A Compiler for an Action-Based Object-Oriented Programming Language

By
Guanrong Lou, BSc. Hon. Computer Science

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Master of Science
Department of Computing and Software
McMaster University

MASTER OF SCIENCE(2004) McMaster University
(Computer) Hamilton, Ontario

TITLE: A Compiler for an Action-Based Object-Oriented Programming Language

AUTHOR: Guanrong Lou, BSc. Hon. Computer Science(McMaster University)

SUPERVISOR: Dr. Emil Sekerinski

NUMBER OF PAGES: ix, 97

Abstract

Lime is an action-based object-oriented concurrent programming language, which was developed by Dr. Sekerinski from McMaster University. The development of Lime is based on the observation that more and more applications will be implemented on networks of processors in the future and those are significantly more ambitious than current applications. It is difficult for programmers to do multiprogramming using current object-oriented programming languages. Action systems, which model concurrency by nondeterministic choice among atomic actions (e.g. Lamport's Temporal Logic of Actions, J. Misra and K. M. Chandy's Unity Logic and Back's Action Systems), can help us to simplify both the specification and design of concurrent applications. But all of that is still theoretical. Such action systems have not been implemented and experimented with very thoroughly.

Lime is an object-oriented programming language that is based on the action system. The closest approach is Seuss, which was developed by J. Misra et.al. at The University of Texas at Austin, but the limitation of Seuss is that it is not fully object-oriented. One purpose of my research is to find out how to schedule actions for applications running on a multiprocessor environment. I implemented a compiler for this new technique that can translate programs written in Lime to Java assembly language.

Object-oriented programming techniques are used widely in software development, and Lime can make a good combination of these techniques and action systems. So we can make the development of object-oriented concurrent programs much easier and efficient.

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete this thesis.

I am deeply indebted to my supervisor, Dr. E. Sekerinski whose help, stimulating suggestions and encouragement helped me in all the time of research and writing of this thesis.

Especially, I would like to give my special thanks to my wife Hongliang and my son Jayson whose love enabled me to complete this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Why Lime

Concurrent programming is desirable for many reasons, such as improved response and better exploitation of hardware. There are many real life cases that can benefit from the advantages provided by concurrency. Additionally, object-oriented program design provides benefits such as reusability and maintainability.

Java and C# are popular object-oriented programming languages today and both of them support concurrent programming. But they were invented as general purpose languages and concurrency features are not critical for them, so this support has some limitations. There also exist a large number of research proposals and experimental languages for developing new object-oriented languages for concurrent programming. However, all of them have some limitations in concurrent software development.

Lime was invented as a truly concurrent object-oriented programming language. Concurrency is one of the most important features of the Lime, and are also critical for it. Lime is based on an action system. The purpose of the development is to simplify both the specification and design of object-oriented concurrent applications.

1.2 Contributions

My contributions in the research of Lime include:

- Complete the design of Lime, including its functionality and syntax;
- Design the translation schemes to translate Lime to Jasmin, a Java assembly language;
- Design and implement a compiler for Lime.

Lime is an ongoing research project and many aspects are likely to evolve over time. Therefore the design and implementation of the compiler should be as general as possible. We use an object-oriented approach to present and implement the compiler. The code of the compiler is written entirely in Java. The target language of the Lime compiler is Jasmin, a Java assembly language. We made this choice since we want the compiled Lime programs to run on the Java Virtual Machine and without looking deeply into Java class files.

1.3 Structure of the Thesis

- In Chapter 2, the concurrency features of some programming languages and limitations of them are discussed.
- In Chapter 3, a brief background of action systems is first given. Then Lime is introduced by describing its features, its syntax and some sample programs.
- Chapter 4 discusses the Java Virtual Machine and Jasmin. Jasmin is an assembler that translates from an intermediate language to Java bytecode. Lime uses Jasmin to handle the final translation to Java.
- Chapter 5 and Chapter 6 describe the translation schemes that are used to translate Lime to its target language, Jasmin.
- Chapter 7 contains the description of the design and implementation of the Lime compiler.
- Chapter 8 outlines the conclusion of the thesis, in addition to discussing future work.

Chapter 2

Related Work

In this chapter, some programming languages, especially some object-oriented languages, which support concurrent programming, are discussed. The intention is to have a general idea about the nature of those languages and how they support concurrent programming as well as some problems and limitations.

Most modern programming languages support concurrent programming. There also exist a large number of research proposals for developing new object-oriented languages for current programming.

There are different ways to classify concurrent object-oriented programming languages. For example, Peter Wegner introduces the classification according to whether objects are internally sequential, quasi-concurrent or concurrent [24]:

Sequential objects possess a single active thread of control. Objects in Ada tasks and Seuss are examples of sequential objects.

Quasi-concurrent objects have multiple threads but only one thread may be active at a time. Control must be explicitly released to allow interleaving of threads.

Concurrent objects do not restrict the number of internal threads. New threads are created freely when accepting requests. Languages like Java and C# support concurrent objects. From the point of view of the called objects, a new local thread is effectively created whenever a method is activated in response to a message.

This section is going to give an overview of the concurrent features of some programming languages and some research proposals. The purpose is to give a general idea about how these languages support concurrent programming and their limitations.

2.1 Ada

Ada was developed for the U.S. Department of Defense to be the standard language for programming defense applications [11]. The concurrency features of Ada are one of the most important parts of the language, and are also critical for its intended uses.

In Ada, the basic construct for concurrent processes is the *task*. An Ada task is a unit of modularization comprising a specification and a body, and also a data object. A task terminates when it reaches the end of the begin-end code of its body. Tasks may either be static (created at start of execution and never terminate) or dynamic, e.g. create a new task for each new radar trace in a radar system.

A task can be temporarily paused by delay statements. The delay statement has the following two formats:

- *Delay xyz*: where *xyz* is an expression of type *duration*, causes the execution of the thread to be delayed for (at least) the given amount of time;
- *Delay until tim*: where *tim* is an expression of type *time*, causes execution of the thread to be delayed until (at the earliest) the given *tim*.

Ada provides *select* statements to support nondeterministic task interaction.

- **Selective accept**: Select statement allows a choice of actions. For example:

  select
  *entry*1 $(\cdots)$ *do* $\cdots$ *end*;
  or
  *when some_condition entry*2 $(\cdots)$;
  or
  delay $\cdots$ *ddd* $\cdots$;
  end select;

  Ada takes whichever open entry arrives first, or if none arrives by end of *delay*, do *ddd*;

- **Conditional entry call**: Make a call only if it will be accepted. The conditional entry call allows the task client to withdraw the offer to communicate if the server task is not prepared to accept the call immediately [5]. For example:

  select
  entry − *call* $\cdots$
  else
  statements
  end select;

If *entry-call* is accepted immediately, the body of the entry will be executed; otherwise the *else* statements will be executed;

Ada was first standardized in 1983 [11]. Ada 83 introduced rendezvous as the primary interprocess communication mechanism and the sole synchronization mechanism. Everything else has to be programmed using rendezvous.

A second version of Ada was standardized in 1995 [11]. The two new features of Ada 95 for concurrent programming are *protected type* and the *request* statement. The *protected type* supports the synchronized access to shared data. A *request* statement supports the synchronization and scheduling that depends on the arguments of calls. A protected type encapsulates shared data and synchronizes access to it. Each instance of a protected type is similar to monitors.

Tasks can share variables; however, they cannot assume these variables are updated except at synchronization points. The *abort* statement allows one task to terminate another task. There is also a mechanism for setting the priority of a task.

The following is an example of how to implement a semaphore in Ada using protected types.

```
protected type Semaphore is
   entry P;
   procedure V;
   private Lock  : Boolean := False;
end Semaphore;
protected body Semaphore is
   entry P when not Lock is
   begin
        Lock := True;
   end P;
   procedure V is
   begin
        Lock := False;
   end V;
end Semaphore;
task body Semaphore is
begin
     loop
          accept P;
          accept V;
     end loop;
end Semaphore;
```

2.2 Java

Java is an object-oriented language developed by Sun Microsystems, released to the public in 1995 [12].

Concurrent programming in Java is based on *Java threads*. A Java thread is an object of the *java.lang.Thread* class or an object that implements the *Runnable* interface. The thread maintains housekeeping and control of its activity. Every program consists of at least one main thread. Other internal background threads may also be started during Java Virtual Machine (JVM) initialization. All user-level threads are explicitly constructed and started from the main thread, or from any other threads that they in turn create.

Threads in Java execute concurrently, at least conceptually. So they can simultaneously access shared variables. Java supports mutual exclusion by means of the keyword *synchronized*, which can be used in the entire method or in a sequence of statements. In Java, a monitor is used to synchronize the access to a shared resource. All Java objects have a monitor and each object can be used as a mutually exclusive lock. When a *synchronized* method is invoked, the invoked thread waits to obtain the lock on the object, then executes the body of the method, and finally releases the lock.

For the communication between different threads, Java uses the signal oriented methods *wait*(), *notify*() and *notifyAll*() that are inherited from the root class *Object*. A thread calls *O.wait*() when it needs to be suspended until the object referenced by O has a particular state. Another thread, on detecting that this object's state has the desired value, invokes either *O.notify*() or *O.notifyAll*().

The following example shows a semaphore implemented in Java using *synchronized* methods, and the methods call *wait*() and *notify*().

```
public class Semaphore{
    private boolean lock;
    public Semaphore() {
        lock = false;
    }
    public synchronized void P(){
        while(lock){
            try {
                wait();
            }
            catch(InterruptedException e){}
        }
```

```
        lock = true;
    }
    public synchronized void V(){
        lock = false;
        notify();
    }
}
```

To use this semaphore, one simply creates an object s of the class *Semaphore* and calls the method $s.P()$ and $s.V()$.

```
Semaphore s = new Semaphore();
s.P();
s.V();
```

2.3 C#

C# is an object-oriented programming language for building applications for the Microsoft .NET development environment. C# borrows heavily from Java, including the concurrency features.

The common language runtime (CLR) provides the following three synchronization methods [6].

Synchronized code regions The synchronization can be done on either the entire method or a part of them using the *Monitor* class.

Classic manual synchronization Use the various synchronization classes (like the *WaitHandle*, *Mutex*, *ReaderWriterLock*, *ManualResetEvent*, *AutoResetEvent* and the *Interlocked*) to create the synchronization mechanisms. Any field or method that the programmer want to synchronize must be manually defined as such.

Synchronized contexts *SynchronizationAttribute* can be used to enable simple, automatic synchronization for *ContextBoundObject* objects. This technique is useful if only the attributes and methods need to be synchronized. All objects in the same context domain share the same lock.

Generally speaking, the mechanisms used in C# are similar to those used in Java. But C# provides a wider libraries collection than Java does.

Following example shows a semaphore implemented in C#.

```
using System;
using System.Threading;
public class Semaphore{
   private boolean lock;
   public Semaphore(){
      lock = false
   }
   public void P()
   {
      try{
         lock(this){
            try{
               Monitor.Wait(this, Timeout.Infinite);
            }
            catch(ThreadInterruptedException e){}
         }
      }
      catch(SynchronizationLockException){}
      lock = true;
   }
   public void V(){
      try{
         lock(this){
            lock = false;
         }
      }
      catch(SynchronizationLockException){}
   }
}
```

2.4 Seuss

Seuss is an experimental programming language proposed by Misra et. al. at the University of Texas at Austin [20]. Unlike other languages such as Java, C# and Ada, Seuss does not support built-in concurrency or is commitment to either shared memory or message-passing style of implementation. Seuss also does not provide a mechanism for specific communication or synchronization mechanism, except the procedure call. Computation and communication, process specifications and interface

specifications are not distinguished.

Consequently, Seuss does not have many of the traditional multiprogramming concepts we have discussed previously. For example, there is no mechanism like processes, locking, rendezvous, waiting, interference or deadlock. Typical multiprograms employing message passing over bounded or unbounded channels can be encoded in Seuss by declaring the processes and channels as the components of a program. Shared memory multiprograms can be encoded by having processes and memories as components.

A Seuss program contains cats and boxes. Cats are roughly equivalent to classes in traditional object-oriented languages. A box is an instance of a cat. Cats can contain data elements as well as two types of procedures, partial and total procedures. A procedure can be either a method or an action. An action is executed autonomously an infinite number of times during a program execution. A method is called from another procedure [16, 20].

A partial procedure has an execution condition. When the procedure is called, if the condition does not hold, the call is rejected to preserve the program state. Total procedures do not have a condition so that the call to it is always successful. Total procedures can only call other total procedures.

The following program is an example of a semaphore implemented in Seuss using a partial procedure to implement the P operation and a total method to implement the V operation of the semaphore. Then a partial action is used by the program to access this semaphore.

$$
\begin{array}{l}
cat\ Semaphore \\
\quad var\ lock : boolean\ init\ false \\
\quad partial\ method\ P :: not\ lock\ \rightarrow\ lock := true \\
\quad total\ method\ V :: lock := false \\
end \\
box\ s : Semaphore \\
box\ RunSemaphore \\
\quad partial\ action :: not\ s.P\ \rightarrow\ \{do\ some\ work\}; s.V \\
end
\end{array}
$$

2.5 $\pi o \beta \lambda$

$\pi o \beta \lambda$ is a concurrent language that is inspired by POOL [15]. Concurrency in $\pi o \beta \lambda$ can be done in two ways. One is creating *non–default* bodies. (Default bodies can only sequentially respond to arriving messages. So the synchronous message passing highly prevents concurrency). In this way, messages are accepted in an arbitrary

order.

The other way for $\pi o\beta\lambda$ to support concurrent programming is called *post–processing*. $\pi o\beta\lambda$ allows the programmer to specify that further execution can be done concurrently once a return is called, so this is also called early return. Return causes the caller to continue execution with the value given. The callee continues execution of the rest of the method body. The following program shows an example of a priority queue implemented by this mechanism. The priority queue delivers and removes the smallest element by a remove method and also new elements can be added to this queue by another add method [15].

```
Priq class
   vars m : [N] ← nil; l : private ref(Priq) ← nil
   add(e : N) method
    begin
       return
       if m = nil then(m ← e; l ← new Priq)
       elif m < e then l!add(e)
       else(l!add(m); m ← e)
       fi
    end
   rem() method r : N
    begin
       return
       if m <> nil then m ← l!rem()
            if m = nil then l ← nil
            fi
       fi
    end
end Priq
```

In both the *add* and *rem* methods the return statement is on the first line. If we make an assumption that only one method in a particular class instance can run at any given time then it is easy to see that the structures will not be compromised by the concurrent addition to and removal from the queue structures. In a traditional programming language the return statement would have been at the end for the *add* method. This implementation allows the value to be put into the queue and forgotten about, as the queue itself as an independent entity takes care of it. The calling procedure is free to continue its work. In the method *rem*, a traditional language would also have placed the return statement at the end of the method, even though

the value to return is know at the very beginning. On top of this, the value to return would have to be stored in an extra variable to avoid destroying it while updating the rest of the queue structures. In this implementation the return value can be sent as soon as it is known so that the calling procedure can continue its work, while the queue finishes updating its structures as independently.

The semaphores can not be implemented in $\pi o \beta \lambda$ since there is no *wait* statement in the language. Once a call to the P operation of a semaphore fails, the call finishes instead of staying in the *wait* state.

2.6 Other Languages

Objective CAML (Categorically Abstract Machine Language) is an object-oriented programming language based on the ML family. A class declaration defines a type with the same name as the class. There are two kinds of polymorphism in the language. One of them is parametric polymorphism (parameterized classes). The other one is called inclusive polymorphism, which uses the subtyping relation between objects and delayed binding. There is no method overloading notion in Objective CAML [10]. The concurrent programming is supported through the *threads* library. The following modules are provided by the *threads* library [17]:

- *Thread*: the lightweight thread that handles threads creation and termination.
- *Mutex*: used to implement critical sections and protect shared mutable data structures against concurrent accesses.
- *Condition*: used when one thread wants to wait until another thread has finished doing something.
- *Event*: used to implement synchronous inter-thread communications over channels.
- *ThreadUnix*: used to re-implement some of the functions from Unix so that they only block the calling thread, not all threads in the program, if they cannot complete immediately.

The *threads* library in Objective CAML is implemented by time-sharing on a single processor. It will not take advantage of multi-processor machines. It will not make the programs run faster by using this library (even much slower for the cost of context switching). However, it may make it easy to write some programs by using several communicating processes [17].

The concurrent programming is supported by Ruby in the way that is similar to that of Objective CAML – by the *thread* library that is implemented by time-sharing on a single processor. Three classes are provided by the *thread* library: the class *Thread* that represents user-level threads, the class *Mutex* that is used to protect shared data against concurrent accesses and the class *ConditionVariable* that is used when one thread wants to wait until another thread finishing its job [7] .

Python is an object-oriented programming language originally developed for scripting. Python supports method overloading and exception handling. It also supports dynamic typing and multiple inheritance. Python threads are tightly controlled by a global interpreter lock and scheduler. The thread module provides low-level access to threads such as thread creation and simple mutex locks. The threading module is a high-level access to threads. The threads are implemented as classes (similar to Java) The threading module provides the following synchronization primitives: mutual exclusion locks, reentrant locks, conditional variables, semaphores and events [3]. Some work has been done to enhance the concurrency feature of them. For example, Papathomas et al. developed an experimental framework based (roughly) on inheritable synchronization constraints [21].

DisCo (Distributed Cooperation) is an experimental specification language for reactive systems, based on the execution model of joint actions. The notion of *joint action systems* was developed to "describe distributed systems and to support a rigorous methodology for their design" [13]. Unlike other action systems, the ideas of DisCo were formulated in such a way that temporal logic could be applied. DisCo is both action-oriented and object-oriented.

Bonsangue et al. proposed an approach called OO-action systems [4]. They took a class-based approach to object-orientation: OO-action systems model classes and instances of classes, i.e., objects, are created at run time. The objects themselves can be *distributed* and *active*. Communication between objects takes places via remote procedure calls and shared variables.

Back et al. [2] proposed an approach to show how objects are added into Action-Oberon, an Oberon-like language for parallel programming. They extend the Oberon language for executing action systems with type-bound actions. Type-bound actions combine the concepts of type-bound procedures (methods) and actions, bringing object orientation to action systems. Type-bound actions are created at runtime along with the objects of their bound types. They permit the encapsulation of data and code in objects. Allowing an action to have more than one participant gives us a mechanism for expressing *n-ary* communication between objects. By showing how type-bound actions can logically be reduced to plain actions, they give an extension a firm foundation in the refinement calculus.

2.7 Conclusion

Ada′s *protected type* mechanism offers mutual exclusion and synchronization control which support concurrent programming, but without object-oriented programming feature (such as inheritance). The combination of concurrency and object-oriented programming in Ada needs a lot of work.

Java and C# can be considered to be *pure* object-oriented languages. Object inheritance could become a problem with concurrent programming. When a class overrides a *synchronized* method, it must also explicitly declare the overriding method as *synchronized* otherwise the method will not be synchronized in the child. For example, if a new version of a class C is produced (with new functionality or a revised implementation, etc.), a user who had defined a subclass *C1* of the original class should be able simply to re-link with the new version. But if either version of class C contains synchronization code, this will potentially not work. The user will need the source code for the new version and may need to modify the source code for *C1*. On the other hand, threads of Java and C# needs to be created and handled explicitly. Java and C# were designed as general purpose programming languages, they were not designed for concurrent programming.

Seuss is the only programming language, which is based on action systems, but Seuss is not fully object-oriented. There is no inheritance and objects can not be created dynamically. In order to make action execution atomic, each action can only call a guarded method once and a call must be the first statement of the action. Currently, Seuss is only translated into a sequential programming language and does not take advantage of object-oriented programming.

The most important advantage of $\pi o \beta \lambda$ is that early return can be implemented by introducing some background actions to do the real work when the needed value has been returned to the caller. But synchronization is achieved by one-thread-at-a-time concept since every data is considered to be an object. There is no mechanism in the language to synchronize threads working on multiple objects. $\pi o \beta \lambda$ also lacks subtyping and inheritance.

The new object-oriented language, Lime, which is introduced in the next chapter, will solve these problems in some way. Lime is a good combination of an object-oriented programming language and action systems. Lime can simplify both the specification and design of object-oriented concurrent applications.

Chapter 3

Lime

In this chapter, a brief background of action systems is first given. Then, Lime, a new object-oriented concurrent programming language that is based on the action system, is introduced. Lime language is introduced by first given its main features and its syntax then several sample programs written in it.

3.1 Action System

Action systems were originally proposed as a formalism for parallel systems and distributed systems. This formalism is based on an extended version of Dijkstra's *guarded commands language*, which introduced the concept of *guards* and committed choice *nondeterminism* [9]. A guard is a boolean expression attached to a statement definition specifying when that statement can be executed. In an action system, which action is chosen for execution is not determined. This is called nondeterminism.

The formalism of action systems can be extended from the original form to object-orientation.

In general, *actions* are guarded commands of the form $p \rightarrow A$ where p is a boolean condition and A a statement. An action is *enabled* if the guard of it is *true*; otherwise the action is *disabled*.

An action system $\mathcal{A}$ is a set of actions operating on local and global variables. It has the form:

$$\begin{aligned} \mathcal{A} = \quad & |[\ \ var\ l := l_0;\ g := g_0 \\ & \ \ do\ A_i [] \ \cdots \ [] \ A_m\ od \\ &]| \end{aligned}$$

Action systems describe computations in which the list of local variables l and the list of global variables g are first created and initialized to be l_0 and g_0, respectively.

The local and global variables are assumed to be distinct.

After the initialization of the state variables, an enabled action A_i is selected for execution nondeterministically. Actions operating on disjoint sets of variables can be executed in parallel. Actions are considered to be *atomic*, which means that when an action is picked up for execution, it will be completed without any interference. So two actions A_i and A_j that are enabled at the same time and do not have any read-write conflicts can be executed in any order. The computation terminates if no action is enabled; otherwise it continues indefinitely. Hence we can model parallel behavior with actions taking the view of interleaving action systems.

An action system is *fair* if it is guaranteed that every action in the system gets a chance to execute. In other words, a continuously enabled action is eventually selected for execution.

3.2 What Is Lime

Lime is an action-based object-oriented concurrent programming language, which was developed by Dr. Emil Sekerinski at McMaster University [23] [22]. This language is intended to make a good combination of action systems and object-oriented programming.

The most significant difference between Lime and the traditional object-oriented languages is that *actions* and *guarded methods* are added in order to implement action systems.

3.2.1 Actions

Actions in Lime are executed automatically. They are not referenced within the program but are invoked by the scheduler. Actions have no arguments or return values.

Each action declaration has the format:

action A ***when*** b ***do*** S or ***action*** A ***do*** S

where A is the name of the action, b is a boolean condition and S is the body of the action. This action is *enabled* if b holds, otherwise it is *disabled*. The execution of an action is accepted if the action is enabled. The following is an example of an action in Lime:

```
action A
when b do begin
   x := x + 1;
```

$b := false$
end

When the scheduler chooses to execute this action (which means that the object this action belongs to is enabled), it first checks the boolean condition of this action, b=*true*. If the condition holds then the body of the action, x:=x+1; b:=*false*, is executed.

An action with the format ***action*** A ***do*** S means that the condition to execute this action is always true. In other words, action A is always enabled.

3.2.2 Methods

Unlike actions, methods in Lime may have arguments and return values. There are two types of methods in Lime, *guarded* methods and *unguarded* methods.

An unguarded method is the same as a method in other object-oriented programming languages, such as a method in Java. Here is an example of a class with an unguarded method:

class C
  method addX $(x : integer) : integer$
  begin
    $x := x + 1;$
    return x
  end
end

The unguarded method *addX* takes an integer as its argument, increases it by 1 and returns the new value to the caller. The call to an unguarded method is always accepted.

Guarded methods may accept or reject the call. They have the following format:

method M ***when*** b ***do*** S

where M is the name of the guarded method, b is a boolean condition and S is the body of the method. This guarded method is *enabled* if b holds, otherwise it is *disabled*. The following is an example of a guarded method:

class C
  attr $x : integer$

```
initialization x := 0
method addX : integer
   when x < 10 do
   begin
      x := x + 1;
      return x
    end
end
```

When this method is called, the condition, $x< 10$, is checked. If it holds, this call is accepted and the body of the method, $x:=x+1$; *return* x, is executed; if the condition does not hold, the call is rejected.

3.2.3 Program Structure and Execution

A Lime program consists of several class definitions and one program definition. The attributes, actions and methods of the class are declared in the class definition. In the program definition we control the flow of the program, allocate whatever resources are needed, and run any methods that provide the functionality for the application. It is similar to the *main* method in Java. The following example shows how Lime program is structured:

```
class C1
// body of class C1
class C2
// body of class C2
program P
begin
   //...
end
```

Execution of Lime is similar to that of traditional languages. The main difference is that we have some background threads (the number of the threads depends on the number of processors and various translation schemes) to pick up and execute the actions. We require that every action can be chosen infinitely often during the execution.

3.3 Lime Syntax

A programming language is an infinite set of sentences well formed according to its syntax. In this section, we introduce the syntax of Lime by giving the *context-free grammar* (*grammar* for short) of this language, described in EBNF (extended Backus-Naur Formalism) [1].

3.3.1 Notations

The grammar of Lime consists of terminals, nonterminals, a start symbol and a set of productions.

Terminals are the basic symbols from which strings are formed. Terminals are also called *tokens*.

Nonterminals are syntactic variables that denote sets of strings. The nonterminals define sets of strings that help to define the language generated by the grammar. They also impose a hierarchical structure on the language that is useful for both syntax analysis and translation.

Start symbol is one special nonterminal in the grammar, and the set of strings it denotes is the language defined by the grammar.

Productions of the grammar specify the manner in which the terminals and nonterminals can be combined to form strings. A production rule defines a nonterminal symbol in terms of other nonterminal symbols and/or terminal symbols. A production rule has the following form:

$$\textit{LeftHandSide} ::= \textit{RightHandSide}$$

The "::=" sign separates the definition (*RightHandSide*) from the defined symbol (*LeftHandSide*). As the result of a production rule, *RightHandSide* replaces left side wherever *LeftHandSide* appears in a production rule.

In the rest of this section, a nonterminal symbol is denoted by italic font, such as N; and a terminal symbol is put between two double quotes, such as "T".

Let M and N be two given symbols, we have the following notations:

- $M \mid N$ denotes the alternative between M and N (either M or N);
- MN denotes that the concatenation of M and N (M followed by N);

- [*M*] denotes that *M* is optional;
- { *M* } denotes a sequence of *M* (the sequence may be empty);
- (*MN*) denotes that *M* and *N* are grouped, i.e. (*M* | *N*)*M* means *MM* or *NM*.

3.3.2 Representation of Lime Programs

In a programming language, the representation of symbols in terms of characters depends on the underlying character set. The ASCII (American Standard Code Information Interchange) set is used in Lime. Following symbols are first defined:

- White space: is a non-empty sequence of blanks, new lines and tabs. White spaces are ignored by the compiler.
- Letter and Digit: are used to define other symbols, such as identifiers and numbers.
- Operators and delimiters: are either special characters or *reserved words*. Reserved words are also called keywords, they have a predefined meaning to the compiler and therefore must not be used for any other purpose.

  The operators and delimiters composed of special characters are:

+	addition
−	subtraction
*	multiplication
/	division
:=	assignment
=	equal
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
<>	not equal
[	left square bracket
]	right square bracket
(	left parentheses
)	right parentheses

The reserved words are enumerated in the following list; their meaning will be explained throughout this thesis:

abstract	action	array	and
assert	attr	begin	boolean
while	class	const	do
div	else	end	extend
false	if	implement	import
inherit	initialization	integer	interface
new	nil	not	method
when	of	or	procedure
program	protected	public	redefine
real	repeat	return	then
true	type	var	

- Comment: Lime supports three types of comments. The first type is a single-line comment, which begins with the characters "//" and continues until the end of the current line. The second kind of comment is a multi-line comment. It begins with the characters "/*" and continues, over any number of lines, until the characters "*/". Any text between the "/*" and the "*/" is ignored by the compiler. This type of comment cannot be nested (i.e., one "/* */" comment cannot appear within another one). The third type of comment is a special case of the second. If a comment begins with "/**", it is regarded as a special doc comment. Like regular multi-line comments, doc comments end with "*/" and cannot be nested.

- EOF: a special token, which is used to indicate the end of the input file.

3.3.3 Identifiers and Numbers

In Lime, identifiers are defined as sequences of letters and digits. The first character must be a letter. Identifier is defined as follows:

Identifier ::= *Letter*{*Letter* | *Digit*}

IdentifierList is a sequence of identifiers of the same type, it has the form:

IdentifierList ::= *Identifier*{"," *Identifier*}

Numbers are either integers or real numbers. Currently, only integers are supported in Lime. Numbers are denoted by sequences of digits without any space between these digits. Integer in Lime is defined as follows:

IntegerLiteral ::= [“0”-“9”]{[“0”-“9”]}

3.3.4 Compilation Unit

A *compilation unit* is a unit of source code that can be compiled. In Lime, the compilation unit has the form:

CompilationUnit ::= {*Import*“;”}{*Declaration*} *EOF*

This means that a compilation unit consists:

- A sequence of imported classes, which has the form:

  Import ::= “import” *QualifiedIdentifier*

- A sequence of declarations, including constant declarations, class declarations, procedure declarations and program declarations:

  Declaration ::= *ConstDeclaration* | *ClassDeclaration*
  | *ProcedureDeclaration* | *ProgramDeclaration*

- Finally, EOF, a special token indicating the end of the input file.

3.3.5 Constant, Procedure and Program Declarations

All identifiers must be declared before they are used, unless they are imported from another class. If an identifier is to denote a constant value, it must be introduced by a *ConstDeclaration* declaration that indicates the value for which the constant identifier stands. A constant declaration has the form:

ConstDeclaration ::= “const” *Identifier* “=” *Expression* | *Statement*

The following examples show the use of constant declaration:

$const\ A = 50 + 100$
$const\ b = false$

The compiler determines the data type of the constant automatically.
The procedure declarations and program declaration have the following forms:

ProcedureDeclaration ::= "procedure" *Identifier* [*ValueParameter*] [*ResultParameter*][*Statement*]

ProgramDeclaration ::= "program" [*VarList*] [*Statement*]

3.3.6 Class Declarations

A class is declared by giving it a name, optionally stating the class being inherited, and then listing all the attributes, initializations, methods, and actions. Class declarations in Lime have following form:

ClassDeclaration ::= "class" *Identifier*{*Inherit*}{*Feature*} "end"

- *Identifier* represents the name the class.
- *Inherit* indicates the relationship of this class to other classes or interfaces, it is defined as following:

  Inherit ::= ("inherit" | "extend" | "implement")*Identifier*

  This means that the current class can inherit, and/or extend other classes, and/or implement some interfaces.

- The body of the class is defined in *Feature*. It consists a sequence of declarations of methods, attributes, initializations, actions and constants. Each of them will be introduced in the following sub-sections.

  Feature ::= *Method* | *Attrbute* | *Initialization* | *Action* | *ConstDeclaration*

Method

Methods may have both value parameters and result parameters. Method declarations in Lime have the form:

Method ::= "method" *Identifier* [*ValueParameter*]
[*ResultParameter*][["when" *Expression* "do"]*Statement*]

Identifier is the name of the method. *ValueParameter* is the list of input parameters and their types. *ValueParameter* has the form:

ValueParameter ::= "(" *TypedIdentifierList* ")"

and *TypedIdentifierList* has the following form:

TypedIdentifierList ::= *IdentifierList* ":" *Type* {"," *IdentifierList* ":" *Type*}

ResultParameter shows the return type of the method, which has the format:

ResultParameter ::= ":" *Type*

Attribute

Attribute give the attributes of the class. It has the form:

Attribute ::= "attr" *VariableDeclarator* {"," *VariableDeclarator*}":" (*Type*)

VariableDeclarator is used to give the declaration of a variable, it has the following form:

VariableDeclarator ::= *VariableDeclaratorId* [":=" *VariableInitializer*]

where *VariableDeclaratorId* gives the name of the variable and *VariableInitializer* gives possible initialization of the variable. They have the form:

VariableDeclaratorId ::= *Identifier*{"[" "]"}

VariableInitializer::= *ArrayInit* | *Expression*

Initialization

Initialization initializes the attributes as needed. It has the following form:

Initialization ::= "initialization" [*ValueParameter*][*Statement*]

Action

Action in Lime is defined as:

Action ::= "action" *Identifier* ["when" *Expression* "do"] *Statement*.

3.3.7 Types

There are two types in Lime, the reference type and primitive type.

Type ::= *ReferenceType* | *PrimitiveType*

Primitive Types

The primitive type represents the elementary data types in Lime, integer and boolean.

PrimitiveType ::= "boolean" | "integer"

- Type "integer": This type represents the whole integer numbers, and any value of type "integer" is therefore an integer. Operators applicable to integers include following basic arithmetic operations:

+	add
−	subtract
∗	multiply
div	divide
mod	remainder of division

- Type "boolean": A "boolean" value is one of the two logical *truth values* denoted by the standard identifiers "true" and "false". A set of logical operations is provided which, together with "boolean" variables, form "boolean" expressions. These operators are "and", "or" and "not".

a and b = "if *a* then *b* else *false*"
a or b = "if *a* then *true* else *b*"
not a = "if *a* then *false* else *true*"

It needs to be mentioned that *lazy evaluation* is used in Lime boolean operations. That is to say, when evaluating *a and b*, if *a* is false, the result of *false* is returned immediately instead of evaluating the value of *b*. Similarly, when evaluating *a or b*, if *a* is true, the result of *true* is returned immediately instead of evaluating the value of *b*.

Reference Types

The reference type has the following format:

ReferenceType ::= (*ClassOrInterfaceType*{"[" "]" }) |
(*PrimitiveType*("[" "]")+)

ClassOrInterfaceType ::= *Ident* [*TypeArguments*]
"." *Ident*[*TypeArguments*]}

TypeArguments ::= "<" *ActualTypeArgument*{"," *ActualTypeArgument*}
(((">") | (">>") | (">>>"))

ActualTypeArgument ::= "?" [("extends" | "super")*ReferenceType*] |
ReferenceType

3.3.8 Expression

An *expression* is in general composed of several operands and operators. Its evaluation consists of applying the operators to the prescribed sequence, in general taking the operators from left to right.

The syntax of *Expression* in Lime is defined as follows:

Expression ::= *CondExpr* { ":=" *Expression* }

CondExpr ::= *CondOrExpr* {"?" *Expression* ":" *CondExpr*}

CondOrExpr ::= *CondAndExpr* { "or" *CondAndExpr* }

CondAndExpr ::= *EqExpr* { "and" *EqExpr* }

EqExpr ::= *RelExpr* { "=" *RelExpr* | " <> " *RelExpr* }

RelExpr ::= *AddExpr* { "<" *AddExpr* | ">" *AddExpr*
| "<=" *AddExpr* | ">=" *AddExpr* | "=" *AddExpr*
| "<>" *AddExpr*}

AddExpr ::= *MulExpr* { "+" *MulExpr* | "−" *MulExpr* }

MulExpr ::= *UnaryExpr* {"∗" *UnaryExpr*
| "div" *UnaryExpr* | "mod" *UnaryExpr* }

UnaryExpr ::= "not" *UnaryExpr* | *PrimExpr*

PrimExpr ::= *PrimPrefix* | {*PrimSuffix*}

PrimPrefix ::= *Literal* | "this"
| {*Identifier* "."} "super" "." *Identifier*
| "(" *Expression* ")" | *AllocatExpr*
| *Type* "." "class" | *Name*

Name ::= *Identifier* ["." *Identifier*]

AllocatExpr ::= "new" (*ClassOrInterfaceType*
(*ArrayDimsAndInits* | *Arguments* {*Feature*}
| "(" *Expression* ")" | *AllocatExpr*
| ("integer" | "boolean") *ArrayDimsAndInits*)

Arguments ::= "(" {*ArgumentList* } {"," } ")"

ArgumentList ::= *Expression* ["," *Expression*]

PrimSuffix ::= "." ("this" | "super" | *AllocatExpr*
| { *ReferenceTypeList* } *Identifier*) | "[" *Expression* "]"
| *Arguments* | *ReferenceTypeList*

ReferenceTypeList ::= "<" *ActualTypeArgument* ["," *ActualTypeArgument*]
| { *ReferenceTypeList* } *Identifier*) | "[" *Expression* "]"

| *Arguments* | *ReferenceTypeList*

ArrayDimsAndInits ::= ("[" *Expression* "]")+ { "[" "]" }
| ("[" "]")+ *ArrayInit*

ArrayInit ::= "{" [*VariableInitializer* { "," *VariableInitializer* }]
[","] "}"

There are a few basic rules governing expressions in Lime, for example, every variable in an expression must have been previously defined and assigned a value; two operators must never be written side by side. For instance, $a*-b$ is not permitted and must be written as $a*(-b)$.

3.3.9 Statements

The specification of an action is called a *statement*. *Statement* in Lime has the following form:

Statement ::= *Compound* | *If* | *While* | *Return* | *Var* | *StatementExpression*

Statement Expression

The *statement expression* has the following form:

StatementExpression ::= *PrimExpr* [":=" *Expression*]

Compound and Return Statements

The *compound* statement represents a sequence of statements, starting with "begin" and ending with "end". *Compound* is defined as:

Compound ::= "begin" *Statement* { ";" *Statement* } "end"

For example, if a statement S_1 follows S_0, it is expressed as:

begin S_0; S_1 *end*

A *Return* statement consists of the symbol "return", possibly followed by an expression. It indicates the termination of a method, and the expression specifies the value returned as result of the method call. Its type must be assignment compatible with the result type specified in the method declaration. *Return* statement has the following format.

Return ::= "return" [*Expression*]

Control Structures

It is a prime characteristic of a programming language that individual actions can be selected, repeated, or performed conditional depending on some previously computed results. This is determined by the control structures. There are two control structures in Lime, *While* and *If* statement.

- The *While* statement is defined as follows:

  While ::= "while" *Expression* "do" *Statement*

  The *Expression* is of type "boolean" which was discussed in the section on data types. Following is an example of using the *While* statement to calculate the sum of first 100 positive integers (i and *sum* are initialized to be 0, respectively):

  $$\begin{array}{l} while\ i <= 100\ do \\ \quad begin\ sum := sum + 1;\ i := i + 1\ end \end{array}$$

- The if statement has the form

  If ::= "if" *Expression* "then" *Statement* ["else" *Statement*]

  The *Expression* here is also of type "boolean".

  The following two examples illustrate general form of *If* statement:

  $$\begin{array}{l} if\ a = 0\ then\ b := 1 \\ if\ b > 0\ then\ a := b\ else\ a := -b\text{W} \end{array}$$

3.4 Lime Examples

It is much easier to develop concurrent programs in Lime than in other traditional object-oriented languages. The programmer only needs to create the actions the objects have and does not need to worry about how those actions are scheduled in the background. All that scheduling work is left to the Lime compiler.

3.4.1 An Integer Generator

Following example shows how a Lime class, *IntegerGenerator*, implements a simple arbitrary integer number generator using actions. The class has two actions, *inc* and *dec*, which increment and decrement the counter respectively.

```
class IntegerGenerator
    attr n : integer
    attr m : integer
    initialization n := 0
    action inc
        n := n + 1
    action dec
        when n > 0 do n := n − 1
    method fetch : integer
    begin
        m := n;
        n := 0;
        return m
    end
end
```

Note that the action *inc* is always enabled.

3.4.2 A General Semaphore

Synchronization is a fundamental aspect of a concurrent program. Semaphores are the first and remain one of the most important synchronization tools. The following example shows a general semaphore implemented in Lime.

```
class Semaphore
    attr value : integer
    initialization value := 0
```

```
    method P
    /*to access and lock the object*/
        when value > 0 do value := value − 1
    method V
    /*to release the lock on the object*/
        value := value + 1
end
```

Since the V operation on the semaphore is always accepted, it is implemented as an unguarded method. The P operation is only accepted when *value*> 0, so it is implemented as a guarded method.

3.4.3 Reader-Writer

The following example shows how to use monitors to implement the reader/writer problem in Lime. Reader processes query a database and writer processes examine and alter it. We need to ensure that a resource is either accessed by up to R readers or a single writer. Attribute n is the number of readers that are currently reading the resource, N is the maximum number of readers can read the resource at the same time, both attribute are initialized to be R [23].

```
class ReaderWriter
    attr n, N : integer
    initialization(R : integer)
        begin n := R; N := R end
    method StartRead
        when n > 0 do n := n − 1
    method startWrite
        when n = N do n := 0
    method endRead
        n := n + 1
    method endWrite
        n := N
end
```

3.4.4 Dining Philosophers

The *Dining Philosophers* problem is a traditional problem in the history of concurrency. N philosophers are sitting around a circular table on which are placed N plates and N forks, one plate in front of each philosopher, one fork between each pair of

philosophers. Philosophers sit and think for a while, then get hungry, pick up the fork on each side, and eat. Then they return them to the table, and return to thinking.

The next example shows how to implement the Dining Philosophers problem in Lime. The eating process of each philosopher is implemented as an action; and one philosopher can only pick up the fork when the forks on his both sides are available. Here the monitor mechanism is used. The class *Philosopher* contains an action *eat*, and the action is enabled when both the left and right forks of the philosopher are available. In the program *NPhilosopher*, we only need to create the array of the philosophers and the forks, and assign the forks to the philosophers. The schedule of how the philosophers picking up the forks is left to the Lime compiler.

```
//The class of Philosopher
class Philosopher
   attr leftFork, rightFork : Fork
   action eat
   when leftFork.avail and rightFork.avail do begin
      leftFork.avail := false;
      rightFork.avail := false;
   end
   action finishEat
   when notleftFork.avail and notrightFork.avail do begin
      leftFork.avail := true;
      rightFork.avail := true
   end
end
//The class of Forks
class Fork
   attr avail : boolean
   initialization avail := true
end
//The class of N philosopher problem
class NPhilosopher
   attr N : integer
   attr i : integer
   attr philosopher[] : Philosopher
   attr fork[] : Fork
   method start
   begin
      //create philosophers and forks
```

```
        philosopher = new Philosopher[N];
        fork = new Fork[N];
        i := 0;
        //assign forks to philosophers
        while (i < N − 1) do begin
            philosopher[i].leftFork := fork[i];
            philosopher[i].rightFork := fork[i + 1];
            i := i + 1
        end;
        philosopher[i].leftFork := fork[i];
        philosopher[i].rightFork := fork[0]
    end
end
```

3.4.5 A Screen Saver

Our next example implements a program that locks a terminal each time the unguarded method *lock* is invoked. No more interaction is possible until the correct password is entered at the keyboard. The variable *pword* is the password that is used to unlock the terminal.

```
class ScreenSaver
    attr b : boolean
    attr input, pword : integer
    initialization b := true
    method getKeyboardInput : integer
    /*to get the input integer from the keyboard*/
    method outputDisplay(i : integer)
    /*to display the input integer on the terminal*/
    method setPassword(i : integer)
        pword := i
    method lock
        b := false
    action checkPassword
        when not b do begin
            input := getKeyboardInput;
            b := (pword = input)
        end
```

```
action displayInput
    when b do begin
        input := getKeyboardInput;
        outputDisplay(input)
    end
end
```

Two actions, *checkPassword* and *displayInput* are nondeterminately selected to execute during the program execution. When the screen is locked, the action *checkPassword* is enabled. The input from the keyboard is used to compare with the password (*pword*) and to unlock the screen if they are the same. When the screen is not locked, the action *displayInput* is enabled. Then the input from the keyboard will be displayed on the screen.

3.4.6 Fish Screen Saver

This example was first introduced in [2]. Some fish swim around the screen. The position of each fish is given by horizontal axis x and vertical axis y. The fish is either moving up (when *up* is true) or down, and either right (when *right* is true) or left. When a fish reaches the border of the screen, it will change the direction.

In the following implementation, each possible movement of a fish is implemented as an action.

```
class Fish
    attr x, y : integer
    attr up, right : boolean
    const WIDTH = 20
    const HEIGHT = 10
    initialization
    begin
        x := 0; y := 0; up := true; right := true
    end
    action moveUp
        when y < HEIGHT and up do y := y + 1
    action moveRight
        when x < WIDTH and right do x := x + 1
    action moveLeft
        when x > 0 and not right do x := x − 1
    action moveDown
```

```
        when y > 0 and not up do y := y − 1
    action bounceUp
        when y = 0 and up do up := false
    action bounceDown
        when y = HEIGHT and not up do up := true
    action bounceRight
        when x = WIDTH and right do right := false
    action bounceLeft
        when x = 0 and not right do right := true
end
```

Then we can implement the screen saver program as follows. We can create as many as *numFish* fish and do not need to worry about how to schedule those actions, the Lime compiler will do it for us.

```
programFishScreenSaver
    var i : integer
    var f : Fish
    const numFish = 10
    begin
        i := 0;
        while i < numFish do begin
            f := new Fish; i := i + 1 end
    end
```

Lime is an object-oriented programming language so we can define a class in terms of another class. The definition of the subclass inherits all the operations of the superclass. Both methods and actions can be overridden. Previously we define a class *Fish* that has some actions making the fish move on the screen. The following example defines a class *FastFish* that inherits from *Fish*, actions in *FastFish* overrides the ones in its superclass.

```
class FastFish extends Fish
    action moveUp
        when y < HEIGHT and up do y := y + 2
    action moveRight
        when x < WIDTH and right do x := x + 2
    action moveLeft
        when x > 0 and not right do x := x − 2
```

```
    action moveDown
        when y > 0 and not up do y := y − 2
end
```

3.4.7 Priority Queue

In the previous chapter, an example of a priority queue using the *early return* mechanism in $\pi o\beta\lambda$ was introduced. It is complicated to implement *early return* in most programming languages, such as Java and C#. The *early return* can be implemented in Lime by actions. The following program shows this priority queue example implemented in Lime:

```
class Priq
    attr m : integer
    attr l : Priq
    attr a : boolean
    attr p : integer
    initialization a := false
    method add(e : integer)
        when not a do
            if l = nil then begin m := e; l := new Priq end
            else begin p := e; a := true end
    action doAdd
    //The action do the real adding work
        when a do
        begin
            if m < p then l.add(p)
            else begin l.add(m); m := p end;
            a := false
        end
    method remove : integer
        when not a do
        begin
            var x : integer;
            x := m; m := l.remove(); return x end
end
```

In this example, an action *doAdd* is added to perform the real task of adding new

element to the proper position of the queue. This action is enabled when its guard, boolean variable *a* holds (means there is an element that needs to be added). The method *add* is guarded, it is only enabled when *a* is *false* (means that there is no other element being added). The method *add* returns immediately to the caller after setting *a* to be true and leaves all the adding work to the background action, *doAdd*.

3.4.8 A Simple Symbol Table

Our last example shows how to implement *early return* in Lime. There is a simple symbol table implemented in class *SymbolTable*. The data in this table is an object of class *SymbolEntry* and each entry has a key which is of type integer. The data in this symbol table is ordered by the key value. The method *insert* tries to insert new data in the table. It only inserts the data when it can find the correct place for the data immediately. Otherwise the method only sets the boolean variable *a* to be *true* to enable the action *doInsert* which can do the complex insertion work, and returns to the caller.

```
class SymbolTable
    attr mk, md, k1, d1 : integer
    attr l, r : SymbolTable
    attr a : boolean
    initialization begin a := false; mk := 0; md := 0; l := nil; r := nil end
    method insert(k : integer, d : integer)
        when not a do
            if mk = 0 then begin mk := k; md := d end
            else if mk = k then md = d
                else begin a := true; k1 := k; d1 := d end
    /*The action to do the real insertion work*/
    action doInsert
        when a do begin
            if k1 < mk then begin
                if l = nil then l := new SymbolTable();
                l.insert(k1, d1); mk := k1; md := d1 end
            else begin
                if r = nil then r := new SymbolTable();
                r.insert(k1, d1); mk := k1; md := d1 end;
            a := false
        end
    method search(k : integer) : integer
```

```
        when not a do
            if k = mk then return md
            else if k < mk then return l.search(k)
                else return r.search(k)
end
```

3.4.9 Other Examples

The array data type is important all programming languages. Lime also supports the array data type. The following is an example that shows how to operate on an array of integers:

```
class IntArray
    attr x : integer
    attr i : integer
    attr b[] : integer //declare but not create
    method A
    begin
        i := 0; x := 10;
        b := new integer[x]; //create the array
        while i <= 9 do begin
            b[i] := x * 5;
            i := i + 1
        end
    end
end
```

Lime methods may have parameters associated with them. These parameters are assigned copies of the actual parameters when the method is called. The parameter then acts as a local variable within the called method. Any changes made to the value of this variable last for the lifetime of the method and do not affect the original value. There are two ways in Lime for passing arguments. If the arguments are of primary type, they are passed by their value; if they are of object or array, then they are passed by reference.

```
class ParameterPassing
   attr a : integer
   attr b[] : integer
```

```
    method foo
    begin
        a := 1;
        b := new integer [2];
        b[0] := 1;
        b[1] := 2;
        m(a);
        n(b)
    end
    method m(p : integer)
    //parameter is passed by its value
        p := p + 2
    method n(b : integer[]) begin
    //parameter is passed by its reference
        p[0] := p[0] * 2;
        p[1] := p[1] * 2
    end
end
```

Lime is compiled into java class file and executed on the Java Virtual Machine. Methods defined in Java can be called by Lime. In this way we can take advantage of Java features and develop complex programs. The following example shows the use of Java method *System.out.println* to print the output to the screen.

```
class C
    attr i, k : integer
    method m begin
        i := 0; k := 0;
        while i < 100 do begin
            k := k + i;
            i := i + 1
        end
        //call Java method
        System.out.println(k)
    end
end
```

Chapter 4

Java Virtual Machine and Jasmin

The target language of the Lime compiler is a Java assembly language called Jasmin. The assembly language code can then be translated to Java byte-code (Java class file) by the Jasmin assembler such that it can be executed on a Java Virtual Machine (JVM for short).

The reason to translate Lime into the Java assembly language instead of Java source code is that we need to control when to enter and exit the monitor (to lock or release an object). The entering and exiting of the monitor need to come in pair by using the keyword *synchronized* in Java. So it is not possible to control the entering and exiting of a monitor from the Java source code.

In this chapter, some basic knowledge of the components of JVM and some JVM instructions we use in our compilation are first introduced; then we give a brief introduction to Jasmin. These are fundamentals for our translation schemes.

4.1 Java Virtual Machine

JVM is an abstract machine used as a target for Java compilation. It is a platform-neutral runtime engine used to execute Java programs. The relevant components of JVM are introduced in this section [18].

4.1.1 Components of Java Virtual Machine

Class File Loader

The interface to the JVM is through the class file. A class file has a strict format. The strictness of the format is required in part so that the JVM specification remains well defined. The class file loader is used to read the class file, verify it and initialize its static fields. There is information on constants, fields, methods and attributes in the

class file. Function γ is defined to return the information contained in the class file. Let *ClassName* be the name of a class, then $\gamma(ClassName) = (\gamma_K, \gamma_F, \gamma_M, \gamma_P, S, C)$ where:

Constant Pool γ_K : a pool for constants and literals used by the class. For arrays, it contains entries (n, t) where n is the dimension of the array and t is the type of elements in the array. For fields, it contains entries of the form (c, f) where c is the name of the class and f is the field's name.

Field Table γ_F : a list of fields in the class. Entries in the field table have the form (t, sd, a) where t is the type of the field and sd is either *static* or *dynamic*. If sd is static then a is the address of the static field otherwise a is an offset of the non-static field.

Method Table γ_M : a list of methods in the class (with flags and method signatures). Entries in the table have the form (c, p, n, l, sd, ms) where c is the name of the class that the method belongs to; p is the pointer to the byte-code of the method; n and l are the number of arguments and local variables of the method. ms is the maximum number of stack cells needed by the method execution and sd is either *static* or *dynamic*.

Byte-Code Programs γ_P : sequences of cells of size 1 byte. For some instructions, the arguments are encoded in the instruction itself (e.g. *aload_0*, *iconst_4*). For most instructions, the instruction and its arguments are stored in consecutive cells.

Superclass S : the superclass of the current class and any interfaces this class implements.

This Class C : code segments that implement this class' methods.

Execution Environment

The JVM has 4 registers used for program execution [18]:

PC: program pointer, points to a position in the program store;

VARS: all local variables are addressed relative to this register;

OPTOP: points to the topmost cell of the operand stack;

FRAME: points to the first cell of the execution environment.

JVM Instruction	Meaning
add	add two integers
neg	negate an integer
isub	subtract two integers
imul	multiple two integer
idiv	divide two integers

Table 4.1: JVM instructions for integer arithmetic

Store

The *store*, *stack* and *heap* are sequences of cells of size 4 bytes; stack and heap are disjoint areas of the store [18].

4.1.2 Java Virtual Machine Instruction Set

A Java Virtual Machine instruction consists of an opcode specifying the operation to be performed, followed by zero or more operands embodying values to be operated upon. There are 166 instructions in the current version of JVM. The meanings of instructions that are used in Lime compiler are briefly introduced in the remaining of this section [18, 19]. The details of these instructions, especially the usages and how they operate on the stack are introduced in [19].

The JVM instructions are grouped into the following eight category according to their usage in the Lime compiler:

- Instructions for integer arithmetic are introduced in Table 4.1;
- Instructions for stack manipulations are introduced in Table 4.2;
- Instructions for integer comparison and jumps are introduced in Table 4.3;
- Instructions for creating new arrays are shown in Table 4.4;
- Instructions for accessing arrays or objects are introduced in Table 4.5;
- Instructions for method returns are shown in Table 4.6;
- Instructions for method calls are introduced in Table 4.7;
- Instructions for assignments are introduced in Table 4.8.

JVM Instruction	Meaning
iconst_n	push the constant n(0-5) onto the stack
bipush n	push one-byte signed integer
ldc n	push single-word constant onto stack
istore n	store integer in local variable n
iload n	push integer from local variable n

Table 4.2: JVM instructions for stack manipulation

JVM Instruction	Meaning
goto a	branch to address
ifeq a	jump if equal to zero
ifgt a	jump if greater than zero
ifle a	jump if less than or equal to zero
if_icmpeq a	jump if two integers are equal
if_icmpne a	jump if two integers are not equal
if_icmple a	jump if first integer is less than or equal to the second one
if_icmpge a	jump if first integer is greater than or equal to the second one
if_icmplt a	jump if first integer is less than the second one
if_icmpgt a	jump if first integer is greater than the second one

Table 4.3: JVM instructions for integer comparison and jumps

JVM Instruction	Meaning
newarray t	allocate new array for integer number or boolean
anewarray t	allocate new array for objects of type t
multianewarray t d	allocate multi-dimensional array of type t with dimension d

Table 4.4: JVM instructions for creating new arrays

JVM Instruction	Meaning
aload *v*	retrieve object reference from local variable
aaload	retrieve object reference from array
iaload	retrieve integer from array
astore *v*	store object reference in local variable
aastore	store object reference in array
iastore	store in integer array

Table 4.5: JVM instructions for accessing arrays or objects

JVM Instruction	Meaning
return	return from a method whose return type is void
ireturn	return from method with integer result
areturn	return from method with object reference result

Table 4.6: JVM instructions for method return

JVM Instruction	Meaning
iaload	retrieve integer from array
invokespecial	invoke method belonging to a specific class
invokevirtual *i*	call an instance method

Table 4.7: JVM instructions for method call

JVM Instruction	Meaning
getstatic *i*	get value of static field
putstatic *i*	set value of static field
getfield *i*	get value of object field
putfield *i*	set value of object field

Table 4.8: JVM instructions for assignments

4.2 Jasmin

The Lime compiler uses Jasmin as the target language. Jasmin was developed by Jon Meyer [19]. It takes ASCII descriptions for Java classes, written in a simple assembler-like syntax and using the JVM instruction set. Jasmin converts them into binary Java class files suitable for executing on a JVM [19].

Jasmin takes care of the generation of the constant pool, so that one can write numbers, method signatures and types as strings in Jasmin files. The mapping of labels to addresses in the byte-code is also taken care by Jasmin, so that one can use strings as labels in jump instructions.

Jasmin source files consists of a sequence of newline-separated statements. There are three types of statements:

- Directives: used to give Jasmin meta-level information. A directive statement consists of a directive name, and then zero or more parameters separated by spaces, then a newline. All directive names start with a '.'. For example:
  - *.method public* $M()\,V$: declares a public method M whose return type is *void*;
  - *.field public foo* I: declares a public integer field *foo*;
  - *.class* C: declares a class C.
- Instructions: which are JVM instructions.
- Labels: consist of a name followed by a ':', and a newline. Labels can only be used within method definitions and are local to that method.

A Jasmin file starts by giving information on the class being defined in the file – such as the name of the class by *.class*, the name of the superclass by *.super*, etc. The following section of the Jasmin file is a list of field definitions by the directive *.field*. The rest of the Jasmin file lists methods defined by the class.

The following example shows a Jasmin program that is generated by the Lime compiler. The original Lime program is the example shown in Section 3.3.3. Note that a Jasmin comment starts with a ";" character, and terminates with the newline character at the end of the line [19]. The methods P and V are translated differently because P is a guarded method while V is an unguarded one. The details of the translation schemes will be introduced in the next two chapters.

```
    ; Generated By Lime Compiler at 6:27:1 on 4/2/2004.
;————————————————————————————-*/
.class public Semaphore
```

```
.super java/lang/Object
.implements ActiveObject
.field value I

;The standard class initializer, from the initialization part of the program
.method public <init>()V
.limit stack 2
.limit locals 1
    aload_0
    invokespecial java/lang/Object/<init>()V
    ;initialize the variables
    aload_0
    iconst_0
    ; type of attribute is I
    putfield Semaphore/value I
    aload_0
    iconst_1
    putfield Semaphore/inPool
    aload_0
    iconst_0
    ; type of attribute is
    putfield Semaphore/nextAction
    return
.end method

; ———-method P ———
.method P()V
.limit stack 4
.limit locals4
    aload_0
    monitorenter
    goto L_W_TEST
    L_W_BODY :
    L_WAIT_1_BEGIN :
    aload_0
    invokevirtual java/lang/Object/wait()V
    EXCEPTION_HANDLER :
    L_WAIT_1_END :
    .catch java/lang/InterruptedException from L_WAIT_1_BEGIN to
```

```
    L_WAIT_1_END using EXCEPTION_HANDLER
    L_W_TEST :
    aload_0
     getfield Semaphore/value I
    ineg
    iconst_0
    if_icmpgt L_GT_1_1
    iconst_0
    goto L_GT_1_2
    L_GT_1_1 :
    iconst_1
    L_GT_1_2 :
    ifgt L_W_BODY
    aload_0
    aload_0
    getfield Semaphore/value I
    iconst_1
    isub
    ; type of attribute is I
    putfield Semaphore/value I
    ;—- if ... then ...——-
    aload_0
    getfield Semaphore/inPool
    ineg
    ifle L_IF_1_1
    aload_1
    invokevirtual ObjectPool/add()V
    aload_0
     iconst_1
    ; type of attribute is
    putfield Semaphore/inPool
    L_IF_1_1 :
    aload_0
    monitorexit
    return
.end method
; end of method P

;———-method V ———
```

```
.method V()V
.limit stack 4
.limit locals 4
    aload_0
    monitorenter
    ;—- if ... then ...———-
    aload_0
    getfield Semaphore/inPool
    ineg
    ifle L_IF_2_1
    aload_0
    getfield Semaphore/this
    aload_1
    invokevirtual ObjectPool/add()V
    aload_0
    iconst_1
    putfield Semaphore/inPool
    L_IF_2_1 :
    aload_0
    aload_0
    getfield Semaphore/value I
    iconst_1
    iadd
    ; type of attribute is I
    putfield Semaphore/value I
    aload_0
    monitorexit
    return
.end method
; end of method V
```

Chapter 5

Basic Translation Schemes

Lime is an ongoing research project and many aspects are likely to evolve over time. Therefore the design and implementation of the compiler should be as general as possible. The most important feature of Lime is that the actions and guarded methods are added to the language. How to translate the actions and the guarded methods is the key of the Lime compiler. On the other hand, the translation of a Lime program without actions and guarded methods is more straightforward. For this reason, we separate the compilation of a Lime program to two steps. First, the program is translated into an intermediate Lime program without actions and guarded methods. Then the intermediate Lime program is translated into Jasmin. This two-step translation is shown in Figure 5.1.

This chapter introduces the basic translation schemes, which are used to translate Lime without actions and guarded methods into Jasmin.

For the ease of notation of the translation schemes, a function *code* is defined as follows:

$$code\ p\ \rho$$

where p is the Lime program that is being translated; ρ is the address environment of this Lime program [8]. The function is defined by structural induction on p and returns sequences of Jasmin instructions.

The notion of types in Jasmin and in Lime is different. In Jasmin, for the type *integer* the abbreviation "I" is used, and for the type *void* the abbreviation "V" is used. Array types have "[" as a prefix and class types an "L" as a prefix and a ";" as a suffix. Following examples show the Jasmin expressions and their meanings:

LTest	type of an object *Test*
Test/F(I)V	method F in class *Test*, has an *integer* as its

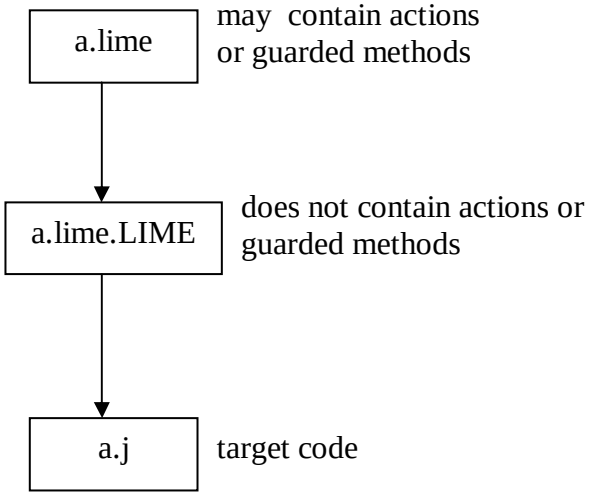

Figure 5.1: Translation of Lime program

	argument and return type is *void*
Test/F(I)I	method *F* in class *Test*, has an *integer* as its argument and returns an *integer*
Test/F(II)I	method *F* in class *Test*, has two *integer*s as its arguments and return type is *integer*
Test/F(LTest;B)LTest	method *F* in class *Test*, has two arguments, one is of type *Test* and one of *boolean*, return type is *Test*
[I	one-dimensional array of type *integer*
[[LTest	2-dimensional array of type *Test*

A function ξ is defined to return the corresponding Jasmin type by given a Lime type t:

$$\xi(t) = \begin{cases} I & t = integer \\ V & t = void \\ B & t = boolean \\ Lt & t \text{ is a class name} \end{cases}$$

5.1 Translation Schemes for Expressions

To translate integer and boolean expressions, JVM instructions for integer arithmetic, stack manipulation and conditional/unconditional jumps are needed.

The translation schemes for some boolean expressions are shown in Table 5.1. Other boolean expressions can be translated by using similar schemes. The integer

arithmetic expressions and assignments to local integer variables are shown in Table 5.2. The assignments to arrays, classes and object fields will be shown later in this chapter.

5.2 Translation Schemes for Statements

The basic control flow statements, *If* and *While* statements, and *Compound* statement are translated by the schemes shown in Table 5.3.

Lime supports both one-dimensional and multi-dimensional arrays. The elements of an array can be primitive types or objects. JVM instructions *newarray*, *anewarray* and *multianewarray* are used to create new arrays of different types.

- Multi-dimensional arrays of both primitive types or objects are created by the instruction *multianewarray*:
  $code(\textbf{new}\ t\ [e_1]\dots[e_k])\ \rho =$
  $code\ e_1\ \rho$
  $\dots$
  $code\ e_k\ \rho$
  $\textbf{multianewarray}\ i\ k$
  where $\gamma_K(i) = (k, t)$.

- One-dimensional arrays for primitive types are created by instruction *newarray*:
  $code(\textbf{new}\ t\ [e])\ \rho =$
  $code\ e\ \rho$
  $\textbf{newarray}\ i$
  where $t \in \{integer, boolean\}$ and $\gamma_K(i) = t$.

- One-dimensional arrays for types other than integer and boolean are created by instruction *anewarray*:
  $code(\textbf{new}\ c\ [e])\ \rho =$
  $code\ e\ \rho$
  $\textbf{anewarray}\ i$
  where c is a class name and $\gamma_K(i) = c$.

The following is the translate scheme to translate the access to a multi-dimensional array of integers. Translation of accessing one-dimensional arrays and arrays of other types is similar.

$$\begin{aligned} code\ x[i_1]\dots[i_k]\ \rho\ &= \textbf{aload}\ \ \rho(x) \\ &\quad\ \ code\ i_1\ \rho \end{aligned}$$

Lime expression	Translation scheme
$code(e1 = e2)\ \rho$	$code\ e1\ \rho$ $code\ e2\ \rho$ **isub** **ifeq** $l1$ **iconst_0** **goto** $l2$ $l1$: **iconst_1** $l2$:
$code(e1\ and\ e2)\ \rho$	$code\ e1\ \rho$ **ifle** $l1$ $code\ e2\ \rho$ **ifle** $l1$ **iconst_1** **goto** $l2$ $l1$: **iconst_0** $l2$:
$code(e1\ or\ e2)\ \rho$	$code\ e1\ \rho$ **ifgt** $l1$;*e1 is true* $code\ e2\ \rho$ **ifle** $l1$;*e2 is true* **iconst_0** ; *both are false* **goto** $l2$ $l1$: **iconst_1** $l2$:
$code(e1 > e2)\ \rho$	$code\ e1\ \rho$ $code\ e2\ \rho$ **isub** **ifgt** $l1$ **iconst_0** **goto** $l2$ $l1$: **iconst_1** $l2$:
$code(not\ e)\ \rho$	$code\ e\ \rho$ **ifeq** $l1$ **iconst_1** **goto** $l2$ $l1$: **iconst_0** $l2$:

Table 5.1: Translation schemes for boolean expressions

Lime expression	Translation scheme
$code(e1 + e2)\ \rho$	$code\ e1\ \rho$ $code\ e2\ \rho$ **iadd**
$code(e1 - e2)\ \rho$	$code\ e1\ \rho$ $code\ e2\ \rho$ **isub**
$code(e1\ div\ e2)\ \rho$	$code\ e1\ \rho$ $code\ e2\ \rho$ **idiv**
$code(e1 * e2)\ \rho$	$code\ e1\ \rho$ $code\ e2\ \rho$ **imul**
$code(e1\ mod\ e2)\ \rho$	$code\ e1\ \rho$ $code\ e2\ \rho$ **imod**
$code\ x\ \rho$ x: an object variable of type t	**getfield** $x\ \ \xi(t)$
$code\ x\ \rho$ x: a class variable of type t	**getstatic** $x\ \ \xi(t)$
$code(-e1)\ \rho$	$code\ e1\ \rho$ **ineg**
$code\ c\ \rho$ c: a constant	**ldc** c
$code\ (x := e)\ \rho$ x: a local variable	$code\ e\ \rho$ **istore** $\rho(x)$

Table 5.2: Translation schemes for integer arithmetic

Lime expression	Translation scheme
code(*begin* st_1; st_2 *end*) ρ	*code* st_1 ρ *code* st_2 ρ
code (***if*** *e* ***then*** *st*) ρ	*code e* ρ **ifeq** *label* *code st* ρ label:
code(**if** *e* **then** st_1 **else** st_2) ρ	*code e* ρ **ifeq** label_1 *code* st_1 ρ *goto* label_2 label_1 : *code* st_2 ρ label_2 :
code (***while*** *e* ***do*** *st*) ρ	label_1 : *code e* ρ **ifeq** label_2 *code st* ρ *goto* label_1 label_2 :

Table 5.3: Translation schemes for compound and control flow statements

aaload
⋮
code i_{k-1} ρ
aaload
code i_k ρ
iaload

If k is smaller than the dimension of the array stored in x, then the JVM instruction **aaload** should be used instead of **iaload**. **aaload** should also be used when x contains an array of objects.

Translation Schemes for Class Declarations

The following is the scheme used to translate class declarations:

code(**public class** *ClassName* **extends** *SuperClass body*) ρ =
.class public *ClassName*
.super *SuperClass*
$code_{Fields}$ *body* ρ
.method public <**init**>()**V**
aload_0
invokespecial *SuperClass*/ <**init**>()**V**
$code_{InitObjectFields}$ *body* ρ
return
.end method
.method public <**clinit**>()**V**
return
.end method
code body ρ

In this translation scheme, function $code_{Fields}$ and $code_{InitObjectFields}$ are used to traverse the *body* of the class declaration and process field declarations [8]. They are defined as follows:

- $code_{Fields}(attr\ x : type)\ \rho$ =**.field public** $x\ \xi(type)$
- $code_{InitObjectFields}(x := e)\ \rho = code(x := e)\ \rho$

Each method is uniquely determined by its signature. The signature of a method contains:

- The name of the class C where the method is defined in;

- The name of the method, M;
- The types t_i of its parameters (and as a result implicitly the number n of its parameters);
- The return type of the method, t.

This signature of the method is stored in the constant table. Further information on this method can be found in the method table according to the signature.

The translation scheme for method declarations is as follows:

$$
\begin{aligned}
&\mathit{code}(m(p_1 : t_1, \cdots, p_k : t_k) : t\ \mathit{MethodBody})\ \rho = \\
&\qquad\qquad \textbf{.method public}\ m(\xi(t_1) \cdots \xi(t_k))\xi(t) \\
&\qquad\qquad\quad \mathit{code}\ \mathit{MethodBody}\ \rho \\
&\qquad\qquad \textbf{.end method}
\end{aligned}
$$

Declarations of local variables are translated as follows:

- When $t \in \{integer, boolean\}$:
  $\mathit{code}(\ \textbf{var}\ x : t)\ \rho = \textbf{istore}\ \rho(x)$
- When t is a reference type:
  $\mathit{code}(\ \textbf{var}\ x : t)\ \rho = \textbf{astore}\ \rho(x)$

In Jasmin, a **return** statement must be executed at the end of each method. The translation schemes for method returns are as follows:

- $\mathit{code}\ \textbf{return}\ \rho = \textbf{return}$
- $\mathit{code}(\textbf{return}\ e)\ \rho =$
  $\mathit{code}\ e\ \rho$
  ireturn
  (if $infer\_type(e) \in \{integer, boolean\}$)
- $\mathit{code}(\textbf{return}\ e)\ \rho =$
  $\mathit{code}\ e\ \rho$
  areturn
  (if $infer\_type(e)$ is a reference type)

The function $infer\_type(e)$ above is defined to return the inferred type of an expression e.

Method Call

For a method call, the reference of the object also needs to pass to the method. The following is the translation scheme for the call to a virtual method, where C is the type inferred for expression e. If there is no expression e, the instruction **aload_0** is generated to get a reference to the current object. C is the name of current class.

$$
\begin{aligned}
&\textit{code } e.m(e_1, \cdots, e_k)\ \rho = \\
&\qquad\qquad \textit{code } e\ \rho \\
&\qquad\qquad \textit{code } e_1\ \rho \\
&\qquad\qquad \vdots \\
&\qquad\qquad \textit{code } e_k\ \rho \\
&\qquad\qquad \textbf{invokevirtual } C/m(\xi(t_1)\cdots\xi(t_k))\xi(t)
\end{aligned}
$$

5.3 Translation Schemes for Objects

The initialization of class and object fields can be done by two special methods, <*init*> and <*clinit*>.

- The body of the method <*clinit*> contains the assignments for static fields. <*clinit*> is called once when the class is loaded to initialize the static fields.
- The body of method <*init*> contains the assignments of other fields. <*init*> is called when an object is created and initializes its object fields.

To create a new object, the JVM instruction **dup** (to duplicate top single-word item on the stack) and **new** (to create an object) are used. The translation scheme for creating a new object is shown as follow:

$$
\begin{aligned}
&\textit{code } \textbf{new } c()\ \rho = \\
&\qquad\qquad \textbf{new } c \\
&\qquad\qquad \textbf{dup} \\
&\qquad\qquad \textbf{invokespecial } c/ < init > ()V
\end{aligned}
$$

5.4 Translation Schemes for Assignments

In Lime, we can assign values to local variables, object fields or class fields.

The translation schemes of assignments ares as follows:

- $code(c.x := e)\ \rho =$
  $code\ e\ \rho$
  putstatic $\xi(c)/x \quad \xi(t)$
  where x is a class field and t its type.

- $code(e_1.x := e_2)\ \rho =$
  $code\ e_1\ \rho$
  $code\ e_2\ \rho$
  putfield $\xi(c)/x \quad \xi(t)$
  where x is an object field and $Type(e_1) = c$ and t the type of x.

- $code(x := e)\ \rho =$
  $code\ e\ \rho$
  istore $\rho(x)$
  where x is a local variable and $Type(e) \in \{integer, boolean\}$.

- $code(c := e)\ \rho =$
  aload 0
  $code\ e\ \rho$
  putfield $\xi(c)/x \quad \xi(t)$
  where x is an object field of current class c and t the type of x.

- $code(c := e)\ \rho =$
  $code\ e\ \rho$
  putstatic $\xi(c)/x \quad \xi(t)$
  where x is a class field of the current class c and t the type of x.

The following is the translation scheme for assignments to arrays:

$code(e_1.x[d_1]\cdots[d_k] := e_2)\ \rho =$
$code\ e_1\ \rho$
getstatic $\xi(c_1)/\xi(t)$
$code\ d_1\ \rho$
aaload
$\vdots$
$code\ d_{k-1}\ \rho$
aaload
$code\ d_k\ \rho$
$code\ e_2\ \rho$
aastore

Chapter 6

Advanced Translation Schemes

The translation schemes used to translate a Lime program with actions or guarded methods into an intermediate Lime program that is without actions and guarded methods are introduced in this chapter. Actions and guarded methods are translated into unguarded methods, and some routines are added into the intermediate Lime program to handle actions and guarded methods [23].

6.1 Translation Principles

For the ease of discussion, following terminology is defined to distinguish objects in Lime programs:

- Guarded Object: An object that has either guarded methods or actions, or has both guarded methods and actions.
- Unguarded Object: An object that has neither guarded methods nor actions.
- Active Object: An object that has actions.
- Passive Object: An object that does not have any actions.
- Enabled Object: An active object that has at least one enabled action.
- Disabled Object: An active object that has no enabled actions.
- Blocking: Blocking means that a method of an object is called but the guard of that method is false.

During program execution, there exist several working threads that execute the methods and actions of the objects. An object can only be accessed by one thread at a time, which means that the object needs to be locked when a method or an action of it is being executed by a thread. When a thread calls a method of an object that is locked by another thread, this thread becomes *suspended*. The suspended thread may *resume* the call if the lock is released by another thread later and this thread obtains the lock on the object.

We impose a constraint that one thread can lock at most one object at a time. This means that a thread must release the lock on object A before the thread obtaining the lock on object B. On the other hand, a thread also needs to release the lock on an object when the thread blocks. In this way, other working threads have the chance to call the methods of execute the actions of the object. So that on the exit from a method or action, the blocked thread has to be notified to evaluate the guard again and resume if possible or suspend again.

During the execution of the program, two pools are maintained. One is the *thread pool* and the other is the *object pool*. The thread pool contains all working threads that call the methods of the objects or execute the actions of the objects. The object pool is used to hold the active objects and is initialized to be empty. When an active object is created, a pointer to that object is placed in the object pool. Each active object has an extra boolean attribute *inPool* indicating whether a pointer to it is in the object pool.

When a working thread in the thread pool becomes available, it requests a reference to an active object from object pool. If the selected object is disabled, the thread removes the pointer to this object from the object pool and also resets the *inPool* attribute. If the object is enabled, the thread executes an enabled action and leaves the object in the object pool.

Each working thread locks an object when working on it, i.e., executing one of its methods or actions. The thread unlocks the object when finishing executing the method or action. The lock is also released at a call to another object and obtained again at re-entry from the call.

Fairness among the actions of an object is ensured by evaluating the action guards cyclicly. Fairness among the objects is also ensured by retrieving active objects from the object pool in a cycling fashion.

This translation principle is shown in Figure 6.1 [22]. *Obj1* is a passive object. *Obj2*, *Obj3* and *Obj4* are active objects, thus each has a boolean attribute *inPool*. *Obj2* and *Obj4* are partially enabled objects and *Obj3* is an disabled object. Thin arrows from an object to another object represent references. Thick arrows from a thread to an object represent references with locks.

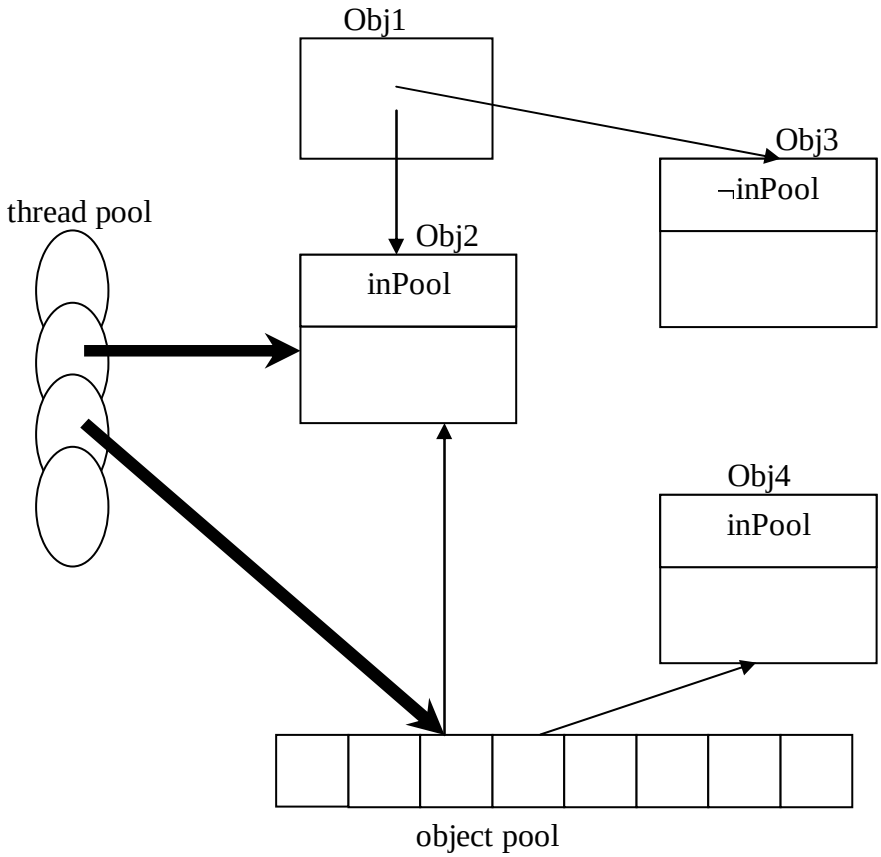

Figure 6.1: The translation principle

When a working thread executes a guarded method of an object, the thread first evaluates the guard of the methods. The thread continues to execute the body of the method if the guard is true; otherwise the thread waits.

All waiting threads are notified to reevaluate the guards when a thread exits from a guarded object. This is because the execution of the method or action of a guarded object may affect the guards of other guarded methods or actions.

With this principle, action guards are only evaluated when a working thread select it from the object pool. Method guards are only re-evaluated when another thread has exited the object and thus possibly affected the guard. The memory overhead is that every active object requires one bit for the *inPool* attribute, one integer for the index to the last evaluated action guard, and one pointer in the object pool.

6.2 Run Time Library

A run time library is created to schedule the selection of active objects and actions. This library includes:

- An interface *ActiveObject* that declares a method *doAction*. All classes of active objects will implement this interface. The method *doAction* is used to handle the selection of actions within the object. *ActiveObject* is implemented as follows:

```
interface ActiveObject
{
    void doAction();
}
```

- Class *ObjectPool* is used to handle the pool for the active objects. This class contains two methods: method *put* puts the pointers to the active objects onto the object pool (indicates that the object is enabled); method *get* gets a pointer to an active object from the pool so that this object is ready for a working thread to execute. The pool is implemented as a dynamic array of active objects.
  - Method *put* is used to put an object into the end of the array. When the array becomes full, a new array with doubled size is created and the original array is copied into the new one;
  - Method *get* is used to get the object from the array. Note that this is done in a cyclic fashion; which action is selected from the array is controlled by the static attribute *curr*.

  The class *ObjectPool* is implemented as follows:

```
class ObjectPool{
    private static int max = 128;
    private static int size = 0;
    private static int curr = 0;
    private static ActiveObject[ ] pool = new ActiveObject[max];
    private static Boolean lock = new Boolean(false);
    public static void put(ActiveObject o){
        synchronized(lock){
            if(size == max){
                max = max * 2;
```

```
            ActiveObject[ ] n = new ActiveObject[max];
            System.arraycopy(pool, 0, n, 0, size);
            pool = n;
        }
        if(size == 0)
            lock.notify();
        pool[size] = o;
        size++;
    }
}
public static synchronized ActiveObject get(){
    synchronized(lock){
        try{
            while(size == 0)
                lock.wait();
        }catch(InterruptedException ie){}
        curr = curr%size;
        final ActiveObject o = pool[curr];
        size--;
        pool[curr] = pool[size];
        pool[size] = null;
        curr++;
        return o;
    }
}
}
```

- Class *ObjectThread* is used to create the running threads for Lime execution. The number of the working threads is implemented as a constant. No more threads will be created even if all threads are suspended. This means that false deadlocks are possible.

  The action threads are implemented as daemon threads, meaning that the whole program is going to terminate when the main program terminates. The implementation of the class is as follows:

```
class ObjectThread extends Thread{
    private static final int T = 3;
    public static void startThreads(){
        for(int i = 0; i < T; i++){
```

```
            ObjectThread ot = new ObjectThread();
            ot.setDaemon(true);
            ot.start();
        }
    }
    public void run(){
        while(true)
            (ObjectPool.get()).doAction();
    }
}
```

6.3 Translation Schemes

New Object Creation

- For the creation of a new passive object, the translation is simply to copy the statement.

- Whenever a new active object is created, a pointer to that object is placed into the object pool. This is done by calling the method *add* of class *ObjectPool*. So the creation of active objects "$x := new\ C$" is translated as follows:
  $x := new\ C$;
  $ObjectPool.add(x)$

Method Call

All method calls in Lime need to be synchronized. So the call statement is put between the two instructions, *monitorenter* and *monitorexit*. Note that these are JVM instructions. They are used in intermediate Lime to mark the beginning and the ending of a synchronized blocks in a Lime program.

A method call "$x.m(e)$" is translated as follows:
$monitorexit(this)$;
$x.m(e)$;
$monitorenter(this)$

Action Declaration

Actions are translated into unguarded methods that return *boolean*. The return value indicates whether this action was enabled and the body of the action was success-

fully executed. If the guard of the action holds and the action's body is executed successfully, the method returns *true*, otherwise it returns *false*.

- Action "*action A when C do S*" is translated as follows:

```
method A : boolean
    monitorenter(this);
        if C then begin
            S;
            return true
        end
        else
            return false
    monitorexit(this)
```

- An always enabled action "*action A do S*" is translated as follows:

```
method A : boolean
    monitorenter(this);
        begin
            S;
            return true
        end
    monitorexit(this)
```

Methods Declaration

For all guarded classes, the execution of the method may affect the guards of actions or guarded methods. So when a method is successfully executed, all threads need to be notified so that the suspended threads can re-evaluate the guards.

For all active objects, when a method is successfully executed, the guards of the actions may be affected. This may change a disabled object to be enabled. So for a disabled object, when a method is executed, the object needs to be put into the object pool.

The body of a guarded method can be executed if the guard holds, otherwise the method is blocked and the thread executing the method gets suspended.

- An unguarded method "*method m(p) MethodBody*" in a passive class is translated as follows:

```
method m(p)
begin
    monitorenter (this);
        MethodBody;
```

```
        notifyAll();
      monitorexit(this)
    end
```

- An unguarded method "*method m (p) MethodBody*" in an active class is translated as follows:

```
method m(p)
begin
    monitorenter(this);
        MethodBody;
        if not inPool then
            begin
            ObjectPool.add(this);
            inPool := true
        end;
        notifyAll();
    monitorexit(this)
end
```

- A guarded method "*method m (p) when C do S*" in a passive class is translated as follows:

```
method m(p)
begin
    monitorenter(this);
        while not C do wait();
        S;
        notifyAll();
    monitorexit(this)
end
```

- A guarded method "*method m (p) when C do S*" in an active class is translated as follows:

```
method m(p)
begin
    monitorenter(this);
        while not C do wait();
        S;
        if not inPool then
        begin
            ObjectPool.add(this);
```

```
            inPool := true
        end;
        notifyAll();
    monitorexit(this)
end
```

Class Declaration

- For passive classes, the translation is just a copy of the class declaration.
- For active class, three attributes, *inPool*, *nextAction* and *nextAction* are added to the intermediate Lime code to handle the selection of actions and objects.

  A class declaration "*class C ClassBody end*" is translated as follows:

```
class C implements ActiveObject
    ClassBody
    attr inPool : boolean
    attr nextAction : integer
    attr numAction : integer
    method doAction
        // ···
end
```

  Method *doAction* is used to select the actions of the object cyclicly. All actions are translated into unguarded methods whose return types are *boolean*. Thus actions in the object $A_1, A_2, \cdots, A_n$ all become methods $A_1, A_2, \cdots, A_n$. The evaluation of the guards is in a cyclic fashion. This is done by implementing a while loop. The loop exits when an action is enabled and successfully executed or all actions are disabled. If an action is executed then the object is put into the object pool again; if no action is enabled then the object is removed from the object pool. The mechanism of handling the actions is shown in Figure 6.2. The method *doAction* that implements this mechanism is inserted into each active class. The code for the *doAction* is shown below. Note that the code is written in Lime since this translation happens when the compiler translate the original Lime program to the intermediate Lime program. Also note that *numAction* is the class attribute that is used to indicate the number of actions in the class. The method *doAction* is implemented as follows:

```
method doAction
begin
    var start : integer;
```

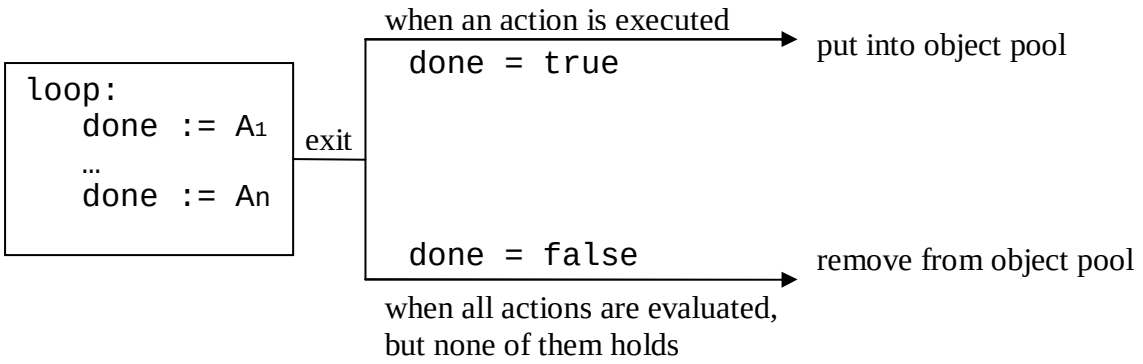

Figure 6.2: The mechanism to handle the selection of actions

```
var done : boolean;
start := nextAction;
done := false;
nextAction := (nextAction + 1) mod numAction
while not done or nextAction <> start do
begin
    if nextAction = 0 then done := A1;
    ...
    if nextAction = n − 1 then done := An;
    nextAction := (nextAction + 1) mod numAction
end; //of the loop
if done then
begin
    notifyAll;
    ObjectPool.put(this)
end
else
    inPool := false
end
```

Note that *doAction* gets called only when the object is partially enabled (*inPool* is true). This scheme supports inheritance of active classes and even overriding of actions. However, if the class C extends an active class, then the declaration of the attributes *inPool* and *nextAction* have to be left out, only *doAction*() has to be generated.

Program Declaration

When a Lime program starts, the threads that are used to execute the actions at background needs to be created and started.

A program declaration "*program P S*" is translated as follows:

program P
begin
  ObjectThread.startThreads();
  S
end

Chapter 7

Implementation of Lime Compiler

The Lime compiler was developed under Mac OS X 10.2, which uses Darwin (originally released in March 1999, a version of the BSD UNIX operating system supporting both Macintosh and UNIX file systems) 6.6 as its core. The JVM for the development is version 1.3.1. Because of the portability of Java, the Lime compiler can run at any other platforms with JVM installed.

7.1 Lime Compiler Overview

Compilation is the process of translating source code into an object program, which is composed of machine instructions along with the data needed by those instructions. The Lime compiler consists of the following four major components, which are also shown in Figure 7.1:

Scanner The scanner reads the characters from source program; groups the characters into lexemes (sequences of characters, each lexeme corresponds to a token); then returns the next token to the parser. The scanner is also called lexical analyzer.

Parser The parser groups tokens according to the grammar, discovers the underlying structure of the source program and finds syntax errors.

Symbol table The symbol table is used for type checking.

Code generator The code generator generates target code from the abstract syntax tree.

The scanner and parser are generated by JavaCC; the other components are written in Java. The details of the implementation are introduced in the following sections.

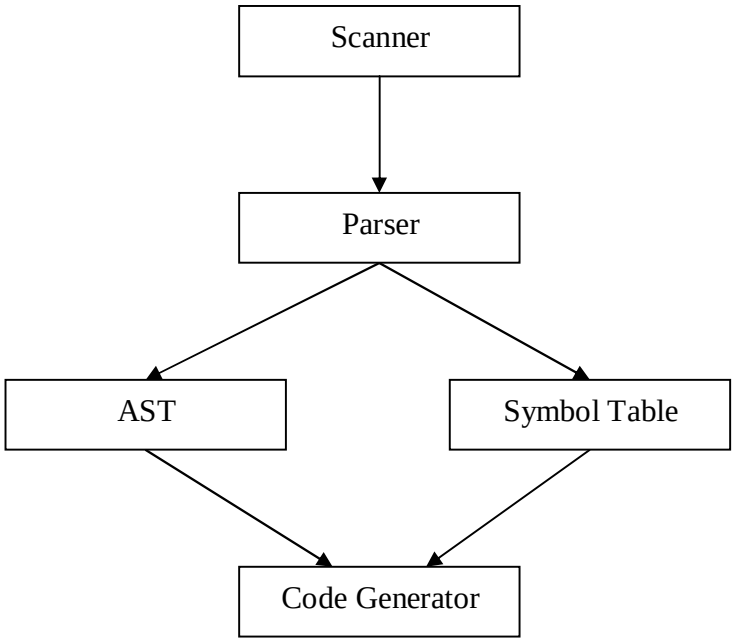

Figure 7.1: Components of the implementation of Lime compiler

7.2 Scanner and Parser

The scanner and parser of Lime compiler are generated using Sun's Java Compiler Compiler – JavaCC. JavaCC is an $LL(k)$ scanner/parser generator for Java, comparable to the well-known LR parser generator yacc [14] for C. LL(k) stands for "left-right, leftmost derivation with k tokens of look-ahead." A grammar is said to be LL if, for each distinct pair of productions with the same left-hand side, there is no confusion about which left-hand side to apply for k symbols of look-ahead.

The input file to JavaCC for generating the scanner and parser for Lime is *LimeParser.jjt*. The structure of *LimeParser.jjt* is shown in Figure 7.2. The file contains four parts: option settings, Java compilation unit, token definitions and grammar production rules. Each of these parts will be gone through in detail in the rest of this section.

LimeParser.jjt starts with the settings for the options offered by JavaCC. Three options are set in this part:

- *MULTI* is set to "*true*" to generate a multi mode parse tree. The default for this is false, generating a simple mode parse tree; in multi mode the type of the parse tree node is derived from the name of the node. Since the implementations for code generation are provided in the node classes, this option needs to be set

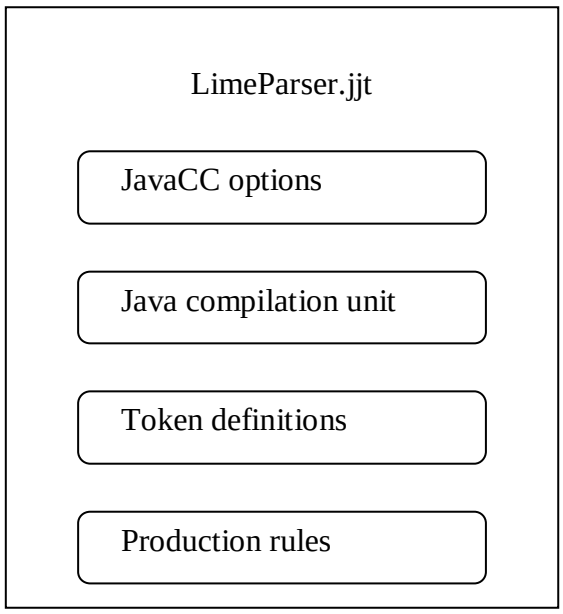

Figure 7.2: The structure of the input file to JavaCC

to "*true*".

- *NODE_SCOPE_HOOK* is set to "*true*" to insert calls to parser methods on entry and exit of every node scope;
- *NODE_PREFIX* is set to "*AST_*" so that all class names constructed for node on abstract syntax tree start with "*AST_*".

```
options{
   NODE_PREFIX = "AST_";
   MULTI = true;
   NODE_SCOPE_HOOK = true;
}
```

The second part of *LimeParser.jjt* is called Java compilation unit and is purely written Java. This part is enclosed between *PARSER_BEGIN*(*LimeParser*) and *PARSER_END*(*LimeParser*), where *LimeParser* is used as the prefix for all generated parser classes and as the name for the generated main class. The parser code that JavaCC generates is inserted immediately before the closing brace of the main Java compilation unit. Following tasks are performed in the compilation unit:

- All global variables are declared and initialized if needed;
- Two methods, *jjtreeOpenNodeScope* and *jjtreeCloseNodeScope*, are defined to open and close each node in the abstract syntax tree respectively (The signatures of these two methods are generated by JavaCC since the option *NODE_SCOPE_HOOK* is set to "*true*".) ;
- Method *WriteHeader* and *WriteOut* are defined to write the generated code into the target file;
- Method *printSymbolTable* is defined to print out the contents of the symbol table on the standard output. This method is mainly used for debug purpose;
- Method *usage* is defined to print out the usage of the compiler;
- Method *process* is the most important method in this class. It opens and reads the input file; creates the output file; determines the compilation mode based on given flag. If everything is fine then the method creates an empty symbol table and starts the compilation .

The implementation of this part is listed as follows:

```
PARSER_BEGIN(LimeParser)
  package lime.compiler.parser;
  import java.io.*;
  import java.util.Vector;
  public class LimeParser{
    // the root of the SymbolTable tree
    private static SymbolTable rootTable;
    public  static PrintStream out = null;
    public static boolean flag = true;
    public static boolean flag1 = true;
    // the SymbolTable of the procedure currently being parsed.
    private static SymbolTable currentScope;
    private static ImportedClassList importClass;
    private static Token expr_token;
    private String fileName;
    public static Vector actionName;
    public static Vector v;
    public static String currentClass;
    public static boolean hasAction = false;
```

```
public static Vector target;
public static String methodReturnType;
//*********************************
public static int ifCount = 0;
public static int andCount = 0;
public static int orCount = 0;
public static int eqCount = 0;
public static int leCount = 0;
public static int ltCount = 0;
public static int geCount = 0;
public static int gtCount = 0;
public static int waitCount = 0;
public static boolean inil = false;
public static int arrayId = 0;
//*********************************
final static void jjtreeOpenNodeScope(Node n)
{
   ((SimpleNode)n).firstToken = getToken(1);
   Token s = getToken(0);
}
final static void jjtreeCloseNodeScope(Node n)
 {
    ((SimpleNode)n).lastToken = getToken(0);
 }
public static void process(String file)
{
   Token tokenList = null;
   LimeParser parser = null;
   InputStream in = null;
   importClass = new ImportedClassList();
   rootTable = new SymbolTable(null,(char)0);
   currentScope = rootTable;
   try
   {
      if (flag)
         out = new PrintStream(new FileOutputStream
               (file.substring(0, file.length() - 10) + ".j"));
      else
          out = new PrintStream
```

```
                (new FileOutputStream(file + ".LIME"));
    }
    catch(Exception e)
     {
        System.out.println(e.getMessage());
        return;
    }
    try
    {
        in = new FileInputStream(file);
        parser = new LimeParser(in);
    }
    catch(java.io.FileNotFoundException e)
    {
        System.out.println("Lime Parser : File" + file + "not found.");
        return;
    }
    try
    {
        TokenList ttt = parser.CompilationUnit(out);
        System.out.println("Lime Parser :
                        Lime program parsed successfully.");
    }
    catch(ParseException e){
        System.out.println("LimeParser : Encountered errors during parse.");
        System.out.println(e.getMessage());
        return;
    }
}//End of process
private void WriteHeader(PrintStream out)
{
    out.println(";This Jasmin file is generated by the Lime compiler");
}
private void WriteOut(PrintStream out,TokenList t)
{
    WriteHeader(out);
    t.printWithSpecials(out);
}
// Prints the SymbolTable. Mostly for debugging.
```

```
    public static void printSymbolTable()
    {
        printSymbolTable(rootTable);
    }
    // Recursively prints the root SymbolTable.
    private static void printSymbolTable(SymbolTable table)
    {
        java.util.Enumeration enum = table.elements();
        while(enum.hasMoreElements())
        {
            SymbolTableEntry s = (SymbolTableEntry)enum.nextElement();
            if(s instanceof ProcedureEntry)
                printSymbolTable(((ProcedureEntry)s).table);
            if(s instanceof ClassEntry)
                printSymbolTable(((ClassEntry)s).table);
            if(s instanceof MethodEntry)
                printSymbolTable(((MethodEntry)s).table);
            System.out.println(s);
        }
    }
    public static void usage()
    {
        System.out.println("Lime Parser : Usage is one of :");
        System.out.println("java LimeParser -IP <inputfile>");
        System.out.println("to generate intermidiate Lime program for Passive class");
        System.out.println("java LimeParser -IA <inputfile>");
        System.out.println("to generate intermidiate Lime program for Active class");
        System.out.println("java LimeParser -J <inputfile>");
        System.out.println("totranslate the intermidiate Lime to Jasmin.");
        System.exit(0);
    }
}//LimeParser
PARSER_END(LimeParser)
```

The Java compilation unit is followed by the list of the definitions of the tokens. There are two kinds of tokens for the Lime compiler:

- TOKEN: regular tokens in the grammar. The token manager creates a *Token* object for each match of such a regular expression and returns it to the parser;

- SPECIAL_TOKENS: Special tokens are like tokens, except that they do not have significance during parsing – that is the EBNF productions will ignore them. Special tokens are still passed on to the parser so that parser actions can access them.

the following list shows how we defined tokens for the Lime compiler:

```
SPECIAL_TOKEN :/* WHITE SPACE */
{
" "
| "\t"
| "\n"
| "\r"
| "\f"
}
SPECIAL_TOKEN : /* COMMENTS */
{
< SINGLE_LINE_COMMENT : "//"(~["\n","\r"])*("\n"|"\r"|"\r\n") >
|  < FORMAL_COMMENT : "/**"(~["*"])*"*"("*"|(~["*","/"](~["*"])*"*"))*"/" >
|  < MULTI_LINE_COMMENT : "/*"(~["*"])*"*"("*"|(~["*","/"](~["*"])*"*"))*"/" >
}
TOKEN : /* RESERVED WORDS AND LITERALS */
{
| < ACTION : "action" >
| < ARRAY : "array" >
| < AND : "and" >
| < ATTRIBUTE : "attr" >
| < BEGIN : "begin" >
| < BOOLEAN : "boolean" >
| < CHAR : "char" >
| < CLASS : "class" >
| < CONST : "const" >
| < DO : "do" >
| < DIV : "div" >
| < ELSE : "else" >
| < END : "end" >
| < EXTEND : "extend" >
| < FALSE : "false" >
```

```
| < IF : "if" >
| < IMPLEMENT : "implement" >
| < IMPORT : "import" >
| < INHERIT : "inherit" >
| < INITIALIZATION : "initialization" >
| < INTEGER : "integer" >
| < MOD : "mod" >
| < NEW : "new" >
| < NOT : "not" >
| < METHOD : "method" >
| < OF : "of" >
| < OR : "or" >
| < PROCEDURE : "procedure" >
| < PROGRAM : "program" >
| < PUBLIC : "public" >
| < REDIFINE : "redefine" >
| < RETURN : "return" >
| < THEN : "then" >
| < TRUE : "true" >
| < VAR : "var" >
| < WHEN : "when" >
| < WHILE : "while" >
}
TOKEN : /* LITERALS */
{
< INTEGER_LITERAL : ["0" - "9"](["0" - "9"])* >
}
TOKEN :/* IDENTIFIERS */
{
< IDENTIFIER :< LETTER > (< LETTER > | < DIGIT >)* >
|
< #LETTER :
[
"\u0024",
"\u0041" - "\u005a",
"\u005f",
"\u0061" - "\u007a",
"\u00c0" - "\u00d6",
"\u00d8" - "\u00f6",
```

```
“ \ u00f8” − “ \ u00ff”,
“ \ u0100” − “ \ u1fff”,
“ \ u3040” − “ \ u318f”,
“ \ u3400” − “ \ u3d2d”,
“ \ u4e00” − “ \ u9fff”,
“ \ uf900” − “ \ ufaff”
]
>
|
< #DIGIT :
[
“ \ u0030” − “ \ u0039”,
“ \ u0660” − “ \ u0669”,
“ \ u06f0” − “ \ u06f9”,
“ \ u0966” − “ \ u096f”,
“ \ u09e6” − “ \ u09ef”,
“ \ u0a66” − “ \ u0a6f”,
“ \ u0ae6” − “ \ u0aef”,
“ \ u0b66” − “ \ u0b6f”,
“ \ u0be7” − “ \ u0bef”,
“ \ u0c66” − “ \ u0c6f”,
“ \ u0ce6” − “ \ u0cef”,
“ \ u0d66” − “ \ u0d6f”,
“ \ u0e50” − “ \ u0e59”,
“ \ u0ed0” − “ \ u0ed9”,
“ \ u1040” − “ \ u1049”
]
>
}
TOKEN :/* SEPARATORS */
{
< LPAREN : “(” >
| < RPAREN : “)” >
| < LBRACE : “{” >
| < RBRACE : “}” >
| < LBRACKET : “[” >
| < RBRACKET : “]” >
| < SEMICOLON : “;” >
| < COMMA : “,” >
```

```
| < DOT : "." >
}
TOKEN :/* OPERATORS */
{
< ASSIGN : ":=" >
| < GT : ">" >
| < LT : "<" >
| < EQ : "=" >
| < LE : "<=" >
| < GE : ">=" >
| < PLUS : "+" >
| < MINUS : "-" >
| < STAR : "*" >
| < SLASH : "/" >
}
```

The next part of the JavaCC file contains all production rules of the Lime grammar. These production rules are expressed in EBNF form. Java code are also applied for non-context free parts of the grammar. The left hand side of a EBNF production is expressed by a Java method declaration. JavaCC generates for each nonterminal an identically named method in the parser class (*LimeParser.java*). Parameters and return values are be declared to pass values up and down the parse tree. The right hand side of the EBNF production rule starts with a set of Java declarations and code, which is generated into the beginning of the method, and thus carried out every time this non-terminal is used. The subsequent expansion unit, or parser actions, of the non-terminals instruct the generated parser on how to make choices. It is enclosed within braces and can consist of any number of Java declarations and code. The expansion unit can also include a local lookahead, specified either as a lookahead limit (to limit the maximum number of tokens of lookahead that may be used for choice determination purposes), a syntactic lookahead (to test the input stream against a regular expressions), or a semantic lookahead (to test the tokens of the input stream with a boolean expression). The following is a piece of program on how the methods *CompilationUnit* and *AdditiveExpression* are defined in JavaCC.

```
//CompilationUnit ::= { Import ";" } { Declaration } EOF
TokenList CompilationUnit(PrintStream outFile) :
    {Token head;  String name;}
    {
        {head = getToken(1);}
```

```
        (Import()";")*
        (Declaration())*
        < EOF >
        {
           if(flag)  jjtThis.interpret(outFile, "");
           else jjtThis.interpretI(outFile);
           return new TokenList(head, getToken(0));
        }
    }
// ...
/* AdditiveExpression ::=MultiplicativeExpression{
"+" MultiplicativeExpression|"-"MultiplicativeExpression}*/
void AdditiveExpression() #void : {}
    {
        MultiplicativeExpression()
        (
        "+" MultiplicativeExpression() #AddNode(2)
        |
        "-" MultiplicativeExpression() #SubtractNode(2)
        )*
    }
```

JavaCC generates the following Java classes for the scanner and parser:

- *TokenMgrError*: a simple error class; it is used for errors detected by the lexical analyzer and is a subclass of *Throwable*;
- *ParseException*: another error class; it is used for errors detected by the parser and is a subclass of *Exception* and hence of *Throwable*;
- *Token*: a class representing tokens. Each *Token* object has an integer field *kind*, representing the kind of the token (PLUS, NUMBER, or EOF) and a String field image, representing the sequence of characters from the input file that the token represents;
- *SimpleCharStream*: an adapter class that delivers characters to the lexical analyzer;
- *AdderConstants*: an interface defining a number of classes used in both the lexical analyzer and the parser;
- *LimeParserTokenManager*: the lexical analyzer;

- *LimeParser*: the parser.

7.3 Symbol Table

The symbol table is one of the core data structures in a compiler. The design of the Lime symbol table focuses on following issues.

The symbol table must support multiple definitions for a given identifier. All symbols that share the same identifier at a particular scope level are contained in the same table. An identifier may be an attribute of a class, a method or an action etc. There may also be multiple instances for a given kind of definition. The symbol table is searchable by identifier type (attribute, method or action, etc.) and it can quickly be determined whether there is more than one definition of a given type (leading to an ambiguous reference). If the object is named, the symbol will have a field that points to the symbol for its parent (e.g, a method or class).

Lime has a large global scope, since all classes and packages are imported into the global name space. Global symbols must be stored in a high capacity data structure that supports fast lookup. We use *hash table* to implement the symbol table since hash table can support a large number of symbols without developing long hash chains.

The symbol table must also deal with the package information correctly and efficiently. Once a package is imported it should not be referenced again. The imported classes are referenced as if they were defined in the current compilation unit (e.g., via simple type names). Package definitions are kept in a separate table. Packages are imported into the global scope of the compilation unit from this table. Package information is live for as long as the main compilation unit is being compiled (e.g., through out the compile process).

The symbol table also needs to support the hierarchical scope of Lime, which is the same as that of Java. So each symbol table needs to contain a pointer to the symbol table in the next scope up.

The symbol table must be searchable by symbol type. The semantic analysis phase knows the context for the symbol it is searching for (e.g., whether the symbol should be a member, method or class). The symbol table hierarchy is searched by identifier and its type.

The symbol table must be able to determine whether a symbol definition is ambiguous quickly and correctly. The report of errors must also be efficient.

The implementation of the symbol table of Lime compiler are grouped into three catalogues:

Symbol Table Entries

The classes for entries of the symbol table includes an abstract class *SymbolTableEntry* and each kind of entry has a class that extends it. The following entries are each represented in a class: *ProgramEntry*, *ClassEntry*, *ProcedureEntry*, *MethodEntry*, *ActionEntry*, *AttributrEntry*, *VariableEntry* and *ConstantEntry* The class hierarchy of the symbol table entries are shown in Figure 7.3.

The implementation of the class *SymbolTableEntry* is shown as follows:

```
package lime.compiler.parser;
public abstract class SymbolTableEntry
{
   //attributes
   String name;
   int type = −1;
   String typeName = "";
   SymbolTable parent;
   String rType, pType;
   //constructor
   public SymbolTableEntry(String n, SymbolTable p)
   {
      this.name = n;
      this.parent = p;
   }
   //get the return type of the entry, if it exists
   public String get_rType()
   {
      return this.rType;
   }
   //get the parameter type, if it exists
   public String get_pType()
   {
      return this.pType;
   }
   //get the name of the current entry
   public String get_name()
   {
      return this.name;
   }
   //get the type of current entry
```

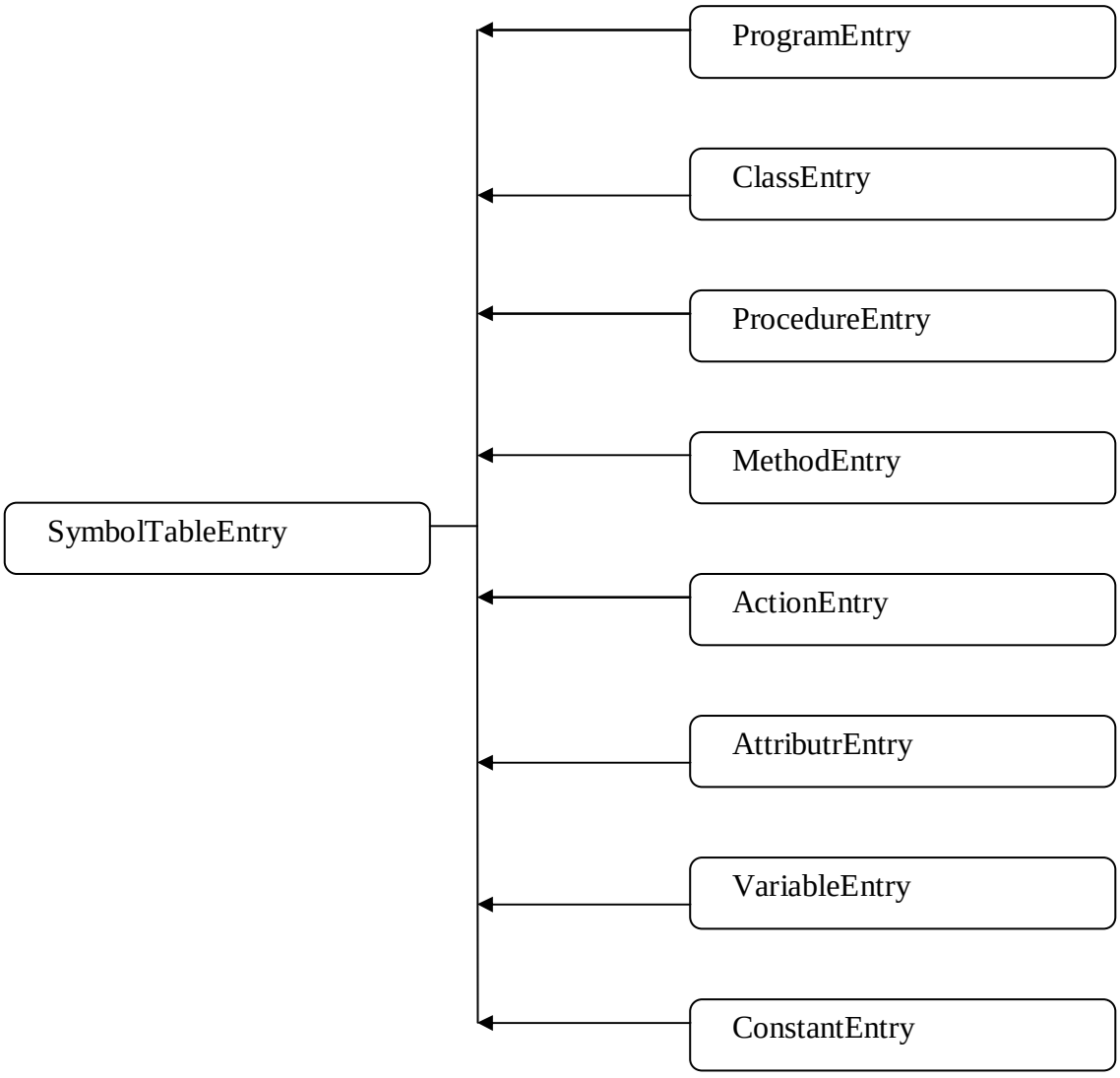

Figure 7.3: The class hierarchy of the symbol table entries

```
    public int get_type()
    {
       return type;
    }
    //get the symbol table that the current entry belongs to
    public SymbolTable get_parent()
    {
       return this.parent;
    }
    //compare two entries
    public boolean equals(SymbolTableEntry entry)
    {
       if (this.name.equals(entry.name)&&
          this.type == ((SymbolTableEntry)entry).type&&
          this.parent == ((SymbolTableEntry)entry).parent)
           return true;
       else
           return false;
    }
    //following two methods are used to convert the current entry to a string,
    //this is mainly for debugging
    public String getLocalizedMessage()
     {
       return (this.getClass().getName() + "[name = " + this.name + "]");
    }
     public String toString()
     {
       return this.getLocalizedMessage();
    }
}
```

Symbol Table

The class *SymbolTable* extends *java.util.Hashtable*. It defines all the attributes and methods for the symbol table.

```
package lime.compiler.parser;
public class SymbolTable extends java.util.Hashtable
{
```

```
SymbolTable parent;
char static_level;
short const_count;
short variable_count;
short procedure_count;
short attribute_count;
short class_count;
short action_count;
short method_count;
//constructor
public SymbolTable(SymbolTable p, char l)
{
   //call java.util.Hashtable's constructor
   super(16);
   //initialize data fields
   this.parent = p;
   this.static_level = l;
   this.variable_count = 0;
   this.procedure_count = 0;
   this.attribute_count = 0;
   this.class_count = 0;
   this.action_count = 0;
   this.method_count = 0;
   this.const_count = 0;
}
//set the parent table for the current table
public void set_parent(SymbolTable p)
{
   this.parent = p;
}
//get the parent table for the current table
public SymbolTable get_parent()
{
   return this.parent;
}
//get static level of the current table from root table
public char get_static_level()
{
   return this.static_level;
```

```
}
//get the number of variables in the current table
public short get_variable_count()
{
   return this.variable_count;
}
//find if the table has entry with given key
public synchronized boolean has(String key)
{
//prepare to search up table
   SymbolTable table = null;
   for(table = this;table! = null && !table.containsKey(key);
   table = table.parent);
   //if not found
   if(table == null)
      return false;
   //if found – get using Hashtable's get method
   else
      return true;
}
//get the entry from the table by given key
public synchronized SymbolTableEntry get(String key)
      throws TypeErrorException
{
   //prepare to search up table
   SymbolTable table = null;
   for(table = this;table! = null && !table.containsKey(key);
   table = table.parent);
   //if not found
   if(table == null)
   {
      TypeErrorException e = new TypeErrorException("Identifier '"
            +key + "' was never declared",0);
      e.fillInStackTrace();
      throw e;
   }
   //if found – get using Hashtable's get method
   else
      return (SymbolTableEntry)((java.util.Hashtable)table).get(key);
```

```
}
//test whether two entries are of the same type
public boolean sameType(SymbolTableEntry s1, SymbolTableEntry s2)
{
   if ((s1 instanceof VariableEntry)&&(s2 instanceof VariableEntry))
      return true;
   if ((s1 instanceof ConstantEntry)&&(s2 instanceof ConstantEntry))
      return true;
   if ((s1 instanceof AttributeEntry)&&(s2 instanceof AttributeEntry))
      return true;
   if ((s1 instanceof ClassEntry)&&(s2 instanceof ClassEntry))
      return true;
   if ((s1 instanceof ActionEntry)&&(s2 instanceof ActionEntry))
      return true;
   if ((s1 instanceof MethodEntry)&&(s2 instanceof MethodEntry))
      return true;
   if ((s1 instanceof ProcedureEntry)&&(s2 instanceof ProcedureEntry))
      return true;
   return false;
}
// Puts a new entry in the SymbolTable.
public synchronized SymbolTableEntry put(String key, SymbolTableEntry value)
      throws TypeErrorException
{
   //if key is already in table
   if((this.containsKey((value.get_key())))&&(sameType(this.get(key), value)))
   {
      TypeErrorException e = new TypeErrorException("Identifier '"
            +key + "' was already declared", 0);
      e.fillInStackTrace();
      throw e;
   }
   //if not in table
   else
   {
      if (value instanceof VariableEntry)
         this.variable_count++;
      if (value instanceof ClassEntry)
         this.class_count++;
```

```
        if (value instanceof AttributeEntry)
            this.attribute_count++;
        if (value instanceof ConstantEntry)
            this.const_count++;
        if (value instanceof ActionEntry)
            this.action_count++;
        if (value instanceof MethodEntry)
            this.method_count++;
        if (value instanceof ProcedureEntry)
            this.procedure_count++;
        return (SymbolTableEntry)super.put(key, value);
      }
   }
   public synchronized SymbolTableEntry remove(String key)
   {
        return (SymbolTableEntry)super.remove(key);
   }
   public synchronized String toString()
   {
      return ("@" + Integer.toHexString(this.hashCode())+
          ", variable_count = " + this.variable_count+
          "\nClass_count" + this.class_count+
          "\nAttr_count" + this.attribute_count+
          "\nConst_count" + this.const_count+
          "\nAction_count" + this.action_count);
   }
}
```

Type Checking

A new symbol table, *rootTable* is created when the compilation starts. Once a new class, method, procedure or program is encountered, a new symbol table is also created for it. The type checking is performed by going through the chain of the symbol tables to look for the symbol. If the symbol is not found, an error is reported.

The classes used to handle type checking errors, including *CompilerException* that extends *Exception* and *InvalidNumberException*, *TypeErrorException* and *UnknownSymbolException*, which all extend *CompilerException*. The class hierarchy for type checking exceptions is shown in Figure 7.4.

The implementation of class *CompilerException* is shown as follows:

```
package lime.compiler.parser;
public class CompilerException extends Exception
{
   protected String error;
   protected int line_number;
   public CompilerException(String message, String error, int line_number)
   {
    //construct Exception
      super(message);
      this.error = error;
      this.line_number = line_number;
   }
   public String get_error()
   {
      return this.error;
   }
   public int get_line_number()
   {
      return this.line_number;
   }
   public String getLocalizedMessage()
   {
      String s = super.getMessage() + System.getProperty("line.separator");
      s+ = "Error : \"" + this.get_error() + " \ ", online"+
               this.get_line_number();
      return s;
   }
}
```

7.4 Code Generation

An add-on to JavaCC, JJTree, is a pre-processor for JavaCC that inserts abstract syntax tree building actions into the JavaCC source code.

JJTree first generates an interface *Node* and an abstract class *SimpleNode* implementing *Node*, which declares and defines some important methods for the abstract syntax tree. We add the following three methods to it for code generation.

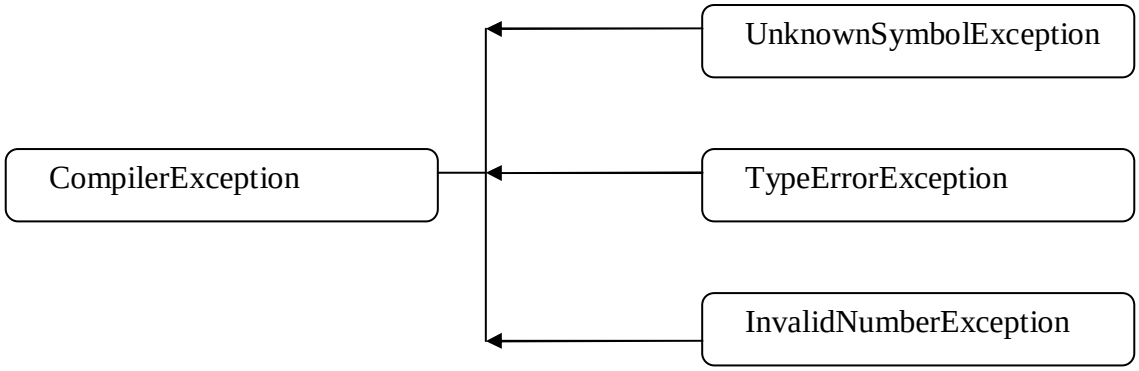

Figure 7.4: The class hierarchy of type checking exceptions

- *interpret*(*PrintStream o*, *String s*): used to translate the current node into Jasmin, write result to output file;
- *interpretI*(*PrintStream o*): used to translate node into Lime without actions and guarded methods, write result to output file;
- *print*(*String s*, *PrintStream ostr*): used to write string *s* to output file.

By default, JJTree generates a class extending *SimpleNode* for each non-terminal symbol in the grammar. The 56 node classes generated are:

AST_Action.java
AST_AddNode.java
AST_AndNode.java
AST_ArrayType.java
AST_IntConstNode.java
AST_Assert.java
AST_Assignment.java
AST_Attrbute.java
AST_Method.java
AST_ClassDeclaration.java
AST_CompilationUnit.java
AST_Compound.java
AST_ConstDeclaration.java
AST_DivNode.java
AST_EQNode.java
AST_Expression
AST_FalseNode.java
AST_ModNode.java
AST_MulNode.java
AST_LTNode.java
AST_NewArray.java
AST_NewObj.java
AST_NotNode.java
AST_While.java
AST_OrNode.java
AST_LENode.java
AST_ProcedureDeclaration.java
AST_Initialization.java
AST_ProgramDeclaration.java
AST_QualifiedIdentifier.java
AST_ResultParameter.java
AST_Return.java
AST_Statement.java
AST_SubtractNode.java

AST_Feature.java	AST_Synchronize.java
AST_GENode.java	AST_TrueNode.java
AST_GTNode.java	AST_Type.java
AST_Ident.java	AST_TypedIdentifierList.java
AST_IdentifierList.java	AST_ValueParameter.java
AST_If.java	AST_Var.java
AST_Import.java	AST_Wait.java
AST_Inhert.java	AST_When.java

The code generation of our compiler is straight forward. We go through the each node in the abstract syntax tree, implement the three methods mentioned above to perform the translation schemes, which are introduced in the previous two chapters. The following shows the implementation of such node, AST_LENode.

```
package lime.compiler.parser;
import java.io.*;
public class AST_LENode extends SimpleNode{
   int count = 0;
   publicAST_LENode(int id){
      super(id);
    }
   public AST_LENode(LimeParser p, int id){
      super(p, id);
   }
   public void interpret(PrintStream o, String s)       {
      String label_1 = "L_LE_" + Integer.toString(count) + "_1";
      String label_2 = "L_LE_" + Integer.toString(count) + "_2";
      int i, k = jjtGetNumChildren();
      for(i = 0; i < k; i++)
         jjtGetChild(i).interpret(o, s);
      print("    if_icmple" + label_1, o);
      print("    iconst_0", o);
      print("    goto" + label_2, o);
      print(label_1 + " : ", o);
      print("    iconst_1", o);
      print(label_2 + " : ", o);
    }
   public void interpretI(PrintStream o);
   {
       // nothing needs to do for current node
```

```
    }
}
```

Finally, a class *LimeC* is used to get information from the command line and to invoke the compiler. The code follows:

```
import lime.compiler.parser.*;
public class LimeC{
public static void main(String args[])
{
    if (args.length == 2)
    {
        System.out.println("Lime Parser :  Reading from file " + args[1] + "...");
        if (args[0].equals("-J"))
            LimeParser.flag = true;
        else{
            LimeParser.flag = false;
            if (args[0].equals("-IA"))
                LimeParser.flag1 = true;
            else
            {
                if (args[0].equals("-IP"))
                    LimeParser.flag1 = false;
                else
                    LimeParser.usage();
            }
        }
        LimeParser.process(args[1]);
    }else
        LimeParser.usage();
    } // end of main
}//end of LimeC
```

7.5 Testing the Implementation

Testing is a very important concept in software engineering. The testing for the Lime compiler has two phases, black-box testing and white-box testing. Both of these phases are done manually.

White-box testing is used to test what is happening on the inside the program. White-box testing is done during the implementation by unit testing of individual

compiler modules.

Black-box testing is used to test the functional requirements of the compiler. The black-box testing for the Lime compiler includes two phases. The first is to test the translation schemes of that are introduced in the previous two chapters; the other one is to test the compiler itself.

To test the translation schemes, we need to check the correctness of generated code - the the correctness of the generated Jasmin code. A set of Lime programs in various areas are fully tested with the compiler. The programs need to be modified to print out some execution information for the test purpose. For example, when the tests are focused on the scheduling of actions and objects, the test programs need to print out which thread is working, which object is is picked up by the thread, which action is chosen to execution, etc.

To test the compiler, we mainly focus on testing the parser. This includes testing error messages of syntactically incorrect programs and error messages of type incorrect programs. A set of Lime programs with different syntax/type errors are used as test cases.

7.6 Running the Compiler

The Lime compiler is packed as a JAR file, *LimeC.jar*. To install it, the Java Archive Tool provided as part of the Java Development Kit is needed. The installation takes following three steps:

- Make a directory for the compiler, for example, "*/Users/abc/Lime*";
- Move *LimeC.jar* to the destination directory;
- Extract *LimeC.jar* by typing

  jar x LimeC.jar.

Before starting to compile a Lime program, the installation path of the compiler needs to be added to the Java class path. For the above installation example, the class path should be set by

setenv CLASSPATH .:/Users/abc/Lime.

To compile a Lime program, *A.lime*, into a Java class file, following three steps are needed:

- Firstly, compile it into intermediate Lime program, that is, a Lime program without actions and guarded methods.

  - For passive class, the option *IP* is used. For example,
    java LimeC -IP A.lime;
  - For active class, the option *IA* is used. For example,
    java LimeC -IA A.lime.

  This will generate an intermediate Lime program, *A.lime.LIME*.

- Secondly, compile the intermediate Lime program into Jasmin. The option *J* is used in this step. For example,

  java LimeC -J A.lime.LIME.

  This will generate a Jasmin file, *A.j*.

- Finally, use Jasmin assembler to get the Java class file by executing

  jasmin A.j.

  This will generate the final target file, *A.class*.

Chapter 8

Conclusion and Future Work

We present a new object-oriented programming language, Lime, which is based on action systems in this thesis. A technique is developed to schedule the selection of objects and actions. A compiler is implemented to test the technique.

The implementation of the compiler and the library contains following three parts:

- A JavaCC file to generate Lime parser;
- Classes for type checking;
- Classes of abstract syntax trees for code generation.

The implementation contains one JavaCC file and 63 Java classes, with altogether about hand-written 4500 lines.

We did several experiments to implement some traditional problems in concurrent programming (in Chapter 3). It can be seen that it is much easier and more convenient to solve these problem with Lime (i.e. examples presented in Chapter 3). Programmers do not need to deal with the details the multi-threaded programming. Compiler takes care of that. But all our examples are simple because of the short time. More complex examples, especially examples with object-orientation, are need to test both the language and the compiler.

Lime itself needs further development to make it more complete and useful. One of the most important things is to make it more object-oriented. For example, the supports of more data types, the supportive of exception handling, the supports of assertions, etc. need to be added.

The translation schemes need more development. Currently we maintain two pools, one to hold all executing threads and the other for all active objects. Fairness among the actions within an object and among all active objects is ensured in a cyclic fashion. The compiler does not do any optimizations like eliminating synchronization

statements when not needed and detecting which guards do not need re-evaluation after specific exits.

The current implementation of the compiler mainly focuses on the functionality of Lime. More work needs to be done on the compiler. For example, although inheritance and assertions are enabled in the language grammar, they are just ignored by the compiler during the compilation. The symbol table and type checking need to be improved so that they can support object orientation.

The test of the compiler is done manually. Some tools need to be involved for the testing so that the tool can run a list of scenarios and report useful result. This will save a lot of time on the testing.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.

[2] Ralph Back, Martin Büchi, and Emil Sekerinski. Action-based concurrency and synchronization for objects. In M. Bertran and T. Reus, editors, *Proceedings of the Fourth AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software (ARTS)*, volume 1231 of *Lecture Notes in Computer Science*, pages 248–262. Springer Verlag, May 1997.

[3] David M. Beazley. Advanced python programming. *OReilly Open Source Conference*, 2000.

[4] Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. An approach to object-orientation in action systems. *Lecture Notes in Computer Science*, 1422:68+, 1998.

[5] Alan Burns and Andy Wellings. *Concurrency in Ada*. Number ISBN: 0-521-62911-X. Cambridge University Press, second edition, Nov 1997.

[6] Eric Buttow and Tommy Ryan. *C# – Your visual blueprint for building .NET applications*. Hungry Minds Inc., 2002.

[7] with Chad Fowler Dave Thomas and Andy Hunt. *Programming Ruby, The Pragmatic Programmer's Guide, Second Edition*. The Pragmatic Programmers, LLC, 2004.

[8] Stephan Diehl. Translation schemes for tasskaf. 1997.

[9] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *J-CACM*, 8(9), September 1965.

[10] Pascal Manoury Emmanuel Chailloux and Bruno Pagano. *Developing Applications with Objective CAML*. O'Reilly France.

[11] A. Nico Habermann and Dewayne E. Perry. *Ada for Experienced Programmers*. Addison-Wesley Publishing, USA, 1983.

[12] Guy Steele Gilad Bracha James Gosling, Bill Joy. *The Java Language Specification, Second Edition*. Sun Microsystems, Inc., 2000.

[13] Kurki-Suonio R Jarvinen H-M. The disco language and temporal logic of actions, 1990.

[14] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[15] Cliff B. Jones. An object-based design method for concurrent programs. Report 1992-12-04, University of Manchester, Department of Computer Science, 1992. ftp://m1.cs.man.ac.uk/pub/TR.

[16] Ingolf Heiko Krüger. Master thesis: An experiment in compiler design for a concurrent object-based programming language, the university of texas at austin. 1996.

[17] Xavier Leroy. The objective caml system (release 3.00) documentation and user's manual. 2000.

[18] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification, Second Edition*. Sun Microsystems, Inc., 1998.

[19] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly & Associates, Inc., 1997.

[20] Jayadev Misra. A discipline of multiprogramming: Programming theory for distributed applications. 2001.

[21] Michael Papathomas and Anders Andersen. Concurrent object-oriented programming in Python with ATOM. In *Proceedings of the 6th International Python Conference*, pages 77–87, San Jose, Ca., October 1997.

[22] Emil Sekerinski. Concurrent object-oriented programs: From specification to code. In *Proceedings of the First International Symposium on Formal Methods for Components and Objects, FMCO 02*, Lecture Notes in Computer Science. Springer Verlag, November 2003.

[23] Emil Sekerinski. A simple model for concurrent object-oriented programming. *International Conference Internet, Processing, Systems, Interdisciplinaries, IPSI 2003, Sveti Stefan, Montenegro, IEEE*, 8(9), 2003.

[24] Peter Wegner. Dimensions of object-based language design. In *Proc. of the OOPSLA-87: Conference on Object-Oriented Programming Systems*, pages 168–182, Languages and Applications, Orlando, FL, 1987.