case where multiple actions are enabled at the same time, the selection among them is nondeterministic. The execution of actions is restricted by the following fairness condition: each action is executed repeatedly until program terminates. In an implementation, the program execution may be stopped when it is detected that a final state has been reached.

In addition to actions, if one allows objects to evolve autonomously, by attaching actions to objects, these objects become a natural "unit" of concurrency. With actions defined within objects, concurrency is expressed by allowing objects to have arbitrary number of actions, in addition to fields and procedures. Similar to actions, procedures defined within objects can also be guarded by a predicate.

The most important concepts in concurrent programming are communication and synchronization. With the action-based model, communication between objects is expressed by procedure calls; synchronization is realized by the guards of procedures.

Within an object, at most one action or procedure can executed at a time. However, from program-wide perspective, all objects in principle can execute concurrently.

The appeal of this model is that it conceptually simplifies development and analysis of multithreaded programs. Programmer is freed from using conventional concurrent programming constructs (e.g. wait and signal), because these constructs are absent from the model. If no interference is allowed between actions, reasoning about concurrent executions of actions in this model can be reduced to reasoning about some sequential execution, as if there is no interleaving among the actions. ABC Pascal does not prohibit interference but rather limits interference with its built-in synchronization mechanisms. The action-based model can be used to describe and analyze both shared-variable and message-passing concurrency.

1.2 Goals of The Thesis

The aim of this thesis is to develop a simple language that conforms to the action-based concurrency model. The prototype implementation is an experimental attempt as a proof of feasibility of such a model. It also serves the purpose of identifying potential issues for achieving reasonable efficiency. Hence, ABC Pascal, an object-based language was created. ABC Pascal is an extension of Pascal0. Pascal0 is a subset of Pascal, a sequential language suited for teaching purpose [19].

Previous effort on action-based concurrency was the development of a language called Lime [13]. The prototype implementation of Lime translates source code to assembly code for JVM. During program execution, a fixed number of Java threads are created to run the methods belonging to objects. The problem with this implementation of Lime is that a false deadlock is possible, when all Java threads are conditionally blocked but there are other Lime methods eligible to execute [13]. This is a false deadlock situation because the Lime source program is not deadlocked but correct. The implementation of ABC Pascal maps one thread to each object that has action(s) attached, so false deadlock is avoided. This simple implementation scheme benefits from the recent development of Pthreads library and Linux kernel — it is no longer a problem to efficiently create and manage a large number of threads. Therefore, the development of ABC Pascal is intended as an alternate implementation of action-based concurrency, which provides enhanced robustness.

As the name suggests, ABC Pascal is simple and particularly suitable for explaining essentials of concurrent programming. To reduce complexity of the implementation, ABC Pascal was specifically designed to be processed with a single-pass compiler. The compiler directly generates assembly code for Intel 32-bit platforms without optimizing the target code. In order to make performance comparisons, we implemented several classical concurrent programs in ABC Pascal, Java, Ada, and C/Pthreads respectively. The benchmark result shows that ABC Pascal is capable of achieving performance comparable to the other mainstream languages.

1.3 Structure of The Thesis

Chapter 2 provides a quick survey of commonly used concurrent programming constructs and features from some mainstream languages. The remainder of this thesis focuses on ABC Pascal. Chapter 3 covers the informal language specification. Chapter 4 covers the implementation details. The testing plans for the compiler are addressed in Chapter 5. The performance test result is outlined in Chapter 6. In Chapter 7, we discuss the future directions in which ABC Pascal might go, regarding to language specifications and implementation improvement.

Chapter 2

Related Work

Concurrency can be provided to programmers in the form of explicitly concurrent languages, compiler-supported extensions to traditional languages (e.g. OpenMP for C and Fortran), or library packages outside the language proper (e.g. Pthreads for C and C++). The latter two alternatives have been historically more common. Concurrency also appears in recent mainstream languages, such as Ada, Modula-3, Java, and C#.

With the popularity of Java and C#, this trend is beginning to change. However, it is likely to take some time before explicitly concurrent programming languages displace Fortran, C, and C++ for multithreaded applications.

In this chapter, we give a quick survey of common shared-variable concurrent programming constructs (semaphores and monitors), followed by multiprogramming in Ada and Java, the two mainstream languages equipped with, to some extent, similar concurrency features to ABC Pascal.

2.1 Scheduler-based Concurrency Mechanisms

Busy-wait synchronization in general is easy to implement. Condition synchronization can be realized by a simple busy-waiting loop, which a process enters and cycles until the condition holds. Mutual exclusion can be realized with spin locks, which usually requires atomic machine instruction such as `test_and_set`. A process trying to acquire the lock spins until the lock appears to be free:

```
while not test_and_set(Lock)
   /* do nothing */
```

The problem with busy-wait synchronization is that the blocked thread still consumes processor cycles, so that the processor is unavailable for other computations. To ensure acceptable performance, most concurrent programming languages employ scheduler-based synchronization mechanisms, which switch to a different thread when the current running thread is blocked. Semaphores and monitors are typical scheduler-based mechanisms in concurrent programming.

Semaphores were described by Dijkstra in the mid-1960s [5]. They are still used today, both in library packages and in languages. A semaphore is basically a non-negative counter with two associated operations, P() and V(). A thread calling P() waits until the counter to be positive and then atomically decrements the counter. A thread calling V() atomically increments the counter and wakes up one waiting thread, if any. Semaphores are widely considered to be "low-level" constructs for well-structured code, because in practice they suffer from one major problem – the use of a given semaphore tends to get scattered throughout a program, making it difficult to track them down for the purpose of software maintenance [17].

Monitors were also suggested by Dijkstra [6], and were developed more thoroughly by Hansen [10] and formalized by Hoare [11] in the early 1970's. They have been incorporated into a number of languages that have been influential.

A monitor is a module with operations (entries), internal variables, and condition variables. Only one operation of a given monitor is allowed to be active at a time; thus, operations of the same monitor are mutually exclusive. A thread calling a monitor operation must wait until the monitor becomes free, that is no other thread is currently executing an operation of this monitor. On behalf of its calling thread, a monitor operation may suspend itself by waiting on a condition variable. To resume execution of suspended threads, an operation may also signal a conditional variable, in which case one of the waiting threads might be resumed. Monitor signaling can follow either the signal-and-exit or signal-and-continue discipline. In the former, a signaling thread must leave the monitor immediately, at which point it is guaranteed that the signaled thread is the next one in the monitor. In the latter, the signaling thread retains control of the monitor.

One advantage monitors possess is that a monitor is an abstraction in which all operations on private data, including synchronization, are collected together in one place. Listing 2.1 shows a monitor-based solution to a bounded buffer, where signaling follows the signal-and-continue discipline. Operations `insert` and `remove` require exclusive access to the monitor's data.

Listing 2.1: Monitor-based code for a bounded buffer

```
monitor Bounded_buffer
  buf: array[1..SIZE] of data;
  next_full: integer := 1;
  next_empty: integer := 1;
  full_slot: integer := 0;
  is_full, sis_empty: condition;

  entry insert(d: data)
    begin
      while full_slots = SIZE do    {while-loop is needed; signals are only hints}
        wait(is_empty);
      buf[next_empty] := d;
      next_empty := (next_empty+1) mod SIZE;
      full_slots := full_slots+1;
      signal(is_full);
    end;

  entry remove
    begin
      while full_slots = 0 do
        wait(is_full);
      next_full := (next_full+1) mod SIZE;
      full_slots := full_slots - 1;
      signal(is_empty);
    end;
```

2.2 Ada Tasking

Ada supports concurrent programming through tasks. An Ada program may contain multiple tasks that execute concurrently. Each task represents a separate thread of control, which proceeds independently and concurrently between the points where it interacts with other tasks.

The first standardized Ada language, Ada 83, introduced the *rendezvous* mechanism for synchronization and communication of Ada tasks. Rendezvous is an example of synchronized message passing, suited to programming client/server interactions. One task, the server, declares a set of services that it is prepared to offer to other tasks, the clients. Such services are declared as entries in the task specifications. A rendezvous is requested by one task making an entry call on an entry of another task. During the time rendezvous is established, the calling task waits while the accepting task executes. A rendezvous is accepted by using Ada's `accept` keyword. When the accepting task ends the rendezvous, both tasks are freed to continue their execution.

Listing 2.2: A rendezvous example

```
-- Producer-Consumer using a buffer task.
procedure TaskPC is
  type Index is mod 8;
  type Buffer_Array is array(Index) of Integer;
  task Buffer is
    entry Append(I: in Integer);
    entry Take(I: out Integer);
  end Buffer;

  task body Buffer is
    B: Buffer_Array;
    Count: Index := 0;
  begin
    loop
      select
        when Count < Index'Last =>
        accept Append(I: in Integer) do
          B(Count) := I;
        end Append;
        Count := Count + 1;
      or
        when Count > 0 =>
        accept Take(I: out Integer) do
          I := B(Count);
        end Take;
        Count := Count - 1;
      end select;
    end loop;
  end Buffer;

  task type Producer;
  task body Producer is
  begin
    for N in 1..200 loop
      Buffer.Append(N);
    end loop;
  end Producer;

  task type Consumer(ID: Integer);
  task body Consumer is
    N: Integer;
  begin
    loop
      Buffer.Take(N);
    end loop;
  end Consumer;

  P1: Producer; C1: Consumer(1); C2: Consumer(2);
  begin
```

```
    null;
  end TaskPC;
```

Ada 95 enhances concurrency features of Ada 83 by providing protected types, which support synchronization based on shared data objects rather than a thread of control.

A protected type encapsulates shared data and synchronizes access to it. Each instance of a protected type is similar to a monitor. It has an interface that can contain functions, procedures, and entries, as shown in Listing 2.3.

Listing 2.3: Elements of protected object

```
protected type Signal_Object is
  entry Wait;
  procedure Signal;
  function Is_Open return Boolean;
  private Open : Boolean := False;
end Signal_Object;

protected body Signal_Object is
  entry Wait when Open is
    begin
      Open := False;
    end Wait;
  procedure Signal is
    begin
      Open := True;
    end Signal;
  function Is_Open return Boolean is
    begin
      return Open;
    end Is_Open;
end Signal_Object;
```

Protected functions provide concurrent read-only access to encapsulated data. A protected procedure provides mutually exclusive read/write access to the data encapsulated. Nevertheless, calls to a protected function are still executed mutually exclusive with calls to a protected procedure. A protected entry is similar to a protected procedure, except that it is guarded by a predicate, called a barrier. If the barrier evaluates to false when the entry call is made, the calling task is suspended until the barrier becomes true and no other task is currently active inside the protected object. Hence, it is common to use protected entry calls to implement condition synchronization.

In Listing 2.4, the buffer in the producer-consumer problem is now implemented with a protected object.

Listing 2.4: Buffer implemented with a protected object

```
protected Buffer is
  entry Append(I: in Integer);
  entry Take(I: out Integer);
  private
    B: Buffer_Array;
    Count: Index := 0;
end Buffer;
protected body Buffer is
  entry Append(I: in Integer) when Count < Index'Last is
    begin
      B(Count) := I;
      Count := Count + 1;
    end Append;
  entry Take(I: out Integer) when Count > 0 is
    begin
      I := B(Count);
      Count := Count - 1;
    end Take;
end Buffer;
```

When a call on a protected entry is executed, the barrier is evaluated first; if the barrier is closed (false), the calling task is queued. When the execution of protected procedure or entry is completed, all barriers of the queued entry calls are re-evaluated, potentially, entry bodies are executed.

The GNAT Implementation Notes

GNAT, an acronym for GNU NYU Ada Translator, is a front-end and runtime system for Ada 95. Ada's concurrency features are implemented on top of Pthreads. A large runtime library has been developed to support Ada tasking. The runtime library consists of three layers: GNU Ada Run-time Library (GNARL), GNU Low-level Library (GNULL), and Pthreads operations [14].

Each Ada task corresponds to a POSIX thread. The runtime library provides a set of routines which use Pthreads operations, so that the threads' behavior complies with Ada's semantics for tasks. The scheduling of the threads is directly under control of the Pthreads scheduler, and runtime stack allocation is under the control of the Pthread implementation. In many cases, the mapping between an Ada task and

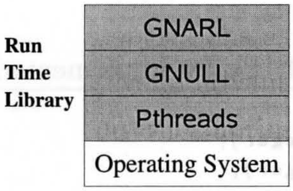

Figure 2.1: The GNAT Run Time Library

the corresponding POSIX thread is straightforward. However, the semantic differences are enough to disallow a direct feature-to-feature mapping. For example, the distinction between task creation and activation and the rules for task termination, are very different from their Pthreads counterparts, and must be implemented entirely by the Ada runtime system. These differences impose additional overhead on implementation of Ada tasking. Nevertheless, the implementers find it practical to implement Ada tasking using Pthreads operations as primitives. Implementing Ada tasking as a layer over Pthreads offers several advantages. First, it simplifies the job of the implementation. Secondly, building the runtime system on Pthreads enhances portability of the implementation. Therefore, the GNAT team chose Pthreads over using a runtime library specially designed to support Ada.

2.3 Java Threads

Java supports concurrent programming by means of threads, shared-variable synchronizations, message-passing, and remote procedure calls. In this section, we only give an introduction of Java threads, as well as synchronization and communication mechanisms for threads; the latter two concurrent programming techniques will not be discussed here.

In Java, threads are programmed by extending the `Thread` class or by implementing the `Runnable` interface. Like a monitor, each Java object has a lock associated to it. The per-object lock provides mutual exclusion for accessing the object's data. This lock is automatically acquired when a `synchronized` method of the object is invoked, and released if the method ever exits. Synchronized method body will not be executed until the object lock has been successfully acquired.

Conditional synchronization is provided by built-in constructs: `wait`, `notify`, and `notifyAll`. The semantics of these constructs are same as monitor's condition wait and signal operations. Note that they must be called within a synchronized block of the code, and hence can only be used when the object lock is obtained by the calling thread. The `wait` method releases the object lock and suspends the calling thread by putting the caller in the waiting queue of the object [9]. The `notify` method awakens the first thread in the waiting queue, if there is one. The awakened thread is moved from the waiting queue to ready queue, meaning that it is now ready to be scheduled for execution. The thread that calls `notify` continues to hold the object lock. Hence, in Java, `notify` has a signal-and-continue semantics. A call to `notifyAll` broadcasts the signal to awaken all threads in the waiting queue.

Listing 2.5: Producer-Consumer implemented with Java threads

```
class Buffer {
  static final int MAX = 8;
  private int[] B = new int[MAX];
  private int count;
  public synchronized void Append(int i) {
    while (count == MAX - 1) {
      try {wait();
      } catch (InterruptedException e) {}
    }
    B[count] = i;
    count++;
    notify();
  }
  public synchronized int Take() {
    int temp;
    while (count == 0) {
      try {wait();
      } catch (InterruptedException e) {}
    }
    temp = B[count];
    count--;
    notify();
    return(temp);
   }
  public Buffer(){
```

```
    count = 0;
  }
}
class Producer extends Thread{
    private Buffer buf;
    public Producer(Buffer b){
      this.buf = b;
    }
    public void run() {
    for (int r = 0; r < 200; r++) {
      buf.Append(r);
    }
   }
}
class Consumer extends Thread{
    private Buffer buf;
    public Consumer(Buffer b){
      this.buf = b;
    }
    public void run() {
    int x;
    for (int r = 0; r < 200; r++) {
      x = buf.Take();
    }
   }
}
public class PC{
  public static void main(String[] args) {
    Buffer b = new Buffer();
    Consumer cx = new Consumer(b);
    Producer px = new Producer(b);
    px.start(); cx.start();
  }
}
```

Implementation of Java threads largely depends on the Java Virtual Machine, and its performance is influenced by the underlying operating system. Implementations are historically either user-level "green threads" or native threads provided by the OS.

Green threads are simulated threads within the virtual machine. The green thread library is a pure user-level implementation of threads. They exist only at user-level and are not mapped to kernel threads. That means, when a multithreaded Java program is running on multiprocessor machines, it is impossible to take advantage of the multiple processors. Therefore, green threads are no longer used and replaced by native threads since Java 1.2.

Native threads are directly provided by the native OS. The virtual machine (VM) performs bytecode interpretation using native threads. Native threads can improve performance by allowing real parallelism on multiprocessors. The table below shows the common default native threads enviroment on different OS's.

Solaris	Solaris Threads/Pthreads
Linux	Pthreads
Windows	win-32 Threads

Thus, the native OS threading model and environment influences Java application's performance. On many OS's, the native thread library keeps a simple 1-on-1 thread mapping between Java threads and the kernel threads; others still adopt an M-on-N mapping.

2.4 The Seuss Model

Jayadev Misra developed Seuss, a new action-based model to design distributed applications.

A key observation leading of the research on Seuss is that sizable concurrent programs consist of sequential programs. Design, verification, and implementation of the latter has been well studied and understood. The major goal is to simplify concurrent programming, by separating the concern of concurrency from the core program design. The programming model is minimal; no specific communication or synchronization mechanism, except procedure calls, are built in. Seuss disallows interference among actions. With this restriction, the execution of a multithreaded program can be understood as a single thread of control. As a consequence, it is possible to reason about a multithreaded program from a single execution thread.

The Seuss model consists of three components: box, clone, and procedure. A box definition is similar in many ways to a class definition in object-oriented languages. Clones are just instances of boxes. A box definition may contain procedures, which can either be actions or methods. An action runs autonomously and infinitely often during program execution, whereas a method may only be invoked by other procedures.

In Seuss, two kinds of procedures are distinguished: total and partial. The former corresponds to conventional procedures in sequential programming, and the latter introduces the notion of accepting and rejecting a procedure call. A call to a total procedure will always succeed; yet, a call to a partial procedure might be rejected and leave program state unchanged. Detailed description on Seuss components can be found in [15].

Listing 2.6: Seuss example - mutual exclusion

```
program MutualExclusion
  box Semaphore
    integer v:=1;  {the semaphore value is initially set to 1}
    partial method P:: v>0 -> v:=v-1
    total method V:: v:=v+1
  end{Semaphore}

  clone s,t: Semaphore;

  box user
    boolean hs:=false;
    boolean ht:=false;
    partial action s_acquire :: not hs; s.P -> hs:=true
    partial action t_acquire :: not ht; t.P -> ht:=true
    partial action execute:: hs and ht ->
                 {critical section};
                 s.V;  t.V;  hs:=false;  ht:=false
  end{user}

  clone ux, uy : user
```

A Seuss program consists of a set of clones that in turn may contain arbitrary number of actions or methods. Listing 2.6 shows an example of a Seuss program containg two clones `ux`, `uy` of box type `user`, and two clones `s,t` of box type `semaphore`. The body of an action can run only if the `user` has acquired both semaphores; thus, mutual exclusion is guaranteed. The execution of a Seuss program is conceptually simple: every action is executed infinitely often. This execution rule defines a logical view of the execution; in an implementation, the program execution may be stopped once a final state has been reached.

The major advantage of Seuss is that programmers can reason about a Seuss program as if it is being executed in a single thread of control; this simplifies the correctness proof for multithreaded programs.

Some concrete programming languages based on the Seuss model have also been attempted. For example, I.K. Krüger implemented SeussCpp [12]. As the name indicates, the sequential parts described in Seuss are written in C++. To provide a flexible basis for future changes in SeussCpp's language specification and target architecture, the compiler generates C++ code and uses PVM [8] as the basis of the runtime system. Since PVM provides interfaces for program development over computer networks, SeussCpp is intended to run on a network of computers. Its synchronization mechanism is based on message-passing. For each clone declaration, there is an instance process , which is in turn implemented with a PVM process. Communication among instance processes is realized by sending and receiving requests over the network; execution of a procedure body described in Seuss is implemented with Remote Procedure Calls.

2.5 The Thread Library

A process is an operating system abstraction that allows one computer system to support many units of execution. Each process typically represents a separate running program. A thread is an independent stream of instructions that can be scheduled to execute by the operating system. Threads and processes are closely related. A thread exists within a process and uses this process's resource, while a process can have multiple threads each of which has its own independent flow of control.

One common way to write concurrent programs is to use thread libraries or packages with a sequential language. A typical thread library provides a collection of API's (usually C routines) for thread creation and management. These API's are sufficient for one to develop concurrent programs in langauges supported by the thread packages.

This section give a overview of three common thread libraries available on some Unix-based systems.

2.5.1 Pthreads

The name Pthreads is the acronym for POSIX Threads, based on the IEEE POSIX 1003.1c standard. This standard defines the application programming interfaces for developing multithreaded program in POSIX environment. The POSIX 1003.1c standards was finalized in 1995 based on its draft 10 specification. The Pthreads library defines a set of standardized C routines for creating and managing threads. It is now widely available on various flavors of Unix-based systems, because the standard is created to provide a portable programming interface for writing concurrent programs across different operating systems. It is expected that Pthreads will be the common standards for multi-programming on UNIX systems.

It is worth to stress that Pthreads is a set of specifications for multi-programming. The implementation of Pthreads library varies on different operating systems. That means a program using Pthreads will behave exactly the same across different operating systems, but the performance will probably vary.

Commonly used Pthreads routines

The Pthreads API is used to implement ABC Pascal's action-based concurrency. This section provides a listing of some Pthreads library routines commonly used for managing and synchronizing threads.

- pthread_attr_init()

  ```
  int pthread_attr_init(
  pthread_attr_t *attr);
  ```

  Initializes a thread attribute object.

- pthread_attr_setdetachstate()

  ```
  int pthread_attr_setdetachstate(
  pthread_attr_t *attr, int detachstate);
  ```

  Adjusts the detached state of a thread. A thread's `detachstate` can be joinable or detached. If the programmer knows in advance that the thread will never need to join with another thread, the programmer can consider creating it in a detached state. If a thread requires joining, one may explicitly creating it as joinable.

- pthread_cond_init()

  ```
  int pthread_cond_wait(
  pthread_cond_t *cond, const pthread_condattr_t *attr);
  ```

Initializes a condition variable with the attributes specified in the condition variable attribute object, `attr`. If `attr` is NULL, the default attributes are used.

- pthread_cond_wait()

  ```
  int pthread_cond_wait(
  pthread_cond_t *cond, pthread_mutex_t *mutex);
  ```

  Atomically unlocks the specified mutex, and places the calling thread into a waiting state.

- pthread_cond_broadcast()

  ```
  int pthread_cond_broadcast(
  pthread_cond_t *cond);
  ```

  Unblocks all threads that are waiting on a condition variable, `cond`.

- pthread_cond_signal()

  ```
  int pthread_cond_signal(
  pthread_cond_t *cond);
  ```

  Unblocks one thread waiting on a condition variable, if there is any.

- pthread_mutex_init()

  ```
  int pthread_mutex_init(
  pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
  ```

  Initializes a `mutex` with attributes specified in mutex attribute object `attr`. If `attr` is NULL, the default attributes are used.

- pthread_mutex_lock()

  ```
  int pthread_mutex_lock(
  pthread_mutex_t *mutex);
  ```

  Locks an unlocked `mutex`. If the `mutex` is already locked, the calling thread blocks.

- pthread_mutex_unlock()

```
int pthread_mutex_unlock(
pthread_mutex_t *mutex);
```

Unlocks a mutex.

- pthread_create()

  ```
  int pthread_create(
  pthread_t *thread, const pthread_attr_t *attr,
  void *(*start_routine)(void *), void *arg);
  ```

  Creates a thread with the attributes specified in `attr`. The **`thread`** argument receives a thread handle for the new thread. The new thread starts execution in `start_routine` and is passed the single specified argument, **`arg`**.

- pthread_join()

  ```
  int pthread_join(
  pthread_t thread, void **value_ptr);
  ```

  Causes the calling thread to wait for the specified thread's termination. The `value_ptr` parameter receives the return value of the terminating thread.

NPTL

Native POSIX Thread Library (NPTL) today is the standard POSIX thread library on Linux. Since we developed ABC Pascal on Linux, the Pthreads library we are using is NPTL by default. NPTL is a recent implementation of Linux threading that conforms to the POSIX specification [7]. This new Pthreads library, shipped as a component of the GNU C library, offers significant improvement in terms of performance and stability.

NPTL is implemented with 1-on-1 model. That means each user-level thread maps to an underlying kernel thread. Hence, the whole library is a relative thin layer on top of kernel functions. The development of NPTL benefits from the improvement of Linux kernel, such as the *clone()* and *futex()* system call. These changes were specifically made to ensure a decent thread library implementation for Linux. Performance measurements on thread creation and lock contention demonstrated NPTL's advantages. Tests also showed that NPTL is able to support large number of user-level threads, for example, 100,000 threads on Intel architecture. Thus, we can worry less about the limitation on the number of threads that can be created within a process.

2.5.2 UI Threads

Another thread standard that is popular on UNIX systems is based on UNIX International specification. Thus, this interface is also referred to as UI Threads. Operating systems such as Solaris 2.X and Unixware support the UI threads interface. The UI Threads interface is very similar to Pthreads, so one should use Pthreads instead of UI threads if portability becomes a concern. Solaris Threads is a subset of the UI Threads interface.

2.5.3 C-Threads

Mac OS X provides a high-level interface for developing multithreaded applications called C-Threads. The C-Threads APIs are broadly classified into thread management, thread synchronization, thread-specific data, scheduling and priority. In fact, the legacy of Pthread lies in C-Threads. Therefore, C-Threads are very similar to Pthreads, except that Pthread have been extended with new features. On the Mac OS X, both Pthreads and C-Threads are provided as a part of the operating system.

Chapter 3

ABC Pascal: The Language

As a derivation of Pascal0 [19], ABC Pascal inherits the basic facilities of Pascal to support sequential programming. It further extends Pascal0 with support for action-based concurrency. In this chapter, we introduce the language, followed by examples developed to solve real problems.

3.1 An Overview of ABC Pascal

ABC Pascal inherits the basic sequential programming features from Pascal, and it extends these features by including support for action-based concurrency. The basic issues of concurrent programming, such as exclusive access to resources, deadlock, waiting on conditions, and signaling, will be revisited.

With ABC Pascal, the methodology to design concurrent programs is simple: represent each programming entity by an object and have the objects communicate by calling each other's methods. Because it is intended to run on shared-memory multiprocessor machines, synchronization in ABC Pascal is achieved through shared-variables. More specifically, actions are synchronized by objects that are shared among actions. If an object contains no actions, it is referred to as *passive*; otherwise, it is called an *active* object. An action runs autonomously, so concurrency is realized by having multiple actions that are eligible to run simultaneously. Like Pascal, an ABC Pascal program has a main body. Code enclosed in the main body runs independently from the actions, if any.

Listing 3.1 shows a concrete example that solves the dining philosophers problem. We will go through this example to briefly describe the unconventional concepts in ABC Pascal that are not found in commonly used high-level languages.

Listing 3.1: The dining philosophers

```
program DP;
  const ROUNDS = 1000; SEATS = 5;
  class Fork
    var up: boolean;
    procedure pickup when not up;
      begin
        up := true
      end;
    procedure putdown;
      begin
        up := false
      end;
  begin {initialization block}
    up := false
  end;

  class Host
    var occupants: integer;
    procedure enter_sitdown when occupants < SEATS - 1;
      begin
        occupants := occupants + 1
      end;
    procedure getup_leave;
      begin
        occupants := occupants - 1
      end;
  begin {initialization block}
    occupants := 0
  end;

  var butler: Host;
      F: array [0..SEATS - 1] of Fork;

  class Philosopher
    var seat: integer; awake: boolean;
    procedure wakeup(s: integer) ;
      begin
        seat := s; awake := true
      end;
    action start when awake;
    var r: integer;
      begin
        r := 0;
        while r < ROUNDS do
          begin
            butler.enter_sitdown;
```

```
          F[(seat + 1) mod SEATS].pickup;
          F[seat].pickup;
          r := r + 1;
          F[(seat + 1) mod SEATS].putdown;
          F[seat].putdown;
          butler.getup_leave
        end;
      awake := false
    end;
begin {initialization block}
  awake := false
end;

var P: array [0..SEATS - 1] of Philosopher;  s: integer;
begin {main body}
  s := 0;
  while s < SEATS do
    begin P[s].wakeup(s); s := s + 1 end
end.
```

In Listing 3.1, each *philosopher* is represented by an object. The philosopher object contains two fields `seat` and `awake`, one procedure `wakeup`, and one action `start`. The action defined within each philosopher object runs autonomously in parallel with the program main body, as the program starts its execution. The *forks* and the *host* are represented by objects in a similar way, with the exception that they are passive objects since they contain no actions. Thus, this simple ABC Pascal program has six threads of control — five threads for the philosophers plus one for the program main body. Synchronization is ensured by mutual exclusion among actions and procedures from the same object, and by the optional guarded expression in an action or procedure declaration. In addition to fields, actions, procedures, an object has an initialization block that marks the end of the class definition.

An object may contain multiple methods, which can either be procedures or actions. Within an object, only one method is eligible to run at a time; no concurrent execution is allowed among methods from the same object. Yet, a method may be guarded with a predicate. This predicate, also known as the *guard*, is evaluated first, before the body of method is executed. The method's body is executed only if the guard is true; otherwise, the current thread is blocked on the condition imposed by the guard, until the guard becomes true in the future.

This dining philosophers example also reveals another essential concept of ABC Pascal: built-in concurrent programming constructs are absent. Listing 3.2 shows a partial implementation of dining philosophers problem in Java – the class definition for `Fork`. In class `Fork`, `synchronized` is a built-in modifier; moreover, `wait` and `notifyAll` are built-in constructs to provide condition synchronization.

Listing 3.2: The 'Fork' class

```
class Fork {
  private boolean up = false;
  synchronized void pickup() {
    while (up) {
      try{wait();}
      catch (InterruptedException e) {}
    }
    up = true;
  }
  synchronized void putdown() {
    up = false;
    notifyAll();
  }
}
```

In this thesis, we argue and demonstrate that explicit concurrent construct may be excluded from multiprogram development.

3.2 The Grammar

The concrete grammar of ABC Pascal is listed below. All reserved words are presented in boldface. ABC Pascal is syntactically case sensitive.

Lexical Elements of ABC Pascal

ident ::= letter { letter | digit | "_" }
integer ::= digit { digit }

Grammar for ABC Pascal

selector ::= { "." *ident* | "[" *Expression* "]" }
factor ::= *ident selector* | *integer* | "(" *Expression* ")" | **not** *factor*
term ::= *factor* { ("*" | **div** | **mod** | **and**) *factor*}

SimpleExpression ::= ["+" | " − "] *term* {(" + " | " − " | **or**) *term*}
Expression ::= *SimpleExpression* [(" = " | " <> " | " < " | " <= " |
    " > " | " >= ") *SimpleExpression*]

assignment ::= *ident selector* " := " *Expression*
ActualParameters ::= "(" [*Expression* {"," *Expression*}] ")"
ProcedureCall ::= *ident selector* [*ActualParameters*]
CompoundStatement ::= **begin** *statement* { ";" *statement* } **end**
IfStatement ::= **if** *Expression* **then** *Statement* [**else** *Statement*]
WhileStatement ::= **while** *Expression* **do** *Statement*
Statement ::= [*assignment* | *ProcedureCall* |
    CompoundStatement | *IfStatement* | *WhileStatement*]
IdentList ::= *ident* { "," *ident* }
ArrayType ::= **array** "[" *Expression* ".." *Expression* "]" **of** *type*
FieldList ::= [*IdentList* " : " *type*]
RecordType ::= **record** *FieldList* { ";" *FieldList* } **end**
type ::= *ident* | *ArrayType* | *RecordType*
FPSection ::= [**var**] *IdentList* " : " *type*
FormalParameters ::= "(" [*FPSection* { ";" *FPSection* }] ")"
guard ::= [**when** *Expression*]
ProcedureDeclaration ::= **procedure** *ident* [*FormalParameters*]
    guard ";" *declarations CompoundStatement*
ActionDeclaration ::= **action** *ident guard* ";"
    declarations CompoundStatement
ClassDeclaration ::= **class** *ident* [**var** { *IdentList* " : " *type* ";" }]
    { *ProcedureDeclaration* ";" } { *ActionDeclaration* ";" }
    CompoundStatement ";"
declarations ::= [**const** { *ident* " = " *Expression* ";" }]
    [**type** { *ident* " = " *type* ";" }]
    { *ClassDeclaration* ";" | **var** { *IdentList* " : " *type* ";" } | { *ProcedureDeclaration* ";" } }

program ::= **program** *ident* ";" *declarations CompoundStatement*

Besides the constructs inherited from Pascal0, the only three syntactic constructs added are:

- *guard*
- *ActionDeclaration*
- *ClassDeclaration*

The semantics of these added features will be explained with examples in the later sections.

Within declarations, constant and type definition, if any, must appear prior to the class definitions, variable declarations, or procedure declarations. However, classes, variables, and procedures may be declared in any order.

Not shown in the grammar listing, programming comments in ABC Pascal are marked by a pair of curly brackets, {... *comments*... }.

3.3 The Program Structure

With ABC Pascal, a concurrent program consists of a set of objects that in turn may contain arbitrary number of methods. It orchestrates the execution of all actions, which are sequential programs, by specifying the conditions under which each action is enabled for execution.

The widely-used concurrent programming construct – waiting, signaling, and mutual exclusion – are absent from the language. The built-in primitives are powerful enough to encode communication and synchronization protocols in a compact way.

The Basic Structure and Scope Rules

Programming in ABC Pascal is simple. An ABC Pascal program consists of two sections:

- declaration section
- program main body

```
program X;
 {declaration section}
     constant declarations
     type definitions
     procedure, class and variable declarations
begin
 {program main body}
end.
```

In ABC Pascal, every identifier must be declared before its first use. ABC Pascal thus disallows forward reference. The purpose of enforcing this rule is to make it easier to implement ABC Pascal with a single-pass compiler.

Constants are accessible anywhere in the program. Global procedures are visible to the program main body and classes, and they are not allowed to have a guard, behaving exactly the same as a procedure in Pascal. All global variables are visible to the program main body. Global variables can be accessed anywhere in the program.

Class fields are only accessible within the containing class. As for a guarded procedure, parameters of the procedure may appear in the guard. However, a method's guard cannot contain global variables of non-class type. This is because accessing global variables of non-class type is not synchronized.

The program main body contains sequential code that can run in parallel with all actions, if there are any.

Types and Values

ABC Pascal is statically typed, meaning that every variable and expression has a type that is known at compile time, and it prohibits the application of any operation to any object that is not intended. The types of ABC Pascal programming language are divided into two categories: primitive types and structured types.

The primitive types are `integer` and `boolean`. The structured types are array types, record types, and class types. The value of a primitive type always holds a value of that exact type, while a variable of structured type is a pointer that points to an array, a record, or an object. Different from most object-oriented languages, an object in ABC Pascal is a statically created instance of a class type when it is declared. Dynamic object creation is not supported.

Only variables of primitive types can appear in assignment statements. It is an illegal operation to assign a variable of a structured type to another variable.

Variables of primitive types can either be passed as value parameters or reference parameters. Variables of structured types can only be passed as reference parameters.

Standard Procedures

Four standard procedures are predefined in ABC Pascal to provide basic I/O and randomization of integer.

- **read(var x: integer)**
  Read an integer from the standard input, and store the value in variable x.
- **write(x: integer)**
  Write the value of variable x to standard output.
- **writeln**
  Take no argument; write the end-of-line character to standard output.
- **random(var x: integer)**
  Generate a random positive integer, and store the value in variable x.

Class Definition

A class definition declares a name for a new class and provides description of its field declarations, method definitions, and initialization block.

```
class X
 {... variable declarations ...}
 {... method definitions ...}
begin
  {... initialization block...}
end;
```

Within the scope of a class definition, global variables of non-class types can be used freely. However, as mentioned earlier, these global variables are unprotected because accessing them is not synchronized. Hence, bad race conditions are still possible if variables of non-class type are shared among objects.

The current language specification prohibits actions to call a procedure of the same object.

The initialization block serves the purpose to initialize the object. All initialization blocks are set to execute autonomously before any action or the program main body start running.

Method Definition

Methods can either be procedures or actions. Procedures may have optional parameters, yet actions are parameterless.

Methods can have an optional guard, a predicate. The guard ensures that the body of the method will not be executed until its guard becomes true. Methods can have optional local variables, with the restriction that local variables cannot be class type.

```
class Account
  var c: integer;
  action transfer when c>1000;
    begin
    { body of the action }
    end;
  procedure withdraw when c>0;
    begin
    { body of the procedure }
    end;
  procedure deposit;
    begin
    { body of the procedure }
    end;
begin
  c:=0;
end;
```

For example, class `Account` contains three methods: `transfer`, `withdraw`, and `deposit`. Action `transfer` will only start its execution when condition $c > 1000$ is satisfied. The caller of procedure `withdraw` will be blocked until condition $c > 0$ is satisfied. The caller of procedure `deposit` never blocks.

Each object continuously selects actions to execute. The choice made by an object is nondeterministic. The nondeterminacy follows the weak fairness condition, which implies no guard that is always true is always skipped. If no action is eligible to execute, the program enters its final state and terminates immediately.

3.4 Essentials for Multiprogramming

ABC Pascal supports concurrent programming by providing mechanisms for shared-variable synchronization. Methods of the same object are executed in mutual exclusion, ensuring the execution of each method is atomic. Communication among the objects and the main body are realized through procedure calls. In the dining philosophers example (Listing 3.1), it is clear to see how the main body and the objects "communicate" with each other:

- The main body interacts with each `Philosopher` by calling `wakeup`.
- A `Philosopher` interacts with the forks by calling `pickup` and `putdown`.
- A `Philosopher` interacts with the `butler` by calling `enter_sitdown` and `getup_leave`.

Condition synchronization is realized by the guards. For a guarded procedure, the caller that invokes this procedure is blocked until the guard evaluates to true. For a guarded action, the execution of the action body will not start until the guard evaluates to true. When a method `M` of object `OA` exits, it will unblock all actions that were waiting on some conditions (imposed by the guards) to access `OA`. An unblocked action will have to re-check the condition, and can only resume its execution if the condition holds; otherwise, the action is blocked again on the condition.

3.5 Program Execution in a Multithreaded Context

By declaring active objects, an ABC Pascal program may have multiple threads of control. As for program execution, the actions and the program main body are set to run concurrently and interact with each other, but they are not necessarily running in parallel because of synchronization conditions.

Two actions are eligible to run concurrently if no object is shared by both actions. It is obvious that actions of the same object cannot be executed concurrently. As for actions belonging to different objects, if they share the same object but do not access the shared object simultaneously, they are also eligible to run concurrently.

Listing 3.3: Producer and Consumer

```
program PAC;
  const MAXSIZE = 10; MAXROUND = 5;
  class Buffer
    var b: array [0..MAXSIZE-1] of integer;
```

```
      in, out, n, max: integer;
    procedure put (x: integer) when n < MAXSIZE;
      begin b[in] := x; in := (in + 1) mod S; n := n + 1 end;
    procedure get (var x: integer) when n > 0;
      begin x := b[out]; out := (out + 1) mod S; n := n - 1 end;
    begin
      in := 0; out := 0; n := 0
    end;

  var ch: Buffer;
  class Producer
    var x: integer;
    action produce when x < MAXROUND;
      begin x := x + 1; write(x); ch.put(x) end;
    begin x := 0 end;

  class Consumer
    var r: integer;
    action consume when r < 2 * MAXROUND;
      var y: integer;
      begin ch.get(y); write(y); r := r+1 end;
    begin r := 0 end;

  var p1, p2: Producer; c1: Consumer;
  begin end.
```

We here use a simplified version of Producer/Consumer program to explain the rule of program execution in ABC Pascal. As program execution starts, the initialization blocks of all objects is set to run first. That is initialization block of object `ch`, `p1`, p2, and `c1` will be executed first. Following the initialization phase, the actions along with the program main body will start concurrently. In particular, action `produce` of `p1` and `p2`, action `consume` from `c1`, and the main body will start concurrently.

The terminating condition for a program's execution is:

- there are no actions eligible to execute, that is all actions are blocked.
- the program main body has completed its execution.

Both conditions must be satisfied in order to terminate an ABC Pascal program. In the example `PAC.pas`, if we remove the guards from action `produce` and action `consume`, both actions' body will be executed infinitely often, thus, the program becomes nonterminating. The guards attributed to the methods play two critical roles:

1. specifying the conditions to execute the method body.

2. specifying the terminating conditions of the program, if the program is meant to terminate.

3.6 ABC Pascal Examples

This section presents several ABC Pascal examples that solve concurrent programming problems. We are trying to explore the expressiveness of ABC Pascal.

3.6.1 One-lane Car Control

CC.pas simulates a car control system for a one-lane bridge. Cars coming from two opposite directions arrive at a one-lane bridge. Cars heading in the opposite direction cannot cross the bridge at the same time. Cars heading the same direction can cross the bridge together, but with a restriction on capacity, which specifies the maximum number of cars allowed on the bridge. No preference is given to cars traveling in either direction.

Listing 3.4: CC.pas

```
program CC;

const
  ROUNDS = 5000;
  CARS_S2N = 250;
  CARS_N2S = 350;
  CAPACITY = 10;

class Bridge
  var s2n, n2s: integer;
  procedure s2n_arrive when (n2s = 0) and (s2n < CAPACITY);
    begin
      s2n := s2n + 1
    end;
  procedure s2n_leave;
    begin
      s2n := s2n - 1
    end;
  procedure n2s_arrive when (s2n = 0) and (s2n < CAPACITY);
    begin
      n2s := n2s + 1
    end;
  procedure n2s_leave;
    begin
      n2s := n2s - 1
```

```
    end;
begin
  s2n := 0;
  n2s := 0
end;

var brg: Bridge;

class Car_s2n
  var round: integer;
  action cross_bridge when round < ROUNDS;
    begin
      brg.s2n_arrive;
      round := round + 1;
      brg.s2n_leave
    end;
begin
  round := 0
end;

class Car_n2s
  var round: integer;
  action pass_bridge when round < ROUNDS;
    begin
      brg.n2s_arrive;
      round := round + 1;
      brg.n2s_leave
    end;
begin
  round := 0;
end;

var cars_to_north: array [1..CARS_S2N] of Car_s2n;
    cars_to_south: array [1..CARS_N2S] of Car_n2s;
begin
 { empty main body}
end.
```

3.6.2 Dining Philosophers

Five philosophers sit around a circular table. In the center of the table is a large plate of spaghetti. A philosopher needs two forks to eat the spaghetti. However, there are only five forks placed on the table – one fork is placed between each pair of philosophers. They all agree to use only the forks to the immediate left and right. To prevent possible deadlock situations, a butler will enforce that at most four philosophers are allowed to be seated.

Listing 3.5: DP.pas

```
program DP;

const
  ROUNDS = 100000;
  SEATS = 5;

class Fork
  var up: boolean;
  procedure pickup when not up;
    begin
      up := true
    end;
  procedure putdown;
    begin
      up := false
    end;
begin
  up := false
end;

class Host
  var occupants: integer;
  procedure enter_sitdown when occupants < SEATS - 1;
    begin
      occupants := occupants + 1
    end;
  procedure getup_leave;
    begin
      occupants := occupants - 1
    end;
begin
  occupants := 0
end;

var butler: Host;
    F: array [0..SEATS - 1] of Fork;

class Philosopher
  var seat: integer;
       awake: boolean;
  procedure wakeup(s: integer);
    begin
      seat := s; awake := true
    end;
  action start when awake;
    var r: integer;
    begin
      r := 0;
      while r < ROUNDS do
        begin
          butler.enter_sitdown;
```

```
          F[(seat + 1) mod SEATS].pickup;
          F[seat].pickup;
          F[(seat + 1) mod SEATS].putdown;
          F[seat].putdown;
          butler.getup_leave;
          r := r + 1
        end;
      awake := false
    end;
begin
  awake := false
end;

var P: array [0..SEATS - 1] of Philosopher;
    s: integer;

begin
  s := 0;
  while s < SEATS do
    begin P[s].wakeup(s); s := s + 1 end
end.
```

3.6.3 Multiple Resource Allocator

There are a set of resources and a set of users. The problem is each user needs to consume a subset of these resources exclusively. A user cannot consume any resource until it successfully acquires all needed resources. After consuming the required resources, a user releases all resources it holds.

Listing 3.6: MRA.pas

```
program MRA;

const
  ROUNDS = 5000;
  RESOURCES = 10;
  USERS = 500;
  RES_PER_USER = 4;

class Resource
  var avail: boolean;
  procedure acquire when avail;
    begin
      avail := false
    end;
  procedure release;
    begin
      avail := true
    end;
```

```
begin
  avail := true
end;

var R: array [0..RESOURCES - 1] of Resource;

class User
  var round: integer;
      needs: array [0..RESOURCES - 1] of boolean;
      d: integer; {idle: -1; acquiring: 0 .. RESOURCES - 1; acquired: RESOURCES}

  action request_resources when (d = -1) and (round < ROUNDS);
    var x, c: integer;
    begin {assign needs[i] randomly}
      c := 0;
      while c < RES_PER_USER do
        begin
          random(x);
          needs[x mod RESOURCES] := true;
          c := c + 1
        end;
      d := 0
    end;
  action acquire_one_resource when (d > -1) and (d < RESOURCES) and (round < ROUNDS);
    begin {acquire resource d if needed}
      if needs[d] then R[d].acquire;
      d := d + 1
    end;
  action release_resources when (d = RESOURCES) and (round < ROUNDS);
    begin
      while d > 0 do
        begin
          d := d - 1;
          if needs[d] then
            begin
              R[d].release;
              needs[d] := false;
            end;
        end;
      d := -1;
      round := round + 1
    end;
begin
  d := 0;
  while d < RESOURCES do
    begin needs[d] := false; d := d + 1 end;
  round := 0;
  d := 0
end;

var U: array [1..USERS] of User;

begin
```

```
end.
```

3.6.4 Producer/Consumer

This program simulates the interaction between producers and consumers. A producer process produces data and appends data to the shared buffer. A consumer process consumes and removes one element from the shared buffer.

Listing 3.7: PC.pas

```
program PC;
  const  MAXSIZE = 10; MAXROUND = 5;
  class Buffer
    var b: array [0..MAXSIZE-1] of integer;
        in, out, n, max: integer;
    procedure put (x: integer) when n < MAXSIZE;
      begin b[in] := x; in := (in + 1) mod S; n := n + 1 end;
    procedure get (var x: integer) when n > 0;
      begin x := b[out]; out := (out + 1) mod S; n := n - 1 end;
    begin
      in := 0; out := 0; n := 0
    end;

  var ch: Buffer;
  class Producer
    var x: integer;
    action produce when x < MAXROUND;
      begin x := x + 1; write(x); ch.put(x) end;
    begin x := 0 end;

  class Consumer
    var r: integer;
    action consume when r < 2 * MAXROUND;
      var y: integer;
      begin ch.get(y); write(y); r := r+1 end;
    begin r:=0; end;

  var p1, p2: Producer; c1: Consumer;
  begin end.
```

3.6.5 Reader/Writer

Readers and writers can access a shared database. The rule is that multiple readers may access the database concurrently, but only one writer may access the database at a time. This program allows either concurrent read or an exclusive write.

Listing 3.8: RW.pas

```
program RW;

const
  ROUNDS = 5000;
  RD = 350;
  WR = 250;

class RW_arbiter
  var rw: integer; {-1: one writer; 0: idle; > 0: #readers}
  procedure start_read when rw >= 0;
    begin
      rw := rw + 1
    end;
  procedure end_read;
    begin
      rw := rw - 1
    end;
  procedure start_write when rw = 0;
    begin
      rw := -1
    end;
  procedure end_write;
    begin
      rw := 0
    end;
begin
  rw := 0
end;

var rwa: RW_arbiter;

class Reader
  var round: integer;
  action read_cycle when round < ROUNDS;
    begin
      rwa.start_read;
      {read access}
      rwa.end_read;
      round := round + 1
    end;
begin
  round := 0
end;

class Writer
  var round: integer;
  action write_cycle when round < ROUNDS;
    begin
      rwa.start_write;
      {write access}
      rwa.end_write;
```

```
      round := round + 1
    end;
begin
  round := 0
end;

var
  readers: array [1..RD] of Reader;
  writers: array [1..WR] of Writer;

begin
end.
```

3.6.6 A Sorting Network

The idea behind a sorting network is to repeatedly – in parallel – merge two sorted lists into a longer sorted list. Assume that the ends of the input channels are marked by a sentinel `EOS`. Each merger has three channels: two as input channels, one as output channel. Each `merger` receives values from two sorted input channels and produces one sorted output channel. The program requires the number of input values to be a power of 2, so that the resulting communication pattern forms a full binary tree. Figure 3.1 shows the buffers and mergers in a sorting network, where n represents the total number of input.

Listing 3.9: SORT.pas

```
program SORT;

const CAPACITY = 8; EOS = -1; INPUTS = 1024; {power of 2}

class Buffer
  var b: array [0 .. CAPACITY - 1] of integer;
      in, out, n: integer;
  procedure put(x: integer) when n < CAPACITY;
    begin b[in] := x; in := (in + 1) mod CAPACITY; n := n + 1 end;
  procedure get(var x: integer) when n > 0;
    begin x := b[out]; out := (out + 1) mod CAPACITY; n := n - 1 end;
  begin in := 0; out := 0; n := 0 end;

var ch: array [1 .. INPUTS * 2 - 1] of Buffer;

class Merger
  var in1, in2, out: integer; {indices of input & output channels}
       ready: boolean;      {merger configured}
  procedure configure(i1, i2, o: integer);
    begin in1 := i1; in2 := i2; out := o; ready := true end;
```

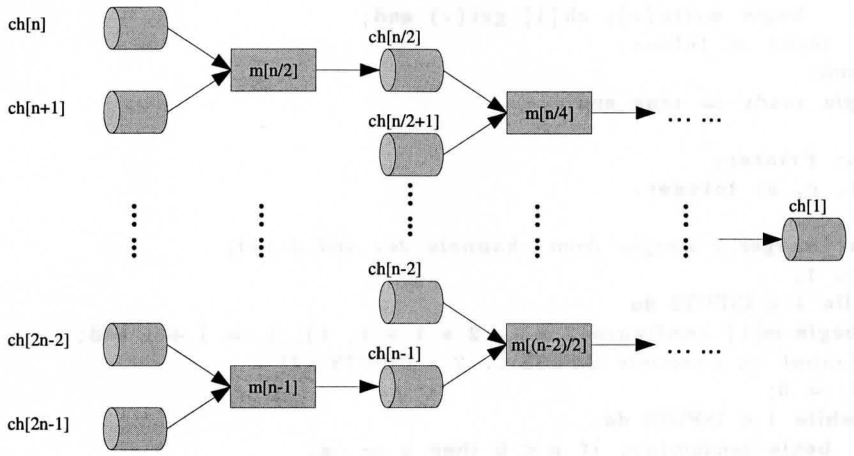

Figure 3.1: Buffers and mergers in the Sorting Network

```
  action merge when ready;
    var v1, v2: integer; {values to be compared}
    begin
      ch[in1].get(v1); ch[in2].get(v2);
      while (v1 <> EOS) and (v2 <> EOS) do
        if v1< v2 then
          begin ch[out].put(v1); ch[in1].get(v1) end
        else
          begin ch[out].put(v2); ch[in2].get(v2) end;
        if v1 = EOS then
          while v2 <> EOS do
            begin ch[out].put(v2); ch[in2].get(v2) end
        else
          while v1 <> EOS do
            begin ch[out].put(v1); ch[in1].get(v1) end;
        ch[out].put(EOS);
        ready := false
      end;
    begin ready := false end;

var m: array [1 .. INPUTS - 1] of Merger;

class Printer
  var ready: boolean;
  action print when ready;
    var v: integer;
```

```
      begin ch[1].get(v);
        while v <> EOS do
          begin write(v); ch[1].get(v) end;
        ready := false;
      end;
    begin ready := true end;

var p: Printer;
    i, r, s: integer;

begin {merger i merges from channels 2*i and 2*i+1}
  i := 1;
  while i < INPUTS do
    begin m[i].configure(2 * i, 2 * i + 1, i); i := i + 1 end;
    {input on channels INPUTS .. 2 * INPUTS -1}
    i := 0;
    while i < INPUTS do
      begin random(s); if s < 0 then s := -s;
        ch[INPUTS + i].put(s); i := i + 1
      end;
    i := 0;
    while i < INPUTS do
      begin ch[INPUTS + i].put(EOS); i := i + 1 end
  end.
```

3.6.7 The Sieve of Eratosthenes

The Sieve of Eratosthenes is a classic problem for determining which numbers in a given range are prime. Parallel algorithms have been developed to solve this problem. With a pipeline of filter processes, each filter receives a stream of numbers from its predecessor and sends a stream of numbers to its successor. The first number that a filter receives is the next largest prime, *p*; it passes on to its successor all numbers that are not multiples of *p*. The program is shown in Listing 3.10. Note that the first filter (Sieve[1]) sends all the odd numbers to the second filter.

Listing 3.10: SV.pas

```
program SV;

const MAX = 1000;

class Channel
  var data: integer; empty: boolean;
  procedure put(x: integer) when empty;
    begin data := x; empty := false end;
  procedure get(var x: integer) when not empty;
    begin x := data; empty := true end;
```

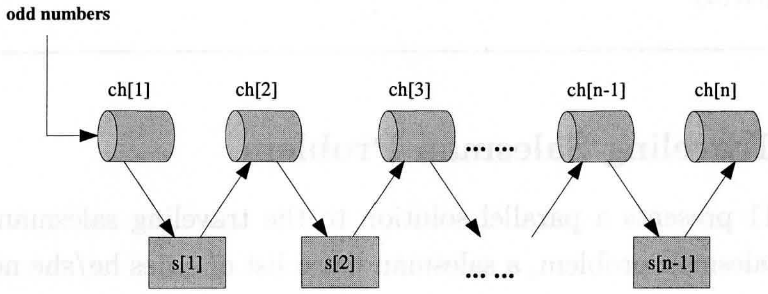

Figure 3.2: Communications between sieves and channels

```
    begin empty := true end;

  var ch: array [1 .. MAX] of Channel;

  class Sieve
    var c: integer;
    procedure configure(i: integer);
      begin c := i end;
    action filter when c > 0;
      var p, n: integer;
      begin ch[c].get(p); n := p;
        while n <> 0 do {0 is sentinel}
          begin ch[c].get(n);
            if n mod p <> 0 then ch[c + 1].put(n);
          end;
        if p <> 0 then write(p);
        ch[c + 1].put(0); {pass on sentinel}
        c := 0
      end;
    begin c := 0 end;

  var s: array[1 .. MAX − 1] of Sieve;
    i: integer;

  begin {configure s[i] to receive from ch[i] and send to ch[i+1]}
    i := 1;
    while i < MAX do
      begin s[i].configure(i); i := i + 1 end;
    {send odd numbers 3,5,7, ... MAX to ch[1]}
    i := 3;
    while i < MAX do
```

```
    begin ch[1].put(i); i := i + 2 end;
  {send sentinel}
  ch[1].put(0)
end.
```

3.6.8 Traveling Salesman Problem

Listing 3.11 presents a parallel solution to the traveling salesman problem. In the traveling salesman problem, a salesman has a list of cities he/she needs to visit and a list of cost for travelling between each pair of cities. The problem is to find a "tour" for the salesman, which minimizes the total cost of the trip. A tour is an ordered list of all cities that starts and ends with the salesman's hometown.

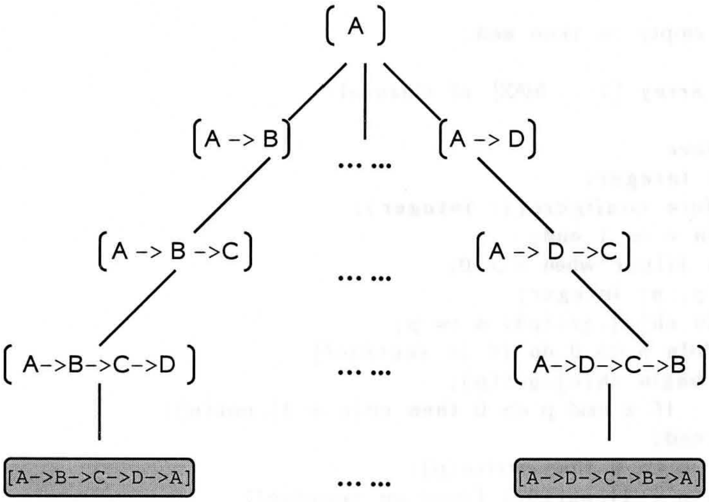

Figure 3.3: The tree representing all possible tours

The idea is that in searching for solutions, we build a "tree". Leaves of the tree correspond to tours, and other rest nodes correspond to "partial" tours — trips that have visited some, but not all, of the cities. Figure 3.3 shows a search tree for a four-city tour, in which the starting point is city A. This parallel solution is a form of tree search. Each node of the tree has an associated cost: the cost of the partial tour.

We can use this to eliminate some nodes of the tree. Thus, we want to keep track of the cost of the best tour so far, and, if we find a partial tour or node of the tree that couldn't possibly lead to a less expensive complete tour, we shouldn't bother searching the children of that node. In the parallel solution, we have each `Search_agent` search the tours starting with its assigned 2-city partial tours. The only "communication" between the `Search_agent` will occur when they need to access the current best tour.

Listing 3.11: TSP.pas

```
program tsp;
const
  INFINITY = 20000;
  NO_CITY  = -1;
  INITIAL_STACK_SIZE = 1000;
  MAX_SIZE = 1000;
  MID_SIZE = 1000;
  ROUND = 1;
  CITY_COUNT = 4; {number of cities to travel}
  THREAD_COUNT = 2; {number of search_agent to be created}

type
  Tour_t = record
    cities: array [0..CITY_COUNT] of integer;
    count: integer;
    cost: integer;
  end;
  Stack_elt_t = record
    tour_p : Tour_t;
    city : integer;
  end;
  {Ary_of_stack_elt_t = array[1..MID_SIZE] of Stack_elt_t;  }
  My_stack_t = record
    list: array [0..MAX_SIZE] of Stack_elt_t;
    top: integer;
    allocated: integer;
  end;
  Ary_of_int = array [0..CITY_COUNT * CITY_COUNT] of integer;

var
  best_tour: Tour_t;
  mat: Ary_of_int;

procedure init_mat(var m: Ary_of_int);
  var i, j, n, x: integer;
begin
  i := 0; j := 0; n := CITY_COUNT;
  while i < n do
    begin
      j := i;
      while j < n do
```

```
        begin
          if i = j then
            begin mat[n * i + j] := 0; write(mat[n * i + j]); j := j + 1 end
          else
            begin
              random(x);
              mat[n * i + j] := x;
              mat[n * j + i] := x;  {the matrix is symetric to the diagonal}
              write(mat[n * i + j]);
              j := j + 1
            end
        end;
      i := i + 1
    end
end;

procedure empty(var stack_p: My_stack_t; var ret: boolean);
begin
  ret := stack_p.top = -1;
end;

procedure initialize_tour(var tour_p: Tour_t);
  var i,n:integer;
begin
  i := 0;
  n := CITY_COUNT;
  while i < n + 1 do
    begin
      tour_p.cities[i] := NO_CITY;
      i := i + 1;
    end;
  tour_p.cost := 0; tour_p.count := 0;
end;

procedure find_initial_cities(my_rank: integer;
    var first: integer; var last: integer);
var
  quo, rem, my_city_count , m , n: integer;
begin
  n := CITY_COUNT;
  m := THREAD_COUNT;
  quo := (n - 1) div m;
  rem := (n - 1) mod m;
  if my_rank < rem then
    begin
      my_city_count := quo + 1;
      first := my_rank * my_city_count + 1;
      last := first + my_city_count - 1;
    end
  else
    begin
      my_city_count := quo;
      first := my_rank * quo + rem + 1;
```

```
      last := first + my_city_count - 1;
    end;
  writeln
end;

procedure Dup_tour(var tour_p:Tour_t; var temp_p: Tour_t);
  var i:integer;
begin
  i := 0;
  while i < CITY_COUNT do
    begin
      temp_p.cities[i] := tour_p.cities[i];
      i := i + 1
    end;
  temp_p.cost := tour_p.cost;
  temp_p.count := tour_p.count
end;

procedure push(var tour_p: Tour_t; city: integer; var stack_p: My_stack_t);
var
  t: Tour_t;
  i, j: integer;
  new_list: array[1..MAX_SIZE] of Stack_elt_t;
begin
  Dup_tour(tour_p, t);
  if stack_p.top = stack_p.allocated - 1 then
    begin
      stack_p.allocated := 2 * stack_p.allocated;
      if stack_p.allocated > MAX_SIZE then write(-9999);
      { '[push operation failed] Stack size limit has been exceeded!'}
    end;
  stack_p.top := stack_p.top + 1;
  i := 0;
  stack_p.list[stack_p.top].city := city;
  while i < CITY_COUNT do
    begin
      stack_p.list[stack_p.top].tour_p.cities[i] := t.cities[i];
      i := i + 1;
    end;
  stack_p.list[stack_p.top].tour_p.count := t.count;
  stack_p.list[stack_p.top].tour_p.cost := t.cost
end;

procedure create_init_records(my_rank: integer; var stack_p: My_stack_t);
  var
    city: integer;
    tour: Tour_t;
    my_first_city, my_last_city: integer;
begin
  find_initial_cities(my_rank, my_first_city, my_last_city);
  initialize_tour(tour);
  tour.cities[0] := 0;
  tour.count := 1;
```

```
  city := my_first_city;
  while city < my_last_city + 1 do
    begin
      tour.cost := mat[city];
      push(tour, city, stack_p);
      city := city + 1;
    end;
end;

procedure visited(nbr: integer; var tour_p: Tour_t; var is_visited: boolean);
  var i, c :integer;
begin
   i := 0;
   c := 0;
   while i < tour_p.count do
     begin
       if tour_p.cities[i] = nbr then c := c + 1;
       i := i + 1;
     end;
   is_visited := c > 0
end;

procedure pop(var tour_pp:Tour_t; var city_p:integer; var stack_p:My_stack_t);
var top:integer;
       i:integer;
begin
  top:= stack_p.top;
  tour_pp.count:= stack_p.list[top].tour_p.count;
  tour_pp.cost := stack_p.list[top].tour_p.cost;
  i:=0;
  while i <= CITY_COUNT do
     begin
       tour_pp.cities[i]:= stack_p.list[top].tour_p.cities[i];
       i:=i+1;
     end;
  city_p:= stack_p.list[top].city;
  stack_p.top:= stack_p.top-1;
end;

procedure feasible(city: integer; nbr: integer; var tour_p: Tour_t;
                     loc_best_cost:integer; var is_feasible:boolean);
  var is_visited :boolean;
begin
  visited(nbr, tour_p, is_visited);
  is_feasible := (not is_visited) and
                 (tour_p.cost+mat[CITY_COUNT*city+nbr] < loc_best_cost)
end;

procedure initialize_stack(var stack_p: My_stack_t);
begin
  stack_p.top := -1;
  stack_p.allocated := INITIAL_STACK_SIZE
end;
```

```
class Best_tour_seeker
  var n: integer;

  procedure check_best_tour(city:integer; var tour_p:Tour_t;
      var loc_best_cost_p:integer);
    var i:integer;
  begin
    if tour_p.cost+mat[city * n] < best_tour.cost then
      begin
        i:=0;
        while i < tour_p.count do
          begin
             best_tour.cities[i] := tour_p.cities[i];
             i := i + 1;
          end;
        best_tour.cities[n] := 0;
        best_tour.count := n + 1;
        loc_best_cost_p := tour_p.cost + mat[city * n];
        best_tour.cost:= loc_best_cost_p
      end
    else if loc_best_cost_p > best_tour.cost then
      loc_best_cost_p := best_tour.cost
  end;
begin
  n := CITY_COUNT;
end;

var tour_finder: Best_tour_seeker;

class Search_agent
  var
    init_done: boolean;
    rank: integer;
    r: integer;

  procedure initialize(i: integer);
  begin
    init_done := true;
    rank := i
  end;

  action search when init_done and (r < ROUND);
    var
      is_empty, is_feasible: boolean;
      stack: My_stack_t;
      tour_p: Tour_t;
      nbr, loc_best_cost, city : integer;
  begin
    loc_best_cost := INFINITY;
    initialize_stack(stack);
    create_init_records(rank, stack);
    empty(stack, is_empty);
```

```
    while not is_empty do
      begin
        pop(tour_p, city, stack);
        tour_p.cities[tour_p.count] := city;
        tour_p.count := tour_p.count+1;
        if tour_p.count = CITY_COUNT then
          tour_finder.check_best_tour(city, tour_p, loc_best_cost)
        else
          begin
            nbr := 1;
            while nbr < CITY_COUNT do
              begin
                feasible(city, nbr, tour_p, loc_best_cost, is_feasible);
                if is_feasible then
                  begin
                    tour_p.cost := tour_p.cost + mat[CITY_COUNT*city + nbr];
                    push(tour_p, nbr, stack);
                    tour_p.cost := tour_p.cost - mat[CITY_COUNT*city + nbr]
                  end;
                nbr := nbr + 1
              end;
          end;
        empty(stack, is_empty);
      end;
    r := r + 1;
    write(best_tour.cost)
  end;

begin
  init_done := false;
  r := 0
end;

var
  m: integer;
  SA: array [0..THREAD_COUNT - 1] of Search_agent;
begin
  m:=0;
  init_mat(mat);
  initialize_tour(best_tour);
  best_tour.cost := INFINITY;
  while m < THREAD_COUNT do
    begin
      SA[m].initialize(m);
      m := m + 1
    end;
end.
```

Chapter 4

Detailed Description of Implementation

This implementation of ABC Pascal was developed and tested on Linux for Intel 32-bit platforms. The compiler produces assembly code, which in turn is assembled and linked to obtain executables.

This chapter describes the structure of the compiler and run-time system, including design decisions, followed by a detailed exposition on how the compiler generates assembly code.

4.1 Structure of Compiler and Run-time System

The implementation is composed of two main parts: a single-pass compiler and a run-time system. The compiled code communicates with run-time system through function calls. Figure 4.1 presents the overall structure of the system. Assembler and linker are not parts of our work, but they are needed to produce executables.

4.1.1 The Compiler

The implementation of the compiler is derived from the Pascal0 compiler [19]. The Pascal0 compiler produces stack-based code to be interpreted by a virtual machine.

ABC Pascal was explicitly designed to be easy to implement with a single-pass compiler, by top-down recursive descent parsing. A "pass" here means a complete

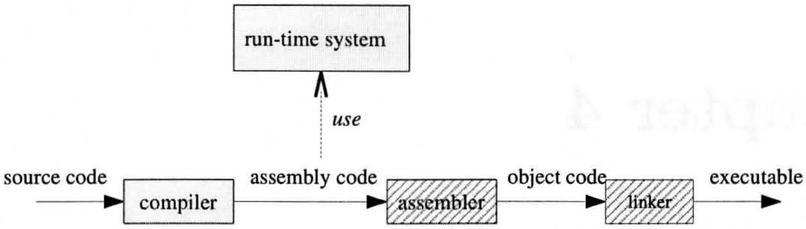

Figure 4.1: Overall structure

traversal of the source program, or internal representation of the source program. A single-pass compiler makes only one pass over the source code, parsing, analyzing and generating code all at once. There is no need to construct an explicit representation of the source code, such as an abstract syntax tree. Moreover, with a single-pass compiler, the phases that perform syntax analysis, semantic analysis, and code generation are combined together. Figure 4.2 shows the dependency diagram of this single-pass compiler.

The compiler consists of four modules. Each module is implemented by a unit.

- Scanner module:

  Read the source code and produce a stream of tokens. This module provides interfaces (shown in Listing 4.1) to

  - parse the next symbol in the source,
  - catch compile-time errors.

```
unit scanner
  interface
  const
    IdLen = 40; {number of significant characters in an identifier}
  type
    Symbol = (null, ExpSym, TimesSym, DivSym, ModSym, AndSym, PlusSym, MinusSym,
              OrSym, EqlSym, NeqSym, LssSym, GeqSym, LeqSym, GtrSym, PeriodSym,
              CommaSym, ColonSym, RparenSym, RbrakSym, OfSym, ThenSym, DoSym,
```

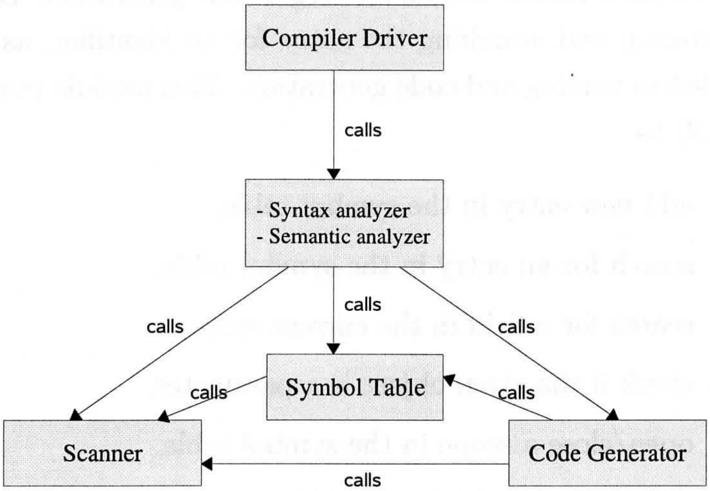

Figure 4.2: Dependency diagram

```
          LparenSym, LbrakSym, NotSym, BecomesSym, NumberSym, IdentSym,
          SemicolonSym, EndSym, ElseSym, IfSym, WhileSym, WhenSym, ArraySym,
          RecordSym, MonitorclassSym, ConstSym, TypeSym, VarSym,
          ProcedureSym, ActionSym, BeginSym, ProgramSym, EofSym);
  Identifier = string[IdLen];

var
  sym: Symbol;      {the next symbol to be parsed}
  v: longint;       {numeric value of a number, if sym = NumberSym}
  id: Identifier;   {literal string for an identifier, if sym = IdentSym}
  error: Boolean;   {indicates whether a compile-time error has occurred}

procedure Mark (msg: string);
   {Reports on a compile-time error.}
procedure Warn (msg: string);
   {Reports on a warning message.}
procedure GetSym;
   {Scan the next symbol from the source code and save that symbol to 'sym'.}
```

Listing 4.1: Interfaces of Scanner module

- Symbol Table module:

  Maintains for every identifier the context-dependency information, as well as attribute information needed for target code generation. Encapsulates procedures for storing and searching the entry for an identifier, as well as data structure needed to parsing and code generation. This module provides interfaces (shown in 4.2) to

- add new entry in the symbol table,
- search for an entry in the symbol table,
- search for a field in the current scope,
- check if the given object is a parameter,
- open/close a scope in the symbol table,
- define the standard procedures,

```
unit symboltable
  interface
  uses scanner;
  type
    Class=(HeadClass, VarClass, ParClass, ConstClass, FieldClass, TypeClass,
           ProcClass,SProcClass, RegClass, EmitClass, CondClass, MonitorClass,
           ActionClass);
    Form =(Bool, Int, Arry, Rcrd, Monitor);  {supported types}

  OA_List = ^ObjNode;
   {to store all actions and object offset. needed when calling pthread API}

  ActionList = ^ActionNode;  {list of actions}

  ObjNode = record
        obj_name: Identifier;
        val : longint;
        actions : ActionList;
        next : OA_List;
  end;  {represent one object in a linked list}

  ActionNode = record
        id: Identifier;
        next:  ActionList;
  end;  {represent one action in a linked list}

  Objct = ^ObjDesc;
  Typ = ^TypeDesc;

  Item = record
      mode: Class;
      lev: longint; {level of scope}
      tp: Typ;
```

```
    a: longint;  {value of the item; offset address of the item}
    b: longint;  {saves the current pc which is used to make up the jumping
                  labels}
    c: longint;  {the relational operation offset}
    r: longint;  {saves the current pc for making up jumping labels, similar
                  to .b}
    o: longint;  {offset value of a record field}
    indirect : boolean;  {whether requires indirect addressing}
    bool_set : boolean;  {whether the item's boolean value is set}
    sc : boolean;    {whether the conditional expression is short-circuited}
    push_placeholder: boolean;
        {set to true initially but set to false after the first call to
         placeholder}
    parSize : longint    {parameter size, if procedure}
  end ;

  ObjDesc = record
    cls: Class;
    lev: longint;
    next, dsc: Objct;
    tp: Typ;
    name: Identifier;
    val: longint;
    isGuarded : boolean;  {indicates if a procedure is guarded}
    isAParam : boolean;
    parSize : longint     {total size of all parameters}
  end;      {represents one entity in the symbol table}

  TypeDesc = record
    form: Form;
    typename: Identifier;
    fields: Objct;            {for records and monitor class}
      {if form=record then record fields;
       if form=monitor then the list vars and procedures and actions}
    a_list: ActionList;       {to store all actions of one class}
    action_count: longint;    {number of actions}
    base: Typ;                {the base type for arrays}
    lower, size, len: longint;
              {lower bound, required memory size, and length for arrays}
  end ;   {represents a type descriptor}

  var
  topScope: Objct;
   {current scope, where search for an identifier starts, this is the global
    variable to the intelcompiler program}

  guard: Objct;    {topScope and universe are linked lists. they are ended
                    with 'guard'}
  boolType, intType: Typ;    {predefined primitive types}

  procedure NewObj (var obj: Objct; cls: Class);
      {Insert an object to the current scope. If the 'id' is not found, insert
       to the end of the list; otherwise, do not insert, return the object
```

```
      that is already defined.}
  procedure Find (var obj: Objct);
     {Search for 'id' - the last symbol scanned - from the topscope and up.
      If object is found, return it to obj. Stop the search when universe
      level is reached.}
  procedure FindField (var obj: Objct; list: Objct);
     {Given a record or an object, search in the symbol table for its
      field(s) or procedure(s).}
  function IsParam (obj: Objct): boolean;
     {Return true if 'obj' is a parameter.}
  procedure OpenScope;
     {Open a new scope in the symbol table based on the current scope}
  procedure CloseScope;
     {Return to the parent scope of the current scope}
  procedure PreDef (cl: Class; n: longint; name: Identifier; tp: Typ);
     {Define standard identifiers: true, false, read, write, writeln, random.}
```

Listing 4.2: Interfaces of Symbol Table module

- Code Generator module:

  Provides data structures and procedures for emitting target code. This module provides interfaces (shown in Listing 4.3) for the compiler driver to

  - generate proper assembly code for language constructs as parsing keeps going,
  - save the generated code to a file if no compile-time error is caught.

```
unit IntelGenerator;
  interface
  uses Scanner, symboltable;

  const
    ACTION_LIMIT = 499; {maximum number of actions in a class}
    ADDOP = 0; SUBOP = 1; MULOP = 2; DIVOP = 3; MODOP = 4; CMPOP = 5;
    BEQOP = 15; BNEOP = 16; BLTOP = 17; BGEOP = 18; BLEOP = 19; BGTOP = 20;
  type
    action_entry = record
      action_body: Identifier;
        {a label in assembly code that identifies the action body}
      action_guard:Identifier;   {a label that identifies the action guard}
    end;
    AQ = array [0..ACTION_LIMIT] of action_entry;
  var
    curlev, pc, trap_line: longint;   {trap_line is for debugging purpose}
    fileID: Identifier;               {the output file name }
    File_GAS: text;                   {the generated .s file}

  procedure Call_object_proc(temp_str, temp_id: string; param_size: longint);
```

```
  {Generate code to call a procedure of an object}
procedure Obj_array_addr(temp_str: string);
  {Load the base address of an array of objects}
procedure Save_addr_in_edi;
  {Temporarily save the address in the edi register}
procedure Push_addr_in_edi;
  {Load the address saved in edi register}
procedure Restore_stack_ptr(temp_str: string);
  {Clean up the stack after a procedure call}
procedure MakeGuardLabel(classid, procid: string);
  {Make a label for procedure guard}
procedure MakeGuardJumpLabel(classid, procid: string);
  {Make a label to jump to the condition wait loop}
procedure Lock_mutex(mtx_addr: string);
  {Lock the object mutext variable}
procedure Make_jump_and_wait_label(procid: string);
  {Jump to the guard evaluation block and make a label for the condition
   wait loop}
procedure Eval_bool_val(opstr: string; var expr: Item);
  {Evaluate the boolean expression and make a conditional jump}
procedure Get_bool_val;
  {Get the boolean value if it is already set}
procedure Wait_on_condition(mtx, cv: longint);
  {Generate code for the condition wait block}
procedure Broadcast_and_unlock(cv, mtx: longint);
  {Generate code to broadcast on condition variable and unlock the mutex}
procedure MakeActionGuard(action_id: string);
  {Make a label to identify the action guard block}
procedure Return_from_action_guard;
  {Generate code to return from evaluation of action guard}
procedure MakeActionGuard_open(action_id: string);
  {Generate code for an unguarded action}
procedure MakeActionLabel(action_id: string);
  {Make label to identify an action body}
procedure Make_conjunction_negation_label(monitor_id: string);
  {Make label to identify the conjunction of negation of the guards}
procedure Eval_action_guard(action_guard: string; var anchor: longint);
  {Evaluate the action guard and push the result onto stack}
procedure Op_logical_and;
  {Performs logical AND operation}
procedure Generate_obj_thread_prelogue(worker_id, local_block: string);
  {Code as the prelogue of the body of an object thread}
procedure Init_n_and_next(offset_n, n, offset_next: longint);
  {Initialize variables n and next}
procedure Init_count_and_done(offset_count, offset_n, offset_done: longint);
  {Initialize variables count and done}
procedure If_not_done_then_wait(offset_done,offset_count,offset_n,anchor:
                                longint; var end_if_not_done,
                                start_second_inner_loop,end_second_inner_loop:
                                longint; monitor_id: string; mtx,cv:longint);
  {If no action is eligible to run, then wait on condition until one action
   is enabled}
procedure Update_val_next(offset_next, offset_n: longint);
```

```
  {Update the value of variable next — next:= (next+1) mod n}
procedure Generate_obj_thread_epilogue(start_while_true, end_while_true,
                                        local_block_size:longint);
  {Generate code as the epilogue of an object thread body}
procedure Init_obj(mtx, cv: longint);
  {Initialize an object by setting up the mutex and condition variable}
procedure Generate_text_section_label;
  {Generate directive for .text section in assembly code}
procedure Generate_main_action_label(main_action: string);
  {Generate label for the main action}
procedure Call_init_block(init_func_name, offset: string);
  {Execute the initialization blocks of all objects}
procedure Setup_pthread_attribute;
  {Set up pthread attribute for thread creation and management}
procedure Setup_randomization;
  {Set up for random number generation}
procedure Create_main_action(main_action_label: string);
  {Create a thread associated to the program main body}
procedure Setup_global_counter(temp: string);
  {Save the number of object thread to variable 'global_counter'}
procedure Setup_action_base_ptr(offset: string);
  {Calculate the address of an active object and save it in eax}
procedure Create_obj_thread(m: longint; action_name: string);
  {Create a pthread for each active object}
procedure Join_main_action;
  {Wait for the program main body(main action thread) to finish}
procedure Check_termination(var anchor1: longint);
  {Check the termination condition - whether the global conter reaches zero}

procedure PlaceHolder;
  {Generates code to put a placeholder on top of the stack, for accessing
   array elements of record fields.}
procedure IncLevel (n: longint);
  {Increase the scope level by n.}
procedure MakeConstItem (var x: Item; tp: Typ; val: longint);
  {Make a new item that reprents a constant.}
procedure MakeItem (var x: Item; y: Objct);
  {Given object y, make a new item to represent y.}
procedure LoadItem_32(var x:Item; LeaveAddress: boolean);
  {When item x needs to be used, code is geneated depending on what kind of
   item x is. LeaveAddress indicates if the address or the value of x is
   needed}
procedure Field_32 (var x: Item; y: Objct);
  {Make an item for a record field, where y represents a field.}
procedure Index_32 (var x, y: Item);
  {Computes the offset of array element (x[y]), and push this offset onto
   stack}
procedure Op1_32 (op: Symbol; var x: Item);
  {Given operator 'op', makes an item(x) and generates code for x:=op x}
procedure Op2_32 (op: Symbol; var x, y: Item);
  {Given operator 'op', makes an item(x) and generates code for x:=x op y}
procedure Relation_32 (op: Symbol; var x, y: Item);
  {Given a relational symbol 'op', makes an item(x) and generates code
```

```
  for x:=x op y}
procedure Store_32 (var x, y: Item);
 {Store some value(represented by y) to the address of the given variable(x)}
procedure Parameter_32 (var x: Item; ftyp: Typ; cls: Class);
 {Generates code when parsing actual parameters of some procedure. 'x' is
  the parameter; 'ftyp' is the desired type; 'cls' indicates whether it is
  a value or a reference paramter.}
procedure CJump_32 (var x: Item);
 {Given a boolean condition ,represented by 'x',generates code for making a
  condition jump}
procedure Place_boolean_val(var x: Item);
 {Push the boolean value of an expression onto stack, if it is not
  on stack yet}
procedure Call_32 ( name: Identifier );
 {Making a procedure call, whose name is given by 'name'}
procedure IOCall_32 (var x, y: Item);
 {Making a call to standard procedure.}
procedure Header_var (size: longint; num_thr :longint);
 {Generate code for the .data and .bss section. 'size' gives the total
  memory storage needed for all variables. 'num_thr' indicates total number
  of threads needed to create.}
procedure Header_code;
 {Generate code for handling runtime errors, and generates label for 'main'.}
procedure Enter_32 (size: longint);
 {Generate the prelogue code of a procedure call.}
procedure Return_32(size: longint);
 {Generate the postlogue code of a procedure call; return from the call}
procedure GenerateFuncPrefix(id: Identifier);
 {Make a prefix for a procedure body in assembly code.}
procedure GenerateLabel(id: Identifier);
 {Make a label to mark a procedure body in assembly code.}
procedure MakeJumpLabel(s: string; pc0: longint);
 {Make a label for jumps.}
procedure Jumpto(op:string; s: string; pc_label: longint);
 {Issue assembly code for a condional jump}
procedure inner_loop_one_and_three(n, offset_next, offset_count, offset_n,
                                   offset_done: longint; action_queue:AQ);
 {Generate code that implements the body of an object-thread.}
procedure Open;
 {Initialize current level and pc to zero.}
procedure Close_32;
 {Issue assebly code to terminate the program execution.}
procedure Load;
 {Write the generated code to target .s file}
function TransformToStr(rel_op:longint): string;
 {Transform the given relational operator(represented by a number) to
  its corresponding assembly instruction.}
```

Listing 4.3: Interfaces of Code Generator module

- Compiler Driver:

Contains both syntactical analyzer and semantic analyzer. Uses all three modules listed above.

Optimization in the global scope is impossible. We focus on functionality for this prototyped implementation, rather than performance. A single-pass compiler is sufficient to deliver a simple but correct implementation.

4.1.2 Run-time System

The concurrency features of ABC Pascal match closely to the Pthread API, so direct mappings of ABC Pascal's multiprogramming mechanism to Pthread operations becomes feasible. Action-based concurrency featured by ABC Pascal is fully implemented with the Pthreads API. Therefore, the run-time system is comprised of a program termination checker, an object-thread implemented in assembly, and Pthreads operations.

For concurrent programming languages, the run-time system (RTS) is a key component in the implementation. Multithreading is typically supported by the underlying RTS.

The implementation choices for RTS are either developing a thread system from scratch or building the RTS on top of some thread packages, like Pthreads. As developing the implementation of ABC Pascal, effort has been devoted to the former choice. However, we soon realized that building a thread system on our own means we are building mechanisms that already exist within the underlying implementation of Pthreads. To build a replacement of a sophisticated Pthreads implementation is impractical, because a thread system should always keep up with the advances of the OS kernel. Therefore, a "live" thread system should be improved in response to the constantly increasing support of multithreading found on most operating systems. Nevertheless, by building the RTS on top of Pthreads, we leave the development and maintenance of low-level mechanisms to the Pthreads implementors, and it further enhances the portability of the implementation. As mentioned in the introductory chapter, Pthreads API are standardized interfaces widely available on Unix-like operating systems. An ABC Pascal RTS built on top of Pthreads can be easily portable from one Unix-like operating systems to another.

Many Unix-based operating systems are shipped with efficient implementation of Pthreads. On Linux, the OS we developed ABC Pascal, the default Pthreads implementations is NPTL [7]. Now, it is part of the GNU C library on Linux. NPTL has been proved to be an optimized Pthreads implementation for Linux. One important reason is that NPTL takes fully advantage of Linux kernel improvements intended to support threads.

In summary, we advocate to build the RTS on top of Pthreads or some widely available alternatives in future development of ABC Pascal.

4.2 Code Generation Scheme

This section on code generation is divided into two parts: the scheme for fundamental sequential constructs, and the scheme for concurrent constructs.

The assembly code produced by the compiler follows the AT&T opcode syntax. AT&T syntax uses a separate character at the end of mnemonics to reference the data size used in the operation. The AT&T instruction,

```
movl $5, %eax
```

moves a 32-bit operand, constant 5, to the `eax` register. Thus, the generic form of the mov instruction becomes: `movx`, where x can be the following:

- l for a 32-bit double word value
- w for a 16-bit word value
- b for an 8-bit byte value

A list of Intel instructions that appear in this chapter is shown below.

mov	move data between registers and memory
add/sub	arithmetic operations for add/subtract
imul/idiv	arithmetic operations for signed multiplication/division
and/or	logical operations for and/or
push	push the operand onto stack
pop	pop data from stack and store it to the operand
jmp	unconditional jump to an address
cmp	compare the values of two operands; only affect the bits in the EFLAGS register. The EFLAGS register contains 32-bit information that are mapped to represent control flag information
jeq	jump if equal, based on the state of EFLAGS register
jne	jump if not equal, based on the state of EFLAGS register
jge	jump if greater or equal, based on the state of EFLAGS register
jl	jump if less than, based on the state of EFLAGS register
call	calling a function
ret	return from a function

4.2.1 General Code Translation Strategy

Emitting code for sequential constructs mostly follows the conventional techniques introduced in compiler textbooks [3, 18]. Bearing in mind that ABC Pascal generates assembly code for Intel 32-bit platforms, and the compiler is a single-pass compiler, we decided to let the compiler emit stack-based code, rather than register-based code commonly produced by commercial compilers.

The assembly code shown in this chapter contains four functions: `addr()`, `val()`, `eval()` and `code()`.

addr(x): computes the address of x and push it onto stack, where x is a variable.

val(x): retrieves the value of x and push it onto stack, where x is a variable of primitive types.

eval(E): evaluates an expression E, and push the result onto stack, where E can either be boolean or arithmetic.

code(S): provides the set of instructions generated for statement(s) S.

Stack-based target code

General-purpose registers are scarce resource on IA-32, and most of them either will be affected by certain operations or need to be preserved. For instance, the **eax** register is affected by integer multiplication, and **edx** is affected by a modulo operation, while the **ebx** register must be preserved for accessing uninitialized data on some operating systems [4].

Register	Description
eax	Accumulator for operands and results data
ebx	Pointer to data in the data memory segment
ecx	Counter for string and loop operations
edx	I/O pointer
edi	Data pointer for destination of string operations
esi	Data pointer for source of string operations
ebp	Block pointer
esp	Stack pointer

Another difficulty is register allocation. On a CISC architecture (e.g. Intel), it is possible to frequently run out of architectural registers. A naive register allocator, must then spill one or more registers to memory, reuse the register for another purpose, and pops the saved value back into the register [17]. These operations on handling register spill are expensive. A sophisticated register allocator cannot be built with single-pass compilers. In this implementation, only *eax* and *ecx* are used to save temporary data.

The ABC Pascal compiler saves intermediate values by pushing the content onto the stack, and pops the content from the stack whenever it is needed for computations.

For example, the compiled code for assignment statement `x := y+10;` can be represented by the code listed below.

```
addr(x)
val(y)
popl    %ecx            # ecx= value of y
popl    %eax
```

```
addl    $10, %ecx       # ecx= y+10
movl    %ecx, (%eax)    # store y+10 in the memory location pointed by eax
```

The generated assembly code consists of three blocks: the initialized data section, the uninitialized data section, and the text section. The initialized data section, declared by `.data` directive, is the memory block reserved for initialized data elements. The uninitialized data section, declared by `.bss` directive, is the memory block reserved for uninitialized data elements. The text section is where all instruction code are placed, and it is declared by `.text` directive. To define the starting point for execution, an assembly program must declare a label `main`. The `main` label is used to indicate the instruction from which the program should start running. A basic template for an assembly program looks like the following:

```
.section .data
  # initialized data
.section .bss
  # unintialized data
.section .text
main:
  # instructions go here
```

Figure 4.3 shows the layout of an assembly program in the virtual memory space of a process. Note that the heap area is the memory that is dynamically allocated through system calls (e.g. `malloc`). Because ABC Pascal does not support dynamic object creation, the heap area will not be used in this implementation.

Global variables

All global variables in an ABC Pascal program, will are referenced by a single label in the uninitialized data section from the assembly code. Label `global_var` is the label that marks the base address of all global variables. The integer value `totalsize` is the memory space needed to allocate all global variables.

```
section  .bss
   global_var  totalsize
```

The memory space needed for the primitive types are listed below:

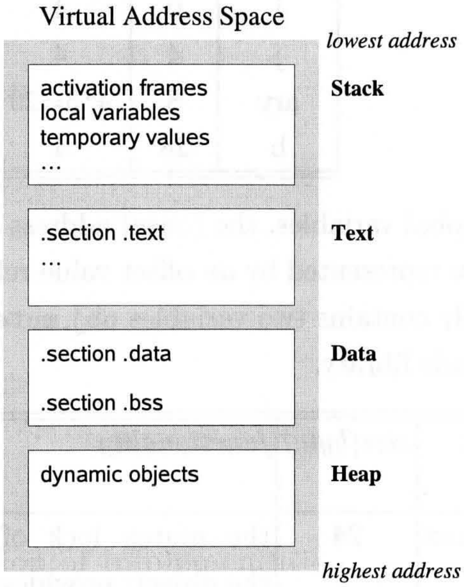

Figure 4.3: Virtual memory layout

type	*size*
integer	4 bytes
boolean	4 bytes

Each global variable has an entry in the symbol table. The compiler calculates and assigns an offset value for each global variable. This offset value is used to obtain the address of that variable.

For example, a program that declares four global variables, `i`, `j`, `ary`, `b`:

```
var
 i,j : integer;
 ary : array[1..5] of integer;
 b: boolean;
```

The offset value of the first variable is always be zero, and offsets of the rest global variable follows this equation:

$$\text{offset}(\textit{next variable}) = \text{offset}(\textit{current variable}) + \text{sizeof}(\textit{current variable})$$

So the compiler will assign the offset values as follows, for `i`, `j`, `ary`, and `b` respectively.

variable	*offset*	*size(byte)*
i	0	4
j	4	4
ary	8	4*5=20
b	28	4

Just like other global variables, the (base) address of an object, that is a variable of class types, will be represented by an offset value relative to the label `global_var`. Each object implicitly contains two variables `obj_mutex`, and `obj_cv`, of which types are defined in Pthreads library.

variable	*size(byte)*	*functionality*	*offset to the base address of object*
`obj_mutex`	24	the mutex lock of the object; provides mutual exclusion	0
`obj_cv`	48	the condition variable of the object; used to implement guarded methods	24

Thus, the offset address for objects private variables starts from 24 + 48 = 72. For a class E that has three private variables of integer type, the offset address for private variables x, y, z will be shown below.

```
class E
  var x,y,z : integer;
  ... ...
  begin
    x:=0
  end;
var ob: E;
```

fields	*size*	*offset address*
x	4	72
y	4	76
z	4	80

With the offset values setting up, it is straight forward to calculate the address for any private variables from any object. For example, we can calculate the address of private variable `y` of object `ob`, where function `address(X)` returns the absolute address of variable `X`:

```
address(ob.y) = offset(y) + offset(ob) + address(global_var)
```

Parameter passing

In ABC Pascal, parameters can either be passed by value or by reference. For value parameters, the value of the actual parameter is pushed onto stack before the procedure call. As for reference parameters, the address of the actual parameter is pushed onto stack before the procedure call. A reference parameter can be accessed by first fetching the its address saved on the stack, then fetching its content.

Method calls

A procedure of an object serves as an entry call for other objects to modify the state of the containing object. To invoke a procedure of an object, the correct format is:

`obj_name.procedure_name` (*argument list*)

The compiler appends the objects base address to the end of argument list, therefore, translate `obj_name.procedure_name`(*argument list*) to:

`obj_name.procedure_name`(*argument list*, *address of object*)

At the assembly level, the activation frame of an object's procedure is illustrated in Figure 4.4. As the procedure call returns, the stack pointer, esp, has to be restored to its address before the invocation is taken. The address of the object may always be referenced by taking the content of (%ebp+8).

```
pushl  first_parameter
... ...
... ...
pushl  last_parameter
addr(obj_name)          # push the address of obj_name onto stack
call   procedure_name
addl   $(parameter_size + 4), %esp
```

The autonomous execution of an action follows the same pattern as the procedures, except that an action has no parameters. At runtime, the body of an action is executed by:

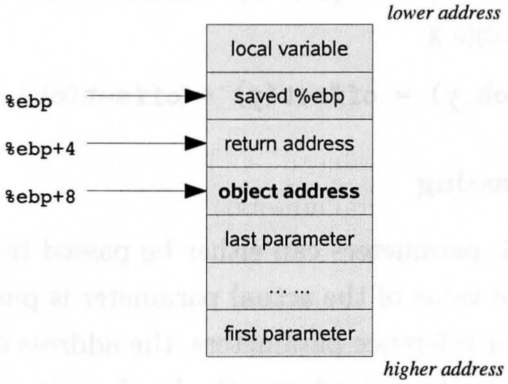

Figure 4.4: The activation frame

```
addr(obj)
call   action_name
addl   $4, %esp
```

Object initializations

Each object has an initialization block. This initialization block is set to execute as the first portion of the emitted assembly code, before any Pthread is created for the actions. Two Pthread operations are implicitly performed to the initialization block, to initialize the mutex and the conditional variable.

```
class E
  var x,y,z : integer;
  {..methods definitions..}

  begin {initializations}
    x:=0;
  end;
var ob: E;
```

The equivalent C code generated for the initialization block of object ob is listed below.

```
void ob_init(E *b){
```

```
    b->x = 0;
    pthread_mutex_init(&b->mutex, NULL);
    pthread_cond_init(&b->cv, NULL);
}
```

Using labels to implement jumps

One main reason that we choose to emit assembly code is the free use of labels in assembly programming. With the labels provided by the assemblers, forward jumps and backward jumps cannot be easier to implement.

To demonstrate the convenience by using labels, we use an ABC Pascal code that must involve both forward and backward jumps.

```
while i<0 do
  begin
    loop_body
  end;
```

This while-do statement is translated to the following assembly code pattern with the use of labels. Two labels are used to mark the starting point of the while-do statement and the exit point.

```
startwhile_101:  # assume pc=101 at this point
   eval(i<0)     # is i<0 true?
         # push True(1) onto the stack if i<0; otherwise push False (0).
   pop  %eax
   cmp  $0, %eax
   je   is_false_101   # jump if (i<0) is False
   code(loop_body)
   jmp  starwhile_101  # unconditional jump
is_false_101:
   # end of the while-loop
```

In fact, to avoid duplications of labels, every jumping label generated by the compiler consists of two portions: name of the label and current program counter (*pc*). These two portions are to be connected by an underscore symbol. As shown in the above code listing, suppose that value of *pc* is 101 when the compiler emits the starting label of the `while` statement, therefore, the label is named `startwhile_101`. Similarly, another label to mark the exit point is named `is_false_101`, because this `while` statement can be uniquely identified by $pc = 101$ in a global context, thus the *pc* value is sufficient to distinguish this statement from the others. There is no need to use two distinct *pc* values to mark the labels for a `while` statement.

4.2.2 Translations for Basic Sequential Constructs

Arithmetic expressions

Arithmetic operations can be directly mapped to the arithmetic instructions in assembly. We outline the correspondence and the translation scheme for all arithmetic operators.

- addition (+)

  Maps to instruction `add`. Evaluation of expression $y + z$, given y and z are integer variables, is translated to:

```
val(y)
val(z)
popl  %eax
popl  %ecx
addl  %ecx, %eax   # value of (y+z) is saved in %eax
```

- subtraction (−)

  Maps to instruction `sub`. Translation for subtraction is similar to addition operation.

- multiplication (∗)

  Maps to instruction `imul`. Evaluation of expression $y * z$, given y and z are integer variables, is translated to:

```
val(y)
val(z)
popl   %eax
popl   %ecx
imull  %ecx    # value of (y*z) is placed in %eax
```

- division (*div*)

  Maps to instruction `idiv`. Evaluation of expression $y \; div \; z$, given y and z are integer variables, is translated to:

```
val(y)
val(z)
popl   %ecx
popl   %eax
idivl  %ecx    # (y div z) is placed in %eax; the remainder is in %edx
```

- modulo (*mod*)
  Maps to instruction `idiv`, since the `idiv` operation places the remainder to `edx` register.

- unary minus (−)
  Implemented with `sub`. Evaluation of expression $-y$, given y is an integer variable, is translated to:

```
val(y)
movl  $0, %eax
subl  (%esp), %eax    # -y is saved in %eax
```

Boolean expressions

Boolean expressions provide a special and important opportunity for code improvement. The compiler can short-circuits boolean expression evaluations. The ABC Pascal impelmentation performs short-circuit evaluation of boolean expressions; thus, it generates code that skips the rest of computation when the overall value has already been determined.

For boolean expression `e1 or e2`, if `e1` evaluates to true, `e2` will not be evaluated, and the evaluation of the whole predicate returns true. For boolean expression `e1 and e2`, if `e1` evaluates to false, `e2` will not be evaluated, and the evaluation of the whole predicate returns false. Example 4.2.1 shows the generated assembly code for boolean expression `(d>-1) and (d<10)`, assuming `d` is a variable. At the end of the evaluation, either true or false is pushed onto the stack for further evaluation.

Example 4.2.1: assembly code generated for `(d>-1) and (d<10)`

```
# process the first half,(d>-1); if it evaluates to false, skip the second half.
  val(d)        # push value of d onto stack
  popl  %eax
  cmp   $-1, %eax
  jle   branchfrom_pc_a
  jmp   exitfrom_pc_a
 branchfrom_pc_a:
  pushl $0        # constant value 0 for FALSE
  jmp   false_and_others_pc_a
 exitfrom_pc_a:   # continue to evaluate the second part
# now process the second half (d<10)
```

```
   val(d)
   popl  %eax
   cmp   $10, %eax
   jge   false_and_others_pc_a
   pushl $1
 false_and_others_pc_a:
```

It is clear to see the code generation scheme from this concrete example. If the evaluation of the first part returns false, the execution control immediately jumps to the end, which is markd by label `false_and_others_pc_a`.

Assignment statement

Assignment statement has generic form of `x:=expr`. The compiler first computes the address of `x` and pushes it onto the stack; then computes value of `expr` and pushes it onto the stack. In the final step, value of `expr` is stored in address of `x`.

```
addr(x)
eval(expr)
popl   %eax
popl   %ecx
movl   %eax, %ecx
```

Example 4.2.2 shows the generated code for statement `c[i]:=100`, where array `c` is declared as

`c:  array [0..9] of integer;`

Example 4.2.2: assembly code generated for `c[i]:=100`

```
pushl  $0  # push a placeholder for accessing elements of c
addr(i)       # push address of i onto stack
movl   $0, %eax
cmp    %eax, (%esp)   # check the array index; raise runtime error if illegal
jl .trap              # the .trap routine raises runtime error for illegal index
movl   $9, %ecx
cmp    %ecx, (%esp)
jg .trap
movl   $4, %eax
popl   %ecx
subl   $0, %ecx
mull   %ecx
addl   %eax, (%esp)
```

```
addr(c)       # push base address of array c
popl   %eax
addl   %eax, (%esp)   # the top of stack contains the absolute address of c[i].
pushl  $100
popl   %eax
popl   %ecx
movl   %eax, (%ecx)   # store constant 100 to the address of c[i]
```

While-do statement

Because forward branches can be easily implemented by labels in assembly code, translation for `while-do` statement is fairly simple.

```
while expr do
  begin
    loop_body
  end;
```

Example 4.2.3 shows the generated code for a generic while-do statement, where pc_a and pc_b denote two uniquely recorded value of program counter.

Example 4.2.3: assembly code generated for while-do statement

```
startwhile_pc_a:        # the staring point of the while-loop
  eval(expr)
  popl  %eax
  cmp   $0, %eax
  je  is_FALSE_pc_b     # if expr=false, jump to the termination point
  code(loop_body)
  jmp  startwhile_pc_a  # jump back to the be starting point of the while-loop
is_FALSE_pc_b:
  # termination point of the while-loop
```

If-then-else statement

For if-then-else statement, labels are frequently used to mark all blocks. Suppose, for example, we generate code for the following source code.

```
if expr_A then         {block A}
      clause_A
else if expr_B then    {block B}
      clause_B
```

```
  else                    {block C}
      clause_C
```

The code generator will produce unique labels for each block using the distinct values of the program counter, pc_a and pc_b, for example. Example 4.2.4 shows the sketch of the generated code.

Example 4.2.4: assembly code generated for nested if-then-else

```
  eval(exprA)
  popl  %eax
  cmp  $0, %eax
  je   is_FALSE_pc_a     # jump to block B if expr_A=false; pc_a=current program counter
  code(clause_A)
  jmp  exitfrom_pc_a # jump to the end upon exiting from clause_A
is_FALSE_pc_a:
  eval(expr_B)
  popl  %eax
  cmp   $0, %eax
  je is_FALSE_pc_b       # jump to block C if expr_B=false; pc_b=current program counter
  code(clause_B)
  jmp exitfrom_pc_b      # jump to the end upon exiting from clause_C
is_FALSE_pc_b:
  code(clause_C)
exitfrom_pc_b:
exitfrom_pc_a:
```

4.2.3 Delayed Code Generation

Most architectures provide arithmetic instructions that allows one operand to be a constant value, so in some cases there is no need to load both two operands to registers if we know one operand is a constant. This gives us the opportunity to delay code emission in certain cases of arithmetic computation until it is definitely known that there is no better solution [18].

With Intel architecture, the addition and subtraction instructions allow the source operand to be a constant:

```
add source, destination

sub source, destination
```

Therefore, it is possible to generate optimal code for addition, and subtraction where the second operand of the 'minus' operation is a constant.

For addition, `x+c` and `c+x`, where `c` is a constant, the code generated is the same. Only the value of `x` is loaded to a register:

```
val(x)    #get the value of x and push it onto stack
popl  %eax
addl  $c, %eax
```

For subtraction, `x-c`, where `c` is a constant, the code generated is similar to additions. Only the value of `x` is loaded to a register:

```
val(x)    #get the value of x and push it onto stack
popl  %eax
subl  $c, %eax
```

However, if the first operand of subtraction is constant, `c-x`, for example, both operands need to be loaded to registers, because the destination operand of the assembly instruction must not be a constant.

For multiplication, division, and modulo, Intel architecture disallows constant value to appear in the assembly instructions, so we are unable to generate optimal code for these arithmetic operations.

4.2.4 The Predefined Procedures

As mentioned in Chapter 3, there are four predefined procedures in ABC Pascal to support basic integer I/O and number randomization. It is possible to implement these procedures with system calls directly provided by Linux. Because the implementation needs to be done for another OS, it would be cumbersome to implement these procedures with another set of system calls from time to time. A more portable solution has been adopted — to use the standard C library functions to implement basic integer I/O and random number generation. Implementation details for all predefined procedures can be found in procedure `IOCall_32` from Listing D.2, in Appendix D.

- `read(var x: integer)`

  Implemented by calling function `scanf`. Given `x` is an integer variable, `read(x)`, is translated to:

```
addr(x)  # push the address of x
pushl  format_string
call   scanf
addl   $8, %esp
```

- `write(x: integer)`

  Implemented by calling function `printf`. Given `x` is an arithmetic expression, `write(x)`, is translated to:

```
eval(x)  # push the value of x
pushl  format_string
call   printf
addl   $8, %esp
```

- `writeln`

  Implemented with function `printf`

- `random(var x: integer)`

  Implemented with functions `srand`, `rand`. Function `srand` is called only once at the beginning of program execution. Given `x` is an integer variable, `random(x)`, is translated to:

```
addr(x)     # push the absolute address of x
call  rand  # rand() returns to %eax
popl  %ecx
movl  %eax, (%ecx)
```

4.2.5 The Object Lock and Waiting Set

ABC Pascal provides shared-variable synchronization. It uses monitors to realize synchronization, in particular mutual exclusion and condition synchronization. There is a lock and a condition variable associated with each object.

The object lock

Mutual exclusion is realized by locks. If we look at an object in isolation, there is an implicit lock associated with the object that ensures exclusive access to its methods. A call to a procedure of the object or an autonomous execution of an action must first acquire the object's lock. That means each method of this object forms a critical section, and each method appears to be atomic (indivisible). Each method will release the object lock upon exit, so that the lock becomes available again.

If one thread (*Thr1*) is holding the object lock and another thread (*Thr2*) attempts to acquire the object lock, *Thr2* will be placed in a waiting queue along with other threads that attempt to grab the lock. Whenever the object lock is released, *Thr2* will have a chance to acquire the object lock.

Methods from the same object cannot call each other. Thus, recursion and mutual recursion are disallowed inside a class definition. From the perspective of a programmer, it may be understood that a thread cannot re-acquire the object lock that it already holds.

ABC Pascal does not prevent, nor require detection of, deadlock situations. Programmers should use conventional techniques for deadlock avoidance.

Thread creation

Since only one action can access an object at a time, there is no need to correspond one thread to each action within an object. It suffices to create one thread (called the object thread) for each active object. Details of this object thread are explained in the next section.

Threads are started after the entire program has been visited by the compiler. The creation of threads for all active objects is implemented by Pthread routine `pthread_create()`.

Waiting and signaling

The guarded expression optional to a method definition is the only built-in construct for conditional synchronization. Conditional synchronization is typically implemented by conditional variables. The implementation details are addressed in the next section. We instead adopt the notions of waiting and signaling to understand how conditional synchronization works in ABC Pascal.

Each object, in addition to having a lock, has an associated waiting set, which is a set of threads. This waiting set is empty when an object is first created.

```
class Host
  var c:integer;
```

```
  procedure leave_room when c=0;
    begin
      { procedure body }
    end;

  procedure reset_room;
    begin
      c:=0;
    end;
begin
  {initialization code}
end;

var butler : Host;
```

Suppose a thread T1 – an action or the program main body in ABC Pascal – calls `butler.leave_room`. According to ABC Pascal's semantic, T1 will first acquire the lock associated to object `butler`. If it successfully grabbed the lock, T1 will start to evaluate the guarded expression, `c=0`. Then, the body of `leave_room` will be executed only if the condition `c=0` holds; otherwise, T1 is to be added to the waiting set, and becomes blocked. The blocked threads of the waiting set are to be unblocked (awakened) by signals. Signals are issued by methods from the same object upon completion of executing the method body. For example, suppose another thread T2 calls `bulter.reset_room` and successfully exits. Upon T2's exit from method `reset_room`, it will signal all threads in the waiting set of object `butler`, including T1. After a signal is sent, every thread in the waiting set for the object is removed from the waiting set and unblocked for thread scheduling.

The unblocked threads will then reacquire the object lock and then reevaluate the guard. Of course, these unblocked threads will not be able to proceed until the current thread releases the object lock.

4.2.6 Mapping Actions-based Concurrency to Pthreads

ABC Pascal's action-based concurrency can be directly implemented with Pthreads operations. In this subsection, we present the rationale for using Pthreads to implement ABC Pascal action. For clarity and readability, some C style pseudo-code is used to represent the target code.

The implementation details for guarded actions and procedures are presented first, and then, the "big picture" of the entire multithreaded execution. Each active object

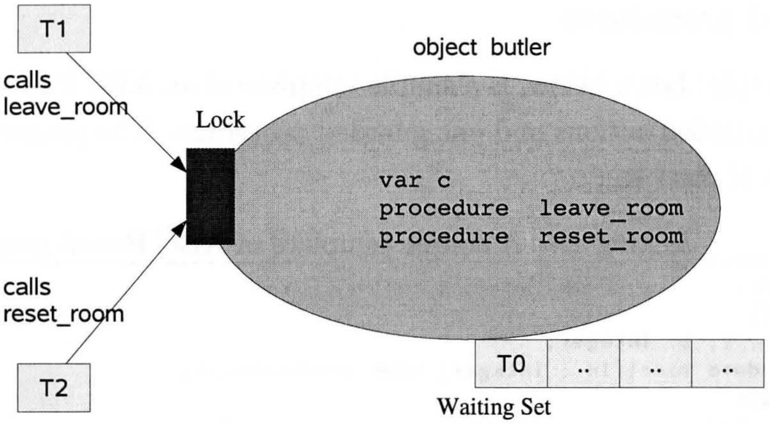

Figure 4.5: Object as an egg-shell

(i.e. it contains at least one action) is associated to an object thread at run-time. This object thread continuously selects enabled actions to run, and therefore never terminates. The entire program terminates only if all object threads are blocked on a condition such that it can no longer find an enabled action to run.

Calling C functions from assembly code

Calling a C function from assembly code is not much different from calling a function implemented in assembly. If the C function being called returns a value other than `void` type, this returned value is by default saved in `eax` register after the function returns.

As making a function call in assembly code, one needs to push the actual paramenters onto the stack before making a C function call. By convention, the last parameter is to be pushed onto stack first, and the first parameter goes last. For example, calling the C function `pthread_cond_init(&mtx,NULL)` can be performed by assembly code:

```
pushl $0    # NULL=0
```

```
addr(mtx)  # push the address of mtx
call  pthread_cond_init
addl  $8, %esp
```

Guarded procedures

The example shown below, is a simple template of an ABC Pascal program. Class `E1` has two guarded actions and one guarded procedure. The program declares an array of object of class `E1`.

Listing 4.4: A simple template of ABC Pascal programs

```
program ex;
  class E1
    var x, y, z: integer;
    procedure proc1(in : integer) when predicate_p1;
      begin
      {... body of proc1 ...}
      end;
    action act1 when predicate_a1;
      begin
      {... body of proc2 ...}
      end;
    action act2 when predicate_a2;
      begin
      {... body of act2 ...}
      end;
    begin
      {the initialization block}
    end;

  var ex: array [1..10] of E1;
begin
  {main body}
end.
```

The guarded procedure `proc1` will be translated to C style pseudo-code shown in Listing 4.5.

Listing 4.5: Translated code for `proc1`

```
void proc1(int in, E1 *e){
  pthread_mutex_lock(&e->obj_mutex);
  while (! predicate_p1){
    pthread_cond_wait(&e->obj_cv , &e->obj_mutex);
  }
  /* the body of proc1 */
  pthread_cond_broadcast(&e->obj_cv);
  pthread_mutex_unlock(&e->obj_mutex);
}
```

As it is shown in Listing 4.5, a thread trying to invoke procedure `proc1` must first acquire the object lock (`obj_mutex`), and then wait on the guard if it evaluates to false. A call to `pthread_cond_wait` will cause the current thread to automatically release the lock and suspend itself by placing it on the waiting set of the object. After the execution of a procedure body, it is necessary to wake up all the threads that are waiting on the objects condition variable (`obj_cv`). The waking-up routine is implemented by calling `pthread_cond_broadcast`. This routine removes all threads in the waiting set and makes them eligible to compete for the lock; thus, the caller does not block on the waking-up routine – it signals and then continues. Because each object has a single conditional variable, all procedure guards and action guards share this condition variable. Therefore, it suffices to wake up all threads waiting on the object's condition variables at the end of each procedure, regardless that the truth value of guarded expressions might be affected by any state changes of the object.

Suppose a blocked thread, T1, awakened by another thread T2, eventually obtains the lock, it will resume its execution immediately after exiting from `pthread_cond_wait`, so T1 will re-enter the loop and reevaluate the negation of the guard [16].

An alternative strategy is to associate each guard to a condition variable, and at the end of each procedure, the current thread only signals the threads from the waiting set, of which the guard associated to it changes from false to true. However, this strategy requires re-evaluation of all guards before sending the signals. It is still not an optimal solution.

Actions and object threads

Because the object thread repeatedly selects an enabled action to run, actions from one object are executed in sequential order. The body of an action is translated to assembly code as if it were an ordinary global procedure; no concurrency aspect needs to be concerned. Thus, we just need one thread of control for each active object, that is the object thread. For example, the object thread constructed for class `E1` from Listing 4.4 is shown in Listing 4.6. However, the solution below does not take account of program termination. The full solution is to be presented later in Listing 4.7.

Listing 4.6: The object thread and main thread

```
void object_thread(El *e) {
  while (true) {
    pthread_mutex_lock(&e->obj_mutex);
    /* search for an enabled action to run */
    if (searching fails) {
      while(!predicate_a1 && !predicate_a2)
        { pthread_cond_wait(&e->obj_cv , &e->obj_mutex);}
    }
    pthread_cond_broadcast(&e->obj_cv);
    pthread_mutex_unlock(&e->obj_mutex);
    }
}

void main_thread {
  /* execute initialization blocks */
  /* create the thread to run main body*/
  /* create all object thread(s) */
  /* wait for the program main body to finish */

  int  gc = num_of_object_thread;

  while (gc > 0) {
    usleep(3); /*sleep for 3 microseconds*/
  }
  return;
}
```

The object thread contains an infinite loop. It never exits, but it will be blocked on the object's condition variable as soon as none of the action is eligible to run. When the object thread is blocked, it implicitly releases the object lock so that other threads may have a chance to modify the object's state.

At the point when all object threads are blocked, the program may terminate. That means we let the main thread exit under this termination condition, without waiting for the object threads to terminate.

Program termination

Recall the two conditions for a program to terminate:

1. the program main body has completes its execution.

2. all actions are being blocked.

The main thread (see Listing 4.6) examines the termination condition by repetitively checking a global counter (gc) shared among the object threads. The main thread terminates when the counter reaches zero. More specifically, we let the main thread exit upon the program terminating condition, without waiting for the object threads to terminate. The initial value of the counter (gc) is the number of object threads. Each object thread (see Listing 4.7) atomically decrements this counter as soon as it cannot find any enabled action to execute, right before calling `pthread_cond_wait`. It atomically increments the counter as soon as it exits from the blocking call, `pthread_cond_wait`.

Listing 4.7: Final version of the object thread

```
void object_thread(El *e) {
  while (true) {
    pthread_mutex_lock(&e->obj_mutex);
    /* search for an enabled action to run */
    if (searching fails) {
      while(!predicate_a1 && !predicate_a2) {
        /*atomic decrement gc; */
        pthread_cond_wait(&e->obj_cv, &e->obj_mutex);
        /*atomic increment gc */
      }
    }
    pthread_cond_broadcast(&e->obj_cv);
    pthread_mutex_unlock(&e->obj_mutex);
  }
}
```

Intel architecture provides *lock* prefix to ensure the prefixed instruction is executed atomically [1]. This is real fine-grained atomicity directly implemented in hardware. For example, with Intel assembler, one can make an atomic increment using lock prefix:

```
lock addl $1, gc
```

Because a read access to a memory location is always atomic at assembly level, there is no need to use the lock prefix in the main thread. To reduce the CPU time consumed in the busy waiting loop, we let the main thread sleep for a few micro seconds every time it enters the while loop. This small change significantly reduced the total CPU time when the number of object threads is not large.

Selection among actions

If two or more guards evaluates to true, the selection among the actions will be nondeterministic. We maintain a circular list of the actions for each object. Once the object thread grabs the lock, it will go one round of the list and try to execute each action in a sequential order. Before the object thread releases the lock, it marks the last attempted action in the circular list, as the anchor. Next time the object thread grabs the lock, it starts the round-robin with the action after the anchor saved from previous round.

```
class X

  action_0 when g_0;
   begin {body} end;
  ... ...
  action_i when g_i;
   begin {body} end;
  ... ...
  action_m when g_m;
   begin {body} end;

begin {initialization} end;
```

Suppose an instance of class X contains $m + 1$ actions. In the first round of round-robin, the object thread starts with the ith action (for $0 < i < m$). Thus, in the next round of round-robin, the attempt starts with the $(i + 1)$th action. This circular list technique is described in C-like code in Listing 4.8. Guards are arranged in an array `guardlist`, and actions are represented by array `actionlist`, where the `guardlist[j]` denotes the guard of the jth action.

Listing 4.8: Selection among actions

```
void object_thread(E *e) {
  next = i;     /* where i is random number between 0 and m*/
  while (true) {
    pthread_mutex_lock(&e->obj_mutex);
    count = m + 1;
    done  = FALSE;
    while (count > 0) {
      switch (next)
      { case 0: if guardlist[0]
                 { done = TRUE;
                   /* execute action_0 */
                 }
                break;
        // ...
        case i: if guardlist[i]
                 { done = TRUE;
                   /* execute action_i */
                 }
                break;
        // ...
```

```
          case m:  if guardlist[m]
                    { done = TRUE;
                      /* execute action_m */
                    }
                   break;
        }
        next = (next+1) % n;
        count--;
      }
      if (!done) {
      while(!guardlist[0] && ... && !guardlist[m-1]) {
          pthread_cond_wait(&e->obj_cv, &e->obj_mutex);
          }
      }
      pthread_cond_broadcast(&e->obj_cv);
      pthread_mutex_unlock(&e->obj_mutex);
  }
}
```

Ideally, we desire a nondeterministic construct to guarantee fairness. Yet, there are several plausible ways that "fairness" might be defined. Weak fairness basically means the scheduling policy guarantees no guard that is always true is always skipped. A stronger notion of fairness, strong fairness, guarantees that no guard that is true infinitely often is always skipped. The current implementation on nondeterminacy conforms to a weak fairness policy.

Chapter 5

Testing Strategy

Testing for ABC Pascals implementation consists of two phases. In phase one, we focus on the functionality of parsing and semantic checking, while the generated code is deliberately ignored. In the second phase, we focus on the correctness of the generated code, which is extremely hard to justify.

As mentioned in the last chapter, the implementation contains four modules: scanner, symbol table, compiler driver, and code generator. The code generator module is tested separately, because when testing against the parsing and semantic checking, the code generator is excluded. During phase one, the first three modules are being tested. As moving on to phase two, most bugs from the first three module are likely to be caught, therefore, the code generator is expected to account for the majority of the bugs.

All compile-time errors and run-time errors the concurrent implementation copes with are illustrated in Appendix A.

5.1 Testing Syntactical and Semantic Analyzer

Syntactical analysis and semantic analysis are performed by the compiler driver. At early stages of testing, all three modules are modified so that the compiler is able to provide explicit information that helps to locate the bugs. For example, the compiler output all identifiers that are processed successfully.

Two testing suites were created — one for syntactical checking, the other for semantics checking. The testing suite (14 programs) created for semantics checking primarily consists of programs with errors that violates typing rules and scoping rules. In contrast, the testing suite (10 programs) developed for syntactical checking consists of valid programs for which we expect the compiler to generate code.

5.2 Testing the Generated Code

When finishing the test against the syntactical checker and semantics checker, we eventually have to examine the generated assembly code. The generated code needs to be processed by assembler and linker, so some obvious bugs may be caught by the assembler. For instance, the assembler will complain if an instruction is not used properly or two labels are given the same name. However, the assembler is unable to catch further bugs.

To make the task slightly easier to handle, this phase is divided into two subphases: validation for the sequential constructs and then validation for concurrent executions. We want to separate the testing for sequential constructs from the rest, given that testing a sequential program is much simpler than multithreaded one. Hence, another testing suite (36 programs) was developed to validate code generated for:

- assignment statement
- boolean expression evaluation
- arithmetic expression evaluation
- if-then-else
- while-do loop
- accessing array elements and record fields
- value, reference parameter passing and procedure calls
- predefined procedures
- combinations of all above

Ideally, we would like to eliminate all bugs with respect to sequential constructs. In practice, this is hard to achieve. The run-time debuggers (e.g. GNU debugger, KDE debugger) are very effective at locating possible bugs for single-threaded programs. However, they are not as effective dealing with concurrent programs, because it is impossible to trace executions of multiple threads simultaneously. Of course, the last but effective option is to trace the execution by hand.

During the last phase of testing, we test the implementation with real examples (10 programs), some of which have been presented in this thesis.

Chapter 6

Performance

In this chapter, we discuss the performance of ABC Pascal programs in comparison to other languages. Although optimization and performance were not the main focus in our prototyped implementation, comparing ABC Pascal's performance against other concurrent programming languages will provide us feedback on how efficient the current implementation is. This feedback can further provide insights on how to enhance the implementation with respect to performance.

Four canonical examples in concurrent programming have been implemented in ABC Pascal, Java, Ada, and C. Because C is a sequential language, the C examples are developed using Pthreads and semaphore libraries.

The metric used to rate the performance is program execution time.

6.1 Coarse-grain measurement

We use the *time* command available on Unix-based systems to take the measurement. The timing results obtained from the time command are coarse-grained measurement, where coarse-grained refers to the measurement resolution. The timing resolution of the time command is one millisecond (10^{-3} second).

```
cuix@mgm:~/> time ./cc
real    0m6.034s
user    0m6.256s
sys     0m3.804s
```

The *time* command returns three values: real time, user time, and system time. Real time is the elapsed time of program execution, which is not useful, because the CPU(s) may be occupied by other threads during the program's execution. Therefore, we are only looking at *user time* and *system time.*

user time The amount of time the CPU was executing the program. Any time spent preempted, blocked for I/O, or running system calls is excluded.

system time The execution time used by the OS while running the program, for example, handling interrupt, I/O, page fault, context switch.

6.2 The Benchmark Test

The four canonical examples used for performance testing are:

- The Car Control problem (CC)
  Cars that are traveling from south to north and the opposite direction share a one-lane bridge. Cars traveling in the same direction can cross the bridge at the same time, but the bridge has a restriction for capacity. In the ABC Pascal implementation, CC.pas, each active object has one action. The number of cars can be set freely.

- The Five Dining Philosophers (DP)
  We employ a simple solution, that is having a butler to ensure at most four philosophers can sit down at the table. In the ABC Pascal implementation, DP.pas, each active object has one action. Yet, the number of philosophers is fixed.

- Simulation of a multiple resource allocator (MRA)
  This program simulates the situation where multiple users are competing for shared resources. The users may require different sets of shared resources. They must first acquire all required resources and then use the resources. In the ABC Pascal implementation, MRA.pas, each active object has multiple actions, and the number of users and resources can be set freely. This is a typical example of having multiple actions in one object.

- The Reader-Writer problem (RW)
  This example allows either concurrent Read or single Write to the data. In the ABC Pascal implementation, RW.pas, each active object has one action. The number of readers and writers can be set freely.

In these examples, the code devoted to computations only accounts for a small portion of the program. In other words, the programs spend the majority of their execution time in synchronizations instead of computations.

For each example, the four implementation versions are carefully developed so that they adhere to the same algorithm and they are idiomatic with respect to the languages' features.

Execution of the tested programs are parameterized by:

N : number of active objects (identical to the number of object thread)

R : number of rounds that a certain task is set to execute.

More specifically, N represents the number of threads created (excluding the thread to execute the program's main body), while R represents the workload assigned to each thread.

To make meaningful observations, N and R are set to some large numbers, except for the Dining philosopher problem where N is fixed at 5. To eliminate the uncertain effect of blocking I/O operations, the programs being measured contain no I/O calls.

Each entry in the benchmark table is filled with the mean and confidence interval (95% CI) of thirty measurements taken in separate rounds. The 95% CI is an interval estimate on the average time of program execution. It specifies the interval in which we have 95% chance that the mean lies. We intend to minimize the measurement side-effect when the cache becomes "hot" towards certain executables.

Car Control(CC), results are in seconds						
	R=5000,N=600		R=10000,N=100		R=500,N=1000	
ABC Pascal	9.08	(8.569, 9.575)	3.27	(3.228, 3.306)	0.85	(0.788, 0.908)
Java	3.60	(3.551, 3.649)	1.27	(1.265, 1.277)	0.92	(0.912,0.934)
Ada	46.75	(42.17, 51.33)	5.87	(5.202, 6.532)	8.77	(8.557, 8.989)
C	3.58	(3.230, 3.922)	1.02	(0.941, 1.095)	0.24	(0.232, 0.242)

Dining Philosophers(DP), results are in seconds

	R=200000,N=5		R=100000,N=5	
ABC Pascal	2.18	(2.068, 2.304)	1.18	(1.117, 1.250)
Java	4.41	(4.303, 4.521)	2.09	(2.021, 2.163)
Ada	5.44	(5.280, 5.598)	2.80	(2.709, 2.889)
C	1.42	(1.370, 1.464)	0.72	(0.694, 0.748)

Multiple Resource Allocator(MRA), results are in seconds

	R=5000,N=500		R=5000,N=100		R=500,N=500	
ABC Pascal	47.43	(42.96, 51.84)	7.12	(7.092, 7.142)	5.38	(5.160, 5.603)
Java	14.60	(14.55, 14.65)	2.95	(2.942, 2.954)	1.69	(1.683, 1.697)
Ada	44.54	(43.37, 45.72)	8.91	(8.870, 8.954)	4.70	(4.663, 4.737)
C	16.60	(15.05, 18.15)	2.13	(2.035, 2.233)	1.95	(1.914, 1.986)

Reader-Writer(RW), results are in seconds

	R=5000,N=600		R=10000,N=100		R=500,N=1000	
ABC Pascal	6.92	(6.240, 7.710)	2.04	(2.002, 2.078)	2.69	(2.591, 2.799)
Java	3.65	(3.606, 3.684)	1.28	(1.271, 1.281)	0.93	(0.918, 0.934)
Ada	16.19	(15.46, 16.91)	5.32	(4.938, 5.710)	3.81	(3.753, 3.861)
C	4.58	(4.247, 4.911)	1.23	(1.187, 1.275)	0.57	(0.458, 0.684)

Java, Ada, and C programs are compiled and executed without specifying any tuning options provided by the vendors. However, by default, some of Java's tuning options are turned on by JVM. For example, `UseBiasedLocking` enables a technique for improving the performance of uncontended synchronization.

From the benchmark table, we make a few interesting observations.

- In the event that each active object contains only one action, ABC Pascal programs perform no worse or even better than the Ada version.
- In the event that an active object contains multiple actions (e.g. MRA.pas), and

both N and R are set to large numbers, performance of ABC Pascal programs lag behind the Ada version, not to mention Java and C.

- As the number of object thread stays small ($N < 100$), or each object thread is not loaded with heavy tasks (R is small), ABC Pascal programs achieve decent performance given that the compiled code is not optimized.
- In general, the performance of C and Java leads the ones of ABC Pascal and Ada, by a sizable factor.
- The Java programs perform better than C in some occasions.

The first three observations provide insights on the possible deficiency in our implementation, and will be addressed later in section 7.1.

The last observation on C and Java will raise some interesting discussions about optimization techniques specialized in speeding up multithreaded programs. First of all, C does not support multiprogramming at its language level. The majority of C compilers will not attempt to optimize with respect to thread synchronizations. Although C compilers usually provide a range of general optimization levels, as well as individual options for specific types of optimization, these optimization options in general have very limited effect on multithreaded C programs implemented with thread libraries or packages.

Contrary to C, Java vendors have always been preoccupied with tuning multithreaded programs. Sun Microsystems launched their first official release of Java HotSpot virtual machine (VM) in 1999. The HotSpot VM has been engineered for maximum performance from the ground up – often providing at least a two-fold increase in speed [2].

The Java HotSpot virtual machine incorporates a breakthrough in thread synchronization which boosts performance by a major factor. In particular, it incorporates techniques for both uncontended and contended synchronization operations which boost synchronization performance by a large factor. The terms uncontended and contended refer to how many threads are operating on a particular lock. A lock that is not held by any thread is an uncontended lock: the first thread that attempts to acquire it immediately succeeds. A contended lock has at least one thread waiting for it; it may have many more. Uncontended synchronization operations, which dynamically comprise the majority of synchronizations, are implemented with constant-time techniques. Contended synchronization operations use advanced adaptive spinning techniques to improve throughput even for applications with significant amounts of lock contention [2]. As a result, synchronization performance becomes so fast that it is not a significant performance issue for the vast majority of Java programs.

Chapter 7

Discussion

Although the first implementation of ABC Pascal has been completed, the research on developing and implementing a new concurrent programming model is very much in progress.

We conclude this thesis by discussions of various design and implementation issues, and give ideas on future directions that ABC Pascal might take.

7.1 Towards a More Efficient Implementation

The first implementation of ABC Pascal is not only an experimental attempt to legitimize our concurrent programming model, it is also meant to identify issues for achieving reasonable efficiency. The benchmark results did reveal some deficiencies in our implementation. More importantly, they show that there is hope to further improve the performance of ABC Pascal.

A potential problem is the execution efficiency under the circumstances that objects contain multiple actions and the number of objects grow large. The C program that simulates the compiler generated code also suffers from the same issue. The bottleneck that sometimes leads to poor performance is the spin-loop in the main thread:

```
while (gc>0){
  usleep(3);
}
```

Although we let the main thread sleep for a couple microseconds in every iteration, it could still waste CPU time while spinning and checking the condition. A more elegant solution is to use semaphores or higher-level mechanisms to check the program termination condition. Unfortunately, our attempt on an alternate semaphore-based solution was not successful.

In terms of memory space occupation, it is possible to estimate and minimize the stack size for each pthread, with the restriction that no method of objects makes calls to a recursive global procedure. The current implementation determines the stack size of each thread by dividing the default stack size by 8. Except for the thread to run the program main body, all other threads are assigned with 1/8 of the default value. If no recursion would occur in the execution of a method, the maximal depth of procedure calls can be determined by the compiler. Hence, it becomes possible to estimate the required stack size for each individual thread. The estimated stack size for an object can be estimated by:

$$\begin{aligned}\text{required size} \approx \text{minimum Pthread stack size} &+ \text{estimated block for stack operations}\\ &+ \text{sum of activation frames}\end{aligned}$$

Another cause of overhead is the intensive use of the stack. The ABC Pascal compiler generates stack-based code for execution. However, stack operations are more expensive than register operations. We here propose a feasible strategy for implementation, if a multiple-pass compiler was ever built for ABC Pascal. The front-end generates stack-based intermediate code which is in turn processed by the back-end. The back-end first transforms the stack-based code to a form of register-based intermediate language, and then performs conventional optimization techniques on this register-based code, such as register allocation.

7.2 Extensions to ABC Pascal

The current ABC Pascal language specification disallows forward references. It also prohibits an action to call procedures in the same object and recursive procedure calls in an object.

The restriction for forward reference can be removed by building a multiple-pass compiler.

In an object, it is useful practice to allow an action to invoke a procedure. To implement this, the lock associated to each object must be a recursive lock, that is it can be locked repeatedly by the owner. The recursive lock may be implemented with Pthread's `PTHREAD_MUTEX_RECURSIVE_NP` option as the mutex variable is initialized.

However, recursive calls within an object have to be treated carefully. The lock doesn't become unlocked until the owner has unlocked for each successful lock request that it has outstanding on the lock. There is another potential danger to overflow the stack assigned to each thread. The stack size assigned to each thread is set before thread creation, and may not be changed dynamically. The handling of stack overflow purely depends on the underlying Pthread implementation. Therefore, we do not intend to add object-wide recursive calls to ABC Pascal, unless there is a more gentle way to handle a stack overflow.

Appendix A

Using the ABC Pascal Compiler

The compiler should be used on Linux. To build the compiler, one needs to place all source files in a one folder, and compiles them with a Pascal compiler to obtain the executable file, intelcompiler. For example, if *freepascal* is the Pascal compiler to be used, just type in command line:

```
fpc intelcompiler.pas
```

To use the compiler to compile an ABC Pascal program, say `test.pas`, one just needs to type:

```
./intelcompiler test.pas
```

After this step, the ABC Pascal compiler has produced assembly code with the same file name, except that the file extension is .s :

```
./intelcompiler test.pas

 Pascal0 Compiler
 ------ compiling >> test
 code generated   190
 Code loaded to: test.s
```

To execute the generated assembly program, one needs to the GNU compiler (gcc) to perform assembling and linking. So compile the generated assembly code with `gcc`, with the -l option to link the pthread library:

```
gcc -lpthread test.s -o test
```

Now, the executable file `test` is ready to run.

Appendix B

Compile-time and Run-time Errors

The compiler is able to catch syntax errors, violations of typing rules, and violations of scoping rules. The compile-time error message includes:

- name undefined
  Attempts to use an identifier that is not declared so far.

- multiple definitions
  Attempts to duplicate declaration of some identifier.

- (array) index is not integer
  Index of an array is not integer type.

- bad type (in expression evaluations)
  Operand is incompatible with the operator.

- incompatible type (in assignment statement)
  Incompatible assignment – type is mismatched.

- bad parameter type
  Actual parameter type does not match its corresponding formal parameter.

- too many/few parameters
  Length of the argument list does not match the length of declared parameters.

- factor is expected
  For the next symbol, parser is expecting a factor. Syntactical rules are voilated.

- ’end’ is expected
  For the next symbol, parser is expecting ’end’.
- an identifier is expected
  For the next symbol, parser is expecting an identifier.
- ’;’ is expected
  For the next symbol, parser is expecting an semicolon.
- class procedure is not defined
  Attempts to call an class procedure that is not defined in that class.
- calling a class procedure here is not allowed
  Methods from the same class can not invoke each other.
- illegal declaration order
  The declarations do not conform to the required order.
- no parameters of structured types can be passed by value
  Can not pass variables of structured types by value.
- global procedures can not have guard
  Trying to declare a global procedure with a guard.
- a class can not have fields of object type
  Can not declare objects as fields of a class. Objects can only be declared globally.
- A method guard can not contain global variables of non-class type
  Variables of non-class types may not appear in a method’s guard.

Three run-time errors can be caught in the current implementation.

- Error: illegal array index
  Array index is out of bound.
- Error: operand invalid
  Operand is not in the valid range of arithmetic operations (−32768..32767).
- Error: thread creation failure
  Fail to create thread at run time.

Appendix C

Canonical Examples

ABC Pascal Examples

Listing C.1: CC.pas

```
program CC;

const
  ROUNDS = 5000;
  CARS_S2N = 250;
  CARS_N2S = 350;
  CAPACITY = 10;

class Bridge
  var s2n, n2s: integer;
  procedure s2n_arrive when (n2s = 0) and (s2n < CAPACITY);
    begin
      s2n := s2n + 1
    end;
  procedure s2n_leave;
    begin
      s2n := s2n - 1
    end;
  procedure n2s_arrive when (s2n = 0) and (s2n < CAPACITY);
    begin
      n2s := n2s + 1
    end;
  procedure n2s_leave;
    begin
      n2s := n2s - 1
    end;
begin
  s2n := 0;
  n2s := 0
```

```
end;

var brg: Bridge;

class Car_s2n
  var round: integer;
  action cross_bridge when round < ROUNDS;
    begin
      brg.s2n_arrive;
      round := round + 1;
      brg.s2n_leave
    end;
begin
  round := 0
end;

class Car_n2s
  var round: integer;
  action pass_bridge when round < ROUNDS;
    begin
      brg.n2s_arrive;
      round := round + 1;
      brg.n2s_leave
    end;
begin
  round := 0;
end;

var cars_to_north: array [1..CARS_S2N] of Car_s2n;
    cars_to_south: array [1..CARS_N2S] of Car_n2s;
begin
 { empty main body}
end.
```

Listing C.2: DP.pas

```
program DP;

const
  ROUNDS = 100000;
  SEATS = 5;

class Fork
  var up: boolean;
  procedure pickup when not up;
    begin
      up := true
    end;
  procedure putdown;
    begin
      up := false
    end;
```

```
begin
  up := false
end;

class Host
  var occupants: integer;
  procedure enter_sitdown when occupants < SEATS - 1;
    begin
      occupants := occupants + 1
    end;
  procedure getup_leave;
    begin
      occupants := occupants - 1
    end;
begin
  occupants := 0
end;

var butler: Host;
    F: array [0..SEATS - 1] of Fork;

class Philosopher
  var seat: integer;
       awake: boolean;
  procedure wakeup(s: integer);
    begin
      seat := s; awake := true
    end;
  action start when awake;
    var r: integer;
    begin
      r := 0;
      while r < ROUNDS do
        begin
          butler.enter_sitdown;
          F[(seat + 1) mod SEATS].pickup;
          F[seat].pickup;
          F[(seat + 1) mod SEATS].putdown;
          F[seat].putdown;
          butler.getup_leave;
          r := r + 1
        end;
      awake := false
    end;
begin
  awake := false
end;

var P: array [0..SEATS - 1] of Philosopher;
    s: integer;

begin
  s := 0;
```

```
  while s < SEATS do
    begin P[s].wakeup(s); s := s + 1 end
end.
```

Listing C.3: MRA.pas

```
program MRA;

const
  ROUNDS = 5000;
  RESOURCES = 10;
  USERS = 500;
  RES_PER_USER = 4;

class Resource
  var avail: boolean;
  procedure acquire when avail;
    begin
      avail := false
    end;
  procedure release;
    begin
      avail := true
    end;
begin
  avail := true
end;

var R: array [0..RESOURCES - 1] of Resource;

class User
  var round: integer;
      needs: array [0..RESOURCES - 1] of boolean;
      d: integer; {idle: -1; acquiring: 0 .. RESOURCES - 1; acquired: RESOURCES}

  action request_resources when (d = -1) and (round < ROUNDS);
    var x, c: integer;
    begin {assign needs[i] randomly}
      c := 0;
      while c < RES_PER_USER do
        begin
          random(x);
          needs[x mod RESOURCES] := true;
          c := c + 1
        end;
      d := 0
    end;
  action acquire_one_resource when (d > -1) and (d < RESOURCES) and (round < ROUNDS);
    begin {acquire resource d if needed}
      if needs[d] then R[d].acquire;
      d := d + 1
    end;
```

```
  action release_resources when (d = RESOURCES) and (round < ROUNDS);
    begin
      while d > 0 do
        begin
          d := d - 1;
          if needs[d] then
            begin
              R[d].release;
              needs[d] := false;
            end;
        end;
      d := -1;
      round := round + 1
    end;
begin
  d := 0;
  while d < RESOURCES do
    begin needs[d] := false; d := d + 1 end;
  round := 0;
  d := 0
end;

var U: array [1..USERS] of User;

begin
end.
```

Listing C.4: RW.pas

```
program RW;

const
  ROUNDS = 5000;
  RD = 350;
  WR = 250;

class RW_arbiter
  var rw: integer; {-1: one writer; 0: idle; > 0: #readers}
  procedure start_read when rw >= 0;
    begin
      rw := rw + 1
    end;
  procedure end_read;
    begin
      rw := rw - 1
    end;
  procedure start_write when rw = 0;
    begin
      rw := -1
    end;
  procedure end_write;
    begin
```

```
      rw := 0
    end;
begin
  rw := 0
end;

var rwa: RW_arbiter;

class Reader
  var round: integer;
  action read_cycle when round < ROUNDS;
    begin
      rwa.start_read;
      {read access}
      rwa.end_read;
      round := round + 1
    end;
begin
  round := 0
end;

class Writer
  var round: integer;
  action write_cycle when round < ROUNDS;
    begin
      rwa.start_write;
      {write access}
      rwa.end_write;
      round := round + 1
    end;
begin
  round := 0
end;

var
  readers: array [1..RD] of Reader;
  writers: array [1..WR] of Writer;

begin
end.
```

Java Examples

Listing C.5: CC.java

```
class Bridge {
  private int s2n = 0; /* s2n >= 0 */
  private int n2s = 0; /* n2s >= 0 */
  /* s2n == 0 || n2s == 0 */
```

```

  public synchronized void s2n_arrive() {
    while (n2s > 0 || s2n == CC.CAPACITY) {
      try {wait();
      } catch (InterruptedException e) {}
    }
    s2n++;
  }
  public synchronized void n2s_arrive() {
    while (s2n > 0 || n2s == CC.CAPACITY) {
      try {wait();
      } catch (InterruptedException e) {}
    }
    n2s++;
  }
  public synchronized void s2n_leave() {
    s2n--;
    notifyAll();
  }
  public synchronized void n2s_leave() {
    n2s--;
    notifyAll();
  }
}

class Car_n2s extends Thread {
  public void run() {
    for (int r = 0; r < CC.ROUNDS; r++) {
      CC.brg.n2s_arrive();
      CC.brg.n2s_leave();
    }
  }
}

class Car_s2n extends Thread {
  public void run() {
    for (int r = 0; r < CC.ROUNDS; r++) {
      CC.brg.s2n_arrive();
      CC.brg.s2n_leave();
    }
  }
}

public class CC {
  static final int ROUNDS = 500;
  static final int CARS_S2N = 250;
  static final int CARS_N2S = 350;
  static final int CAPACITY = 10;

  static Bridge brg = new Bridge();

  public static void main(String[] args) {
    Car_n2s[] cars_to_south = new Car_n2s[CARS_N2S];
```

```
    Car_s2n[] cars_to_north = new Car_s2n[CARS_S2N];
    for (int j = 0; j < CARS_N2S; j++)
      cars_to_south[j] = new Car_n2s();
    for (int j = 0; j < CARS_S2N; j++)
      cars_to_north[j] = new Car_s2n();
    for (int j = 0; j < CARS_N2S; j++)
      cars_to_south[j].start();
    for (int j = 0; j < CARS_S2N; j++)
      cars_to_north[j].start();
  }
}
```

Listing C.6: DP.java

```
class Fork {
  private boolean up = false;

  synchronized void pickup() {
    while (up) {
      try {wait();
      } catch (InterruptedException e) {}
    }
    up = true;
  }
  synchronized void putdown() {
    up = false;
    notifyAll();
  }
}

class Host {
  private int occupants = 0;

  synchronized void enter_sitdown() {
    while (occupants == DP.SEATS - 1) {
      try {wait();
      } catch (InterruptedException e) {}
    }
    occupants++;
  }
  synchronized void getup_leave() {
    occupants--;
    notifyAll();
  }
}

class Philosopher extends Thread {
  private int seat;
  private Host butler;
  private Fork[] f;

  public void run() {
```

```
      for (int r = 0; r < DP.ROUNDS; r++) {
        butler.enter_sitdown();
        f[(seat + 1) % DP.SEATS].pickup();
        f[seat].pickup();
        /* eat */
        f[(seat + 1) % DP.SEATS].putdown();
        f[seat].putdown();
        butler.getup_leave();
      }
    }
    Philosopher(int s, Host b, Fork[] f) {
      seat = s;
      butler = b;
      this.f = f;
    }
}

public class DP {
    static final int ROUNDS = 50000;
    static final int SEATS = 5;

    public static void main(String[] args) {
      Philosopher[] p = new Philosopher[SEATS];
      Host butler = new Host();
      Fork[] f = new Fork[SEATS];
      for (int s = 0; s < SEATS; s++) {
        f[s] = new Fork();
        p[s] = new Philosopher(s, butler, f);
      }
      for (int s = 0; s < SEATS; s++)
        p[s].start();
    }
}
```

Listing C.7: MRA.java

```
class Resource {
    private boolean avail = true;

    synchronized void acquire() {
      while (!avail) {
        try {wait();
        } catch (InterruptedException e) {}
      }
      avail = false;
    }
    synchronized void release() {
      avail = true;
      notifyAll();
    }
}
```

```
class User extends Thread {
  private int d = 0;
  private boolean[] needs = new boolean[MRA.RESOURCES];
  Resource[] r;

  User(Resource[] r) {
    this.r = r;
    for (int i = 0; i < MRA.RESOURCES; i++)
      needs[i] = false;
  }
  void request_resources() {
    for (int c = 0; c < MRA.RES_PER_USER; c++) {
      int x = (int) (Math.random() * MRA.RESOURCES);
      needs[x] = true;
    }
  }
  void acquire_one_resource() {
    if (needs[d]) r[d].acquire();
    d++;
  }
  void release_resources() {
    while (d > 0) {
      d--;
      if (needs[d]) {
        r[d].release();
        needs[d] = false;
      }
    }
  }
  public void run() {
    for (int i = 0; i < MRA.ROUNDS; i++) {
      request_resources();
      while (d < MRA.RESOURCES) acquire_one_resource();
      release_resources();
    }
  }
}

public class MRA {
  static final int ROUNDS = 5000;
  static final int RESOURCES = 10;
  static final int USERS = 500;
  static final int RES_PER_USER = 4;

  public static void main(String[] args) {
    Resource[] r = new Resource[RESOURCES];
    User[] u = new User[USERS];
    for (int i = 0; i < RESOURCES; i++)
      r[i] = new Resource();
    for (int i = 0; i < USERS; i++)
      u[i] = new User(r);
    for (int i = 0; i < USERS; i++)
      u[i].start();
```

```
  }
}
```

Listing C.8: RW.java

```
class RW_arbiter {
  private int rw = 0; // -1: one writer; 0: idle; > 0: #readers

  public synchronized void start_read() {
    while (rw < 0) {
      try {wait();
      } catch (InterruptedException e) {}
    }
    rw++;
  }
  public synchronized void end_read() {
    rw--;
    notifyAll();
  }
  public synchronized void start_write() {
    while (rw != 0) {
      try {wait();
      } catch (InterruptedException e) {}
    }
    rw = -1;
  }
  public synchronized void end_write() {
    rw = 0;
    notifyAll();
  }
}

class Reader extends Thread {
  RW_arbiter arbiter;

  public Reader(RW_arbiter a) {
    arbiter = a;
  }
  public void run() {
    for (int r = 0; r < RW.ROUNDS; r++) {
      arbiter.start_read();
      // read access
      arbiter.end_read();
    }
  }
}

class Writer extends Thread {
  RW_arbiter arbiter;

  public Writer(RW_arbiter a) {
    arbiter = a;
```

```
  }
  public void run() {
    for (int r = 0; r < RW.ROUNDS; r++) {
      arbiter.start_write();
      // write access
      arbiter.end_write();
    }
  }
}

public class RW {
  static final int ROUNDS = 500;
  static final int RD = 350; //350
  static final int WR = 250;

  public static void main(String[] args) {
    RW_arbiter arbiter = new RW_arbiter();
    Reader[] readers = new Reader[RD];
    Writer[] writers = new Writer[WR];
    for (int j = 0; j < RD; j++) {
      readers[j] = new Reader(arbiter);
    }
    for (int j = 0; j < WR; j++) {
      writers[j] = new Writer(arbiter);
    }
    for (int j = 0; j < RD; j++) {
      readers[j].start();
    }
    for (int j = 0; j < WR; j++) {
      writers[j].start();
    }
  }
}
```

Ada Examples

Listing C.9: CC.adb

```
procedure CC is
  ROUNDS: constant := 500;
  CARS_S2N: constant := 250;
  CARS_N2S: constant := 350;
  CAPACITY: constant := 10;

  protected type Bridge is
    entry s2n_arrive;
    entry n2s_arrive;
    procedure n2s_leave;
    procedure s2n_leave;
```

```
  private
    s2n: integer := 0;
    n2s: integer := 0;
  end Bridge;

  protected body Bridge is
    entry s2n_arrive
      when n2s = 0 and s2n < CAPACITY is
    begin
      s2n := s2n + 1;
    end s2n_arrive;
    procedure s2n_leave is
    begin
      s2n := s2n - 1;
    end s2n_leave;
    entry n2s_arrive
      when s2n = 0 and n2s < CAPACITY is
    begin
      n2s := n2s + 1;
    end n2s_arrive;
    procedure n2s_leave is
    begin
      n2s := n2s - 1;
    end n2s_leave;
  end Bridge;

  brg : Bridge;

  task type Car_s2n ;

  task body Car_s2n is
  begin
    for r in 1 .. ROUNDS loop
      brg.s2n_arrive;
      brg.s2n_leave;
    end loop;
  end Car_s2n;

  task type Car_n2s ;

  task body Car_n2s is
  begin
    for r in 1 .. ROUNDS loop
      brg.n2s_arrive;
      brg.n2s_leave;
    end loop;
  end Car_n2s;

  C1: array(1..CARS_S2N) of Car_s2n;
  C2: array(1..CARS_N2S) of Car_n2s;

begin
  null;
```

```
end CC;
```

Listing C.10: DP.adb

```
procedure DP is
  ROUNDS: constant := 50000;
  SEATS: constant := 5;
  type Seat_Index is mod SEATS;

  task type Philosopher is
    entry start (s: Seat_Index);
  end Philosopher;

  protected type Fork is
    entry pickup;
    procedure putdown;
  private
    up: Boolean := False;
  end Fork;

  protected type Host is
    entry enter_sitdown;
    procedure getup_leave;
  private
    occupants: Natural := 0;
  end Host;

  P: array (Seat_Index) of Philosopher;
  F: array (Seat_Index) of Fork;
  butler: Host;

  task body Philosopher is
    seat: Seat_Index;
  begin
    accept start (s: Seat_Index) do
      seat := s;
    end;
    for round in 1 .. ROUNDS loop
      butler.enter_sitdown;
      F(seat + 1).pickup;
      F(seat).pickup;
      F(seat + 1).putdown;
      F(seat).putdown;
      Butler.getup_leave;
    end loop;
  end Philosopher;

  protected body Fork is
    entry pickup when not up is
    begin
      up := True;
    end pickup;
```

```ada
    procedure putdown is
    begin
      up := False;
    end putdown;
  end Fork;

  protected body Host is
    entry enter_sitdown when occupants < SEATS - 1 is
    begin
      occupants := occupants + 1;
    end enter_sitdown;
    procedure getup_leave is
    begin
      occupants := occupants - 1;
    end getup_leave;
  end Host;

begin
  for s in Seat_Index loop
    P(s).Start(s);
  end loop;
end DP;
```

Listing C.11: MRA.adb

```ada
with Ada.Numerics.Discrete_Random;
procedure MRA is
  ROUNDS : constant := 500;
  RESOURCES: constant := 10;
  USERS: constant := 500;
  RES_PER_USER: constant := 4;

  subtype Res_Index is Integer range 0 .. RESOURCES - 1;

  package Random_Res is new Ada.Numerics.Discrete_Random (Res_Index);
  G: Random_Res.Generator;

  protected type Resource is
    entry acquire;
    procedure release;
  private
    avail: Boolean := True;
  end Resource;

  protected body Resource is
    entry acquire when avail is
    begin
      avail := False;
    end acquire;
    procedure release is
    begin
      avail := True;
```

```
    end release;
  end Resource;

  R: array (Res_Index) of Resource;

  task type User;
  task body User is
    d: Integer := 0;
    needs: array (Res_Index) of Boolean := (Others => False);

    procedure request_resources is
      x: Res_Index;
    begin
      Random_Res.Reset(G);
      for c in 1 .. RES_PER_USER loop
        x := Random_Res.Random(G);
        Needs(x):= True;
      end loop;
    end request_resources;
    procedure acquire_one_resource is
    begin
       if needs(d) then R(d).acquire;
       end if;
       d := d + 1;
    end acquire_one_resource;
    procedure release_resources is
    begin
      while d > 0 loop
        d := d - 1;
        if needs(d) then
          R(d).release;
          needs(d) := False;
        end if;
      end loop;
    end release_resources;
  begin
    Random_Res.Reset(G);  -- reset the random number generator
    for i in 1 .. ROUNDS loop
      request_resources;
      while d < RESOURCES loop
        acquire_one_resource;
      end loop;
      release_resources;
    end loop;
  end User;

  U: array (1..USERS) of User;

begin
  null;
end MRA;
```

Listing C.12: RW.adb

```
procedure RW is
 ROUNDS: constant := 500;
 RD: constant := 350;
 WR: constant := 250;

  protected type RW_arbiter is
    entry start_read;
    procedure end_read;
    entry start_write;
    procedure end_write;
  private
    rw: integer := 0;
  end RW_arbiter;

  protected body RW_arbiter is
    entry start_read when rw >= 0 is
    begin
      rw := rw + 1;
    end start_read;
    procedure end_read is
    begin
      rw := rw - 1;
    end end_read;
    entry start_write when rw = 0 is
    begin
      rw := -1;
    end start_write;
    procedure end_write is
    begin
      rw := 0;
    end end_write;
  end RW_arbiter;

  arb : RW_arbiter;

  task type Reader;
  task body Reader is
  begin
    for r in 1 .. ROUNDS loop
      arb.start_read;
      arb.end_read;
    end loop;
  end Reader;

  task type Writer;
  task body Writer is
  begin
    for r in 1 .. ROUNDS loop
      arb.start_write;
      arb.end_write;
    end loop;
```

```
  end Writer;

  readers: array (1..RD) of Reader;
  writers: array (1..WR) of Writer;

begin
  null;
end RW;
```

C/Pthreads Examples

Listing C.13: CC.c

```
#include <pthread.h>
#define ROUNDS 500
#define CARS_S2N 250
#define CARS_N2S 350
#define CAPACITY 10

typedef struct {
  int s2n, n2s;
  /* s2n >= 0 && n2s >= 0 && (s2n == 0 || n2s == 0) */
  pthread_mutex_t mutex;
  pthread_cond_t s2n_cv; /* n2s == 0 && s2n < CAPACITY */
  pthread_cond_t n2s_cv; /* s2n == 0 && n2s < CAPACITY */
} Bridge;

void s2n_arrive(Bridge *b) {
  pthread_mutex_lock(&b->mutex);
  while (b->n2s > 0 || b->s2n == CAPACITY)
    pthread_cond_wait(&b->s2n_cv, &b->mutex);
  b->s2n++;
  pthread_mutex_unlock(&b->mutex);
}

void s2n_leave(Bridge *b) {
  pthread_mutex_lock(&b->mutex);
  if (b->s2n == CAPACITY) /* signal not full */
    pthread_cond_signal(&b->s2n_cv);
  b->s2n--;
  if (b->s2n == 0) /* broadcast free */
    pthread_cond_broadcast(&b->n2s_cv);
  pthread_mutex_unlock(&b->mutex);
}

void n2s_arrive(Bridge *b) {
  pthread_mutex_lock(&b->mutex);
  while (b->s2n > 0 || b->n2s == CAPACITY)
    pthread_cond_wait(&b->n2s_cv, &b->mutex);
```

```
  b->n2s++;
  pthread_mutex_unlock(&b->mutex);
}

void n2s_leave(Bridge *b) {
  pthread_mutex_lock(&b->mutex);
  if (b->n2s == CAPACITY) { /* signal not full */
    pthread_cond_signal(&b->n2s_cv);
  }
  b->n2s--;
  if (b->n2s == 0) { /* broadcast empty */
    pthread_cond_broadcast(&b->s2n_cv);
  }
  pthread_mutex_unlock(&b->mutex);
}

void bridge_init(Bridge *b) {
  b->s2n = 0;
  b->n2s = 0;
  pthread_mutex_init(&b->mutex, NULL);
  pthread_cond_init(&b->s2n_cv, NULL);
  pthread_cond_init(&b->n2s_cv, NULL);
}

Bridge brg;

void *S2N_cross(void *arg) {
  int r;
  for (r = 0;  r < ROUNDS;  r++) {
    s2n_arrive(&brg);
    s2n_leave(&brg);
  }
}

void *N2S_cross(void *arg) {
  int r;
  for (r = 0;  r < ROUNDS;  r++) {
    n2s_arrive(&brg);
    n2s_leave(&brg);
  }
}

int main() {
  pthread_t cars_to_south[CARS_N2S];
  pthread_t cars_to_north[CARS_S2N];
  pthread_attr_t attr;
  int stack_size;
  pthread_attr_init(&attr);
  pthread_attr_getstacksize(&attr, &stack_size);
  pthread_attr_setstacksize(&attr, stack_size/8);

  bridge_init(&brg);
  int i;
```

```
  for (i = 0; i < CARS_N2S; i++)
    pthread_create(&cars_to_south[i], &attr, N2S_cross, NULL);
  for (i = 0; i < CARS_S2N; i++)
    pthread_create(&cars_to_north[i], &attr, S2N_cross, NULL);
  for (i = 0; i < CARS_N2S; i++)
    pthread_join(cars_to_south[i], NULL);
  for (i = 0; i < CARS_S2N; i++)
    pthread_join(cars_to_north[i], NULL);
}
```

Listing C.14: DP.c

```
#include <pthread.h>
#define TRUE 1
#define FALSE 0

#define ROUNDS 50000
#define SEATS 5

typedef struct {
  int up; /* boolean */
  pthread_mutex_t mutex;
  pthread_cond_t forkdown;
} Fork;

void pickup(Fork *f) {
  pthread_mutex_lock(&f->mutex);
  while (f->up)
    pthread_cond_wait(&f->forkdown, &f->mutex);
  f->up = TRUE;
  pthread_mutex_unlock(&f->mutex);
}

void putdown(Fork *f) {
  pthread_mutex_lock(&f->mutex);
  f->up = FALSE;
  pthread_mutex_unlock(&f->mutex);
  pthread_cond_signal(&f->forkdown);
}

void fork_init(Fork *f) {
  f->up = FALSE;
  pthread_mutex_init(&f->mutex, NULL);
  pthread_cond_init(&f->forkdown, NULL);
}

typedef struct {
  int occupants;
  pthread_mutex_t mutex;
  pthread_cond_t notfull;
} Host;
```

```
void enter_sitdown(Host *h) {
  pthread_mutex_lock(&h->mutex);
  while (h->occupants == SEATS - 1)
    pthread_cond_wait(&h->notfull, &h->mutex);
  h->occupants++;
  pthread_mutex_unlock(&h->mutex);
}

void getup_leave(Host *h) {
  pthread_mutex_lock(&h->mutex);
  h->occupants--;
  pthread_mutex_unlock(&h->mutex);
  pthread_cond_signal(&h->notfull);
}

void host_init(Host *h) {
  h->occupants = 0;
  pthread_mutex_init(&h->mutex, NULL);
  pthread_cond_init(&h->notfull, NULL);
}

Fork F[SEATS];
Host butler;

void *Philosopher(void *arg) {
  int seat = (int) arg;
  int r;
  for (r = 0; r < ROUNDS; r++) {
    enter_sitdown(&butler);
    pickup(&F[seat]);
    pickup(&F[(seat + 1) % SEATS]);
    /* eat */
    putdown(&F[seat]);
    putdown(&F[(seat + 1) % SEATS]);
    getup_leave(&butler);
  }
}

int main() {
  pthread_t p[SEATS];
  pthread_attr_t attr;
  int stack_size;

  pthread_attr_init(&attr);
  pthread_attr_getstacksize(&attr, &stack_size);
  pthread_attr_setstacksize(&attr, stack_size/8);

  int s;
  for (s = 0; s < SEATS; s++) fork_init(&F[s]);
  host_init(&butler);
  for (s = 0; s < SEATS; s++)
    pthread_create(&p[s], &attr, Philosopher, (void *) s);
  for (s = 0; s < SEATS; s++)
```

```
    pthread_join(p[s], NULL);
}
```

Listing C.15: MRA.c

```
#include <pthread.h>
#include <stdlib.h>
#define TRUE 1
#define FALSE 0
#define USERS 500
#define ROUNDS 5000
#define RESOURCES 10
#define RES_PER_USER 4

typedef struct {
  int avail; /* boolean */
  pthread_mutex_t mutex;
  pthread_cond_t cv;
} Resource;

void acquire(Resource *r){
  pthread_mutex_lock(&r->mutex);
  while (!r->avail)
    pthread_cond_wait(&r->cv, &r->mutex);
  r->avail = FALSE;
  pthread_cond_broadcast(&r->cv);
  pthread_mutex_unlock(&r->mutex);
}

void release(Resource *r) {
  pthread_mutex_lock(&r->mutex);
  r->avail = TRUE;
  pthread_cond_broadcast(&r->cv);
  pthread_mutex_unlock(&r->mutex);
}

void resource_init(Resource *r) {
  r->avail = TRUE;
  pthread_mutex_init(&r->mutex, NULL);
  pthread_cond_init(&r->cv, NULL);
}

Resource R[RESOURCES];

typedef struct {
  int d;
  int needs[RESOURCES];
} User;

void request_resources(User *u) {
  int c;
  for (c = 0; c < RES_PER_USER; c++) {
```

```
    int x = rand() % RESOURCES;
    u->needs[x] = TRUE;
  }
}

void acquire_one_resource(User *u) {
  if (u->needs[u->d]) acquire(&R[u->d]);
  u->d++;
}

void release_resources(User *u) {
  while (u->d > 0) {
    u->d--;
    if (u->needs[u->d]) {
      release(&R[u->d]);
      u->needs[u->d] = FALSE;
    }
  }
}

void user_init(User *u) {
  int i;
  u->d = 0;
  for (i = 0; i < RESOURCES; i++)
    u->needs[i] = FALSE;
}

void *user_main(void *arg) {
  User *u = (User *) arg;
  int i;
  for (i = 0; i < ROUNDS; i++) {
    request_resources(u);
    while (u->d < RESOURCES) acquire_one_resource(u);
    release_resources(u);
  }
}

User user_data[USERS];

int main() {
  pthread_t user_thread[USERS];
  pthread_attr_t attr;
  int stack_size;

  pthread_attr_init(&attr);
  pthread_attr_getstacksize(&attr, &stack_size);
  pthread_attr_setstacksize(&attr, stack_size/16);

  srand(time(0));
  int i;
  for (i = 0; i < RESOURCES; i++)
    resource_init(&R[i]);
  for (i = 0; i < USERS; i++)
```

```
    user_init(&user_data[i]);
  for(i = 0; i < USERS; i++)
    pthread_create(&user_thread[i], &attr, user_main, (void *)&user_data[i]);
  for(i = 0; i < USERS; i++)
    pthread_join(user_thread[i], NULL);
}
```

Listing C.16: RW.c

```
#include <pthread.h>
#define ROUNDS 500
#define RD 350
#define WR 250

typedef struct {
  int rw; /* -1: one writer; 0: idle; > 0: #readers */
  pthread_mutex_t mutex;
  pthread_cond_t cv;
} rw_arbiter;

void rw_arbiter_init(rw_arbiter *a) {
  a->rw = 0;
  pthread_mutex_init(&a->mutex, NULL);
  pthread_cond_init(&a->cv, NULL);
}

void start_read(rw_arbiter *a) {
  pthread_mutex_lock(&a->mutex);
  while (a->rw < 0)
    pthread_cond_wait(&a->cv, &a->mutex);
  a->rw++;
  pthread_mutex_unlock(&a->mutex);
}

void end_read(rw_arbiter *a) {
  pthread_mutex_lock(&a->mutex);
  a->rw--;
  pthread_cond_broadcast(&a->cv);
  pthread_mutex_unlock(&a->mutex);
}

void start_write(rw_arbiter *a) {
  pthread_mutex_lock(&a->mutex);
  while (a->rw != 0)
    pthread_cond_wait(&a->cv, &a->mutex);
  a->rw = -1;
  pthread_mutex_unlock(&a->mutex);
}

void end_write(rw_arbiter *a) {
  pthread_mutex_lock(&a->mutex);
  a->rw = 0;
```

```
  pthread_cond_broadcast(&a->cv);
  pthread_mutex_unlock(&a->mutex);
}

rw_arbiter arbiter;

void *Reader(void *arg) {
  int r;
  for (r = 0; r < ROUNDS; r++) {
    start_read(&arbiter);
    /* read access */
    end_read(&arbiter);
  }
}

void *Writer(void *arg) {
  int r;
  for (r = 0; r < ROUNDS; r++) {
    start_write(&arbiter);
    /* write access */
    end_write(&arbiter);
  }
}

int main() {
  pthread_t writers[WR];
  pthread_t readers[RD];
  pthread_attr_t attr;
  int stack_size;

  pthread_attr_init(&attr);
  pthread_attr_getstacksize(&attr, &stack_size);
  pthread_attr_setstacksize(&attr, stack_size/8);

  rw_arbiter_init(&arbiter);
  int i;
  for (i = 0; i < WR; i++)
    pthread_create(&writers[i], &attr, Writer, NULL);
  for (i = 0; i < RD; i++)
    pthread_create(&readers[i], &attr, Reader, NULL);
  for (i = 0; i < WR; i++)
    pthread_join(writers[i], NULL);
  for (i = 0; i < RD; i++)
    pthread_join(readers[i], NULL);
}
```

Appendix D

Source Code

Listing D.1: intelcompiler.pas

```
(*
 ******************************************************************************
 Author: Xiao-lei Cui .
 Date  : Nov, 2008.
 Remark: This Pascal0 comopiler generates assembly code for a Intel 32-bit assembler.

 To build and run the compiler :
   > fpc intelcompiler.pas
   > ./intelcompiler fid.pas

 To compile and link the asembly program with GCC:
   $ gcc -lpthread fid.s -o fid
   $ ./fid
 OR use the gnu assembler:
   $ as -o fid.o  fid.s
   $ ld -o fid  fid.o
   $ ./fid
 ******************************************************************************
 *)
program IntelCompiler (input, output);
  {This is the compile-driver of our single-pass compiler.
   It is comprised by syntax analyzer and semantic analyzer.
   This module uses the other three modules: scanner08, symboltable08, intelgenerator .
  }

uses scanner, symboltable, IntelGenerator;

  const
    WordSize = 4;  { one word = 4 bytes}
    {the sizes for Pthreads objects are the size needed on Linux; the sizes may vary for other OS}
```

```
  sizeof_pthread_mutex = 24;
  sizeof_pthread_cv = 48;

  {first/follow sets}
  MoreExp = [EqlSym, NeqSym, LssSym, GeqSym, LeqSym, GtrSym];
  MoreSimpleExp = [PlusSym, MinusSym, OrSym];
  MoreTerm = [TimesSym, DivSym, ModSym, AndSym];
  FirstFactor = [IdentSym, NumberSym, LparenSym, NotSym];
  FollowFactor = [TimesSym, DivSym, ModSym, AndSym, OrSym, PlusSym,
    MinusSym, EqlSym, NeqSym, LssSym, LeqSym, GtrSym, GeqSym, CommaSym,
    SemicolonSym, ThenSym, ElseSym, RparenSym, DoSym, PeriodSym, EndSym];
  DeclSyms = [ConstSym, TypeSym, VarSym, ProcedureSym, MonitorclassSym];
  StrongSyms = [ConstSym, TypeSym, VarSym, ProcedureSym, WhenSym, MonitorclassSym,
      WhileSym, IfSym,
    BeginSym, EofSym];
  FirstStatement = [IdentSym, IfSym, WhileSym, BeginSym];
  FollowStatement = [SemicolonSym, EndSym, ElseSym, BeginSym];
  FirstType = [IdentSym, RecordSym, ArraySym];
  FollowType = [SemicolonSym];
  FollowDecl = [BeginSym, EndSym, ProcedureSym, EofSym];
  FollowProcCall = [SemicolonSym, EndSym, ElseSym, IfSym, WhileSym];
  FollowMclassDeclaration = [ProcedureSym, BeginSym, EndSym, WhenSym];

{A statement may have three mode:
 - Normal (main body, ordinary procedure body)
 - ClassProc ( class procedures or actions )
 - ClassInit ( class initialization block)
}
type Statement_code_gen_mode = ( Normal, ClassProc, ClassInit );

procedure expression(c_mode: Statement_code_gen_mode; var x: Item; g: boolean);
    forward;
procedure statement(code_gen_mode: Statement_code_gen_mode); forward;
procedure selector(code_gen_mode: Statement_code_gen_mode; var x: Item; LeftSide:
    boolean); forward;
procedure ProcedureDecl(var g: boolean; var pid: Identifier; monitor_proc:boolean;
    classid:Identifier; mutex, cv: Objct); forward;
procedure Monitor_decl; forward;

procedure factor(c_mode: Statement_code_gen_mode; var x: Item; g: boolean);
var obj: Objct;
begin {sync}
  if not (sym in FirstFactor) then
    begin Mark('factor?');
      repeat GetSym until sym in FirstFactor + StrongSyms + FollowFactor
    end;
  if sym = IdentSym then
    begin
      find(obj);
      {if ((c_mode<>Normal)and(obj^.lev=0)and((obj^.tp^.form<>Monitor)or(obj^.tp^.
          base<>nil) and (obj^.tp^.base^.form<>Monitor))and(obj^.Cls<>ConstClass))
          then
        Mark('Can not access global variable of non-class type.');
```

```
      }
      if (g and (obj^.lev=0) and ((obj^.tp^.form<>Monitor)or((obj^.tp^.base<>nil)
          and (obj^.tp^.base^.form<>Monitor))) and (obj^.Cls<>ConstClass)) then
         Mark('A method guard can not contain global variables of non-class type.')
           ;
      GetSym; MakeItem(x, obj);
      selector(c_mode, x, false);
               {leftside=false, meaning that x is on the right side of assignment
                   statment}
      if x.mode <> ConstClass then LoadItem_32(x, false);
                                        {leaveaddress=false: meaning that place val
                                            (x) on stack}
    end
  else if sym = NumberSym then
    begin MakeConstItem(x, intType, v); GetSym end
                      {v is the numeric value of the numberSym, defined in
                          scanner08}
  else if sym = LparenSym then
    begin
      GetSym; expression(Normal, x, g);
      if sym = RparenSym then GetSym else Mark (')?')
    end
  else if sym = NotSym then
    begin GetSym; factor(c_mode, x, g); Op1_32(NotSym, x) end
  else begin Mark('factor?'); MakeItem(x, guard) end
end;

procedure term(c_mode: Statement_code_gen_mode; var x: Item; g: boolean);
var y: Item; op: Symbol;
begin factor(c_mode, x, g);
  while sym in moreTerm do
    begin
      op := sym; GetSym;
      if op = AndSym then Op1_32(op, x);
      factor(c_mode, y, g); Op2_32(op, x, y)
    end
end;

procedure SimpleExpression(c_mode: Statement_code_gen_mode; var x: Item; g: boolean
  );
var y: Item; op: Symbol;
begin
  if sym = PlusSym then begin GetSym; term(c_mode, x, g) end
  else if sym = MinusSym then
    begin GetSym; term(c_mode, x, g); Op1_32(MinusSym, x) end
  else term(c_mode, x, g);
  while sym in moreSimpleExp do
    begin op := sym; GetSym;
      if op = OrSym then Op1_32(op, x);
      term(c_mode, y, g); Op2_32(op, x, y)
    end
end;
```

```
procedure expression(c_mode: Statement_code_gen_mode; var x: Item; g: boolean);
  var y: Item; op: Symbol;
begin
  SimpleExpression(c_mode, x, g);
  if sym in MoreExp then
    begin
      op := sym; GetSym;
      SimpleExpression(c_mode, y, g); Relation_32(op, x, y)
    end;
end;

procedure param(c_mode: Statement_code_gen_mode; var fp: Objct); {process one
    actual parameter}
  var x: Item;
begin
  expression(c_mode, x, false);
  {# Note: expression(x) is the last and only last proc called before Parameter_32
      (,,).}
  {# Hence,the steps for backrolling in Parameter_32(,,) should change accordingly
      if Loaditem_32 changes}
  if IsParam(fp) then
    begin Parameter_32(x, fp^.tp, fp^.cls); fp := fp^.next end
  else Mark ('too many parameters')
end;

procedure CompoundStatement(c_mode: Statement_code_gen_mode);
begin
  if sym <> EndSym then {don't try when CompoundStatement is empty}
    begin
      statement(c_mode);
      while sym <> EndSym do
        begin
          if sym = SemicolonSym then {skip}
            repeat GetSym until sym <> SemicolonSym
          else mark (';?');
          if sym <> EndSym then statement(c_mode);
          if sym in DeclSyms then
            begin mark ('end?'); break end
        end;
    end;
end;

procedure statement(code_gen_mode: Statement_code_gen_mode);
var
  par, obj, temp, mproc_obj: Objct;
  x, y: Item; L: longint;
  temp_str, temp_id: Identifier;
  param_size: longint;

  procedure sparam(c_mode:Statement_code_gen_mode ;var x: Item; WhichCall : longint
      );
                  {for standard (pre-defined) procedure's parameters}
  begin
```

```
    if sym = LparenSym then GetSym else Mark ('(?');
    if WhichCall = 2 then expression(c_mode, x, false)
       {expression(x) will eventually leave the value of item x on the stack. }
    else
      begin  {# read(x), random(x) }
        if sym = IdentSym then
          begin
            find (obj); GetSym; MakeItem (x, obj); selector(c_mode, x, false);
            if x.mode <> ConstClass then LoadItem_32 (x, true)
            {## we need to leave the address of x on stack }
          end
        else Mark('read(): expects an identifier argument.');
      end;

    if sym = RparenSym then GetSym else Mark (')?')
  end;

begin { Statement }
  obj := guard;
  if not (sym in FirstStatement) then
    begin Mark ('statement?');
      repeat GetSym
      until sym in FirstStatement + StrongSyms + FollowStatement
     end;
  if sym = IdentSym then
    begin
      find (obj);
      MakeItem (x, obj);
      GetSym;
      if (obj^.tp<>nil) and ((obj^.tp^.form = Monitor) or ((obj^.tp^.form=Arry) and
         (obj^.tp^.base^.form=Monitor))) then
      { obj is a monitor variable OR an array of monitor variable}
        begin
          if obj^.tp^.form = Monitor then
            begin
              if sym<>PeriodSym then Mark(' . is expected here.')
              else
                 begin {if sym=periodsym}
                   GetSym;
                   if sym = IdentSym then
                     begin
                       temp:=x.tp^.fields;
                       FindField(mproc_obj, temp);
                       GetSym;
                       if (mproc_obj<>guard) and (mproc_obj^.cls=ProcClass)  then
                         begin
                           par := mproc_obj^.dsc;
                           param_size :=0;     {initiliaze param_size}
                           if sym = LparenSym then
                             begin
                               GetSym;
                               if sym = RparenSym then GetSym
                               else
```

```
                           while true do
                             begin
                               param (code_gen_mode, par);
                               {knowing that parameters have uniform size,
                                   which is 4}
                               param_size := param_size + 4;
                               if sym = CommaSym then GetSym
                               else if sym = RparenSym then
                                 begin
                                   GetSym;
                                   break;
                                 end
                               else if sym in FollowProcCall + StrongSyms
                                   then break
                               else Mark (') or , ?')
                             end;
                       end;

                     if mproc_obj^.val < 0 then Mark ('forward call')
                     else if not IsParam (par) then
                        begin
                          {push the addr offset of x, where x is an object}
                          Str(x.a, temp_str);
                          temp_id:= x.tp^.typename + '_' + mproc_obj^.name;
                          param_size := param_size + 4;
                          Call_object_proc(temp_str,temp_id, param_size);
                        end
                     else Mark ('too few parameters');

                   end{ if mproc_obj<>guard and .cls=procClass}
                   else Mark('monitor procedure is not defined.');
                end  {sym=ident}

                else Mark ('ident? for a monitor procedure call.')
          end {sym=period}
    end; { end of 'obj^.tp^.form = Monitor then begin'}

if (obj^.tp^.form=Arry) and (obj^.tp^.base^.form=Monitor) then
  begin
    {array of monitor(class typed) variables.}
    if sym = LbrakSym then
      begin
        x.indirect := true;
        Str(x.a, temp_str);
        Obj_array_addr(temp_str);
        GetSym;  expression(code_gen_mode, y, false);
        if x.tp^.form = Arry then
          begin
            Index_32(x, y);
            Save_addr_in_edi;
          end
        else Mark ('not an array');
        if sym = Rbraksym then GetSym else Mark (']?');
```

```
  end
else Mark('[ is missing.');
  {Do not call loaditem here.}
  {Index_32 has placed the addr offset of the object on top of stack}
  if sym<>PeriodSym then Mark(' . is expected here.')
  else
    begin
      GetSym;
      if sym = IdentSym then
        begin
          temp:=x.tp^.fields;
          FindField(mproc_obj, temp);
          GetSym;
          if (mproc_obj<>guard) and (mproc_obj^.cls=ProcClass)  then
            begin
              par := mproc_obj^.dsc;
              param_size :=0;
              if sym = LparenSym then
                begin
                  GetSym;
                  if sym = RparenSym then GetSym
                  else
                    while true do
                      begin
                        param (code_gen_mode, par);
                        param_size := param_size + 4;
                        if sym = CommaSym then GetSym
                        else if sym = RparenSym then
                          begin
                            GetSym;
                            break;
                          end
                        else if sym in FollowProcCall + StrongSyms then
                            break
                        else Mark (') or , ?')
                      end;
                end;

                if mproc_obj^.val < 0 then Mark ('forward call')
                else if not IsParam (par) then
                  begin
                    Push_addr_in_edi;
                    param_size := param_size + 4;
                    temp_id:= x.tp^.typename + '_' + mproc_obj^.name;
                    Call_32(temp_id);
                    Str(param_size , temp_str);
                    Restore_stack_ptr(temp_str);
                  end
                else Mark ('too few parameters')

            end{ if mproc_obj<>guard and .cls=procClass}
            else Mark('monitor procedure is not defined.');
      end  {sym=ident}
```

```
            else Mark ('ident? for a monitor procedure call.')
        end {sym=period}
      end;
    end  { end of monitor procedure call}

    else if x.mode in [VarClass, ParClass, FieldClass] then
      {x is variable of boolean/longint, or param(value / reference) or field}
      begin
        if (code_gen_mode<>Normal) and (x.mode = VarClass) and (x.lev=0) then
          begin
            {Mark('Can not access global variable of non-class type in class body
                .');}
          end;
          selector(code_gen_mode, x, true);  {leftside = true}
          if sym = BecomesSym then
            begin GetSym; expression(code_gen_mode, y, false);  Store_32 (x, y);
                end
          else if sym = EqlSym then
            begin Mark (':= ?'); GetSym; expression(code_gen_mode, y, false) end;
      end

    else if (obj^.cls = ProcClass)  then  {ordinary procedure call}
      begin
        if obj^.lev=3 then Mark('Call to a monitor procedure here is not allowed.
            ');
          {lev=3 means obj is monitor procedure, which can not be called directly
              from within the class}
        par := obj^.dsc;
        param_size := 0;
        if sym = LparenSym then
          begin GetSym;
            if sym = RparenSym then GetSym
            else
              while true do
                begin
                  param (code_gen_mode, par);
                  param_size:= param_size + 4;
                  if sym = CommaSym then GetSym
                  else if sym = RparenSym then
                    begin GetSym; break end
                  else if sym in FollowProcCall + StrongSyms then break
                  else Mark (') or , ?')
                end
          end;
        if obj^.val < 0 then Mark ('forward call')
        else if not IsParam (par) then
                 begin
                   Placeholder;
                   {push a dummy value on stack to fill the place of obj address
                       holder,
                    as needed for monitor procedure calls}
                   param_size := param_size + 4;
```

```
                    Call_32(obj^.name);
                    Str(param_size, temp_str);
                    Restore_stack_ptr(temp_str);
                  end
         else Mark ('too few parameters');
       end

    else if (obj^.cls = SProcClass) then    { standard proc call: read , write ,
        random}
       begin
        MakeItem(x, obj);
        if obj^.val <= 3 then sparam(code_gen_mode, y, obj^.val);
        {1->read(); 2->write(); 3->random()}
        {read(), write() each takes one argument, which has to be an variable.
            writeln has no argument, }
        IOCall_32 (x, y);       {# y is the item for the actual parameter of read
            (),write()}
       end
    else mark('invalid assignment or statement');
  end

{parse if-then-else statement}
 else if sym = IfSym then
   begin
     GetSym; expression(code_gen_mode, x, false); CJump_32(x);
     if sym = ThenSym then GetSym else Mark ('then?');
     Statement(code_gen_mode);
     x.b:=pc;
     Jumpto('jmp', 'exitfrom', x.b);    {# jump to exitLabel}
     if sym = ElseSym then
       begin
         GetSym;
         MakeJumpLabel('is_FALSE', x.r);
         Statement(code_gen_mode);
       end
     else
       begin MakeJumpLabel('is_FALSE', x.r) end;
     MakeJumpLabel('exitfrom', x.b);
   end

{parse while-do statement}
 else if sym = WhileSym then
   begin
     GetSym;
     L := pc;
     MakeJumpLabel('startwhile',L);
     expression(code_gen_mode, x, false);
     Cjump_32(x);
     if sym = DoSym then GetSym else Mark ('do ?');
     Statement(code_gen_mode);
     jumpto('jmp', 'startwhile', L);  {back jump to the startlabel of while loop}
     MakeJumpLabel('is_FALSE',x.r);
   end
```

```
  else if sym = BeginSym then
    begin GetSym; CompoundStatement(code_gen_mode);
      if sym = EndSym then GetSym else mark ('end?')
    end;
end;

procedure selector(code_gen_mode: Statement_code_gen_mode; var x: Item; LeftSide:
    boolean);
  var y: Item; obj: Objct;
begin
  {if x.mode=ConstClass then LoadItem_32(x, false);}
  while (sym = LbrakSym) or (sym = PeriodSym) do
  begin
    if sym = LbrakSym then
      begin
        x.indirect := true;
        if x.push_placeholder then
          begin
           PlaceHolder;
           x.push_placeholder := false;
          end;
        GetSym; expression(code_gen_mode, y, false);
        if x.tp^.form = Arry then Index_32(x, y) else Mark ('not an array');
        if sym = Rbraksym then GetSym else Mark (']?')
      end
    else {sym=PeriodSym}
      begin
        GetSym;
        if sym = IdentSym then
          if x.tp^.form = Rcrd then
            begin {parsing a record}
              FindField (obj, x.tp^.fields); GetSym;
              if obj <> guard then Field_32 (x, obj) else Mark ('undef')
            end
          else Mark('not a record.')
        else Mark ('ident?[selector]')
      end;
  end;
  if LeftSide then LoadItem_32 (x, true); {load the address of x onto stack }
end;

{parse an identifier list "a,b,c:", return the first object of the list to 'first'
    }
procedure IdentList(cls: Class; var first: Objct);
var obj: Objct;
begin
  if sym = IdentSym then
    NewObj (first, cls); GetSym;
    while sym = CommaSym do
      begin GetSym;
        if sym = IdentSym then
          begin NewObj (obj, cls); GetSym end
        else Mark ('ident (in identifier list)?')
```

```
      end;
    if sym = ColonSym then GetSym else Mark (':?')
end;

{parse primitive, user defined, and class types:
 var x,y : T1;
}
procedure TypeDecl(var t: Typ);
var obj, first: Objct; x, y: Item; tp: Typ;
begin t := intType;
  if not (sym in FirstType) then
    begin Mark ('type? [IdentSym, RecordSym, ArraySym] is expected here.');
      repeat GetSym until sym in FirstType+FollowType+StrongSyms
    end ;
  if sym = IdentSym then
    begin find (obj); GetSym;
      if (obj^.cls = TypeClass) or (obj^.cls=MonitorClass) then t := obj^.tp
      else Mark('type? [undef type].')
    end
  else if sym = ArraySym then
    begin
      GetSym;
      if sym = LbrakSym then GetSym else mark('[?');
      expression(Normal, x, false); {lower bound}
      if (x.mode <> ConstClass)or(x.tp^.form<>Int) then
        Mark ('bad index(lower bound is not constant integer value)');
      if sym = PeriodSym then GetSym else mark('.?');
      if sym = PeriodSym then GetSym else mark('.?');

      expression(Normal, y, false); {upper bound}
      if (y.tp^.form<>Int) then Mark ('bad index(upper bound is not constant
          integer value)');
      if (y.mode <> ConstClass) or (y.a < x.a) then
        begin
          if y.a < x.a then Mark ('bad index(invalid upper bound)')
          else if y.mode<> ConstClass then
             Mark('bad index(upper bound is not constant)');
        end;
      if sym = RbrakSym then GetSym else mark(']?');
      if sym = OfSym then GetSym else mark('of?');
      TypeDecl (tp); new (t); t^.form := Arry; t^.base := tp;
      t^.lower := x.a; t^.len := (y.a - x.a) + 1; t^.size := t^.len * tp^.size
    end
  else if sym = RecordSym then
    begin GetSym;
      new (t); t^.form := Rcrd; t^.size := 0; OpenScope;
      repeat
        if sym = IdentSym then
          begin
            IdentList (FieldClass, first); TypeDecl (tp); obj := first;
            while obj <> guard do
              begin
                obj^.tp := tp; obj^.val := t^.size;
```

```
                t^.size := t^.size + obj^.tp^.size; obj := obj^.next
              end
            end;
          if sym = SemicolonSym then GetSym
          else if sym = IdentSym then Mark(';?')
      until not (sym in [SemicolonSym, IdentSym]);
      t^.fields := topScope^.next; CloseScope;
      if sym = EndSym then GetSym else Mark('end?')
    end
  else Mark ('ident?[type declaration section]')
end;

procedure declarations (var varsize: longint); {varsize=total size of all global
    variables}
var
  obj, first: Objct;
  x : Item;
  t: Typ;
  is_m_proc, g: boolean; {is_m_proc=TRUE if the procedure is guarded}
  pid, class_id: Identifier;
begin
  if not (sym in DeclSyms+FollowDecl) then
    begin Mark (' declaration?');
      repeat GetSym until sym in DeclSyms+FollowDecl
    end;
  repeat
    if sym = ConstSym then
      begin GetSym;
        while sym = IdentSym do
          begin NewObj (obj, ConstClass); GetSym;
            if sym = EqlSym then GetSym else Mark ('=?');
            expression(Normal, x, false);
            if x.mode = ConstClass then
              begin obj^.val := x.a; obj^.tp := x.tp end
            else Mark ('expression not constant');
            if sym = SemicolonSym then GetSym else Mark (';?')
          end
      end;

    if sym = TypeSym then
      begin
        GetSym;
        while sym = IdentSym do
          begin NewObj (obj, TypeClass); GetSym;
            if sym = EqlSym then GetSym else Mark ('=?');
            TypeDecl (obj^.tp);
            if sym = SemicolonSym then GetSym else Mark (';?')
          end
      end;

    while sym = ProcedureSym do
        begin
          g:=false;
```

```
            is_m_proc:=false;
            class_id:='_';  {makes a dummy class_id}
            ProcedureDecl(g, pid, is_m_proc, class_id, Nil, Nil);
            if sym = SemicolonSym then GetSym else Mark (';?')
          end;

      {parse class definition}
      while sym = MonitorclassSym do
         begin
            monitor_decl;
         end;
      { variable declarations}
      if sym = VarSym then
        begin
          GetSym;
          while sym = IdentSym do
            begin
              IdentList(VarClass, first);
              TypeDecl(t);
              obj := first;
              while obj <> guard do
                begin
                  obj^.tp := t;
                  obj^.lev := curlev;
                  obj^.IsAParam := false;
                  if curlev=0 then  {#distinguish between global&local var: curlev=0
                      means global}
                     begin
                       obj^.val := varsize;
                       varsize := varsize + obj^.tp^.size;
                     end
                  else begin      {#if it is local var, its offset (.val) is a
                      negative number.}
                          varsize := varsize + obj^.tp^.size;
                          obj^.val :=-varsize;
                        end ;
                  obj := obj^.next;
                end ;
              if sym = SemicolonSym then GetSym else Mark ('; ?');
            end;
        end;

      if sym in [ConstSym] then
        Mark ('illegal declaration order');
    until not (sym in [ConstSym, TypeSym, VarSym, MonitorclassSym, ProcedureSym])
  end;

procedure ProcedureDecl(var g: boolean; var pid: Identifier; monitor_proc: boolean;
                          classid:Identifier; mutex, cv: Objct);

  const marksize = 8;
    {4 bytes for return address pushed on stack by call instruction; 4 bytes for obj
        address}
```

```
var
  proc, obj: Objct;
  locblksize, parblksize, pc_Label, relational_op: longint;
  procid, temp: Identifier;
  expr: item;
  opstr: string;

procedure FPSection;
var obj, first: Objct; tp: Typ;   parsize :longint;
begin
  if sym = VarSym then
    begin GetSym; IdentList(ParClass, first) end
  else IdentList(VarClass, first);
  if sym = IdentSym then
    begin
      find (obj); GetSym;
      if (obj^.cls = TypeClass) or (obj^.cls=MonitorClass) then tp := obj^.tp
      else begin Mark('type?'); tp := intType end
    end
  else begin Mark ('ident?'); tp := intType end;
  {# use the param-calculation scheme}
  if first^.cls = VarClass then
    begin
      parsize := tp^.size;
      if tp^.form in [Arry, Rcrd] then Mark('No structured parameters passed by
          value.');
    end
  else parsize:= WordSize;
    obj := first;
    while obj <> guard do
      begin
        obj^.tp := tp;
        parblksize:= parblksize + parsize;
        obj := obj^.next
      end
  end;

begin {ProcedureDecl}
  GetSym;
  if sym = IdentSym then
    begin
      g:=false;   { assume it is not guarded before parsing }
      procid:= id;   {id: the (sym)string constructed by the sanner.}
      pid := id;
      NewObj(proc, ProcClass); GetSym; parblksize := marksize;
      IncLevel(2);
      {cur_lev for procedure local variables and parameters are either 2 (ordinary
          procedure) or 3(monitor procedure)}
      OpenScope; proc^.val := -1; proc^.lev := curlev; proc^.isGuarded := false;

      if sym = LparenSym then
        begin GetSym;
          if sym = RparenSym then GetSym
```

```
    else
      begin
        FPSection;
        while sym = SemicolonSym do
          begin GetSym; FPSection end ;
        if sym = RparenSym then GetSym else Mark(')?')
      end
  end;

proc^.parsize := parblksize;  {parsize = actual par_block_size + 4 }

obj := topScope^.next; locblksize := parblksize;
while obj <> guard do
  begin
    obj^.lev := curlev;
    if obj^.cls = ParClass then locblksize := locblksize - WordSize
    else locblksize := locblksize - obj^.tp^.size;
    obj^.val := locblksize+4;  //always add 4; this does not change while
        marksize changes
    obj^.IsAParam := true;
    obj := obj^.next
  end;

{for guarded expression}
if (sym = WhenSym) and (monitor_proc)  then
  begin
    GetSym;
    MakeGuardLabel(classid , procid);
    proc^.isGuarded := true;
    g:= true;
    { parse the guard expression }
    expression(ClassProc , expr , g);
    Op1_32(NotSym, expr);
    MakeGuardJumpLabel(classid , procid);

  end
else if ((sym=WhenSym) and (not monitor_proc)) then
      Mark('Ordinary procedure should not have guard.');

proc^.parSize := parblksize - marksize;
proc^.dsc := topScope^.next;

if sym = SemicolonSym then GetSym
else
  begin writeln(id); Mark (';? [in procedure declaration]') end;
  {end of signature parsing}
locblksize := 0;
declarations (locblksize);
{ no nested-procedures is allowed in ABC Pascal}
proc^.val := 1;     {# val>0 means procedure has been defined }
if monitor_proc then
  begin
    procid := classid+'_'+procid;
```

```
        GenerateLabel(procid);
      end
    else GenerateLabel(procid);

    Enter_32(locblksize);

    if sym = BeginSym then
      begin
        {# generate asm code}
        if monitor_proc then
          begin
            Str(mutex^.val, temp);
            Lock_mutex(temp);
            if g then
              begin
                pc_Label:=pc;
                MakeJumpLabel('startwhile', pc_Label);
                Make_jump_and_wait_label(procid);
                if (not expr.bool_set) then
                  begin
                    relational_op:= BeqOP + (expr.c);
                    expr.r:= pc;
                    opstr := TransformToStr(relational_op);
                    Eval_bool_val(opstr, expr);
                  end
                else begin
                        relational_op := BeqOP + (expr.c);
                        opstr := TransformToStr(relational_op);
                        expr.r := pc;
                        Get_bool_val;
                        jumpto(opstr, 'is_FALSE', expr.r);
                      end;
          {----------------------------------------------------------}

                Wait_on_condition(mutex^.val, cv^.val);
                Jumpto('jmp', 'startwhile', pc_Label);
                MakeJumpLabel('is_FALSE', expr.r);
              end;
          end; {end of if (monitor_proc)}

          GetSym;
          if monitor_proc then CompoundStatement(ClassProc)
          else CompoundStatement(Normal);
          if monitor_proc then
            begin
              Broadcast_and_unlock(cv^.val, mutex^.val)
            end;
        end {end of 'if sym=begin'}
    else mark('"begin" is expected here.');

    if sym = EndSym then GetSym else Mark ('[procedure decl] end?');

    Return_32(locblksize);
```

```
        CloseScope;
        IncLevel(-2)
    end;
  end;
{end of parsing procedure declarations}

{Class declaration}
procedure Monitor_decl;
var
  obj, obj_mutex, obj_cv: Objct;
  class_var_size, locblksize: longint;
  temp, temp_id, monitor_id, action_id, procedure_name: identifier;
  proc, mobj, first: Objct;
  mtp, t: Typ;
  expr: Item;
  has_guard, is_m_proc: boolean;
  action,a_node: ActionList;

  {new variables for each object -> }
  i, anchor: longint;
  n, local_block_size: longint;
  offset_next, offset_count, offset_n, offset_done: longint;
  worker_id: Identifier;
  start_while_true_loop, end_while_true_loop, start_second_inner_loop,
      end_second_inner_loop: longint;
  end_if_not_done: longint;
  action_queue: AQ;
  {<-}
{------------ body of monitor_decl ------------ }
  begin
    class_var_size :=0;
    GetSym;
    if sym=IdentSym then  {start parsing monitor class}
      begin
        monitor_id :=id;
        NewObj(mobj,MonitorClass); GetSym;
        IncLevel(1);
        mobj^.val:=-1; mobj^.lev := curlev;
        new(mtp); { mtp: store information of this monitor class(type) }
        mtp^.form:=Monitor;
        mtp^.typename:= monitor_id;
        mtp^.a_list:=nil;
        OpenScope; { new topscope to store the class variables }
        temp_id:= id;
        id:= '__mutex_'+ monitor_id;
        NewObj(obj_mutex, VarClass);
        obj_mutex^.val := class_var_size; {.val field is the address offset}
        class_var_size := class_var_size + sizeof_pthread_mutex;
        {sizeof(pthread_mutex) = 24}
        id:= '__cv_'+ monitor_id;
        NewObj(obj_cv, VarClass);
        obj_cv^.val := class_var_size;
```

```
      class_var_size := class_var_size + sizeof_pthread_cv; {sizeof(pthread_cond_var) = 48}

      id:=temp_id;
      if sym = VarSym then   {no constant declaration inside a class}
        begin
          GetSym;
          while sym = IdentSym do
            begin
              IdentList(VarClass, first);
              TypeDecl(t);  {parse  v1,v2,v3,...  :TYPE }
              if t^.form=Monitor then
              Mark('Can not declare object fields in class variable section.');
              obj := first;
              while obj <> guard do
                begin
                  obj^.tp := t;
                  obj^.lev := curlev;
                  obj^.IsAParam := false;
                  if curlev = 1 then   {#distinguish between global(curlev=0) & local}
                    begin
                      obj^.val := class_var_size;
                      class_var_size := class_var_size + obj^.tp^.size;
                    end;
                  obj := obj^.next;
                end;
              if sym = SemicolonSym then GetSym else Mark ('; ?') ;
            end; {end of while sym= IdentSym do}
        end;

      if sym=ProcedureSym then {if sym = ProcedureSym, start parsing monitor proc(s)}
      while sym = ProcedureSym do
        begin
          has_guard := false;
          is_m_proc := true;
          ProcedureDecl(has_guard, procedure_name, is_m_proc, monitor_id, obj_mutex, obj_cv);
          if sym = SemicolonSym then GetSym else Mark('; ?');
        end ; {end of 'if sym=proceduresym'}

      while sym=ActionSym do
        begin
          GetSym;
          action_id := id;   {id: the (sym)string constructed by the sanner.}
          NewObj(proc, ActionClass); GetSym;
          IncLevel(2);
          OpenScope;  {new top list for the local variables}
          proc^.val := -1; proc^.lev := curlev;
          proc^.isGuarded := false;
          action_id := '_action_' + monitor_id + '_' + action_id;
          if sym = WhenSym then
```

```
        begin
          GetSym;
          {define a function label}
          MakeActionGuard(action_id);
          expression(ClassProc, expr, true);
          place_boolean_val(expr);
          Return_from_action_guard;
          proc^.isGuarded := true;
        end
      else  {an unguarded action}
        begin
          MakeActionGuard_open(action_id);
          proc^.isGuarded := false
        end;

      proc^.dsc := topScope^.next;
      if sym = SemicolonSym then GetSym else Mark (';?');
      {end of the action signature}

      locblksize := 0;
      declarations(locblksize);
      { no nested-procedures is allowed in the Intel version of pascal0}
      proc^.val := 1;      { val>0 => procedure is defined }
      GenerateFuncPrefix(action_id);
      action_queue[mtp^.action_count].action_body := action_id;
      action_queue[mtp^.action_count].action_guard:= action_id + '_guard';
      mtp^.action_count:= mtp^.action_count+1;
      MakeActionLabel(action_id);
      Enter_32 (locblksize);

      if sym = BeginSym then  { process action body }
        begin
          GetSym;
          CompoundStatement(ClassProc);
          Return_32(locblksize);
        end  {end of action body}
      else mark(' "begin" is expected here.');
      if sym = EndSym then GetSym else Mark ('end?');
      if sym = SemicolonSym then GetSym else Mark('";" is needed to end an
          action.');
      CloseScope;
      IncLevel(-2);
    end;  {complete parsing for action, end of while-do loop }

    mtp^.size := class_var_size;
    mtp^.fields:= topScope^.next;
    mobj^.tp:= mtp;

    if mtp^.action_count > 0 then
      begin
        {here we create function conjunction_neg_guard }
        Make_conjunction_negation_label(monitor_id);
        n := mtp^.action_count;
```

```
i := 0;
while i < n do
  begin
    if i = 0 then
      begin
        Eval_action_guard(action_queue[i].action_guard, anchor);
      end;
    if i > 0 then
      begin
        Eval_action_guard(action_queue[i].action_guard, anchor);
        Op_logical_and;
      end;
    i:=i+1;
  end; {end of 'while i<n' loop}

  Return_from_action_guard;
  { here we create the worker thread ->}
  worker_id := '_action_' + monitor_id + '_worker';
  new(action);
  action^.next:=nil;
  action^.id:= worker_id;
  a_node := mtp^.a_list;

  if mtp^.a_list=nil then begin mtp^.a_list:=action end
  else { the a_list is non-empty}
    begin
      while a_node^.next <> nil do a_node:=a_node^.next;
      a_node^.next:=action;
    end;

  GenerateFuncPrefix(worker_id);
  n := mtp^.action_count;
  local_block_size := (n + 4)*4;
  Str(local_block_size, temp);

  Generate_obj_thread_prelogue(worker_id, temp);
  offset_next := -4;
  offset_count := -8;
  offset_n := -12;
  offset_done := -16;

  Init_n_and_next(offset_n, n, offset_next);
  start_while_true_loop := pc;
  MakeJumpLabel('startwhile',start_while_true_loop);

  anchor := pc;
  Str(anchor, temp);
  end_while_true_loop := anchor;
  {end of while(true) do loop }
  Init_count_and_done(offset_count, offset_n, offset_done);
  Str(obj_mutex^.val, temp);
  Lock_mutex(temp);
  inner_loop_one_and_three(n, offset_next, offset_count, offset_n,
```

```
                                          offset_done , action_queue);

                  If_not_done_then_wait(offset_done , offset_count , offset_n , anchor ,
                     end_if_not_done ,
                                    start_second_inner_loop , end_second_inner_loop ,
                                        monitor_id ,
                                    obj_mutex^.val , obj_cv^.val);

                  //# next:= next+1 mod n
                  Update_val_next(offset_next , offset_n);
                  // worker thread:
                  //  at end of each iteration , call broadcast(cv), unlock(mtx)

                  Broadcast_and_unlock(obj_cv^.val , obj_mutex^.val);
                  Generate_obj_thread_epilogue(start_while_true_loop ,
                     end_while_true_loop , local_block_size);
               end;

             {parse the initilization block of the class; create a procedure named by
                temp_id.}
             if sym=BeginSym then
               begin
                 temp_id := monitor_id + '_init';  {naming convention is 'MonitorId_init
                    '}
                 GenerateLabel(temp_id);
                 Enter_32(0);  { since the init procedure(per class) has no local
                    variables}
                 {#call pthread_mutex_init()
                  # call pthread_cond_init(_ _) }
                 Init_obj(obj_mutex^.val , obj_cv^.val);
                 {now, processing class initializtion block}
                 GetSym;
                 CompoundStatement(ClassInit);
                 Return_32(0);
                 CloseScope;
               end;
             if Sym=EndSym then
               begin
                 IncLevel(-1);
                 GetSym;
                 {if id=monitor_id then GetSym
                  else Mark('monitor ID does not match the earlier declared one.');}
               end;
             if Sym=SemicolonSym then GetSym else Mark('; is needed to end monitor
                 declaration.');
          end
          else mark('An identifier is expected.');
   end;
{ the end of parsing monitor class declarations }


procedure SkipIdents; {skip optional identifiers in program clause}
  begin
```

```
    if sym = IdentSym then
      begin GetSym;
        while sym = CommaSym do
          begin GetSym;
            if sym = RParenSym then break;
            if sym = IdentSym then GetSym else mark('ident? [skip ident]')
          end
      end
    else mark('ident?')
  end;

{count the number of actions in action list associated to an object}
function get_action_count(list: ActionList) :longint;
  var x: longint;
  begin
    x:=0;
    while list<>nil do
      begin
        x := x+1;
        list := list^.next;
      end;
    get_action_count := x
  end;

procedure mainProgram;
  var progid: Identifier;
        varsize, anchorl: longint;
        init_func_name, offset, main_action_label: Identifier;
        temp: Identifier;
        s, x: Objct;
        j, lowerbound, upperbound, base_elt_size, length, offset_val: longint;
        number_of_actions, number_of_workers, n, m: longint;
        all_actions, t, oa_node, curnode: OA_list;

  begin
    write('-------- compiling >> ');
    if sym = ProgramSym then
      begin
        GetSym; Open; OpenScope; varsize := 0;
        if sym = IdentSym then
          begin
            progid := id;  FileID := progid + '.s';
            assign(File_GAS ,FileID);
            rewrite(File_GAS);
            writeln(progid);    {# write to the standard output }
            GetSym;
          end
        else Mark('ident?');
        if sym = LParenSym then
          begin
            GetSym; SkipIdents;
            if sym = RParenSym then GetSym else mark(')? ')
          end;
```

```
if sym = SemicolonSym then GetSym else mark(';?');

Generate_text_section_label;  { .section  .text }
declarations(varsize);
if sym = BeginSym then   { parse the main body of the pascal program.}
  begin
    main_action_label := '__main_action';
    { asm code
          .globl main_action_label
            .type  main_action_label  , @function
          main_action_label:
    }
    Generate_main_action_label(main_action_label);
    Enter_32(0);
    GetSym;
    CompoundStatement(Normal);
    Return_32(0);
  end;
if sym = EndSym then
  begin GetSym; if sym<>PeriodSym then Mark('. is missing!'); end
else Mark('[end of program] end?');

Header_Code;
{ global main program:
  - call all objects' initialization function
  - pthread initialization routines
  - call pthread_create, pthread_join
}
{ call all object's init functions}
s := topScope;
x := s^.next;
all_actions := NIL;
number_of_actions := 0;
if x = Guard then Mark('Empty topScope list!!');
while true do
  begin
    if (x^.cls=VarClass) and (x^.tp^.form=Monitor) then
      begin
        Str(x^.val, offset);
        init_func_name := 'proc_' + x^.tp^.typename + '_init';
        Call_init_block(init_func_name, offset);
        { store the variable name and its action names }
        if x^.tp^.a_list<>nil then
          begin
            new(oa_node);
            oa_node^.next := nil;
            oa_node^.actions := x^.tp^.a_list;
            oa_node^.obj_name := x^.name;
            oa_node^.val := x^.val;
            if all_actions = nil then all_actions := oa_node
            else  {all_actions^.next := oa_node}
              begin
                curnode := all_actions;
```

```
            while curnode^.next <> nil do curnode := curnode^.next;
            curnode^.next := oa_node;
          end;
        n := get_action_count(oa_node^.actions);
        number_of_actions := number_of_actions + n;
      end
  end;

if (x^.cls=VarClass) and (x^.tp^.form=Arry) then
  begin {an array of objects}
    if x^.tp^.base^.form = Monitor then
      begin
        {for each element in that array var, call its init function}
        lowerbound := x^.tp^.lower;
        length := x^.tp^.len;
        upperbound := lowerbound + length - 1;
        base_elt_size := x^.tp^.base^.size;
        for j := lowerbound to upperbound do
          begin
            init_func_name := 'proc_' + x^.tp^.base^.typename + '_init';
            offset_val := x^.val + (j-lowerbound)*base_elt_size;
            Str(offset_val, offset);
            Call_init_block(init_func_name, offset);
          end;

        if x^.tp^.base^.a_list<>nil then
          begin
            for j:=lowerbound to upperbound do
              begin
                new(oa_node);
                oa_node^.obj_name := x^.name;
                oa_node^.val := x^.val + (j-lowerbound)*base_elt_size;
                oa_node^.actions := x^.tp^.base^.a_list;
                oa_node^.next:=nil;
                if all_actions=nil then
                  begin all_actions:= oa_node end
                else
                  begin
                    curnode := all_actions;
                    while curnode^.next<>nil do
                      curnode := curnode^.next;
                    curnode^.next := oa_node;
                  end;
              end;
              n:=get_action_count(x^.tp^.base^.a_list);
              number_of_actions := number_of_actions + n*length;
          end;
        end;
     end;

     if x^.next = Guard then break
     else
       begin
```

```
      x:=x^.next; continue;
    end;
end; { end of while true loop}

{call pthread initialization routines, and call pthread_create() }
{pthread_attr_setinheritsched(&thr_attr, PTHREAD_EXPLICIT_SCHED)}
{
   pushl $1
   movl $thread_attr , %eax
   pushl %eax
   call pthread_attr_setinheritsched
   addl $8 , %esp
}

Setup_pthread_attribute();
Setup_randomization;
{call srand(time(0)) to initialize the seed }
{create main_action }
Create_main_action(main_action_label);
number_of_workers :=  number_of_actions;
Str(number_of_workers, temp);
Setup_global_counter(temp);  // global_counter := number of object
    threads

{ create pthread for the worker threads.}
t := all_actions;
m := 0;    {just an index the thread_ptrs}
while t<>Nil do
  begin
    Str(t^.val, temp);
    Setup_action_base_ptr(temp);
    while t^.actions <> nil do
      begin
        Create_obj_thread(m, t^.actions^.id);
        m := m+1;
        t^.actions := t^.actions^.next;
      end;
   t := t^.next;
   end;

   {// asm code:
    pushl $0    // NULL=0
    pushl thread_main_action
    call pthread_join
    addl $8 , %esp
   }
   Join_main_action;
   {while not all_finished do
         begin
           LOAD(global_counter, val);  //atomic
           if val=0 then all_finished:= true;
         end;
   }
```

```
                  Check_termination(anchor1);
                        { program terminates }
                  Close_32;
                  Header_var(varsize, number_of_actions);
                  {place the data section at the end of asembly program}
                  CloseScope;
                  if not error then
                    begin writeln('  code generated', pc :6) end
                  else  writeln('>>> Compilation error(s). No code generated.');
          end
        else Mark ('program?');
  end;

  procedure Init;
    begin
      writeln('Pascal0 Compiler');
      OpenScope;
      PreDef (TypeClass, 1, 'boolean', boolType);
      PreDef (TypeClass, 2, 'integer', intType);
      PreDef (ConstClass, 1, 'true', boolType);
      PreDef (ConstClass, 0, 'false', boolType);
      PreDef (SProcClass, 1, 'read',  nil);
      PreDef (SProcClass, 2, 'write', nil);
      PreDef (SProcClass, 3, 'random', nil);
        {predefined procedure random randomly generates a number between 0~999}
      PreDef (SprocClass, 4, 'writeln', nil);
    end;

  procedure Compile;
    begin
      GetSym; MainProgram
    end;

begin
  Init; Compile;
  if not error then Load;
end.
```

Listing D.2: intelgenerator.pas

```
{
 [By Xiao-lei Cui, Jan 2009]
 - defines procedures to generate assembly code.
 - these procedures are invoked by the compiling driver.
}
unit IntelGenerator;

interface
uses scanner, symboltable;

const
  ACTION_LIMIT = 499; {maximum number of actions in a class}
```

```
  ADDOP = 0; SUBOP = 1; MULOP = 2; DIVOP = 3; MODOP = 4; CMPOP = 5;
  BEQOP = 15; BNEOP = 16; BLTOP = 17; BGEOP = 18; BLEOP = 19; BGTOP = 20;

type
  action_entry = record
    action_body: Identifier;    {a label in assembly code that identifies the action
        body}
    action_guard: Identifier;    {a label that identifies the action guard}
  end;

  AQ = array [0..ACTION_LIMIT] of action_entry;

var
  curlev, pc, trap_line: longint; {trap_line is for debugging purpose}
  fileID: Identifier;  {# the output file name }
  File_GAS: text;      {# the generated .s file}

  procedure Call_object_proc(temp_str, temp_id: string; param_size: longint);
  procedure Obj_array_addr(temp_str: string);
  procedure Save_addr_in_edi;
  procedure Push_addr_in_edi;
  procedure Restore_stack_ptr(temp_str: string);
  procedure MakeGuardLabel(classid, procid: string);
  procedure MakeGuardJumpLabel(classid, procid: string);
  procedure Lock_mutex(mtx_addr: string);
  procedure Make_jump_and_wait_label(procid: string);
  procedure Eval_bool_val(opstr: string; var expr: Item);
  procedure Get_bool_val;
  procedure Wait_on_condition(mtx, cv: longint);
  procedure Broadcast_and_unlock(cv, mtx: longint);
  procedure MakeActionGuard(action_id: string);
  procedure Return_from_action_guard;
  procedure MakeActionGuard_open(action_id: string);
  procedure MakeActionLabel(action_id: string);
  procedure Make_conjunction_negation_label(monitor_id: string);
  procedure Eval_action_guard(action_guard: string; var anchor: longint);
  procedure Op_logical_and;
  procedure Generate_obj_thread_prelogue(worker_id, local_block: string);
  procedure Init_n_and_next(offset_n, n, offset_next: longint);
  procedure Init_count_and_done(offset_count, offset_n, offset_done: longint);

  procedure If_not_done_then_wait(offset_done,offset_count,offset_n,anchor:longint;
                                 var end_if_not_done,start_second_inner_loop,
                                     end_second_inner_loop:longint;
                                 monitor_id: string; mtx, cv:longint);
  procedure Update_val_next(offset_next, offset_n: longint);
  procedure Generate_obj_thread_epilogue(start_while_true, end_while_true,
      local_block_size:longint);
  procedure Init_obj(mtx, cv: longint);

  procedure Generate_text_section_label;
  procedure Generate_main_action_label(main_action: string);
```

```
procedure Call_init_block(init_func_name, offset: string);
procedure Setup_pthread_attribute;
procedure Setup_randomization;
procedure Create_main_action(main_action_label: string);
procedure Setup_global_counter(temp: string);
procedure Setup_action_base_ptr(offset: string);
procedure Create_obj_thread(m: longint; action_name: string);
procedure Join_main_action;
procedure Check_termination(var anchorl: longint);

procedure PlaceHolder;

procedure IncLevel(n: longint);

procedure MakeConstItem(var x: Item; tp: Typ; val: longint);

procedure MakeItem(var x: Item; y: Objct);

procedure LoadItem_32(var x:Item; LeaveAddress: boolean);

procedure Field_32(var x: Item; y: Objct);

procedure Index_32(var x, y: Item);

procedure Op1_32(op: Symbol; var x: Item);

procedure Op2_32(op: Symbol; var x, y: Item);

procedure Relation_32(op: Symbol; var x, y: Item);

procedure Store_32(var x, y: Item);

procedure Parameter_32(var x: Item; ftyp: Typ; cls: Class);

procedure CJump_32(var x: Item);

procedure Place_boolean_val(var x: Item);

procedure Call_32(name: Identifier );

procedure IOCall_32(var x, y: Item);

procedure Header_var(size: longint; num_thr: longint);

procedure Header_code;

procedure Enter_32(size: longint);

procedure Return_32(size: longint);

procedure GenerateFuncPrefix(id: Identifier);
procedure GenerateLabel(id: Identifier);
```

```
  procedure MakeJumpLabel(s: string; pc0: longint);

  procedure Jumpto(op:string; s: string; pc_label: longint);

  procedure inner_loop_one_and_three(n, offset_next, offset_count, offset_n,
      offset_done: longint;
                                                    action_queue: AQ);
  procedure Open;
  procedure Close_32;
  procedure Load;
  function TransformToStr(rel_op: longint): string;

implementation
  const TAB=char(9);

  type  Memory = array [0 .. 499999] of string;
  var code: Memory;

  {Generate code to call a procedure of an object}
  procedure Call_object_proc(temp_str, temp_id: string; param_size: longint);
  var temp: string;
  begin
    code[pc] := 'pushl $' + temp_str + ' #push addr offset of object on stack';  pc
        :=pc+1;
    code[pc] := 'movl $global_var, %eax'; pc:=pc+1;
    code[pc] := 'addl %eax, (%esp)';       pc:=pc+1;
    Call_32(temp_id);
    Str(param_size, temp);
    code[pc] := 'addl  $' + temp + ' ,%esp';       pc:=pc+1;
  end;

  {Load the base address of an array of objects}
  procedure Obj_array_addr(temp_str: string);
  begin
    code[pc] := 'pushl  $' + temp_str;  pc:=pc+1;
    code[pc] := 'movl $global_var, %eax'; pc:=pc+1;
    code[pc] := 'addl  %eax, (%esp)  #NOTE: addr of array is on stack now'; pc:=pc+1;
  end;

  {Temporarily save the address in the edi register}
  procedure Save_addr_in_edi;
  begin
    code[pc] := 'popl  %edi';  pc:=pc+1;
  end;

  {Load the address saved in edi register}
  procedure Push_addr_in_edi;
  begin
    code[pc] := 'pushl  %edi';  pc:=pc+1;
  end;

  {Clean up the stack after a procedure call}
```

```
procedure Restore_stack_ptr(temp_str: string); // restore stack pointer after a function call
begin
  code[pc] := 'addl  $' + temp_str + ' ,%esp';    pc:=pc+1;
end;

{Make a label for procedure guard}
procedure MakeGuardLabel(classid , procid: string);
begin
  code[pc] := '__' + classid + '_' + procid + '_guard:'; pc:=pc+1;
end;

{Make a label to jump to the condition wait loop}
procedure MakeGuardJumpLabel(classid , procid: string);
begin
  code[pc] := 'jmp ' + '__' + classid + '_' + procid + '_while_wait';  pc:=pc+1;
end;

{Lock the object mutext variable}
procedure Lock_mutex(mtx_addr: string);
begin
  code[pc] := 'movl  8(%ebp) , %eax';       pc:=pc+1;
  code[pc] := 'addl  '+'$' + mtx_addr + ', %eax'; pc:=pc+1;
  code[pc] := 'pushl %eax';               pc:=pc+1;
  code[pc] := 'call  pthread_mutex_lock'; pc:=pc+1;
  code[pc] := 'addl  $4 , %esp';            pc:=pc+1;
end;

{Jump to the guard evaluation block and make a label for the condition wait loop}
procedure Make_jump_and_wait_label(procid: string);
begin
  code[pc] := 'jmp ' + '__' + procid + '_guard';  pc:=pc+1;
  code[pc] := '__'+ procid + '_while_wait:';      pc:=pc+1;
end;

{Evaluate the boolean expression and make a conditional jump}
procedure Eval_bool_val(opstr: string; var expr: Item);
begin
  code[pc] := 'popl  %ecx' ;    pc:=pc+1;
  code[pc] := 'cmp   $0, %ecx';  pc:=pc+1;

  Jumpto(opstr, 'branchfrom' ,expr.r);
  code[pc] := 'pushl $0x0';         pc:=pc+1;
  Jumpto('jmp', 'exitfrom' ,expr.r);
  MakeJumpLabel('branchfrom', expr.r);
  code[pc] := 'pushl $0x1'; pc:=pc+1;
  MakeJumpLabel('exitfrom', expr.r);
  expr.r := pc;
  code[pc] := 'popl  %eax';   pc:=pc+1;   // pop the value of boolean expn to %eax
  code[pc] := 'cmp  $0x0, %eax';   pc:=pc+1;
  jumpto('je', 'is_FALSE', expr.r);
end;
```

```
{Get the boolean value if it is already set}
procedure Get_bool_val;
begin
  code[pc] := 'popl  %eax';         pc:=pc+1;
  // move the value of boolean expr to %eax
  code[pc] := 'cmp   $0x0, %eax';   pc:=pc+1;
end;

{Generate code for the condition wait block}
procedure Wait_on_condition(mtx, cv: longint);
var temp: string;
begin
  code[pc] := 'movl 8(%ebp), %eax';                pc:=pc+1;
  Str(mtx, temp);
  code[pc] := 'addl  ' + '$' + temp + ', %eax';         pc:=pc+1;
  code[pc] := 'pushl %eax';                        pc:=pc+1;
  code[pc] := 'movl 8(%ebp)  , %eax';              pc:=pc+1;
  Str(cv, temp);
  code[pc] := 'addl  ' + '$' + temp + ', %eax';   pc:=pc+1;
  code[pc] := 'pushl %eax';                        pc:=pc+1;
  code[pc] := 'call  pthread_cond_wait';                pc:=pc+1;
  code[pc] := 'addl  $8  , %esp';                       pc:=pc+1;
end;

{Generate code to broadcast on condition variable and unlock the mutex}
procedure Broadcast_and_unlock(cv, mtx: longint);
var temp: string;
begin
  code[pc] := 'movl 8(%ebp), %eax';    pc:=pc+1;
  Str(cv, temp);
  code[pc] := 'addl  '+ '$' + temp + ', %eax'; pc:=pc+1;
  code[pc] := 'pushl  %eax';                  pc:=pc+1;
  code[pc] := 'call  pthread_cond_broadcast'; pc:=pc+1;
  code[pc] := 'addl $4, %esp';                 pc:=pc+1;
  code[pc] := 'movl  8(%ebp)  , %eax' ;        pc:=pc+1;
  Str(mtx, temp);
  code[pc] := 'addl  ' + '$' + temp + ', %eax'; pc:=pc+1;
  code[pc] := 'pushl %eax';               pc:=pc+1;
  code[pc] := 'call  pthread_mutex_unlock'; pc:=pc+1;
  code[pc] := 'addl  $4, %esp';             pc:=pc+1;
end;

{Make a label to identify the action guard block}
procedure MakeActionGuard(action_id: string);
begin
  code[pc] := action_id + '_guard:';    pc:=pc+1;
end;

{Code to return from evaluation of action guard}
procedure Return_from_action_guard;
begin
  code[pc] := 'popl  %eax';      pc:=pc+1;  //store the boolean value in eax
  code[pc] := ' ret';            pc:=pc+1;
```

```
end;

{Code for an unguarded action}
procedure MakeActionGuard_open(action_id: string);
begin
  code[pc] := action_id + '_guard:'; pc:=pc+1;
  code[pc] := 'movl  $0x1, %eax'; pc:=pc+1; // store true in %eax
  code[pc] := ' ret'; pc:=pc+1;
end;

{Make label to identify an action body}
procedure MakeActionLabel(action_id: string);
begin
  code[pc] := action_id + ':'; pc:=pc+1;
end;

{Make label to identify the conjunction of negation of the guards}
procedure Make_conjunction_negation_label(monitor_id: string);
begin
  code[pc] := monitor_id + '_conjunction_neg_guard :'; pc:=pc+1;
end;

{Evaluate the action guard and push the result onto stack}
procedure Eval_action_guard(action_guard: string; var anchor: longint);
var temp: string;
begin
  code[pc] := 'call  ' + action_guard; pc:=pc+1;
  code[pc] := 'cmp $0, %eax'; pc:=pc+1;
  anchor := pc;
  Str(anchor, temp);
  code[pc] := 'je  branchfrom_' + temp;  pc:=pc+1;
  code[pc] := 'pushl  $0x0';  pc:=pc+1;
  code[pc] := 'jmp exitfrom_' + temp;  pc:=pc+1;
  code[pc] := 'branchfrom_' + temp+ ':';  pc:=pc+1;
  code[pc] := 'pushl  $0x1';  pc:=pc+1;
  code[pc] := 'exitfrom_' + temp + ':' ;  pc:=pc+1;
end;

{Performs logical AND operation}
procedure Op_logical_and;
begin
  code[pc] := 'popl  %eax';  pc:=pc+1;
  code[pc] := 'popl  %ecx';  pc:=pc+1;
  code[pc] := 'and   %eax, %ecx';  pc:=pc+1;
  code[pc] := 'pushl  %ecx';  pc:=pc+1;
end;

{Code as the prelogue of the body of an object thread}
procedure Generate_obj_thread_prelogue(worker_id, local_block: string);
begin
  code[pc] :=  worker_id + ' :'; pc:=pc+1;
  code[pc] := 'pushl %ebp'; pc:=pc+1;
  code[pc] := 'movl  %esp, %ebp'; pc:=pc+1;
```

```
    code[pc] := 'subl $' + local_block + ', %esp';   pc:=pc+1;
    code[pc] := 'pushl %esi';   pc:=pc+1;
  end;

  {Initialize variables n and next}
  procedure Init_n_and_next(offset_n, n, offset_next: longint);
  var temp: string;
  begin
    // n:= action_count;
    Str(offset_n, temp);
    code[pc] := 'pushl $' + temp;   pc:=pc+1;
    Str(n, temp);
    code[pc] := 'pushl $' + temp;   pc:=pc+1;
    code[pc] := 'popl %eax';   pc:=pc+1;
    code[pc] := 'popl %ecx';   pc:=pc+1;
    code[pc] := 'addl %ebp, %ecx';      pc:=pc+1;
    code[pc] := 'movl %eax, (%ecx)';   pc:=pc+1;
    code[pc] := TAB + '# n:= action_count';   pc:=pc+1;
    // next:=0;
    Str(offset_next, temp);
    code[pc] := 'pushl $' + temp;    pc:=pc+1;
    code[pc] := 'popl  %eax';   pc:=pc+1;
    code[pc] := 'addl  %ebp, %eax';   pc:=pc+1;
    code[pc] := 'movl  $0,  (%eax)     #next:=0;';   pc:=pc+1;
  end;

  {Initialize variables count and done}
  procedure Init_count_and_done(offset_count, offset_n, offset_done: longint);
  var temp: string;
  begin
    Str(offset_count, temp);
    code[pc] := 'pushl  $' + temp;    pc:=pc+1;
    Str(offset_n, temp );
    code[pc] := 'pushl  $' + temp;    pc:=pc+1;
    code[pc] := 'popl  %eax';          pc:=pc+1;
    code[pc] := 'addl  %ebp, %eax';   pc:=pc+1;
    code[pc] := 'pushl (%eax)';        pc:=pc+1;
    code[pc] := 'popl  %eax';          pc:=pc+1;
    code[pc] := 'popl  %ecx';          pc:=pc+1;
    code[pc] := 'addl  %ebp, %ecx';   pc:=pc+1;
    code[pc] := 'movl  %eax, (%ecx)   # count:=n; ';    pc:=pc+1;
    Str(offset_done, temp);
    code[pc] := 'pushl  $' + temp;    pc:=pc+1;
    code[pc] := 'pushl  $0';     pc:=pc+1;
    code[pc] := 'popl  %eax';    pc:=pc+1;
    code[pc] := 'popl  %ecx';    pc:=pc+1;
    code[pc] := 'addl  %ebp, %ecx';      pc:=pc+1;
    code[pc] := 'movl  %eax, (%ecx)    # done := false; ';    pc:=pc+1;
  end;

  {If no action is eligible to run, then wait on condition until one action is
      enabled}
  procedure If_not_done_then_wait(offset_done,offset_count,offset_n,anchor:longint;
```

```
                                         var end_if_not_done ,start_second_inner_loop ,
                                             end_second_inner_loop: longint;
                                         monitor_id: string; mtx, cv:longint);
  var temp, templ: string;
  begin
    Str(offset_done , temp);
    code[pc]:= 'pushl $' + temp;   pc:=pc+1;
    code[pc]:= 'popl  %eax';   pc:=pc+1;
    code[pc]:= 'addl  %ebp, %eax';   pc:=pc+1;
    code[pc]:= 'pushl  (%eax)';      pc:=pc+1;
    end_if_not_done := pc;
    Str(end_if_not_done , temp);
    code[pc]:= 'popl  %eax';   pc:=pc+1;
    code[pc]:= 'cmp  $0x1, %eax';   pc:=pc+1;
    code[pc]:= 'je  is_FALSE_' + temp;   pc:=pc+1;
    //if (not done) body ...
    Str(offset_count , temp);
    code[pc]:= 'pushl $' + temp;   pc:=pc+1;
    Str(offset_n , temp);
    code[pc]:= 'pushl $' + temp;   pc:=pc+1;
    code[pc]:= 'popl  %eax';   pc:=pc+1;
    code[pc]:= 'addl  %ebp, %eax';   pc:=pc+1;
    code[pc]:= 'pushl  (%eax)';   pc:=pc+1;
    code[pc]:= 'popl  %eax';   pc:=pc+1;
    code[pc]:= 'popl  %ecx';   pc:=pc+1;
    code[pc]:= 'addl  %ebp, %ecx';   pc:=pc+1;
    code[pc]:= 'movl  %eax, (%ecx)';   pc:=pc+1;

    start_second_inner_loop := pc;
    Str(start_second_inner_loop , temp);
    code[pc]:= 'startwhile_' + temp + ':';   pc:=pc+1;
    code[pc]:= 'call ' + monitor_id + '_conjunction_neg_guard' ;   pc:=pc+1;
    code[pc]:= 'pushl  %eax';   pc:=pc+1;

    end_second_inner_loop := pc;
    Str(end_second_inner_loop , temp);
    code[pc]:= 'popl  %eax';  pc:=pc+1;
    code[pc]:= 'cmp  $0x0, %eax';  pc:=pc+1;
    code[pc]:= 'je   is_FALSE_' + temp;  pc:=pc+1;
    code[pc]:= 'movl 8(%ebp), %eax';     pc:=pc+1;

    //lock DEC global_counter
    code[pc]:= 'lock subl $1, global_counter';          pc:=pc+1;
    Str(mtx, temp);
    code[pc]:= 'addl  ' + '$' + temp + ', %eax'; pc:=pc+1;
    code[pc]:= 'pushl %eax';                     pc:=pc+1;
    code[pc]:= 'movl 8(%ebp)  , %eax';           pc:=pc+1;
    Str(cv, temp);
    code[pc]:= 'addl  ' + '$' + temp + ', %eax';     pc:=pc+1;
    code[pc]:= 'pushl %eax';                     pc:=pc+1;
    code[pc]:= 'call  pthread_cond_wait';               pc:=pc+1;
    code[pc]:= 'addl  $8 , %esp';                       pc:=pc+1;
```

```
    //lock INC global_counter
    code[pc]:= 'lock addl $1, global_counter';          pc:=pc+1;
    Str(start_second_inner_loop, temp);
    code[pc]:= 'jmp  startwhile_' + temp ;     pc:=pc+1;
    Str(end_second_inner_loop, temp1);
    code[pc]:= 'is_FALSE_' + temp1 + ':';    pc:=pc+1;

    anchor := pc;
    Str(anchor, temp);
    code[pc]:= 'jmp  exitfrom_' + temp;     pc:=pc+1;
    code[pc]:= '  # end of if not done body';     pc:=pc+1;
    Str(end_if_not_done, temp1);
    code[pc]:= 'is_FALSE_' + temp1 + ':';     pc:=pc+1;
    code[pc]:= 'exitfrom_' + temp + ':';     pc:=pc+1;
  end;

  {Update the value of variable next -- next:= (next+1) mod n}
  procedure Update_val_next(offset_next, offset_n: longint);
  var temp: string;
  begin
    Str(offset_next, temp);
    code[pc] := 'pushl  $' + temp;        pc:=pc+1;
    code[pc] := 'pushl  $' + temp;        pc:=pc+1;
    code[pc] := 'popl  %eax';          pc:=pc+1;
    code[pc] := 'addl  %ebp, %eax';   pc:=pc+1;
    code[pc] := 'pushl (%eax)';      pc:=pc+1;

    code[pc] := 'pushl  $1';          pc:=pc+1;
    code[pc] := 'popl  %eax';          pc:=pc+1;
    code[pc] := 'popl  %ecx';          pc:=pc+1;
    code[pc] := 'addl  %ecx, %eax';   pc:=pc+1;
    code[pc] := 'pushl %eax';   pc:=pc+1;

    Str(offset_n, temp);
    code[pc]:= 'pushl  $' + temp;   pc:=pc+1;
    code[pc]:= 'popl  %eax';        pc:=pc+1;
    code[pc]:= 'addl  %ebp, %eax';  pc:=pc+1;
    code[pc]:= 'pushl (%eax)';      pc:=pc+1;

    code[pc]:= 'popl  %ecx';          pc:=pc+1;
    code[pc]:= 'popl  %eax';          pc:=pc+1;
    code[pc]:= 'movl  $0, %edx';      pc:=pc+1;
    code[pc]:= 'idivl  %ecx';         pc:=pc+1;
    code[pc]:= 'pushl %edx';          pc:=pc+1;

    code[pc]:= 'popl  %eax';          pc:=pc+1;
    code[pc]:= 'popl  %ecx';          pc:=pc+1;
    code[pc]:= 'addl  %ebp, %ecx';    pc:=pc+1;
    code[pc]:= 'movl  %eax, (%ecx)'; pc:=pc+1;
  end;

  {Code as the epilogue of an object thread body}
```

```
procedure Generate_obj_thread_epilogue(start_while_true , end_while_true ,
    local_block_size:longint);
var temp: string;
begin
  Str(start_while_true , temp);
  code[pc]:= 'jmp  startwhile_' + temp ;   pc:=pc+1;
  MakeJumpLabel('is_FALSE', end_while_true);
  Str(local_block_size , temp);
  code[pc]:=  'addl $' + temp + ', %esp';   pc:=pc+1;
  code[pc]:=  'popl %esi';                 pc:=pc+1;
  code[pc]:=  'movl %ebp, %esp';           pc:=pc+1;
  code[pc]:=  'popl %ebp';                 pc:=pc+1;
  code[pc]:=  ' ret';                      pc:=pc+1;
end;

{Initialize an object by setting up the mutex and condition variable}
procedure Init_obj(mtx, cv: longint);
var temp: string;
begin
  code[pc]:='movl  8(%ebp) , %eax';        pc:=pc+1;
  Str(mtx, temp);
  code[pc]:='addl  ' +'$'+temp+ ', %eax';  pc:=pc+1;
  code[pc]:='pushl $0';                        pc:=pc+1;
  code[pc]:='pushl %eax';               pc:=pc+1;
  code[pc]:='call  pthread_mutex_init';    pc:=pc+1;
  code[pc]:='addl  $8 , %esp';             pc:=pc+1;
  code[pc]:='movl  8(%ebp) , %eax' ;       pc:=pc+1;
  Str(cv, temp);
  code[pc]:='addl  ' +'$'+temp+ ', %eax';  pc:=pc+1;
  code[pc]:='pushl $0';                        pc:=pc+1;
  code[pc]:='pushl %eax';               pc:=pc+1;
  code[pc]:='call  pthread_cond_init';     pc:=pc+1;
  code[pc]:='addl  $8 , %esp';             pc:=pc+1;
end;

{Generate directive for .text section in assembly code}
procedure Generate_text_section_label;
begin
  code[pc]:= '.section  .text'; pc:=pc+1;
end;

{Generate label for the main action}
procedure Generate_main_action_label(main_action: string);
begin
  code[pc]:= '.globl ' + main_action;   pc:=pc+1;
  code[pc]:= TAB + '.type ' + main_action + ' ,@function';  pc:=pc+1;
  code[pc]:= main_action +':';  pc:=pc+1;
end;

{Execute the initialization blocks of all objects}
procedure Call_init_block(init_func_name , offset: string);
begin
  code[pc]:='pushl  $' + offset;    pc:=pc+1;
```

```
    code[pc]:='addl  $global_var, (%esp)';   pc:=pc+1;
    code[pc]:='call  ' + init_func_name;     pc:=pc+1;
    code[pc]:='addl  $4, %esp';  pc:=pc+1;   //clean up stack
  end;

  {Set up pthread attribute for thread creation and management}
  procedure Setup_pthread_attribute;
  begin
    code[pc]:= 'pushl  $1';      pc:=pc+1;
    code[pc]:= 'movl $thread_attr_join , %eax';   pc:=pc+1;
    code[pc]:= 'pushl  %eax';    pc:=pc+1;
    code[pc]:= 'call  pthread_attr_setinheritsched';  pc:=pc+1;
    code[pc]:= 'addl  $8, %esp';  pc:=pc+1;

    code[pc]:= 'pushl  $1';      pc:=pc+1;
    code[pc]:= 'movl $thread_attr_detach  , %eax';   pc:=pc+1;
    code[pc]:= 'pushl  %eax';    pc:=pc+1;
    code[pc]:= 'call  pthread_attr_setinheritsched';  pc:=pc+1;
    code[pc]:= 'addl  $8, %esp';  pc:=pc+1;

    {pthread_attr_init(&thr_attr);}
    {
      movl $thread_attr  , %eax
      pushl %eax
      call  pthread_attr_init
      addl $4  , %esp
    }
    code[pc]:= 'movl  $thread_attr_join  , %eax';  pc:=pc+1;
    code[pc]:= 'pushl  %eax';              pc:=pc+1;
    code[pc]:= 'call  pthread_attr_init';     pc:=pc+1;
    code[pc]:= 'addl $4  , %esp';             pc:=pc+1;

    code[pc]:= 'movl  $thread_attr_detach  , %eax';  pc:=pc+1;
    code[pc]:= 'pushl  %eax';              pc:=pc+1;
    code[pc]:= 'call  pthread_attr_init';     pc:=pc+1;
    code[pc]:= 'addl $4  , %esp';                 pc:=pc+1;
    { pthread_attr_getstacksize(&thr_attr , &default_stack_size);
    pthread_attr_setstacksize(&thr_attr , default_stack_size/8); }

    code[pc]:= 'pushl  $default_stack_size';  pc:=pc+1;
    code[pc]:= 'pushl  $thread_attr_detach';  pc:=pc+1;
    code[pc]:= 'call pthread_attr_getstacksize';  pc:=pc+1;
    code[pc]:= 'addl $8  , %esp';           pc:=pc+1;

    code[pc]:= 'movl  default_stack_size , %eax';  pc:=pc+1;
    code[pc]:= 'shrl $3, %eax';            pc:=pc+1; {shift 3-bit <=> divide by 8}

    code[pc]:= 'pushl  %eax';           pc:=pc+1;
    code[pc]:= 'pushl  $thread_attr_detach';  pc:=pc+1;
    code[pc]:= 'call pthread_attr_setstacksize';  pc:=pc+1;
    code[pc]:= 'addl $8  , %esp';         pc:=pc+1;

    {pthread_attr_setschedpolicy(&thr_attr , SCHED_FIFO) }
```

```
{
  pushl $1
  movl $thread_attr , %eax
  pushl %eax
  call pthread_attr_setschedpolicy
  addl $8 , %esp
}
code[pc]:= 'pushl $1';   pc:=pc+1;
code[pc]:= 'movl $thread_attr_join , %eax';            pc:=pc+1;
code[pc]:= 'pushl %eax';  pc:=pc+1;
code[pc]:= 'call pthread_attr_setschedpolicy'; pc:=pc+1;
code[pc]:= 'addl $8, %esp';  pc:=pc+1;

code[pc]:= 'pushl $1';   pc:=pc+1;
code[pc]:= 'movl $thread_attr_detach , %eax';            pc:=pc+1;
code[pc]:= 'pushl %eax';  pc:=pc+1;
code[pc]:= 'call pthread_attr_setschedpolicy'; pc:=pc+1;
code[pc]:= 'addl $8, %esp';  pc:=pc+1;
{pthread_attr_setdetachstate(&thr_attr , PTHREAD_CREATE_JOINABLE) }
{
 pushl $0
 pushl %eax  //assume last step, %eax has  $thread_attr
 call pthread_attr_setdetachstate
 addl $8 , %esp
}
code[pc]:= 'pushl $0   #0 <=>joinable';   pc:=pc+1;
code[pc]:= 'movl $thread_attr_join , %eax'; pc:=pc+1;
code[pc]:= 'pushl %eax'; pc:=pc+1;
code[pc]:= 'call pthread_attr_setdetachstate'; pc:=pc+1;
code[pc]:= 'addl $8, %esp';  pc:=pc+1;

code[pc]:= 'pushl $1   #1 <=> detachable';   pc:=pc+1;
code[pc]:= 'movl $thread_attr_detach , %eax'; pc:=pc+1;
code[pc]:= 'pushl %eax'; pc:=pc+1;
code[pc]:= 'call pthread_attr_setdetachstate'; pc:=pc+1;
code[pc]:= 'addl $8, %esp';  pc:=pc+1;

{pthread_attr_setscope(&thr_attr , PTHREAD_SCOPE_SYSTEM) }
{
 pushl $0
 pushl %eax
 call pthread_attr_setscope
 addl $8 , %esp
}
code[pc]:= 'pushl $0   # 0 <=> system scope';   pc:=pc+1;
code[pc]:= 'movl  $thread_attr_join , %eax'; pc:=pc+1;
code[pc]:= 'pushl %eax'; pc:=pc+1;
code[pc]:= 'call  pthread_attr_setscope'; pc:=pc+1;
code[pc]:= 'addl  $8, %esp';  pc:=pc+1;

code[pc]:= 'pushl $0';   pc:=pc+1;
code[pc]:= 'movl  $thread_attr_detach , %eax'; pc:=pc+1;
code[pc]:= 'pushl %eax'; pc:=pc+1;
```

```
    code[pc]:= 'call  pthread_attr_setscope'; pc:=pc+1;
    code[pc]:= 'addl  $8, %esp';  pc:=pc+1;
  end;

  {Set up for random number generation}
  procedure Setup_randomization;
   {call srand(time(0)) to initialize the seed }
  begin
    code[pc]:='pushl  $0';    pc:=pc+1;
    code[pc]:='call  time';  pc:=pc+1;
    code[pc]:='addl $4, %esp';  pc:=pc+1;
    code[pc]:='pushl  %eax';    pc:=pc+1;
    code[pc]:='call srand';     pc:=pc+1;
    code[pc]:='addl $4, %esp';  pc:=pc+1;
  end;

  {Create a thread associated to the program main body}
  procedure Create_main_action(main_action_label: string);
  begin
    code[pc]:= 'pushl $0';    pc:=pc+1;
    code[pc]:= 'pushl $' + main_action_label;  pc:=pc+1;
    code[pc]:= 'pushl $thread_attr_join';    pc:=pc+1;
    code[pc]:= 'pushl $thread_main_action'; pc:=pc+1;
    code[pc]:= 'call pthread_create'; pc:=pc+1;
    code[pc]:= 'addl $16  , %esp'; pc:=pc+1;
    code[pc]:= 'cmp  $0, %eax'; pc:=pc+1;
    code[pc]:= 'jne  .trap_thread_failure  #if thread is created successfully ,it
        returns zero';
               pc:=pc+1;
  end;

  {Save the number of object thread to variable 'global_counter'}
  procedure Setup_global_counter(temp: string);
  begin
    code[pc]:= 'movl  $' + temp + ',global_counter';  pc := pc+1;
  end;

  {Calculate the address of an active object and save it in eax}
  procedure Setup_action_base_ptr(offset: string);
  begin
    code[pc]:= 'movl  $' + offset + ',%eax'; pc:=pc+1;
    code[pc]:= 'addl $global_var, %eax'; pc:=pc+1;
    code[pc]:= 'movl $thread_base_ptr, %ecx'; pc:=pc+1;
  end;

  {Create a pthread for each active object}
  procedure Create_obj_thread(m: longint; action_name: string);
  var temp: string;
  begin
    code[pc]:= 'pushl %eax';       pc:=pc+1;
    code[pc]:= 'pushl $' + action_name;  pc:=pc+1;
    code[pc]:= 'pushl $thread_attr_detach';  pc:=pc+1;
    code[pc]:= 'pushl %ecx';    pc:=pc+1;
```

```
    Str(m*4, temp);
    code[pc]:= 'addl $' + temp + ', (%esp)';   pc:=pc+1;
    code[pc]:= 'call pthread_create';   pc:=pc+1;
    code[pc]:= 'addl $16, %esp';    pc:=pc+1;
    code[pc]:= 'cmp  $0, %eax'; pc:=pc+1;
    code[pc]:= 'jne  .trap_thread_failure  #if thread is created successfully, it
        returns zero'; pc:=pc+1;
  end;

  {Wait for the program main body(main action thread) to finish}
  procedure Join_main_action;
  begin
    code[pc]:= 'pushl $0';  pc:=pc+1;
    code[pc]:= 'pushl thread_main_action';  pc:=pc+1;
    code[pc]:= 'call pthread_join';     pc:=pc+1;
    code[pc]:= 'addl $8, %esp';         pc:=pc+1;
  end;

  {Check the termination condition - whether the global conter reaches zero }
  procedure Check_termination(var anchorl: longint);
  var temp: string;
  begin
    anchorl := pc;
    Str(anchorl, temp);
    code[pc]:= 'startwhile_' + temp +':' ;  pc:=pc+1;

    // pushl $4   # that is 4 micro seconds
    // call   usleep
    // addl   $4, %esp

    code[pc]:= '  pushl $4';         pc:=pc+1;
    code[pc]:= '  call   usleep';    pc:=pc+1;
    code[pc]:= '  addl   $4, %esp';  pc:=pc+1;
    code[pc]:= '  nop';              pc:=pc+1;
    code[pc]:= 'cmp  $0, global_counter';  pc:=pc+1;
    code[pc]:= 'jle   is_FALSE_' + temp;   pc:=pc+1;
    code[pc]:= 'jmp  startwhile_' + temp;  pc:=pc+1;
    code[pc]:= 'is_FALSE_' + temp + ':';   pc:=pc+1;
    // print a message when terminates
    code[pc]:= 'pushl $termination_msg';   pc:=pc+1;
    code[pc]:= 'call printf';        pc:=pc+1;
    code[pc]:= 'addl $4 , %esp';     pc:=pc+1;
  end;

{-----------------------------------------------------------------------------}
  procedure TestRange (x: longint);
  begin
    if (x >= $32767) or (x < -$32768) then Mark ('value too large')   {2^15}
  end;

  function negated (cond: longint): longint;
  begin
    if odd (cond) then negated := cond - 1 else negated := cond + 1
```

```
end;

procedure PlaceHolder;
begin
  code[pc]:= 'pushl  $0' + TAB +'# Placeholder  '; pc:=pc+1;
end;

function TransformToStr(rel_op:longint): string;
begin
  case rel_op of
        15:   TransformToStr:= 'je  ' ;        {15 -> '='}
        16:   TransformToStr:= 'jne  ' ;       {16 -> '<>'}
        17:   TransformToStr:= 'jl  ' ;        {17 -> '<'}
        18:   TransformToStr:= 'jge  ' ;       {18 -> '>='}
        19:   TransformToStr:= 'jle  ' ;        {19-> '<='}
        20:   TransformToStr:= 'jg  ' ;         {20-> '>' }
  end;
end;

{# Conditional jump}
procedure Jumpto(op:string;  s: string; pc_label: longint);
var temp: string;
begin
  {je/jl/jge .../jne  BranchFrom_pc }
  Str( pc_label, temp);
  code[pc]:= op + '  ' + s + '_' + temp + TAB + '# Jumpto(op,str,pc0)';  pc:=pc+1;
end;

{# pass a constant string such as 'branchfrom_', 'exitfrom_' to this procedure.}
procedure MakeJumpLabel(s: string; pc0: longint);
var temp:string;
begin
  Str(pc0, temp);
  code[pc]:= s + '_' + temp + ':' +  TAB + '# MakeJumpLabel(str,pc0)'; pc:=pc+1;
end;

procedure LoadItem_32(var x:Item; LeaveAddress: boolean);
var temp: string;
begin
  if x.mode = ParClass then {x is a argument passed by reference}
  begin
    Str(x.a, temp);
    code[pc]:= 'pushl  ' + '$' + temp;    pc:=pc+1;
    if x.lev>1 then
      begin
        code[pc]:= 'movl %ebp, %eax';      pc:=pc+1;
        code[pc]:= 'addl %eax, (%esp)';  pc:=pc+1;
        code[pc]:= 'popl %eax';       pc:=pc+1;
        code[pc]:= 'pushl  (%eax)'; pc:=pc+1;
      end;

    if x.indirect then
      begin
```

```
      code[pc]:= 'popl %eax';        pc:=pc+1;
      code[pc]:= 'addl %eax, (%esp)'; pc:=pc+1;
    end;
  if x.o<>0 then
    begin
      Str(x.o, temp);
      code[pc]:= 'movl $' + temp + ', %eax';  pc:=pc+1;
      code[pc]:= 'addl %eax, (%esp)';     pc:=pc+1;
    end;
  if not LeaveAddress then
    begin
      code[pc]:= 'popl %eax';        pc:=pc+1;
      code[pc]:= 'pushl (%eax)';     pc:=pc+1;
    end;
end;

if x.mode=VarClass then
  begin
    Str(x.a, temp);
    code[pc]:= 'pushl '+ ' $' + temp ; pc:=pc+1;
    if x.lev=1 then
      begin { no operation needed } end;

    if x.indirect then
      begin
        code[pc]:= 'popl  %eax' ; pc:=pc+1;
        code[pc]:= 'addl  %eax, (%esp)'; pc:=pc+1;
      end;
    if (x.o<>0) then
      begin
        Str(x.o, temp);
        code[pc]:= 'pushl '+ '$' + temp;  pc:=pc+1;
        code[pc]:= 'popl  %eax';    pc:=pc+1;
        code[pc]:= 'addl  %eax, (%esp)';    pc:=pc+1;
      end;
    if not LeaveAddress then
      begin
        if (x.lev=0) then
          begin
            code[pc]:= 'popl  %eax';    pc:=pc+1;
            code[pc]:= 'addl  $global_var, %eax'; pc:=pc+1;
            code[pc]:= 'pushl  (%eax)';    pc:=pc+1;
          end
        else if x.lev=1 then
          begin
            code[pc]:= 'movl 8(%ebp), %eax';  pc:=pc+1;
            code[pc]:= 'addl %eax, (%esp)' ; pc:=pc+1;
            code[pc]:= 'popl  %eax';    pc:=pc+1;
            code[pc]:= 'pushl (%eax)';    pc:=pc+1;
          end
        else { x.lev= 2 or 3}
          begin
            code[pc]:= 'popl  %eax';    pc:=pc+1;
```

```
              code[pc]:= 'addl  %ebp, %eax'; pc:=pc+1;
              code[pc]:='pushl  (%eax)';     pc:=pc+1;
            end;

          if x.tp^.form = Bool then x.bool_set:=true;
        end;
    end
  else if x.mode=ConstClass then
    begin
      Str(x.a, temp);
      code[pc]:= 'pushl '+ '$' + temp;  pc:=pc+1;
      x.mode := EmitClass
    end;
    {# write a stub in the assembly file }
    code[pc]:= TAB + '# End of procedure LoadItem_32(,).' ; pc:=pc+1;
  end;

procedure IncLevel(n: longint);
begin
  curlev := curlev + n
end;

procedure MakeConstItem(var x: Item; tp: Typ; val: longint);
begin
  x.mode := ConstClass;
  x.tp := tp;
  x.a := val
end;

procedure MakeItem(var x: Item; y: Objct);
begin
  x.mode := y^.cls; x.tp := y^.tp; x.lev := y^.lev; x.a := y^.val;
  x.indirect := false;
  x.bool_set:=false;
  x.sc :=false;
  x.push_placeholder := true;
  x.parSize := y^.parSize; x.o := 0 ;
  x.r:=0
end;

procedure Field_32(var x: Item; y: Objct);    { x := x.y }
begin
  x.o := x.o + y^.val;  x.tp := y^.tp;
  code[pc]:= '  # End of procedure Field_32(,)';  pc:=pc+1;
end;

{As this procedure exits, the offset of x[y] is placed on top of stack }
procedure Index_32 (var x, y: Item);    { x := x[y] }
var temp: string;
    upper_bound: longint;
begin
  if y.tp <> intType then Mark ('index not integer');
  if y.mode = ConstClass then
```

```
    begin
      if (y.a < x.tp^.lower) or (y.a >= x.tp^.len + x.tp^.lower) then Mark ('bad
          index');
      x.o := x.o + ((y.a - x.tp^.lower) * x.tp^.base^.size);
      {fixing bugs: x[y] when y is constant}
       Str(x.o, temp);
       code[pc]:= 'addl ' + '$' + temp + ', (%esp)';        pc:=pc+1;
       x.o := 0
    end
  else
    begin
      Str( x.tp^.lower, temp);
      code[pc]:= 'movl '+'$'+temp + ', %eax';    pc:=pc+1;
      code[pc]:= 'cmp  %eax, (%esp)';     pc:=pc+1;
      code[pc]:= 'jl  .trap';             pc:=pc+1;

      upper_bound:= x.tp^.lower + x.tp^.len -1;
      Str(upper_bound, temp);
      code[pc]:= 'movl '+ '$'+ temp + ', %ecx';  pc:=pc+1;
      code[pc]:= 'cmp  %ecx, (%esp)';     pc:=pc+1;
      code[pc]:= 'jg  .trap';            pc:=pc+1;

      Str(x.tp^.base^.size, temp);
      code[pc]:= 'movl '+ '$'+ temp + ', %eax';       pc:=pc+1;
      code[pc]:= 'popl  %ecx';                         pc:=pc+1;
      Str( x.tp^.lower, temp);

      code[pc]:= 'subl  $' + temp + ', %ecx';          pc:=pc+1;
      code[pc]:= 'mull  %ecx   #if operands are too large, %eax may not be large
          enough to hold the result';                 pc:=pc+1;         {use unsigned
           multiplication 'mull'}
      code[pc]:= 'addl  %eax, (%esp)';                 pc:=pc+1;
    end;
  x.tp := x.tp^.base;
  code[pc]:= '  # End of procedure Index_32(,)';  pc:=pc+1;
end;

procedure loadBool(var x: Item);
begin
  if x.tp^.form <> Bool then
    begin  Mark ('[loadBool] Boolean?'); writeln(x.tp^.typename);
    end;
  x.mode := CondClass; x.a := 0; x.b := 0; x.c := 0  {or x.c=1 ?}
end;

procedure PutOp_32(cd: longint; var x, y: Item);
var sw : boolean; temp: string;
begin
  if x.mode = ConstClass then begin TestRange (x.a); {LoadItem_32 (x, false)} end;
  if y.mode = ConstClass then begin TestRange (y.a); {LoadItem_32 (y, false)} end;

  {##— put (cd, 0, 0) }
  {irst pop the top two items on stack, compute, then put the result back on stack}
```

```
{
  code[pc]:=  'popl  %ecx';   pc:=pc+1;
  code[pc]:=  'popl  %eax';   pc:=pc+1;
}
case cd of
  CMPOP:
    begin
      if (x.mode = ConstClass) and (y.mode <> ConstClass) then sw := true
      else sw := false;

      if x.mode = ConstClass then begin TestRange (x.a); LoadItem_32 (x, false)
          end;
      if y.mode = ConstClass then begin TestRange (y.a); LoadItem_32 (y, false)
          end;
      if sw then
        begin
          code[pc]:=  'popl  %eax';   pc:=pc+1;
          code[pc]:=  'popl  %ecx';   pc:=pc+1;
          code[pc]:=  'pushl  %eax';   pc:=pc+1;
          code[pc]:=  'pushl  %ecx';   pc:=pc+1;
        end;
      code[pc]:=  'popl  %ecx';   pc:=pc+1;
      code[pc]:=  'popl  %eax';   pc:=pc+1;

    end;
  SUBOP:
    begin
      if y.mode = ConstClass then
        begin
          Str(y.a, temp); y.mode := EmitClass;
          code[pc]:=  'popl  %eax';     pc:=pc+1;
          code[pc]:=  'subl  $' + temp + ', %eax';     pc:=pc+1;
          code[pc]:=  'pushl  %eax';     pc:=pc+1;
        end
      else if x.mode=ConstClass then
        begin
          code[pc]:=  'popl  %ecx';     pc:=pc+1; {eax=y}
          Str(x.a, temp); x.mode := EmitClass;
          code[pc]:=  'pushl  $'+temp;     pc:=pc+1;
          code[pc]:=  'popl  %eax';     pc:=pc+1; {ecx=x}
          code[pc]:=  'subl  %ecx, %eax';     pc:=pc+1;
          code[pc]:=  'pushl  %eax';     pc:=pc+1;
        end
      else begin
          code[pc]:=  'popl  %ecx';   pc:=pc+1; {ecx=y}
          code[pc]:=  'popl  %eax';   pc:=pc+1; {eax=x}
          code[pc]:=  'subl  %ecx, %eax';    pc:=pc+1;
          code[pc]:=  'pushl  %eax';     pc:=pc+1;
            end;
    end;
  ADDOP:
    begin
      if y.mode=ConstClass then
```

```
      begin
        Str(y.a, temp); y.mode := EmitClass;
        code[pc]:= 'popl  %eax';     pc:=pc+1;
        code[pc]:= 'addl  $'+temp+ ', %eax';   pc:=pc+1;
        code[pc]:= 'pushl  %eax';     pc:=pc+1;
      end
    else if x.mode=ConstClass then
      begin
        code[pc]:= 'popl  %eax';     pc:=pc+1; {eax=y}
        Str(x.a, temp);
        x.mode := EmitClass;
        code[pc]:= 'addl  $'+temp+ ', %eax';   pc:=pc+1;
        code[pc]:= 'pushl  %eax';     pc:=pc+1;
      end
    else begin
        code[pc]:= 'popl  %ecx';  pc:=pc+1; {ecx=y}
        code[pc]:= 'popl  %eax';  pc:=pc+1; {eax=x}
        code[pc]:= 'addl  %ecx,  %eax';   pc:=pc+1;
        code[pc]:= 'pushl  %eax';     pc:=pc+1;
          end;
  end;

MULOP:
  begin
    if x.mode = ConstClass then begin TestRange (x.a); LoadItem_32 (x, false)
        end;
    if y.mode = ConstClass then begin TestRange (y.a); LoadItem_32 (y, false)
        end;
    code[pc]:=  'popl  %ecx';  pc:=pc+1;
    code[pc]:=  'popl  %eax';  pc:=pc+1;
    code[pc]:=  'cmp $32767, %eax'; pc:=pc+1;
    code[pc]:=  'jg  .overflow';      pc:=pc+1;
    code[pc]:=  'cmp $32767, %ecx'; pc:=pc+1;
    code[pc]:=  'jg  .overflow';      pc:=pc+1;
    code[pc]:=  'imull  %ecx, %eax';    pc:=pc+1;
    code[pc]:=  'pushl  %eax' ;        pc:=pc+1;
  end;

DIVOP:
  begin
    if (x.mode = ConstClass) and (y.mode <> ConstClass) then sw := true else sw
         := false;
    if x.mode = ConstClass then begin TestRange (x.a); LoadItem_32 (x, false)
        end;
    if y.mode = ConstClass then begin TestRange (y.a); LoadItem_32 (y, false)
        end;
    if sw then
      begin
        code[pc]:= 'popl  %eax';  pc:=pc+1;
        code[pc]:= 'popl  %ecx';  pc:=pc+1;
        code[pc]:= 'pushl  %eax';  pc:=pc+1;
        code[pc]:= 'pushl  %ecx';  pc:=pc+1;
      end;
```

```
          {first pop the top two items on stack, compute, then put the result back on
             stack}
          code[pc]:= 'popl  %ecx';  pc:=pc+1;
          code[pc]:= 'popl  %eax';  pc:=pc+1;
          code[pc]:=  'movl  $0, %edx';      pc:=pc+1;
          {# quotient is in eax, remainder in edx}
          code[pc]:= 'idivl  %ecx';         pc:=pc+1;
          code[pc]:= 'pushl  %eax';         pc:=pc+1;
        end;

      {A MOD B = x  -> A=n*B+x -> x=A-(A div B)*B }
      MODOP:
        begin
          if (x.mode = ConstClass) and (y.mode <> ConstClass) then sw := true else
             sw := false;
          if x.mode = ConstClass then begin TestRange (x.a); LoadItem_32 (x, false)
             end;
          if y.mode = ConstClass then begin TestRange (y.a); LoadItem_32 (y, false)
             end;
          if sw then
            begin
              code[pc]:= 'popl  %eax';  pc:=pc+1;
              code[pc]:= 'popl  %ecx';  pc:=pc+1;
              code[pc]:= 'pushl  %eax';  pc:=pc+1;
              code[pc]:= 'pushl  %ecx';  pc:=pc+1;
            end;
          {first pop the top two items on stack, compute, then put the result back
             on stack}
          code[pc]:= 'popl  %ecx';  pc:=pc+1;
          code[pc]:= 'popl  %eax';  pc:=pc+1;
          code[pc]:=  'movl  $0, %edx';      pc:=pc+1;
          code[pc]:=  'idivl  %ecx';         pc:=pc+1;
          {# quotient is in eax, remainder in edx}
          code[pc]:=  'pushl  %edx' ;        pc:=pc+1;
        end;
      end;{end of case statement}

    code[pc]:= TAB + '# End of PutOp_32(,,) procedure.';  pc:=pc+1;
  end;

  procedure Op1_32(op: Symbol; var x: Item);   { x := op x }
  var relational_op: longint;
       opstr, temp: string;
  begin
    if op = MinusSym then
      if x.tp^.form <> Int then Mark ('bad type')
      else if x.mode = ConstClass then x.a := -x.a
      else begin
             {movl $0, %eax | subl (%esp),%eax | movl %eax,(%esp) }
             code[pc]:= 'movl  $0, %eax'; pc:=pc+1;
             code[pc]:= 'subl  (%esp), %eax'; pc:=pc+1;
             code[pc]:= 'movl  %eax, (%esp)'; pc:=pc+1;
           end
```

```
else if op = NotSym then
  begin
    if x.mode <> CondClass then loadBool(x);
    x.c := negated(x.c);
  end

else if op = AndSym then
  begin
    if x.mode <> CondClass then loadBool(x);
    if not x.bool_set then
      begin
        relational_op:= BEQOP + negated(x.c);
        x.r:= pc;
        opstr := TransformToStr(relational_op);
        {# compare the top of stack with zero, then do a conditional jump}
        code[pc]:= 'popl  %ecx' ;    pc:=pc+1;
        code[pc]:= 'cmp   $0, %ecx';  pc:=pc+1;
        Jumpto(opstr, 'branchfrom' ,x.r);            { jump to: x:=false}
        code[pc]:= 'pushl $0x1'; pc:=pc+1;
        Jumpto('jmp', 'exitfrom' ,x.r);
        MakeJumpLabel('branchfrom', x.r);
        code[pc]:= 'pushl $0x0'; pc:=pc+1;
        {jmp  false_and_others_ + x.r}
        {'false AND others' => enable short-circuited evaluation}
        Str(x.r, temp);
        code[pc]:= '  jmp   false_and_others_' + temp; pc:=pc+1;
        MakeJumpLabel('exitfrom', x.r);
      end
    else
      begin
        code[pc]:= 'movl (%esp), %ecx' ;    pc:=pc+1;
        x.r:= pc;
        code[pc]:= 'cmp    $0x0, %ecx';   pc:=pc+1;
        {jmp  false_and_others_ + x.r}
        {'false AND others' => enable short-circuited evaluation}
        Str(x.r, temp);
        code[pc]:= '  je   false_and_others_' + temp; pc:=pc+1;
      end;
  end
else if op = OrSym then
  begin
    if x.mode <> CondClass then loadBool (x);
    if not x.bool_set then
      begin
        relational_op:= BeqOP + (x.c);
        x.r:= pc;
        opstr := TransformToStr(relational_op);
        { movl $0, %eax | popl %ebx | cmp %eax, %ebx }
        //code[pc]:= 'movl  $0, %eax';  pc:=pc+1;
        code[pc]:= 'popl  %ecx' ;    pc:=pc+1;
        code[pc]:= 'cmp   $0,  %ecx';   pc:=pc+1;
        Jumpto(opstr, 'branchfrom' ,x.r);
        code[pc]:= 'pushl $0x0'; pc:=pc+1;
```

```
          Jumpto('jmp', 'exitfrom' ,x.r);
          MakeJumpLabel('branchfrom', x.r);
          code[pc]:= 'pushl $0x1'; pc:=pc+1;
          {'true AND others' => enable short-circuited evaluation}
          Str(x.r, temp);
          code[pc]:= ' jmp  true_and_others_' + temp; pc:=pc+1;
          MakeJumpLabel('exitfrom', x.r);
       end
     else
       begin
          code[pc]:= 'movl (%esp), %ecx' ;    pc:=pc+1;
          x.r:= pc;
          code[pc]:= 'cmp   $0x1, %ecx';  pc:=pc+1;
          {jmp  false_and_others_ + x.r}
          {'false AND others' => enable short-circuited evaluation}
          Str(x.r, temp);
          code[pc]:= '  je  true_and_others_' + temp; pc:=pc+1;
       end;

    end;
    code[pc]:= TAB + '# End of Op1_32(,) procedure.';  pc:=pc+1;
end;


procedure Op2_32(op: Symbol; var x, y: Item);   { x := x op y }
var
  relational_op: longint;
  opstr, temp : string;
begin
  if (x.tp^.form = Int) and (y.tp^.form = Int) then
    if (x.mode = ConstClass) and (y.mode = ConstClass) then
      if op = PlusSym then x.a := x.a + y.a
      else if op = MinusSym then x.a := x.a - y.a
      else if op = TimesSym then x.a := x.a * y.a
      else if op = DivSym then x.a := x.a div y.a
      else if op = ModSym then x.a := x.a mod y.a
      else Mark ('bad type')
   else
      if op = PlusSym then PutOp_32 (ADDOP, x, y)
      else if op = MinusSym then PutOp_32 (SUBOP, x, y)
      else if op = TimesSym then PutOp_32 (MULOP, x, y)
      else if op = DivSym then PutOp_32 (DIVOP, x, y)
      else if op = ModSym then PutOp_32 (MODOP, x, y)
      else Mark ('bad type')

   else if (x.tp^.form = Bool) and (y.tp^.form = Bool) then
     begin
       if y.mode <> CondClass then loadBool (y);
       if op = OrSym then
         begin
           if not y.bool_set then
             begin
               relational_op:= BeqOP + (y.c);
```

```
                y.r:= pc;
                opstr := TransformToStr(relational_op);
                //code[pc]:= 'movl  $0, %eax';  pc:=pc+1;
                code[pc]:= 'popl  %ecx' ;    pc:=pc+1;
                code[pc]:= 'cmp   $0, %ecx';  pc:=pc+1;
                Jumpto(opstr, 'branchfrom' ,y.r);          { jump to: x:=true}
                code[pc]:= 'pushl $0x0'; pc:=pc+1;
                Jumpto('jmp', 'exitfrom' ,y.r);
                MakeJumpLabel('branchfrom', y.r);
                code[pc]:= 'pushl $0x1'; pc:=pc+1;
                MakeJumpLabel('exitfrom', y.r);
              end;
            {compute x:=x or y}
            code[pc]:= 'popl  %eax'; pc:=pc+1;
            code[pc]:= 'popl  %ecx'; pc:=pc+1;
            code[pc]:= 'or    %ecx, %eax'; pc:=pc+1;
            code[pc]:= 'pushl  %eax'; pc:=pc+1;
            {true_and_others_ + x.r:
                   nop
            }
            Str(x.r, temp);
            code[pc]:= ' true_and_others_' + temp + ':' ; pc:=pc+1;
          end
        else if op = AndSym then
          begin
            if not y.bool_set then
              begin
                relational_op:= BEQOP + (y.c);
                y.r:= pc;
                opstr := TransformToStr(relational_op);
                {# compare the top of stack with zero, then do a conditional jump}
                 //code[pc]:= 'movl  $0, %eax';  pc:=pc+1;
                code[pc]:= 'popl  %ecx' ;    pc:=pc+1;
                code[pc]:= 'cmp   $0, %ecx';  pc:=pc+1;
                {# new in 1.1}
                Jumpto(opstr, 'branchfrom' ,y.r);          { jump to: x:=false}
                code[pc]:= 'pushl $0x0'; pc:=pc+1;
                Jumpto('jmp', 'exitfrom' ,y.r);
                MakeJumpLabel('branchfrom', y.r);
                code[pc]:= 'pushl $0x1'; pc:=pc+1;
                MakeJumpLabel('exitfrom', y.r);
              end;
            {compute x:=x and y}
            code[pc]:= 'popl  %eax'; pc:=pc+1;
            code[pc]:= 'popl  %ecx'; pc:=pc+1;
            code[pc]:= 'and    %ecx, %eax'; pc:=pc+1;
            code[pc]:= 'pushl  %eax'; pc:=pc+1;
            {false_and_others_ + x.r:
                   nop
            }
            Str(x.r, temp);
            code[pc]:= ' false_and_others_' + temp + ':' ; pc:=pc+1;
            code[pc]:= '      nop'; pc:=pc+1;
```

```
          end;
        x.bool_set := true;
        x.c:= 0;
      end
    else Mark ('bad type');
    code[pc]:= TAB + '# End of Op2_32(,,) procedure.';  pc:=pc+1;
  end;

 procedure Relation_32(op: Symbol; var x, y: Item);   { x := x relational_op y}
 begin
   if x.r=0 then x.r:=pc;
   if (x.tp^.form <> Int) or (y.tp^.form <> Int) then Mark ('bad type')
   else
     begin
       PutOp_32 (SUBOP, x, y);      {# subop = cmpop}
       x.c := ord (op) - ord (EqlSym);
     end;
   x.mode := CondClass; x.tp := boolType; x.a := 0; x.b := 0 ;
   code[pc]:= TAB + '# End of Relation_32(,,) procedure.';  pc:=pc+1;
 end;

{#ia32 proc : Store_32(,) }
 procedure Store_32(var x, y: Item); { x := y }
 var
   relational_op: longint;
   opstr: string;
 begin
   if (x.tp^.form in [Bool, Int]) and (x.tp^.form = y.tp^.form) then
     begin
       if y.mode = CondClass then
         begin
           {# load the boolean value on the stack }
           if not x.bool_set then
              begin
                relational_op:= BeqOP + (y.c);
                x.r:= pc;
                opstr := TransformToStr(relational_op);
                { movl $0, %eax | popl %ebx | cmp %eax, %ebx }
                code[pc]:= 'movl  $0, %eax';  pc:=pc+1;
                code[pc]:= 'popl  %ecx' ;  pc:=pc+1;
                code[pc]:= 'cmp   %eax, %ecx';  pc:=pc+1;

                Jumpto(opstr, 'branchfrom' ,x.r);       { jump to: x:=true}
                code[pc]:= 'pushl $0x0'; pc:=pc+1;
                Jumpto('jmp', 'exitfrom' ,x.r);
                MakeJumpLabel('branchfrom', x.r);
                code[pc]:= 'pushl $0x1'; pc:=pc+1;
                MakeJumpLabel('exitfrom', x.r)
              end;
         end
       else if y.mode = ConstClass then LoadItem_32(y, false);
       if x.mode = VarClass then
         if x.lev = 0 then
```

```
            begin
              {popl %eax | popl %ebx | addl $global_var,%ebx | movl %eax, (%ebx) }
              code[pc]:= 'popl  %eax' ;  pc:=pc+1;
              code[pc]:= 'popl  %ecx' ;  pc:=pc+1;
              code[pc]:= 'addl  $global_var, %ecx'; pc:=pc+1;
              code[pc]:= 'movl  %eax, (%ecx)';   pc:=pc+1;
            end
          else if x.lev = 1 then
            begin
              code[pc]:= 'popl  %eax' ;  pc:=pc+1;
              code[pc]:= 'popl  %ecx' ;  pc:=pc+1;
              code[pc]:= 'addl  8(%ebp), %ecx';   pc:=pc+1;
              code[pc]:= 'movl  %eax, (%ecx)'; pc:=pc+1;
            end
          else
            begin
              {popl %eax | popl %ebx | addl %ebp, %ebx | movl %eax, (%ebx) }
              code[pc]:= 'popl  %eax' ;  pc:=pc+1;
              code[pc]:= 'popl  %ecx' ;  pc:=pc+1;
              code[pc]:= 'addl  %ebp, %ecx'; pc:=pc+1;
              code[pc]:= 'movl  %eax, (%ecx)';   pc:=pc+1;
            end
        else if x.mode = ParClass then
            {popl %eax | popl %ebx | movl (%ebx), %edx | movl %eax, (%edx) }
            begin
              code[pc]:= 'popl  %eax' ;  pc:=pc+1;
              code[pc]:= 'popl  %ecx' ;  pc:=pc+1;
              code[pc]:= 'movl  %eax, (%ecx)';  pc:=pc+1;
            end
        else Mark ('illegal assignment');

        code[pc]:= TAB + '# End of Store_32(,) procedure.';  pc:=pc+1;
      end
    else Mark ('incompatible assignment')
  end;


{##ia32 proc : parameter_32(,,)}
  procedure Parameter_32(var x: Item; ftyp: Typ; cls: Class);
  begin
    if x.tp = ftyp then
      begin
        if cls = ParClass then {reference parameter}
          if x.mode = VarClass then {x in position of a VAR parameter, and x is a
              variable}
            begin
              if x.lev=1 then pc := pc - 5
              else pc:= pc-4;
              {roll back, leave address on stack not the value}
              {roll back, be cautious when the code inside loaditem_32->paraclass
                  changes}
              if (x.lev = 0)  then
                begin
```

```
                { addl $global_var , (%esp) }
                code[pc]:= 'addl $global_var , (%esp)';  pc:=pc+1
              end
            else if (x.lev=1) then
              begin
                { asm code: movl 8(%ebp), %eax | addl %eax, (%esp)}
                code[pc]:= 'movl 8(%ebp), %eax';  pc:=pc+1;
                code[pc]:= 'addl %eax, (%esp)';   pc:=pc+1;
              end
            else
              begin
                { addl %ebp, (%esp) }
                code[pc]:= 'addl %ebp ,(%esp)'; pc:=pc+1
              end
          end
        else if x.mode = ParClass then pc := pc - 3   {be cautious when loaditem_32
            ->paraclass changes }
             {this happens when proc A calls Proc B, when the parameter of B is a
                 param of A}
        else Mark ('illegal parameter mode')
      else { value parameter }
        if x.mode = ConstClass then loadItem_32 (x, false)
        {in case of x.mode=varclass , the item x's value is loaded on the stack
            already(see factor()-> loaditem_32(x, false), it loads the value of x.
         so no need to do anything here.}
    end
  else Mark ('bad parameter type');
  code[pc]:= TAB + '# End of Parameter_32(,,) procedure.'; pc:=pc+1;
end;

{Place_boolean_val: push the bool value of a conditional expr on stack, if it is
    not done yet.}
procedure Place_boolean_val(var x: Item);
var relational_op: longint;
    opstr: string;
begin
  if x.tp^.form = Bool then
    begin
      if x.mode <> CondClass then loadBool (x);

      if not x.bool_set then
        begin
          relational_op:= BeqOP + (x.c);
          x.r:= pc;
          opstr := TransformToStr(relational_op);
          { movl $0, %eax | popl %ebx | cmp %eax, %ebx }
          code[pc]:= 'movl  $0, %eax';  pc:=pc+1;
          code[pc]:= 'popl  %ecx' ;  pc:=pc+1;
          code[pc]:= 'cmp   %eax,  %ecx';  pc:=pc+1;

          Jumpto(opstr, 'branchfrom' ,x.r);          { jump to: x:=true}
          code[pc]:= 'pushl $0x0'; pc:=pc+1;
          Jumpto('jmp', 'exitfrom' ,x.r);
```

```
          MakeJumpLabel('branchfrom', x.r);
          code[pc]:= 'pushl $0x1'; pc:=pc+1;
          MakeJumpLabel('exitfrom', x.r);
        end;

      code[pc]:= TAB + '# End of Place_boolean_val() procedure.';  pc:=pc+1;
    end
  else begin Mark ('[Cjump] Boolean?'); x.a := pc end
end;


{##ia32 proc: Cjump_32();}
procedure CJump_32(var x: Item);
var relational_op: longint;
    opstr: string;
begin
  if x.tp^.form = Bool then
    begin
      if x.mode <> CondClass then loadBool (x);

      if not x.bool_set then
        begin
          relational_op:= BeqOP + (x.c);
          x.r:= pc;
          opstr := TransformToStr(relational_op);
          {movl $0, %eax | popl %ebx | cmp %eax, %ebx }
          code[pc]:= 'movl  $0, %eax';  pc:=pc+1;
          code[pc]:= 'popl  %ecx' ;  pc:=pc+1;
          code[pc]:= 'cmp   %eax, %ecx';  pc:=pc+1;

          Jumpto(opstr, 'branchfrom' ,x.r);        { jump to: x:=true}
          code[pc]:= 'pushl $0x0'; pc:=pc+1;
          Jumpto('jmp', 'exitfrom' ,x.r);
          MakeJumpLabel('branchfrom', x.r);
          code[pc]:= 'pushl $0x1'; pc:=pc+1;
          MakeJumpLabel('exitfrom', x.r);

          {# make jump decisions based on the result of the logical expn  }
          x.r:=pc;
          code[pc]:= 'popl  %eax';  pc:=pc+1;        {# move the value of boolean
              expn to %eax }
          code[pc]:= 'movl  $0x0, %ecx';  pc:=pc+1;
          code[pc]:= 'cmp  %ecx, %eax';  pc:=pc+1;
          jumpto('je', 'is_FALSE', x.r);
          {if the top elmt of stack=0 (false), then jump to the else block}
        end
      else
        begin
          relational_op:= BeqOP + (x.c);
          opstr := TransformToStr(relational_op);
          x.r:=pc;
          code[pc]:= 'popl  %eax';  pc:=pc+1;       {# move the value of boolean
              expn to %eax }
```

```
          code[pc]:= 'movl  $0x0,  %ecx';    pc:=pc+1;
          code[pc]:= 'cmp  %ecx, %eax';    pc:=pc+1;
          jumpto(opstr, 'is_FALSE', x.r);
        end;

      code[pc]:= TAB + '# End of Cjump() procedure.';   pc:=pc+1;
    end
  else begin Mark ('[Cjump] Boolean?'); x.a := pc end
end;

{##ia32 proc: Call_32().}
procedure Call_32 (name: Identifier);
begin
  { call name }
  code[pc]:= 'call ' + 'proc_' + name;    pc:=pc+1;
  code[pc]:= TAB + '# End of call_32() procedure.';   pc:=pc+1;
end;

procedure IOCall_32 (var x, y: Item);
begin
  if x.a < 4 then
    if y.tp^.form <> Int then Mark ('integer is expected');
  if x.a = 1 then {read(X)}
    begin
      {# read the input from console;  store the readed value to %edi; store this
          value to the variable }
        { linux system call: 3->read ; 0->standard input; %ecx->buffer that stores
            the input; %edx-> number of bytes to read }

       code[pc]:= 'popl  %ecx';    pc:= pc+1;
       if y.lev = 0 then
         begin code[pc]:= 'addl  $global_var, %ecx'; pc:=pc+1;  end
       else if y.lev=1 then
         begin
           code[pc]:= 'addl  8(%ebp), %ecx'; pc:=pc+1;
         end
       else begin {y.lev=2 or 3}
               code[pc]:= 'addl  %ebp, %ecx';   pc:=pc+1 ;
             end;

       code[pc]:= 'pushl  %ecx';      pc:=pc+1;      {## push the address of x on
           stack}
       code[pc]:= 'pushl  $strfmt1';   pc:=pc+1;      {## push the address of format
           string}
       code[pc]:= 'call scanf';       pc:=pc+1;
       code[pc]:= 'add  $8, %esp';    pc:=pc+1;

       code[pc]:= TAB + '# End of IOcall->read().'; pc:=pc+1;
    end
  else if x.a = 2 then {write(expn)}
    begin
      if y.mode = ConstClass then loadItem_32 (y, false);
      { ## use C functions instead of system calls.
```

```
       pushl $strfmt
       call printf
       }
      code[pc]:= 'pushl  $strfmt2';  pc:=pc+1;
      code[pc]:= 'call  printf';    pc:=pc+1;
      code[pc]:= 'add  $8, %esp'; pc:=pc+1;
      code[pc]:= TAB + '# End of IOcall->write().'; pc:=pc+1;
    end
  else if x.a=3 then {random(x)}
    begin
      code[pc]:= 'popl  %ecx';    pc:= pc+1;  {store offset of x in %ecx}
      if y.lev = 0 then
        begin code[pc]:= 'addl  $global_var, %ecx'; pc:=pc+1;  end
      else if y.lev=1 then
        begin
          code[pc]:= 'addl  8(%ebp), %ecx'; pc:=pc+1;
        end

      else begin {y.lev=2 or 3}
              code[pc]:= 'addl  %ebp, %ecx'; pc:=pc+1 ;
            end;

      code[pc]:= 'pushl  %ecx  #push the absolute address of x on stack';  pc:=pc
          +1;
       {## push the absolute address of x on stack}
       {call—>   rand_num = (rand()) % 50;}

      code[pc]:='call rand';      pc:=pc+1;
      code[pc]:='movl $10000, %ecx';  {store the divisor,10000, in ecx}   pc:=pc+1;
      code[pc]:='movl $0, %edx  #cleanup the content of edx';       pc:=pc+1;
      code[pc]:='divl %ecx  #the remainder is in edx';              pc:=pc+1; {#the
           remainder is in edx}

      code[pc]:='popl %eax';       pc:=pc+1;
      code[pc]:='movl %edx, (%eax)';  pc:=pc+1;  {store the remainder in absolute
          address of x}
    end

  else  {writeln; use linux system call to implement }
     begin
       code[pc]:= 'movl  $4, %eax';    pc:=pc+1;
       code[pc]:= 'pushl %ebx';        pc:=pc+1;
       code[pc]:= 'movl  $1, %ebx';    pc:=pc+1;
       code[pc]:= 'movl  $eol, %ecx';  pc:=pc+1;
       code[pc]:= 'movl  $1, %edx';        pc:=pc+1;
       code[pc]:= 'int  $0x80';            pc:=pc+1;
       code[pc]:= 'popl %ebx';       pc:=pc+1;
       code[pc]:= '# end of IOcall->writeln.'; pc:=pc+1
     end;
end;

procedure inner_loop_one_and_three(n, offset_next, offset_count, offset_n,
    offset_done: longint;
```

```
                                      action_queue: AQ);
var
  temp: identifier;
  k, anchor, start_first_inner_loop, end_first_inner_loop : longint;
  end_if_branch, end_if_next, end_if_else_nesting: array [0..ACTION_LIMIT] of
      longint;

begin
{start the first inner while-loop}
  code[pc]:= '#  execute the 1st inner loop'; pc:=pc+1;
  start_first_inner_loop := pc;
  Str(start_first_inner_loop, temp);
  code[pc]:= 'startwhile_' + temp + ':' ;  pc:=pc+1;
  Str(offset_count, temp);
  code[pc]:= 'pushl $' + temp;  pc:=pc+1;
  code[pc]:= 'popl  %eax';  pc:=pc+1;
  code[pc]:= 'addl  %ebp, %eax';  pc:=pc+1;
  code[pc]:= 'pushl (%eax)';  pc:=pc+1;
  code[pc]:= 'pushl $0';  pc:=pc+1;
  code[pc]:= 'popl  %ecx';  pc:=pc+1;
  code[pc]:= 'popl  %eax';  pc:=pc+1;
  code[pc]:= 'subl  %eax, %ecx';  pc:=pc+1;
  code[pc]:= 'pushl %eax';  pc:=pc+1;
  anchor:=pc;
  code[pc]:= 'movl  $0, %eax';  pc:=pc+1;
  code[pc]:= 'popl  %ecx';  pc:=pc+1;
  code[pc]:= 'cmp   %eax,  %ecx';  pc:=pc+1;
  Str(anchor, temp);
  code[pc]:= 'jg  branchfrom_' + temp;  pc:=pc+1;
  code[pc]:= 'pushl  $0x0';  pc:=pc+1;
  code[pc]:= 'jmp  exitfrom_' + temp;  pc:=pc+1;
  code[pc]:= 'branchfrom_' + temp + ':';  pc:=pc+1;
  code[pc]:= 'pushl  $0x1';  pc:=pc+1;
  code[pc]:= 'exitfrom_' + temp + ':';  pc:=pc+1;
  end_first_inner_loop := pc;
  code[pc]:= 'popl  %eax';  pc:=pc+1;
  code[pc]:= 'movl  $0x0, %ecx';  pc:=pc+1;
  code[pc]:= 'cmp  %ecx, %eax';  pc:=pc+1;
  Str(end_first_inner_loop, temp);
  code[pc]:= 'je  is_FALSE_' + temp;  pc:=pc+1;

  // the first(or third) inner loop body
  k:=0;
  while k<n do
    begin
      { pushl $-4  # offset(next)=-4 }
      Str(offset_next, temp);
      code[pc]:= 'pushl  $' + temp;   pc:=pc+1;
      code[pc]:= 'popl  %eax';        pc:=pc+1;
      code[pc]:= 'addl  %ebp, %eax';  pc:=pc+1;
      code[pc]:= 'pushl (%eax)';     pc:=pc+1;
      Str(k, temp);
      code[pc]:= 'pushl  $' + temp;  pc:=pc+1;
```

```
        code[pc]:= 'popl  %eax';          pc:=pc+1;
        code[pc]:= 'popl  %ecx';          pc:=pc+1;
        code[pc]:= 'subl  %ecx, %eax';          pc:=pc+1;
        code[pc]:= 'pushl %eax';          pc:=pc+1;

        anchor:= pc;
        Str(anchor, temp);
        code[pc]:= 'movl  $0, %eax';          pc:=pc+1;
        code[pc]:= 'popl  %ecx';              pc:=pc+1;
        code[pc]:= 'cmp   %eax, %ecx';        pc:=pc+1;
        code[pc]:= 'je  branchfrom_' + temp;          pc:=pc+1;
        code[pc]:= 'pushl  $0x0';        pc:=pc+1;
        code[pc]:= 'jmp  exitfrom_' + temp;          pc:=pc+1;
        code[pc]:= 'branchfrom_' + temp + ':';          pc:=pc+1;
        code[pc]:= 'pushl  $0x1';        pc:=pc+1;
        code[pc]:= 'exitfrom_' + temp + ':';          pc:=pc+1;

        end_if_branch[k]:= pc;
        Str(end_if_branch[k], temp);
        code[pc]:= 'popl  %eax';          pc:=pc+1;
        code[pc]:= 'movl  $0x0, %ecx';        pc:=pc+1;
        code[pc]:= 'cmp  %ecx, %eax';         pc:=pc+1;
        code[pc]:= 'je  is_FALSE_' + temp;  pc:=pc+1;

        { call   actions_queue[k].action_guard
          pushl %eax             //  action_guard return the truth value to %eax.
        }
        code[pc]:= 'call ' + action_queue[k].action_guard ;  pc:=pc+1;
        code[pc]:= 'pushl  %eax';       pc:=pc+1;
        anchor:=pc;
        Str(anchor, temp);
        code[pc]:= 'movl  $0, %eax';  pc:=pc+1;
        code[pc]:= 'popl  %ecx';  pc:=pc+1;
        code[pc]:= 'cmp  %eax, %ecx';  pc:=pc+1;
        code[pc]:= 'jne  branchfrom_' + temp;  pc:=pc+1;
        code[pc]:= 'pushl  $0x0';               pc:=pc+1;
        code[pc]:= 'jmp  exitfrom_' + temp;          pc:=pc+1;
        code[pc]:= 'branchfrom_' + temp + ':';      pc:=pc+1;
        code[pc]:= 'pushl  $0x1';        pc:=pc+1;
        code[pc]:= 'exitfrom_' + temp + ':';        pc:=pc+1;

        end_if_next[k]:= pc;
        Str(end_if_next[k], temp);
        code[pc]:= 'popl  %eax';          pc:=pc+1;
        code[pc]:= 'movl  $0x0, %ecx';        pc:=pc+1;
        code[pc]:= 'cmp  %ecx, %eax';         pc:=pc+1;
        code[pc]:= 'je  is_FALSE_' + temp;  pc:=pc+1;

        { # done:= true;}
        Str(offset_done, temp);
        code[pc]:= 'pushl  $' + temp;          pc:=pc+1;
        code[pc]:= 'pushl  $1';        pc:=pc+1;
        code[pc]:= 'popl  %eax';          pc:=pc+1;
```

```
      code[pc]:= 'popl  %ecx';          pc:=pc+1;
      code[pc]:= 'addl  %ebp, %ecx';         pc:=pc+1;
      code[pc]:= 'movl  %eax, (%ecx)';       pc:=pc+1;

      { # run the action body}
      code[pc]:= 'movl 8(%ebp), %eax';          pc:=pc+1;
      code[pc]:= 'pushl  %eax';          pc:=pc+1;
      code[pc]:= 'call   ' + action_queue[k].action_body;          pc:=pc+1;
      code[pc]:= 'addl  $4, %esp';          pc:=pc+1;

      {
      # label the end of the if next=_
        anchor := pc;
        jmp exitfrom_ anchor
        if_FALSE_ end_if_next[k] :
          nop
        exitfrom_ anchor :
          nop
        end_if_else_nesting[k] := pc;
        jmp exitfrom_  end_if_else_nesting[k]
        is_FALSE_  end-of-if-branch[k]:
          nop
      }
      anchor:=pc;
      Str(anchor, temp);
      code[pc]:= 'jmp  exitfrom_' + temp;  pc:=pc+1;
      Str(end_if_next[k], temp);
      code[pc]:= 'is_FALSE_' + temp + ':';  pc:=pc+1;
      code[pc]:= '  nop';       pc:=pc+1;
      Str(anchor, temp);
      code[pc]:= 'exitfrom_' + temp + ':';  pc:=pc+1;
      code[pc]:= '  nop';       pc:=pc+1;

      end_if_else_nesting[k]:=pc;
      Str(end_if_else_nesting[k], temp);
      code[pc]:= 'jmp  exitfrom_' + temp;       pc:=pc+1;

      Str(end_if_branch[k], temp);
      code[pc]:= 'is_FALSE_' + temp + ':';      pc:=pc+1;
      code[pc]:= '  nop';              pc:=pc+1;
      k:=k+1;
    end;

    {the ending labels for the if-else }
    k:= n-1;
    while k>-1 do
      begin
        {asm code:
            exitfrom_  end_if_else_nesting[k]:
              nop
        }
        Str(end_if_else_nesting[k], temp);
        code[pc]:= 'exitfrom_' + temp + ':';          pc:=pc+1;
```

```
    code[pc]:= '  nop';            pc:=pc+1;
    k:=k-1;
  end;
{
  next:= next+1 mod n
}
Str(offset_next , temp);
code[pc] := 'pushl  $' + temp;        pc:=pc+1;
code[pc] := 'pushl  $' + temp;        pc:=pc+1;
code[pc]:= 'popl  %eax';          pc:=pc+1;
code[pc]:= 'addl  %ebp, %eax';    pc:=pc+1;
code[pc]:= 'pushl (%eax)';       pc:=pc+1;

code[pc]:= 'pushl  $1';          pc:=pc+1;
code[pc]:= 'popl  %eax';          pc:=pc+1;
code[pc]:= 'popl  %ecx';          pc:=pc+1;
code[pc]:= 'addl  %ecx, %eax';    pc:=pc+1;
code[pc]:= 'pushl %eax';    pc:=pc+1;

Str(offset_n , temp);
code[pc] := 'pushl  $' + temp;        pc:=pc+1;
code[pc]:= 'popl  %eax';          pc:=pc+1;
code[pc]:= 'addl  %ebp, %eax';    pc:=pc+1;
code[pc]:= 'pushl (%eax)';       pc:=pc+1;

code[pc]:= 'popl  %ecx';          pc:=pc+1;
code[pc]:= 'popl  %eax';          pc:=pc+1;
code[pc]:= 'movl  $0, %edx';          pc:=pc+1;
code[pc]:= 'idivl  %ecx';          pc:=pc+1;
code[pc]:= 'pushl %edx';    pc:=pc+1;

code[pc]:= 'popl  %eax';          pc:=pc+1;
code[pc]:= 'popl  %ecx';          pc:=pc+1;
code[pc]:= 'addl  %ebp, %ecx';    pc:=pc+1;
code[pc]:= 'movl  %eax, (%ecx)';   pc:=pc+1;

{
 #count:= count-1;
}
Str(offset_count , temp);
code[pc] := 'pushl  $' + temp;        pc:=pc+1;
code[pc] := 'pushl  $' + temp;        pc:=pc+1;
code[pc]:= 'popl  %eax';          pc:=pc+1;
code[pc]:= 'addl  %ebp, %eax';    pc:=pc+1;
code[pc]:= 'pushl (%eax)';       pc:=pc+1;
code[pc]:= 'pushl $1';      pc:=pc+1;

code[pc]:= 'popl  %ecx';          pc:=pc+1;
code[pc]:= 'popl  %eax';          pc:=pc+1;
code[pc]:= 'subl  %ecx, %eax';    pc:=pc+1;
code[pc]:= 'pushl %eax';    pc:=pc+1;

code[pc]:= 'popl  %eax';          pc:=pc+1;
```

```
      code[pc]:= 'popl  %ecx';          pc:=pc+1;
      code[pc]:= 'addl  %ebp, %ecx';  pc:=pc+1;
      code[pc]:= 'movl  %eax, (%ecx)';  pc:=pc+1;

      { jump back to evaluate the condition}
      Str(start_first_inner_loop, temp);
      code[pc]:= 'jmp  startwhile_' + temp;   pc:=pc+1;
      MakeJumpLabel('is_FALSE', end_first_inner_loop);
    { End of first inner loop}
      code[pc]:='#   end of the 1st inner loop'; pc:=pc+1;
  end;
  { end of procedure inner_loop_one_three}


  procedure Header_var(size :longint;  num_thr :longint);
  var temp: string;
       thread_ptr_block: longint;
  begin
    code[pc]:= '.section .data';      pc:=pc+1;
    code[pc]:= ' error_msg1:';          pc:=pc+1;
    code[pc]:= TAB+ '.ascii "Error: illegal array index\n"'; pc:=pc+1;
    code[pc]:= ' error_msg2:';          pc:=pc+1;
    code[pc]:= TAB+ '.ascii "Error: operand invalid\n"'; pc:=pc+1;
    code[pc]:= ' error_msg3:';          pc:=pc+1;
    code[pc]:= TAB+ '.ascii "Error: thread creation failure.\n"'; pc:=pc+1;

    code[pc]:= ' termination_msg:';         pc:=pc+1;
    code[pc]:= TAB+ '.ascii "Program has terminated.\n"'; pc:=pc+1;

    code[pc]:= ' eol:' ; pc:=pc+1;
    code[pc]:= TAB+ '.ascii "\n"';    pc:=pc+1;

    code[pc]:= ' strfmt1:';           pc:=pc+1;
    code[pc]:= TAB + '.asciz "%d"';   pc:=pc+1;
    code[pc]:= ' strfmt2:';           pc:=pc+1;
    code[pc]:= TAB + '.asciz "%d\n"';   pc:=pc+1;

    code[pc]:= '.section .bss';       pc:=pc+1;
    Str(size, temp);
    code[pc]:= TAB+ '.comm global_var,' + temp + ', 4';   pc:=pc+1;
    code[pc]:= TAB+ '.comm readbuffer, 4, 4';         pc:=pc+1;
    code[pc]:= TAB+ '.comm writebuffer, 4, 4';         pc:=pc+1;
    code[pc]:= TAB+ '.comm global_counter, 4, 4';        pc:=pc+1;

    thread_ptr_block:= num_thr*4;
    Str(thread_ptr_block, temp);
    code[pc]:= TAB+ '.comm thread_base_ptr, ' + temp +',4';    pc:=pc+1;
    code[pc]:= TAB+ '.comm thread_main_action, 4, 4';    pc:=pc+1;
    code[pc]:= TAB+ '.comm default_stack_size, 4 ,4';    pc:=pc+1;
    code[pc]:= TAB+ '.comm obj_addr_in_array, 4, 4';     pc:=pc+1;
    code[pc]:= TAB+ '.comm thread_attr_join, 36 ,32';         pc:=pc+1;
    code[pc]:= TAB+ '.comm thread_attr_detach, 36 ,32';         pc:=pc+1;
    code[pc]:= TAB+ '.comm sched_policy, 4, 4';          pc:=pc+1;
```

```
    code[pc]:= TAB+ '.comm trap_line, 4';                pc:=pc+1;

    code[pc]:= TAB + '# End of Header_var( ).';         pc:=pc+1;
  end;


  {# trapoverflow: generate label that enables program to exit in case operands
      exceed valid range}
  procedure TrapOverflow;
  begin
    code[pc]:= '.overflow:';         pc:=pc+1;
     {print error msg for invalid operand }
    code[pc]:= 'movl $error_msg2, %ecx';  pc:=pc+1;
    code[pc]:= 'pushl %ecx'; pc:=pc+1;
    code[pc]:= 'call printf' ;    pc:=pc+1;
    code[pc]:= 'addl $4 , %esp'; pc:=pc+1;

    code[pc]:= 'movl  $1, %eax';            pc:=pc+1;
    code[pc]:= 'movl  $0, %ebx';            pc:=pc+1;
    code[pc]:= 'int   $0x80';               pc:=pc+1;
 end;

  {# traproutine: generate label that enables program to exit in case certain runtime
       error(illegal array index)}
  procedure TrapRoutine;
  begin
   { .trap:
            movl  $4, %eax
            movl  $1, %ebx
            movl  $error_msg, %ecx
            movl  $41, %edx
            int   $0x80        # call write

            movl  $1, %eax
            movl  $0, %ebx
            int   $0x80        # call exit
            ret
    }

    code[pc]:= '.trap:';         pc:=pc+1;
    code[pc]:= 'movl  $4, %eax';            pc:=pc+1;
    code[pc]:= 'movl  $1, %ebx';            pc:=pc+1;
    code[pc]:= 'movl  $error_msg1, %ecx'; pc:=pc+1;
    code[pc]:= 'movl  $41, %edx';           pc:=pc+1;
    code[pc]:= 'int   $0x80';               pc:=pc+1;

    code[pc]:= 'movl  $1, %eax';            pc:=pc+1;
    code[pc]:= 'movl  $0, %ebx';            pc:=pc+1;
    code[pc]:= 'int   $0x80';               pc:=pc+1;
  end;

  {trap routine to handle thread creation failure}
  procedure TrapThreads;
```

```
begin
 { .trap:
          movl  $4, %eax
          movl  $1, %ebx
          movl  $error_msg, %ecx
          movl  $41, %edx
          int   $0x80         # call write

          movl  $1, %eax
          movl  $0, %ebx
          int   $0x80         # call exit
          ret
  }

  code[pc]:= '.trap_thread_failure:';       pc:=pc+1;
  code[pc]:= 'movl  $4, %eax';          pc:=pc+1;
  code[pc]:= 'movl  $1, %ebx';          pc:=pc+1;
  code[pc]:= 'movl  $error_msg3, %ecx'; pc:=pc+1;
  code[pc]:= 'movl  $41, %edx';         pc:=pc+1;
  code[pc]:= 'int   $0x80';             pc:=pc+1;

  code[pc]:= 'movl  $1, %eax';          pc:=pc+1;
  code[pc]:= 'movl  $0, %ebx';          pc:=pc+1;
  code[pc]:= 'int   $0x80';             pc:=pc+1;
end;


{# Header_code() does this in gas:
              .globl main
              main:
}
procedure Header_code;
begin
  TrapRoutine;
  TrapThreads;
  TrapOverflow;
  code[pc]:= '.globl main';     pc:=pc+1;
  code[pc]:= ' main:';          pc:=pc+1;
  code[pc]:= TAB +'# End of Header_code.'; pc:=pc+1;
end;

procedure GenerateFuncPrefix(id: Identifier);
begin
  code[pc]:=' .globl ' + id ;  pc:=pc+1;
  code[pc]:=' .type ' + id +', @function';   pc:=pc+1;
end;


procedure GenerateLabel(id: Identifier);  {generate labels for procedures only.}
begin
  code[pc]:=' proc_'+ id +':';    pc:=pc+1;
end;
```

```
{#Enter_32 : the prologue of procedure call:}

procedure Enter_32( size: longint);
var temp: string;
begin
  code[pc]:= 'pushl %ebp';    pc:=pc+1;
  code[pc]:= 'movl  %esp, %ebp';  pc:=pc+1;
  //code[pc]:= 'pushl %esi';    pc:=pc+1;
  Str(size, temp);
  code[pc]:= 'subl '+ '$'+ temp + ' ,%esp';    pc:=pc+1;
  code[pc]:= TAB + '# End of Enter_32().'; pc:=pc+1;
end;


procedure Return_32(size: longint);      {# return from a procedure call}
var temp: string;
begin
  Str(size, temp);
  code[pc]:= 'addl '+ '$'+ temp + ' ,%esp';    pc:=pc+1;
  //code[pc]:= 'popl %esi';    pc:=pc+1;
  code[pc]:= 'movl %ebp, %esp';          pc:=pc+1;
  code[pc]:= 'popl %ebp';                pc:=pc+1;
  code[pc]:= 'ret';                      pc:=pc+1;
  code[pc]:= TAB+'# End of Return_32.';  pc:=pc+1;
end;

procedure Open;
begin
  curlev := 0; pc := 0
end;

procedure Close_32;
begin
  code[pc]:='pushl  $0';    pc:=pc+1;
  code[pc]:='call   exit';  pc:=pc+1;
end;

{load assembly code if no error occurs}
procedure LoadCode_32 (var code: Memory; len: longint);
var i: longint;
begin
  i := 0;
  while i < len do
    begin writeln(File_GAS, code[i]);  i := i + 1 end
end;

procedure Load;
var j : longint;
begin
  LoadCode_32(code, pc);       {# write to the file}
  close(File_GAS);             {# close the file }
  writeln(' Code loaded to: ', FileID);
  if paramcount > 2 then
```

```
      begin
        for j:=0 to pc do writeln(code[j]) ;
      end
  end;

{initialization:
  initialize boolean type and integer type
}
begin
  new (boolType); boolType^.form := Bool; boolType^.size := 4;
  new (intType); intType^.form := Int; intType^.size := 4;
end.
```

Listing D.3: symboltable.pas

```
{
   [By Xiao-lei Cui, Nov 2008]
 - defines data structures used in parsing and code generation;
 - defines procedures to consturct and operate with the symbol table.
}

unit symboltable;
interface
uses scanner;

type
  Class = (HeadClass, VarClass, ParClass, ConstClass, FieldClass, TypeClass,
           ProcClass, SProcClass, RegClass, EmitClass, CondClass, MonitorClass,
           ActionClass);

  Form = (Bool, Int, Arry, Rcrd, Monitor); {supported types}

  {to store all actions and object offset. needed only to call pthread APIs }
  OA_List = ^ObjNode;

  ActionList = ^ActionNode;

  ObjNode = record
    obj_name: Identifier;
    val : longint;
    actions : ActionList;
    next : OA_List;
  end;

  ActionNode = record
    id: Identifier;
    next:  ActionList;
  end;

  Objct = ^ObjDesc;
  Typ = ^TypeDesc;
  Item = record
```

```
    mode: Class;
    lev: longint;
    tp: Typ;
    a: longint;  {value of the item; offset address of the item}
    b: longint;  {saves the current pc which is used to make up the jumping labels}
    c: longint;  {the relational operation offset}
    r: longint;  {saves the current pc for making up jumping labels, similar to .b}
    o: longint;  {offset value of a record field}
    indirect: boolean;  {whether requires indirect addressing}
    bool_set: boolean;  {whether the item's boolean value is set}
    sc: boolean;        {whether the conditional expression is short-circuited}
    push_placeholder: boolean;    {set to true initially but set to false after the
        first call to placeholder}
    parSize: longint {parameter size, if procedure}
  end;

  ObjDesc = record
    cls: Class;
    lev: longint;
    next, dsc: Objct;
    tp: Typ;
    name: Identifier;
    val: longint;
    isGuarded : boolean;
    isAParam : boolean;
    parSize : longint
  end;

  TypeDesc = record
    form: Form;
    typename: Identifier;
    fields: Objct; {for records and monitor class}
        {if form=record then record fields}
        {if form=monitor then the list vars and procedures and actions}
    a_list: ActionList;      {to store all actions of one class}
    action_count: longint;   {number of actions}
    base: Typ;               {the base type for arrays}
    lower, size, len: longint {lower bound, required memory size, and length for
        arrays}
  end;

var
  topScope: Objct;
  {current scope, where search for an identifier starts, this is the global var to
      the intelcompiler module}
  guard: Objct; {topScope and universe are linked lists. they are ended with guard}
  boolType, intType: Typ; {predefined primitive types}

  procedure NewObj (var obj: Objct; cls: Class);
  procedure Find (var obj: Objct);
  procedure FindField (var obj: Objct; list: Objct);
  function IsParam (obj: Objct): boolean;
  procedure OpenScope;
```

```
  procedure CloseScope;
  procedure PreDef (cl: Class; n: longint; name: Identifier; tp: Typ);

implementation

var
  universe: Objct; {final scope with only predefined identifiers}

  {insert an object to the current scope. if the id is not found, insert to the end
      of the list;
   otherwise. do not insert , return the object that is already defined.}
  procedure NewObj (var obj: Objct; cls: Class);
    var n, x: Objct;
    begin
      x := topScope; guard^.name := id; {set sentinel for search}
      while x^.next^.name <> id do x := x^.next;
      if x^.next = guard then
        begin
          new (n); n^.name := id; n^.cls := cls; n^.next := guard;
          x^.next := n; obj := n
        end
      else begin obj := x^.next; Mark ('multiple definitions ?'); end
    end;

    {search for 'id' - the last sym returned by GetSym - from the topscope and up,
        stop when universe level is reached.}
  procedure find (var obj: Objct);
    var s, x: Objct;
    begin s := topScope; guard^.name := id;
      while true do
        begin x := s^.next;
          while x^.name <> id do x := x^.next;
          if x <> guard then begin obj := x; break end;
          if s = universe then
            begin obj := x; write(id, ' '); Mark ('undef'); break end;
          s := s^.dsc
        end
    end;

  procedure FindField (var obj: Objct; list: Objct);
    begin
      guard^.name := id;
      while list^.name <> id do list := list^.next;
      obj := list
    end;

  function IsParam (obj: Objct): boolean;
    begin
      IsParam := obj^.IsAParam
    end;

  procedure OpenScope;
    var s: Objct;
```

```
    begin
      new (s); s^.cls := HeadClass; s^.dsc := topScope;
      s^.next := guard; topScope := s
    end;

  procedure CloseScope;
    begin
      topScope := topScope^.dsc
    end;

  procedure PreDef (cl: Class; n: longint; name: Identifier; tp: Typ);
    {to define standard identifiers: true, false, read, write, writeln}
    var obj: Objct;
    begin
      new (obj);
      obj^.cls := cl; obj^.val := n; obj^.name := name;
      obj^.tp := tp; obj^.dsc := nil;
      obj^.next := topScope^.next; topScope^.next := obj
    end;

{Initialization: start with an empty symbol table}
begin
  new (guard); guard^.cls := VarClass; guard^.tp := intType; guard^.val := 0;
  topScope := nil; openScope; Universe := topScope
end.
```

Listing D.4: scanner.pas

```
{
   [By Xiao-lei Cui, Nov 2008]
 - defines the reserved words.
 - defines procedures to handle lexical analysis
 - defines procedures to raise compile-time errors.
}

unit scanner;
interface
  const
    IdLen = 40; {number of significant characters in identifiers}

  type
    Symbol = (null, ExpSym, TimesSym, DivSym, ModSym, AndSym, PlusSym, MinusSym,OrSym
        ,
              EqlSym, NeqSym, LssSym, GeqSym, LeqSym, GtrSym,PeriodSym, CommaSym,
                  ColonSym,
              RparenSym, RbrakSym, OfSym, ThenSym, DoSym, LparenSym, LbrakSym, NotSym
                  ,
              BecomesSym, NumberSym, IdentSym, SemicolonSym, EndSym, ElseSym, IfSym,
              WhileSym, WhenSym, ArraySym, RecordSym, MonitorclassSym, ConstSym,
                  TypeSym,
              VarSym, ProcedureSym, ActionSym, BeginSym, ProgramSym, EofSym);
    Identifier = string[IdLen];
```

```

  var
    sym: Symbol; {next symbol}
    v: longint; {value of number if sym = NumberSym}
    id: Identifier; {string for identifier if sym = IdentSym}
    error: Boolean; {whether an error has occurred so far}

  procedure Mark (msg: string);
  procedure Warn (msg: string);
  procedure GetSym;

implementation
  const
    KW = 25; {number of keywords; only need 23 for now.}

  type
    KeyTable = array [1..KW] of record sym: Symbol; id: Identifier end;
  var
    ch: char;
    line, lastline, errline: longint;
    pos, lastpos, errpos: longint;
    keyTab: KeyTable;
    fn: string[255]; {name of source file}
    source: text; {source file}

  procedure GetChar;
    begin
      lastpos := pos;
      if eoln  (source) then
        begin pos := 0; line := line + 1 end
      else begin lastline:= line ; pos := pos + 1 end;
      read(source, ch);
    end;

  procedure Number;
    begin
      v := 0; sym := NumberSym;
      repeat
      if v <= maxint - (ord (ch) - ord ('0')) div 10 then
        v := 10 * v + (ord (ch) - ord ('0'))
      else
        begin Mark ('number too large'); v := 0 end;
      GetChar
      until not (ch in ['0'..'9'])
  end;

  procedure Ident;
    var len, k: longint;
    begin len := 0;
      repeat
        if len < IdLen then begin len := len + 1; id[len] := ch; end
        else warn ('Length is too long for identifier');
        GetChar
```

```
    until not (ch in ['A'..'Z', 'a'..'z', '0'..'9', '_']);
    {NOTE: identifier may contain '_' char, but can not start with a '_'}
    setlength(id, len); k := 1;
    while (k <= KW) and (id <> keyTab[k].id) do k := k + 1;
    if k <= KW then sym := keyTab[k].sym else sym := IdentSym
  end;

procedure comment;
  begin GetChar;
    while (not eof (source)) and (ch <> '}') do GetChar;
    if eof (source) then Mark ('comment not terminated')
    else GetChar;
  end;

procedure Mark (msg: string);
  begin
    if (lastline > errline) or (lastpos > errpos) then
      writeln ('error: line ', lastline:1, ' pos ', lastpos:1, ' ', msg);
    errline := lastline; errpos := lastpos; error := true;
    halt;
  end;

procedure Warn (msg: string);
  begin
    writeln ('warning: line ', lastline:1, ' pos ', lastpos:1, ' ', msg);
  end;

procedure GetSym;
  begin {first skip white space}
    while not eof (source) and (ch <= ' ') do GetChar;
    if eof (source) then sym := EofSym
    else
      case ch of
      '^': begin GetChar; sym := ExpSym end;
      '*': begin GetChar; sym := TimesSym end;
      '+': begin GetChar; sym := PlusSym end;
      '-': begin GetChar; sym := MinusSym end;
      '=': begin GetChar; sym := EqlSym end;
      '<': begin GetChar;
             if ch = '=' then
               begin GetChar; sym := LeqSym end
             else if ch = '>' then
               begin GetChar; sym := NeqSym end
             else sym := LssSym
           end;
      '>': begin GetChar;
             if ch = '=' then
               begin GetChar; sym := GeqSym end
             else sym := GtrSym
           end;
      ';': begin GetChar; sym := SemicolonSym end;
      ',': begin GetChar; sym := CommaSym end;
      ':': begin GetChar;
```

```
               if ch = '=' then
                 begin GetChar; sym := BecomesSym end
               else sym := ColonSym
             end;
        '.': begin GetChar; sym := PeriodSym ;  end;
        '(': begin GetChar; sym := LparenSym end;
        ')': begin GetChar; sym := RparenSym end;
        '[': begin GetChar; sym := LbrakSym end;
        ']': begin GetChar; sym := RbrakSym end;
        '0'..'9': Number;
        'A' .. 'Z', 'a'..'z': begin Ident; {writeln(id);} end;
        '{': begin comment; GetSym; end;
      otherwise
        begin GetChar; sym := null end
      end
  end;

{Initialization:
  Initialize line, position indices. Enter the reserved words of Pascal0.
}
begin
  line := 1; lastline := 1; errline := 1;
  pos := 0; lastpos := 0; errpos := 0;
  error := false;
  keyTab[1].sym := DoSym; keyTab[1].id := 'do';
  keyTab[2].sym := IfSym; keyTab[2].id := 'if';
  keyTab[3].sym := OfSym; keyTab[3].id := 'of';
  keyTab[4].sym := OrSym; keyTab[4].id := 'or';
  keyTab[5].sym := AndSym; keyTab[5].id := 'and';
  keyTab[6].sym := NotSym; keyTab[6].id := 'not';
  keyTab[7].sym := EndSym; keyTab[7].id := 'end';
  keyTab[8].sym := ModSym; keyTab[8].id := 'mod';
  keyTab[9].sym := VarSym; keyTab[9].id := 'var';
  keyTab[10].sym := ElseSym; keyTab[10].id := 'else';
  keyTab[11].sym := ThenSym; keyTab[11].id := 'then';
  keyTab[12].sym := TypeSym; keyTab[12].id := 'type';
  keyTab[13].sym := ArraySym; keyTab[13].id := 'array';
  keyTab[14].sym := BeginSym; keyTab[14].id := 'begin';
  keyTab[15].sym := ConstSym; keyTab[15].id := 'const';
  keyTab[16].sym := WhileSym; keyTab[16].id := 'while';
  keyTab[17].sym := RecordSym; keyTab[17].id := 'record';
  keyTab[18].sym := ProcedureSym; keyTab[18].id := 'procedure';
  keyTab[19].sym := DivSym; keyTab[19].id := 'div';
  keyTab[20].sym := WhenSym; keyTab[20].id := 'when';
  keyTab[21].sym := MonitorclassSym; keyTab[21].id := 'class';
  keyTab[22].sym := ProgramSym; keyTab[22].id := 'program';
  keyTab[23].sym := ActionSym;  keyTab[23].id := 'action';

  if paramcount > 0 then
    begin
      fn := paramstr (1);
      {$I-}
      Assign (source, fn); Reset (source);
```

```
      {$I+}
      if IOResult<>0 then
          begin writeln ('File ', fn,' doesn''t exist.'); Mark('Error:file does not
              exist'); end;
      GetChar
    end
  else Mark ('name of source file expected')
end.
```

Appendix E

Glossary of Acronyms

CI Confidence Interval

GNAT GNU NYU Ada Translator

IA-32 Intel 32-bit Architecture

NPTL Native POSIX Thread Library

JVM Java Virtual Machine

PVM Parallel Virtual Machine

Pthreads POSIX Threads

Bibliography

[1] *Intel 64 and IA-32 Architectures Software Developers Manual (Volume 2B).* http://download.intel.com/design/processor/manuals/253667.pdf, 2008.

[2] *The Java HotSpot Performance Engine Architecture (White Paper).* http://java.sun.com/products/hotspot/whitepaper.html, 2008.

[3] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Pearson, 2nd ed., 2007.

[4] R. Blum, *Professional Assembly Language.* Wrox, 2005.

[5] E. W. Dijkstra, "Co-operating sequential processes," *Programming Languages*, pp. 43–112, 1968.

[6] E. W. Dijkstra, "Hierachical ordering of sequential processes," *Operating System Techniques, A.P.I.C. Studies in Data Processing #9*, pp. 72–93, 1972.

[7] U. Drepper and I. Molnar, *The Native POSIX Thread Library for Linux.* http://people.redhat.com/drepper/nptl-design.pdf, 2005.

[8] A. Geist, *PVM, Parallel Virtual Machine. A user guide and tutorial for networked parallel computing.* The MIT Press, 1994.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification.* Sun Microsystems, third ed., 2005.

[10] P. B. Hansen, *Operating System Principles.* Prentice-Hall, 1973.

[11] C. A. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM*, vol. 17, no. 10, pp. 549–557, 1974.

[12] I. H. Krüger, "An Experiment in Compiler Design for a Concurrent Object-Based Programming," Master's thesis, University of Texas at Austin, 1996.

[13] G. Lou, "A Compiler for an Action-based Object-oriented Programming Language," Master's thesis, McMaster University, 2004.

[14] J. Miranda, *A Detailed Description of the GNU Ada Run Time.* http://www.iuma.ulpgc.es/users/jmiranda/gnat-rts/index.htm, 1.0 ed., 2002.

[15] J. Misra, *A Discipline of Multiprogramming: programming theory of distributed applications.* Springer, 2001.

[16] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming: A POSIX Standard for Better Multiprocessing.* O'Reilly, 1996.

[17] M. L. Scott, *Programming Language Pragmatics.* Morgan Kaufmann, second ed., 2005.

[18] N. Wirth, *Compiler Construction.* Addison-Wisley, 1996.

[19] D. Zingaro and E. Sekerinski, *The Pascal0 Compiler.* Course material for CS4TB3, http://www.cas.mcmaster.ca/~cs4tb3/, July 2007.