12-2011

ProcessJ: A process-oriented programming language

Matthew Sowders
University of Nevada, Las Vegas

PROCESSJ: A PROCESS-ORIENTED PROGRAMMING LANGUAGE

by

Matthew Sowders

Bachelor of Science (B.Sc.)
University of Nevada, Las Vegas
2009

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science Degree in Computer Science

School of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
December 2011

THE GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Matthew Sowders

entitled

ProcessJ: A Process-Oriented Programming Language

be accepted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
School of Computer Science

Jan Pederson, Committee Chair

Laxmi Gewali, Committee Member

Evangelos Yfantis, Committee Member

Aly Said, Graduate College Representative

Ronald Smith, Ph. D., Vice President for Research and Graduate Studies
and Dean of the Graduate College

December 2011

Abstract

Java is a general purpose object-oriented programming language that has been widely adopted. Because of its high adoption rate and its lineage as a C-style language, its syntax is familiar to many programmers. The downside is that Java is not natively concurrent. Volumes have been written about concurrent programming in Java; however, concurrent programming is difficult to reason about within an object-oriented paradigm and so is difficult to get right.

occam-π is a general purpose process-oriented programming language. Concurrency is part of the theoretical underpinnings of the language. Concurrency is simple to reason about within an occam-π application because there is never any shared state; also occam-π is based on a process calculus, with algebraic laws for composing processes. It has well-defined semantics regarding how processes interact. The downside is that the syntax is foreign and even archaic to programmers who are used to the Java syntax.

This thesis presents a new language, ProcessJ, which is a general purpose, process-oriented programming language meant to bridge the gap between Java and occam-π. ProcessJ does this by combining the familiar syntax of Java with the process semantics of occam-π. This allows for a familiar-looking language that is easy to reason about in concurrent programs.

This thesis describes the ProcessJ language, as well as the implementation of a compiler that translates ProcessJ source code to Java with Java Communicating Sequential Processes (JCSP), a library that provides CSP-style communication primitives.

Acknowledgements

Many thanks to Dr. Pedersen for challenging me and widening my view of programming languages. Dr. Pedersen was the original developer and designer of ProcessJ and together we redesigned some parts of the language and re-implemented the compiler.

Thank you to my wife who supported me while I was working on this large undertaking. Without the help of my wife Randi, I would never have completed the compiler or my thesis.

Finally, I need to thank my mother and father. My mother has been a constant source of encouragement and my father who convinced me to be an engineer who loves to play the bass. I have made it half way.

Matthew Sowders

University of Nevada, Las Vegas
December 2011

Contents

List of Figures

Chapter 1

Introduction

As multiprocessor and multi-core computers become prevalent, and as computers get cheaper and easier to network, the total number of processors available to the average computer program increases. This drive toward more and more processors, combined with a clock rate that is no longer increasing, requires modern applications to use concurrent programming to perform well on modern computers.

Concurrent programming is difficult. For instance, the 'threads and locks' approach to concurrent programming results in numerous problems that cause bugs. Taking too few locks, or too many locks, using the wrong locks, using locks in the wrong order, etc., lead to bugs that are difficult to find. However, concurrent programming does not have to be difficult.

For instance, Sir Tony Hoare created a process algebra known as Communicating Sequential Processes (CSP). Through this work, he defined a mathematical basis for concurrent programming that allows developers to reason about concurrent programs. It provides clearly defined semantics for executing multiple processes in parallel as well as, their communication and composition.

It is through a mathematical basis of concurrent programming that modern applications will be able to make best use of a growing number of resources while maintaining a code base that is understandable.

With this in mind, this paper describes a new general-purpose, process-oriented programming language called ProcessJ. To start, Chapter 2 will discuss the background of concurrent programming through the use of threads and locks as well as message passing. Chapter 2 will then introduce the ProcessJ language and explain both the design decisions and how it differs from its common ancestors. After the introductory material, chapter 3 will explain the project, how it is organized, documented, and built to ensure easy on-boarding for new developers. Next, chapter 4 will discuss the design and implementation of the compiler and finally chapter 5 will discuss the design patterns used in the implementation.

Chapter 2

Concurrent Programming

Concurrent programs execute in an environment that, "...provide(s) mappings from apparent simultaneity to physical parallelism (via multiple CPU's), or lack thereof, by allowing independent activities to proceed in parallel when possible and desirable, and otherwise by time-sharing" [Lea99]. In other words, concurrent programs are able to execute code at the same time.

Concurrent Programming has become more important in the software development industry since the clock rate of processors has leveled off, and as the number of processors / cores has increased. It has become increasingly important for software developers to take advantage of this increased physical parallelism. However, there are issues that come with concurrent programming, as listed in Clean Code by Martin [Mar08]:

- *Concurrency incurs some overhead*, both in performance as well as writing additional code.
- *Correct concurrency is complex*, even for simple problems.
- *Concurrency bugs aren't usually repeatable*, so they are often ignored as one-offs instead of the true defects they are.
- *Concurrency often requires a fundamental change in design strategy.*

As you read about the existing approaches to concurrent programming, and the new ProcessJ programming language, think of the above issues. Think about overhead, and how compilers have been able to reduce code overhead and increase efficiency since their inception. Also think about complexity; read each example and compare them to each other in how understandable and intuitive they are. Think of how each approach determines or avoids bugs, and finally consider which approaches require the greatest mental stretch during design of an application.

2.1 Approaches

To give some history on the design decisions in developing ProcessJ, consider two approaches to concurrent programming: shared memory and message passing. Since each topic is weighty in its own right, this will be a brief cursory introduction.

2.1.1 Threads and Locks

Goetz [GPB^{+}06]: writes of threads, as follows:

> Threads are an inescapable feature of the Java language, and they can simplify the development of complex systems by turning complicated asynchronous code into simpler straight-line code. In addition, threads are the easiest way to tap the computing power of multiprocessor systems. And, as processor counts increase, exploiting concurrency effectively will only become more important.

Java achieves concurrency by means of threads. According to Lea [Lea99], a thread is “a call sequence that executes independently of others, while at the same time possibly sharing underlying system resources such as files, as well as accessing other objects constructed within the same program.” A Java program must explicitly define threads through the use of the `Thread` class.

In each approach to concurrency, there is an example from the producer/consumer problem. The most basic producer/consumer problem has two threads of control: a producer and a consumer. The producer publishes some value that the consumer then processes. Figure 2.1 illustrates a Java implementation of a producer. The producer takes in a buffer and publishes new values until the program closes.

Since the producer needs to run in its own thread of control, the `Producer` class implements `Runnable`. The `Runnable` interface allows Java threads to execute a command. It is considered good practice to implement `Runnable` rather than extend the `Thread` class.

```
public class Producer implements Runnable {

    private Buffer out;

    public Producer(Buffer out) {
        this.out = out;
    }

    public void run() {
        int x = 0;
        while (true) {
            x++;
            out.produce(x);
        }
    }

}
```

Figure 2.1: Java Producer.

Figure 2.2 illustrates a Java implementation of a consumer. The consumer takes a buffer and processes the values it receives until the program closes. Once again, the `Consumer` class implements `Runnable` because it needs to run in its own thread of control. Like the producer, the consumer is simple.

Now that the producer and consumer are established, there needs to be a way to link them together. The driver is responsible for instantiating the producer and consumer then starting their respective threads. After each has started, the driver needs to wait until both complete. If the driver does not wait for them to complete, the main thread of control completes and the application closes.

Finally, the buffer, shown in Figure 2.4, provides communication between the producer and consumer. The buffer is where all the shared memory synchronization happens. Normally, a `SynchronousQueue` can be used to provide synchronous communication between two threads; however,for the purposes of demonstration the Java locking mechanism is used.

The buffer, shown in Figure 2.4, has two methods: `produce` and `consume`. The `produce` method is synchronized on the buffer, which means that only one thread is allowed in to the method for any given `Buffer` object. The value is set then calls `notify` which informs all threads waiting on this buffer that something has changed. After notifying the other threads, the buffer waits for the value to become null again. In other words, the `produce`

```
public class Consumer implements Runnable {

    private Buffer in;

    public Consumer(Buffer in) {
        this.in = in;
    }

    public void run() {
        while (true) {
            int x = in.consume();
            System.out.println(x);
        }
    }
}
```

Figure 2.2: Java Consumer.

```
public class Driver implements Runnable {

    public static void main(String[] args){
        Driver driver = new Driver();
        driver.run();
    }

    public void run() {
        Buffer comm = new Buffer();
        Producer producer = new Producer(comm);
        Consumer consumer = new Consumer(comm);

        Thread producerThread = new Thread(producer,”Producer”);
        Thread consumerThread = new Thread(consumer,”Consumer”);

        consumerThread.start();
        producerThread.start();

        try {
            producerThread.join();
            consumerThread.join();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Figure 2.3: Java Driver.

method is always called from the producer thread which needs to wait for the consumer to process the latest value before producing more.

To go into the waiting mode, the buffer calls `wait`. Waiting has the effect of putting the thread into a state where it does not idle with control of the processor. Instead, it goes into a waiting state until it is woken up by a call to `notify`.

When `notifiy` is called, the thread is removed from the waiting queue; however, that does not mean that it is run immediately. Something could happen between the time the thread is removed from the waiting queue and when it acquires control of the CPU that causes the condition to become false. A conditional loop, otherwise known as a guarded wait, always surrounds a call to `wait` so that the resuming thread knows the condition it was waiting for has been met. It also may be possible for spurious wakeups to occur, where the system removes a thread from the wait queue without an explicit call to `notify`.

In the `consume` method, the process is reversed from the `produce` method. The consumer waits until there is a value to process; it processes the value, then waits until ready again.

Goetz [GPB$^+$06] writes:

> At this writing, multi-core processors are just now becoming inexpensive enough for mid-range desktop systems. Not coincidentally, many development teams are noticing more and more threading-related bug reports in their projects. In a recent post on the NetBeans developer site, one of the core maintainers observed that a single class had been patched over 14 times to fix threading-related problems. Dion Almaer, former editor of TheServerSide, recently logged (after a painful debugging session that ultimately revealed a threading bug) that most Java programs are so rife with concurrency bugs that they work only "by accident".

It takes considerable thought to grok the shared memory model. It may be better to consider a new model, as suggested by Goetz [GPB$^+$06], "One of the challenges of developing concurrent programs in Java is the mismatch between the concurrency features offered by the platform and how developers need to think about concurrency in their programs."

As if any further proof of the problems inherent in the threads and locks model of concurrent programming were necessary, Simon Peyton Jones [AO07] writes: "Locks are bad."

- *Taking too few locks* - It is easy to forget to take a lock and thereby end up with two

```
public class Buffer {
    private volatile Integer x;

    public synchronized void produce(int value) {
        x = value;
        notify();
        try {
            while(null != x){
                wait();
            }
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    public synchronized int consume() {
        int result;
        try {
            while (null == x) {
                wait();
            }
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        result = x;
        x = null;
        notify();
        return result;
    }
}
```

Figure 2.4: Java Buffer.

threads that modify the same variable simultaneously.

- *Taking too many locks* - It is easy to take too many locks and thereby inhibit concurrency (at best) or cause deadlock (at worst).

- *Taking the wrong locks* - In lock-based programming, the connection between a lock and the data it protects often exists only in the mind of the programmer and is not explicit in the program. As a result, it is all too easy to take or hold the wrong locks.

- *Taking locks in the wrong order* - In lock-based programming, one must be careful to take locks in the "right" order. Avoiding the deadlock that can otherwise occur is always tiresome and error-prone, and sometimes extremely difficult.

- *Error recovery* - Error recovery can be very hard because the programmer must guarantee that no error can leave the system in a state that is inconsistent, or in which locks are held indefinitely.

- *Lost wake-ups and erroneous retries* - It is easy to forget to signal a condition variable on which a thread is waiting, or to retest a condition after a wake-up.

To sum up the consensus regarding the threads and locks model: programming within this model is difficult, locks are bad, and programs seem to only work by accident. In other words, programmers need to change the way they think about concurrent programming. In researching the literature, there was not a single quote that gave a whole hearted approval of this method. Most articles and books took a serious and austere tone with the subject as if it is not something that normal people are able to do and so, if you want to learn it, you better pay attention.

The austerity is not without cause. Consider what happened to the Mars rovers, Opportunity and Spirit, 56 million km away and having problems caused by race conditions [MD06]. The two rovers cost roughly $820 million dollars combined. Luckily they recovered, but imagine if the errors had been more serious. The software for these rovers was developed by some of the most brilliant minds, and no one caught the bug before deployment. As mentioned previously, what is required is a fundamental change in design strategy [Mar08].

2.1.2 Communicating Sequential Processes

In the foreword of Sir Anthony Hoare's book, Communicating Sequential Processes [Hoa85], Dijkstra writes:

> To say or feel that the computing scientist's main challenge is not to get confused by the complexities of his own making is one thing; it is quite a different matter to discover and show how a strict adherence to the tangible and quite explicit elegance of a few mathematical laws can achieve this lofty goal.

Communicating Sequential Processes (CSP) is a formal approach to concurrency and an associated set of design techniques [Lea99]. CSP specifies a language in which processes are defined in terms of an alphabet of events and specifies how processes interact with each other.

From the language specification, CSP declares laws that define how processes behave. The CSP language is composed of events, processes, channels, and operators that define behavior of processes.

An event is something that happens to an object at a particular atomic instant of time. There are event classes, much like a Java class, that are a 'type' of event. In addition, there are also event instances at which a certain event class happens at a certain time. For instance, a car crash is an event class, and the car crash that happened yesterday was an instance of that event class.

Consider a change machine. This change machine interacts with the following event classes:

- `dollarIn` The insertion of a dollar into the machine.
- `coinIn` The insertion of a dollar coin into the machine.
- `tokenOut` The result of a token from the machine.

Processes respond to an alphabet of events, or the set of events to which it is defined to respond. They define the behavior of a system. In CSP, processes are composable; they combine and connect to each other to create process networks. Consider a broken change machine, one that takes in a dollar and stops. Figure 2.5, read `dollarIn` then `STOP`, displays

```
BROKENCHANGE = ( dollarIn → STOP )
```

Figure 2.5: A Broken Change Machine.

```
THEIFCHANGE = ( dollarIn → THEIFCHANGE )

THEIFCHANGE = ( dollarIn → ( dollarIn → THEIFCHANGE ) )
```

Figure 2.6: A Change Machine That Steals Money.

a CSP process that first engages in `dollarIn` then behaves as the process `STOP`. In other words, the machine takes a dollar and then stops working.

Of course, a machine that takes one dollar and stops working is of little use to anyone. Consider instead, a change machine built by an evil genius that will take dollars forever, but will never output any change. Figure 2.6 displays a description of `THEIFCHANGE`, a recursively defined process that engages in the `dollarIn` event, then continues to act as `THEIFCHANGE`.

Although this machine is of great use to its evil creator, no one else would consider it better than the broken machine. Figure 2.7 displays a process that can either take in a dollar or a coin, then output a token. The example is read, "`dollarIn` then `tokenOut` then `CHANGE` *choice* `coinIn` then `tokenOut` then `CHANGE`." The choice operator allows a process to decide how to act, depending on the first event with which the process engages.

Suppose we need to convert coins to tokens, or tokens to coins, but we only want to create a single machine. With this machine, the first event it receives determines how it will act. If at first it receives a token, it will take in tokens and output coins; likewise, if at first it receives a coin, it will convert coins to tokens. Figure 2.8 displays how to compose a process out of other processes. In this example, `INIT` determines how the process will work after engaging with its first event.

In Figure 2.7, the process could take either a dollar or a coin and output a token. Consider a real world change machine with a dollar input slot, and a coin input slot. The

```
CHANGE = ( dollarIn → tokenOut → CHANGE | coinIn → tokenOut → CHANGE
    )
```

Figure 2.7: Example of Choice.

```
INIT = ( coinIn → CONVERTCOIN | tokenIn → CONVERTTOKEN )

CONVERTCOIN = (  tokenOut → coinIn → CONVERTCOIN )

CONVERTTOKEN = ( coinOut → tokenIn → CONVERTTOKEN )
```

Figure 2.8: Example of Process Composition.

```
CUST = ( dollarIn → tokenOut → STOP )

CHANGE = ( dollarIn → tokenOut → CHANGE )

( CUST || CHANGE ) = ( dollarIn → tokenOut → STOP )
```

Figure 2.9: Process Composed of Two Concurrent Processes.

process will not do anything on its own; it needs another process with which to interact. In other words, the change machine needs someone to put dollars and coins into its input. For this scenario, consider a process that is composed of a customer process, `CUST`, that needs change for a dollar and a change machine; the customer process and the change machine run concurrently as in Figure 2.9.

Although the previous example does model the interaction between the `CUSTj` process and the `CHANGE` process, it would be nice to have a higher level abstraction to understand how the interaction is taking place. We have one process, `CUST`; another process, `CHANGE`; and a message, `dollarIn`. But how does the `CUST` process communicate the message to the `CHANGE` process? A human would put the dollar into a slot, the slot would read the dollar, and inform the processor. CSP models this as a special class of event called a communication.

A communication $c.v$ has two parts: the *channel* c, and the message v. Channels are used by processes to communicate between each other. Channels offer unbuffered-synchronous communication. When a process reads from or writes to a channel, it blocks until the other process is ready to send or receive the communication.

In our example, the mechanism that reads the dollar and informs the process would be the channel. The `CUST` process would send the `dollarIn` on the writing end of the channel, and the `CHANGE` process would read from the reading end of the channel. Figure 2.10

```
(c!dollarIn → CUST || c?dollarIn → CHANGE(dollarIn) )
```

Figure 2.10: Processes Communicating Over a Channel.

displays a `CUST` writing a `dollarIn` message with the write operator '!', and the `CHANGE` process reading the message from channel with the '?' operator and finally handling that message.

JCSP

Now that the concept of CSP has been explained, let us consider a way of programming CSP in Java.

Communicating Sequential Processes for Java, JCSP [W^{+}03], is a Java library that allows Java programmers to use a CSP style within Java. To get a cursory understanding of JCSP, I will implement the same producer consumer problem, but this time I will use JCSP. Similar to the first Java example, there are three classes: `Producer`, `Consumer`, and `Driver`. However, unlike the first Java example, there is no need for the `Buffer` because it is replaced by the JCSP abstraction of a channel.

To understand JCSP, it is first necessary to understand two concepts: the process and the channel. The process is where all the action happens, literally; moreover, processes communicate over channels. A process implements the `CSProcess` interface. The `CSProcess` interface is similar to the `Runnable` interface: both abstract a process that can be run in its own thread of control. It is possible to run processes either sequentially or in parallel. In order to communicate, processes use channels. There are several types of channels, but the easiest to understand is a one-to-one channel. As the name suggests, there is one reading end and one writing end. Each end is held by a single separate process. When the writer writes to the channel, it blocks until the reading end receives the message. In other words, the channel provides synchronous, unbuffered, blocking communication between one process writing to the writing end of the channel, and one process reading from the reading end of the same channel.

A one-to-one channel is like the ignition of a car. For example, the car is designated as one process and the driver is another process. The ignition is the channel over which the

```
import org.jcsp.lang.CSProcess;
import org.jcsp.lang.ChannelOutputInt;

public class Producer implements CSProcess {

    private final ChannelOutputInt out;
    public Producer(ChannelOutputInt out){
        this.out = out;
    }
    public void run() {
        int x = 0;
        while(true){
            x++;
            out.write(x);
        }
    }

}
```

Figure 2.11: JCSP Producer.

driver and the car communicate. The car will sit still and not do anything until the driver communicates with it by placing the key in the ignition. Turning the key in the ignition is like writing to the channel, sending a message to the car to turn on. The car turns on and does many things, but also continues to listen on the ignition channel for a message to turn off. Because the driver has the key and is the one sending messages, the driver has the writing end of the channel; since the car is the one awaiting messages, it has the reading end.

It is important to distinguish between a channel and the channel ends. A channel provides unidirectional communication from a writing end to a reading end. The channel is a matched pair of read / write ends. In this text: “X writes to channel Y,” means X writes to the writing end of channel Y where X is a process and Y is a channel. The same goes for the reading end: “X reads from channel Y,” means X reads from the reading end of channel Y.

Figure 2.11 shows the same producer consumer example as for the previous Java implementation. The producer takes a `ChannelOutputInt`, which is the writing end of a channel that sends integers. Just as in the Java example, the producer increments an integer and sends it off for the consumer to process.

```
import org.jcsp.lang.CSProcess;
import org.jcsp.lang.ChannelInputInt;

public class Consumer implements CSProcess {

    private final ChannelInputInt in;
    public Consumer(ChannelInputInt in){
        this.in = in;
    }
    public void run() {
        while(true){
            int x = in.read();
            System.out.println(x);
        }
    }
}
```

Figure 2.12: JCSP Consumer.

Next, the consumer in Figure 2.12 reads integers from `ChannelInputInt`. The `ChannelInputInt` is the reading end of the same channel given to the producer. The consumer reads an integer and then does some processing to it; in this case, it just prints to the screen.

Finally, the driver does the wiring between the processes. The driver in this example is slightly more complicated than necessary in order to show some features of JCSP. As all Java programs, the driver begins with the `main` method. It creates a new `Driver` and from there, starts a new `Sequence` process. The sequence process runs an array of processes sequentially.

Once the driver is running, it declares a one to one channel, instantiates a producer and consumer, and provides them the writing and reading end of the channel respectively. Finally, the driver starts a `Parallel` process. The parallel process executes an array of processes, each within their own thread, and does not return control until all its processes have completed.

Though it is possible to program in a process-oriented style within Java using JCSP, there are some drawbacks to this approach. It also is possible to program in other paradigms or styles within Java. For instance, it is perfectly possible to write programs in a functional style or a declarative style within Java. Writing a library or application within a particular style requires self-constraint on the side of the programmer.

```java
import org.jcsp.lang.CSProcess;
import org.jcsp.lang.Channel;
import org.jcsp.lang.One2OneChannelInt;
import org.jcsp.lang.Parallel;
import org.jcsp.lang.Sequence;

public class Driver implements CSProcess {

    public static void main(String[] args) {
        Driver driver = new Driver();
        CSProcess[] processes = new CSProcess[] { driver };
        new Sequence(processes).run();
    }

    public void run() {
        One2OneChannelInt comm = Channel.one2oneInt();
        Producer producer = new Producer(comm.out());
        Consumer consumer = new Consumer(comm.in());
        CSProcess[] processes = new CSProcess[] { producer, consumer };
        new Parallel(processes).run();
    }
}
```

Figure 2.13: JCSP Driver.

Self-constraint is another way of saying that something is not being checked by the compiler. The Java compiler is built to check an object-oriented language; however, it knows nothing of processes or channels. It does not know that you should not give the same end of a one-to-one channel to two processes. The Java compiler also allows the use of threads and locks, and sharing variables between processes.

Regarding shared variables, pun intended, JCSP does not use true message passing. Instead, JCSP abstracts the message passing paradigm from terms of shared memory. When a process writes an object to a channel, the exact same object is received on the reading end. Now, both processes have a reference to that variable. It takes self-constraint by of the programmer to set that reference on the writer side to `null`, or to refrain from modifying that variable from both threads. This is an issue cause by using a language outside the process-oriented paradigm.

occam-π

To get pure process oriented style, with all the implied semantics, it is necessary to use another language, occam-π [WB05]. occam-π is a process-oriented programming language based on the principles of CSP. This paper does not go into the specifics of occam-π because the language has many features that exceed the scope of this work, and ProcessJ was partially based on the occam-π language.

Figure 2.14 shows the producer consumer in occam-π. Starting with the producer, a process is declared with the `PROC` keyword followed by a name and a list of parameters. The producer takes the writing end of a channel, indicated by the '`!`' character, that writes integers. Next, a new integer `x` is declared with the scope only ranging through the following sequential block, indicated by the `SEQ` keyword.

Within the sequential block, the variable `x` is set to 0 using the assignment operator '`:=`'. Next is an infinite loop where `x` is incremented then written to the `out` channel using the '`!`' operator. If you are not familiar with the syntax of occam-π, just compare to the other producer consumer examples and it is possible to understand what it is doing.

After that, the consumer defines two channels as input. The `in` parameter is the reading end of a channel, indicated by the '`?`' character. The other channel parameter is used to write values to the console, just as with Java's `System.out.print`. The rest is similar to the producer. A note regarding on the reading operation: a read is indicated by '<channel read end> ? <variable>' which reads a value from that channel and assigns it to the variable.

Finally, the driver is the main application. The driver is given the writing end of a channel, which will write to standard out from the system. The driver declares a channel and gives the appropriate ends to the producer and consumer; the producer and consumer are run in parallel. Unlike the Java example, there is no need to explicitly wait for the producer and consumer to finish; the parallel block is not completed until all of the contained processes are complete.

So far this paper has taken into consideration threads and locks, and also has described a message passing system within the context of Java. This paper has also explored CSP and an implementation of a process-oriented language called occam-π. Now, let us look at the new ProcessJ language.

```
#USE "course.lib"

PROC Producer (CHAN INT out!)
  INT x:
  SEQ
    x := 0
    WHILE ( TRUE )
      SEQ
        x := x + 1
        out ! x
:

PROC Consumer (CHAN INT in?, CHAN BYTE screen!)
  INT x:
  WHILE ( TRUE )
    SEQ
      in ? x
      out.int (x, 0, screen!)
:

PROC Driver (CHAN BYTE out!)
  CHAN INT comm:
  PAR
    Producer(comm!)
    Consumer(comm?,out!)
:
```

Figure 2.14: occam-π Producer-Consumer Driver.

2.2 ProcessJ

Sir Anthony Hoare writes of programming languages [BW09]:

> Programming language design is a fascinating topic. There are so many programmers who think they can design a programming language better than one they are currently using; and there are so many researchers who believe they can design a programming language better than any that are in current use. Their beliefs are often justified, but few of their designs ever leave the designer's bottom drawer...
>
> Programming language design is a serious business. Small errors in a language design can be conducive to large errors in an actual program written in the language, and even small errors in programs can have large and extremely costly consequences.

With the ProcessJ programming language, I intend to take Sir Anthony Hoare's words to heart. In doing so, I have researched programming languages in general, and process oriented programming specifically and asked myself why write a compiler for another language. ProcessJ is built upon a solid mathematical foundation. With that foundation I feel confident ProcessJ programmers have a reason to believe in the correctness of their applications. Communicating Sequential Processes has clearly defined separation of control flow, and the communication between processes is well defined. The scaffolding of ProcessJ is composed of two other highly successful languages within their own domain: Java and occam-π. Java is the most popular programming language in the world so it is safe to say that people are familiar with it. On the other hand, occam-π has a strong model for concurrent development. Together, they combine to create a language that looks familiar to a large number of programmers that has a strong model for concurrent development.

When James Gosling was asked, "How do you design a system programming language," he replied [BW09]:

> I tend to not think about languages and features much. In the times when I've done language design, which is tragically too often, it's always motivated by a problem. What is the context in which it is going to be run? What are people going to do with it? Kind of what is different about the universe?

Inherent concurrency is the problem ProcessJ is trying to solve. It is almost impossible to program in ProcessJ without concurrency. The idea is to abstract all the process handling to the run-time so it can be done more efficiently. It could, for instance, be abstracted to run

```
record Point {
  int x;
  int y;
}
```

Figure 2.15: ProcessJ Record.

either on a single computer with a single processor or on a distributed system. Potentially, processes could abstract away the single computer model.

With the design of the language in mind, lets look at some of the features ProcessJ offers.

2.2.1 Modules

Each package in ProcessJ contains one or more modules. A module is a ProcessJ file that contains top-level elements.

2.2.2 Top-Level Elements

ProcessJ has four top-level elements: variable declarations, records, protocols, and processes.

Top-level variable declarations in ProcessJ have module visibility. An element with module visibility can only be seen within the current file. Visibility is an important design feature of the top-level variable declarations because of the constant values between processes.

Records are similar to structs in C. They are the data part of an object in Java. Records are holders of related values. Figure 2.15 is an example of a record that represents a two dimensional point.

Protocols are similar to unions in C; they are used to define the types of messages allowed over a channel. Each protocol can have one or more named cases. Unlike unions, however, protocols only specify content and not necessarily the order they need to appear in memory.

Finally, processes define the behavior of the program. Figure 2.18 displays several

```
protocol ClientProtocol {
  Request : {
    int status;
  }
  Result : {
    int status;
    int result;
  }

}
```

Figure 2.16: ProcessJ Protocol.

examples of processes. The components of a process definition are the modifiers, keyword ‘proc’ a return type, name, parameter list, and body.

2.2.3 Built-In Types

ProcessJ has few built-in types. These are as follows.

- *Primitives*, which include boolean, byte, char, double, float, int, long, and short just as in Java.
- *Array*, which are the same as their Java counterparts.
- *Channels*, which are the communication medium between processes.
- *Channel ends*, the two ends being:
    - *read*, the reading end of a channel is used to read values from that channel, and
    - *write*, the writing end of a channel is used to write values to that channel.
- *Barriers*, which are used as synchronization points between a number of processes.
- *Timers*, which are used to post an event after a pre-determined amount of time.

2.2.4 Statements

Most statements in ProcessJ look exactly like their Java counterparts, which are well covered in the literature. Statements specific to ProcessJ are as follows.

```
alt {
  price = slowPartSupplier.read() : {
    // handle the order from the slowPartSupplier
  }
  price = fastPartSupplier.read() : {
    // handle the order from the fastPartSupplier
  }
}
```

Figure 2.17: ProcessJ alt Block.

- *Alternations* are a collection of cases that executes the 'ready' process. Consider an example process, shown in Figure 2.17, where an inventory manager contacts two order supplies. The two suppliers are represented by the reading end of channels. The algorithm always goes with the first supplier to respond. Similarly, whichever channel end is ready first is executed. The part of the `alt` case before the colon is the guard, comprised of an optional boolean precondition and either a channel read, a channel write, a timer, or a barrier.

- *Blocks* are normal sequential blocks in ProcessJ, and look the same as their Java counter parts. They start with an '{' character and end with a '}' character. However, ProcessJ has parallel blocks that execute each statement in its own thread of control. A parallel block is a normal block with the `par` before the opening '{' character.

- *Channel write* statements write data to a synchronous, unbuffered, blocking channel.

- A *claim* statement is syntactic sugar. In the case of a shared channel, this is expressed as "I'm claiming this channel to write on."

- An *enroll* statement enrolls a process on a barrier.

- A *resume* statement tells the program, where to resume after a suspend.

- *Skip* is a process that is always ready, but does nothing.

- *Stop* is another process that is always ready to halt.

- A *suspend* statement returns control to the initiating process until the process is resumed.

```
public proc void Producer(chan<int>.write out){
  int x = 0;
  while(true){
    x++;
    out.write(x);
  }
}

public proc void Consumer(chan<int>.read in) {
  while(true){
    int x = in.read();
    // do stuff with x
  }
}

public proc void Driver(){
  chan<int> comm;
  par {
    Producer(comm.write);
    Consumer(comm.read);
  }
}
```

Figure 2.18: ProcessJ Producer Consumer Driver.

- A *sync* statement synchronizes on a barrier. Every process enrolled on a barrier will wait at these statements until all processes enrolled on that barrier have reached the synchronization point.

- A *timeout* statement represent a timer process plus a duration.

2.2.5 Producer Consumer

In the ProcessJ producer consumer example, there are three processes: the producer, the consumer, and the driver. Just as in previous examples, the producer takes in the writing end of a channel that carries integers. In the body, it loops forever; it increments the `x` value and writes it to the channel. The consumer, once again, loops forever while reading values for the reading end of a channel. The driver declares a channel, and executes the producer and consumer in parallel.

In Sir Anthony Hoare's acceptance speech for the Turing Award [Hoa81], he said,

> There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it

> so complicated that there are no obvious deficiencies.

Comparing the ProcessJ code to Java code, it is painfully obvious which is so simple that there are obviously no deficiencies and which is so complicated that there are no obvious deficiencies. ProcessJ is simple and elegant with built-in parallelism. ProcessJ has mathematically based semantics to ensure correctness. It took minutes to write this code, which is obviously correct. Each process has just a few lines of code that are easily comprehensible.

However, the Java example took much longer to write. There are many lines of code that have nothing to do with the primary responsibilities of the system, namely the `Buffer` and the parts of the driver that deals with threads. It is complicated, and can be written incorrectly several times before the final implementation; the system is not intuitive.

The example in JCSP is a step in the right direction. The code is much easier to understand. The only problem is that it relies on the self-constraint of the programmer. It requires you to not go outside the model for parallelism because it is not checked by the compiler. However, ProcessJ can make additional semantic checks specific to the process-oriented programming model.

Finally, occam-π is strikingly similar by design. The only criticism to make of occam-π, and a major reason for the development of ProcessJ, is that a large number of people do not like the syntax of this language. Programmers used to the C-family programming languages, such as Java, are not used to mandatory indentation. They do not like prefixing 'SEQ' to what could be implied as sequential code. These are some of the little things that the use of ProcessJ's Java-like syntax hopes to remedy.

Chapter 3

Project

Before getting into the design and implementation of the compiler, details of the ProcessJ compiler project will be explained in this chapter. A project has to do with more than code. If it were just code, it would be simple. For instance, all code could be thrown into one file that was five thousand lines long, there would be one command to build the project, and one command to run the resulting artifact all of which lies in the mind of the creator. However, there is a human component to any project, which means that project needs build processes, structure, and a means for collaboration.

To make the build process, structure and collaboration as simple as possible, this project focused on simplicity and standards so that future developers could focus on coding rather than on the build process. To simplify each of these project components, a combination of Apache Maven, git and `github.com` was chosen.

Git and `github.com` were the tools that people on this project used to collaborate with each other on tasks like issue tracking, code review, and source control. The distributed nature of git allowed the team members to work on and manage their own repositories; they only merged when the time was right. Distributed source control systems, such as git, encourage developers to commit more often, because it is only their own repositories that are affected; if someone needs a change that another developer had made, that person can merge the change.

Apache Maven is a software project management and comprehension tool that has the primary goal of allowing a developer to comprehend the complete state of a development effort in the shortest period of time [Apa03]. Maven was chosen for this project because of its simple build process, common project structure, quality project information, and also because this tool gives general ‘best practice’ guidelines. The rest of this chapter will detail how Maven was used in this project and the effect it had on the project.

3.1 Build Process

A uniform and familiar build process was the goal when developing this project. It was known that there would be source code generation for syntax analysis, and that the compiler's purpose was to generate source files. Code generation needs to happen before the main application can compile. The main application needs to compile before tests can compile, and the tests need to compile before the tests are run. However, the tests do not need to compile in order to run the application.

When looking for a clean way of implementing the build process for the application, the Apache Maven project was evaluated. Maven includes the concept of a life cycle. There are three default life cycles, and each life cycle has a standard set of phases. It was a clean, consistent, and well defined process that is already well-documented.

Maven allows plugins to attach goals to life cycle phases. As the life cycle executes, it iterates through the phases and executes all goals bound to the current phase. Think of a life cycle as a pipeline, and the phases as stages in that pipeline. The goals are similar to actions that execute at a certain stage of the pipeline.

There are three life cycles that are available with the distribution of Apache Maven: clean, default, and site. It is possible to define alternative life cycles, but there was no need to do so for the ProcessJ compiler project.

3.1.1 Clean Life Cycle

The *clean* life cycle is used to remove all generated files from previous builds. When the clean life cycle is invoked, the following three phases are executed in order: *pre-clean*, *clean*, and *post-clean*. The *pre-clean* and *post-clean* phases have no default bindings. However, the *clean* phase is bound to the *clean:clean* goal by default. During the *clean:clean* goal, the

Maven Clean Plugin deletes the project's working directory of generated files. By default, it discovers and deletes the directories configured in the Project Object Model (POM), described in Section 3.2.1; this can be useful if generated sources get out of sync with source code. The following list displays settings in the POM that allow Maven to know what to clean by default.

project.build.directory The `target` directory is the base build directory, which contains all generated sources to provide a clean separation from managed source.

project.build.outputDirectory All generated class files are placed in the `target/classes` directory.

project.build.testOutputDirectory All generated class files from test classes are placed in the `target/test-classes` directory.

project.reporting.outputDirectory Generated site files are placed in `target/site`.

3.1.2 Site Life Cycle

An important feature of Maven is its ability to generate a web site from the project. The site is generated during the *site* lifecycle. There are four phases of the *site* lifecycle: *pre-site*, *site*, *post-site*, and *site-deploy*. Similar to the *clean* lifecycle, the *site* life cycle hast two phases without default bindings: the *pre-site* phase and the *post-site* phase. The *site* phase has the *site:site* goal, which executes each of the reporting plugins then generates the site from the results; the *site-deploy* has the *site:deploy*, which deploys the generated site to the ProcessJ web site server using `scp`.

3.1.3 Default Life Cycle

Finally, the most important life cycle is the *default* life cycle. The *default* life cycle handles the project deployment, which involves everything from generating sources to packaging the final product as a 'jar' file. Below is a list of the phases involved in the *default* life cycle and a description from the Apache Maven site [Apa03].

validate Validates that the project is correct and all necessary information is available.

initialize Initializes build state, e.g., sets properties or creates directories.

generate-sources Generates any source code for inclusion in compilation.

process-sources Processes the source code, for example, to filter any values.

generate-resources Generates resources for inclusion in the package.

process-resources Copies and processes the resources into the destination directory, ready for packaging.

compile Compiles the source code of the project.

process-classes Post-processes the generated files from compilation, for example, to do bytecode enhancement on Java classes.

generate-test-sources Generates any test source code for inclusion in compilation.

process-test-sources Processes the test source code, for example, to filter any values.

generate-test-resources Creates resources for testing.

process-test-resources Copies and processes the resources into the test destination directory.

test-compile Compiles the test source code into the test destination directory

process-test-classes Post-processes the generated files from test compilation, for example, to do bytecode enhancement on Java classes. For use with Maven 2.0.5 and above.

test Run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed.

prepare-package Performs any operations necessary to prepare a package before the actual packaging. This often results in an unpacked, processed version of the package. (Maven 2.1 and above).

package Packages the compiled code in a distributable format, such as a JAR.

pre-integration-test Performs actions required before integration tests are executed. This may involve such actions as setting up the required environment.

integration-test Processes and deploys the package if necessary into an environment in which integration tests can be run.

post-integration-test Performs actions required after integration tests have been executed. This may include cleaning up the environment.

verify Runs any checks to verify the package is valid and meets quality criteria.

install Installs the package into the local repository, for use as a dependency in other projects locally.

deploy This is done in an integration or release environment, and copies the final package to the remote repository for sharing with other developers and projects.

3.2 Structure

There are three top-level elements to the ProcessJ project: the Project Object Model (POM), the source, and target directories. The POM is Maven's method of describing the meta-data of a project and adding functionality into the build process. The source directory is where all project documents are kept including the ProcessJ compiler, the unit tests, run-time resources as well as documentation for the project site. Finally, the target directory is a place to keep all generated source files.

One benefit to using Maven is its aspect of convention over configuration. All aspects of a project are configurable: source files go where the creator desires; unit tests can be kept in the same folder as run-time code; and libraries are expected to be somewhere on a class-path environment variable. There is no limit to how complex the project can be. Fortunately, that amount of flexibility is not necessary. Once a developer has worked on a Maven project, that developer is instantly familiar with the structure of any other Maven project.

This section describes the three aspects common across Maven projects: the POM, the source, and target directories.

3.2.1 Project Object Model

The POM is the central concept of Maven. Maven the Definitive Guide [Son08] states, “The POM is where a project’s identity and structure are declared, builds are configured, and projects are related to one another.” There are four types of information about the project: POM relationships, general project information, build settings, and the build environment.

POM relationships describe how the project relates to other projects. Every Maven project has coordinates and an optional set of dependencies. Coordinates are a unique set of information that identifies a Maven project. The three essential coordinates are `groupId`, `artifactId`, and `version`.

The `groupId` of the ProcessJ compiler is `edu.unlv.cs`; the name follows a convention followed in Java where the top level package is the reverse domain name of the organization that developed the project. Using the `groupId` allows grouping all artifacts of an organization, because it determines the location of the artifact in the Maven repository. The `groupId` is split on ‘.’ to determine the directory structure. Since the value of the `groupId` is `edu.unlv.cs`, the directory structure inside the Maven repository becomes `edu/unlv/cs`.

Next, the `artifactId` for the ProcessJ compiler is `processj-compiler`, and the `version` is 1.0.0-SHAPSHOT. Maven uses the coordinates to determine the name and location of the artifact for each project. After executing the *install* phase, the artifact, which is `processj-compiler-1.0.0-SNAPSHOT.jar`, is located in the repository directory `edu/unlv/cs`. The Maven naming convention is convenient because as the version number changes, it is possible to retain multiple version of the same artifact.

If the coordinates allow other projects to access the current project, then the dependencies section allows the project to define what projects it needs access. The ProcessJ compiler utilizes several open-source projects as dependencies, as listed below.

ANTLR The ANTLR, Another Tool for Language Recognition (ANTLR) project is used to generate the parser and lexer from a grammar file.

Commons CLI The Apache Commons CLI, command line interface, project provides an API for parsing command line arguments and print a help message at the command line.

Commons IO The Apache Commons IO project is used to output the generated classes into files.

JUnit A unit testing framework. This is strictly a test dependency, and is not necessary for the completed ProcessJ compiler to execute.

StringTemplate The StringTemplate project is used to generate the output from a template format.

The general project information defined in the POM indicates such information as the project's name, the organization, the web site URL, a list of contributors, and the license under which the project is distributed. Specifying general information in the POM allows Maven to include this information in the generated site and to know where to deploy the generated site. For instance, the site `http://processj.cs.unlv.edu` contains the generated site for the ProcessJ compiler. The general information is not strictly necessary for the build process, although it is useful for generating the site.

Maven provides access to plugins by defining the plugins used in the POM in the build settings. It is possible to customize the build and site generation processes by adding plugins and attaching them to life-cycle phases. Below is a list of plugins and extensions the ProcessJ compiler utilizes with descriptions from the Apache Maven site.

antlr3 Generates the parser and lexer from an ANTL grammar file.

assembly Builds an assembly (distribution) of sources and/or binaries.

changelog Generates a list of recent changes from the SCM.

checkstyle Generates a report regarding the ode style used by the developers.

clean Cleans up after the build.

compiler Compiles Java sources.

eclipse Generates an Eclipse project file for the current project.

findbugs Performs static analysis of the generated class files and generates a report based on the results.

install Installs the built artifact into the local repository.

jar Builds a JAR from the current project.

javadoc Generates Javadoc for the project.

jxr Generates a source cross reference.

pmd Similar to findbugs, it is a static analysis tool.

resources Copies the resources to the output directory for inclusion in the JAR.

site Generates a site for the current project.

ssh Copies site files to the deployment URL.

surefire Runs the JUnit unit tests in an isolated class-loader, and creates a report based on the results.

Finally, the last collection of settings in the POM are the build environment settings, which consist of profiles.

3.2.2 Source

The `src` directory is where the source files for the project are located. Under the `src` directory, there are three directories: `main`, `site`, and `test`. Each directory clearly separates the concerns of its contents. The `main` directory stores the source code and resources that will eventually produce the executable ProcessJ compiler. The `site` directory contains information necessary to generate the ProcessJ web site, and the `test` directory contains code and resources for testing.

Main Directory

The name of the `main` directory should be a good indicator of the intended purpose. It is the main directory of the whole project because it contains the source files for the ProcessJ compiler. Because 'source' does not necessarily mean '.java' file, inside the `main` directory are the three source code directories: `antlr3`, `java`, and `resources`.

Within the `antlr3` directory is the ANTLR grammar file for ProcessJ, named `ProcessJ.g`. The location and name of the file are significant because it is used to determine the name and package of the generated lexer and parser.

The `java` directory contains the actual source code of the ProcessJ compiler. Within the `java` directory is the base Java package of the project `edu/unlv/cs/processj`. The contents of this directory will be described in Chapter 4.

Finally, the `resources` directory contains the resources used during the run time. Specifically, two files that reside in this directory are `log4j.xml` and `templates/java.stg`.

The `log4j.xml` is a configuration file that tells the system where to output log messages and how they should be formatted. Instead of using `System.out.println`, log4j allows the programmer to configure which log messages are displayed. For instance, there is a normal mode, and a verbose, or debug, mode. During the debug mode, which is a command line option, many more messages are displayed than in the normal execution mode. This will allow compiler developers to isolate problems with the compiler, and to see much more information than would be strictly necessary what is only needed is to compile some ProcessJ source files. Utilizing a configuration file for logging allows the developer to keep the code clear of commented out debug statements.

Another resource file that helps clean up the code, is the `templates/java.stg` file. The `templates` directory was originally meant to contain the string template group files for each of the supported target languages. Since Java is the only currently supported target language, the `java.stg` file is the only template file. However, as more target languages or platforms are added, the `templates` directory will be a single place where all the string template group files can be kept.

Site Directory

The site directory contains the files necessary to generate the content for the ProcessJ compiler website. There are two components of the site directory: the site.xml file and the apt directory.

In order to generate a site, Maven needs to know what to generate. The site.xml file, called the site descriptor, describes the navigation structure of the generated site. The site descriptor is used to split the navigation into information that is useful to intended users. There are three sections to the site descriptor: the overview, developers, and reports.

The overview section is intended to give general information about the compiler to ProcessJ developers. The overview section contains information about the ProcessJ language, the compiler, and more advanced options of the compiler.

Next, the developers section contains information pertinent to developers working on, or interested in working on, the ProcessJ compiler. Much of the developers section contains the design and implementation details outlined in this thesis. The developers section is a starting point when searching for documentation on the ProcessJ compiler.

Finally, the last section of the site descriptor is the reports section, which contains reports that Maven generates about the project. The reports contained in this section allows current developers to maintain quality control of the project.

Once Maven knows the structure of the generated site, it also needs to generate the content. The format used to generate the content is Almost Plain Text (APT) [Apa07a], and the files are in the apt directory. APT is a wiki-like format used to generate the content of the web site. Using the ProcessJ website as an example, the content for the site ranges from how the compiler is used to developer documentation.

The final directory inside the source is the test directory. Tests are necessary to any good software project. In the ProcessJ compiler, the tests are kept in the src/test directory. Keeping test files separate from production code is a good practice because it clearly defines what is necessary code to run the application, and what is only there to verify that the code works as expected.

As of this writing there are 146 tests for the ProcessJ project. The great thing about tests are that they give confidence when making changes. After making a change, running the mvn test command executes all the unit tests. As the number of tests increase, developers can become more confident they are not breaking anything.

There are two test rigs: one for syntax analysis, BaseProcessJGUnitTest, and an assertion action for semantic analysis. The BaseProcessJGUnitTest ensures that the grammar correctly validates source programs. The test utilities provided by this class are based on the gUnit project [JYSP07]. It provides input text, the grammar rule to test, and a simple pass or fail. In this way, several issues with the grammar were found and resolved early in the development cycle.

Semantic analysis is performed by executing actions against the abstract syntax tree. To isolate an action performed against the abstract syntax tree, a mechanism similar to that used in testing the syntax analysis phase. Since the test input is controlled, no errors are expected from the syntax. From the source, an abstract syntax tree is built and a number of actions are run against the abstract syntax tree.

For this compiler, an AssertionAction was created that takes a collection of Assertions. The Assertion is used to run a normal JUnit assert statements against specified element classes. For instance, if the action to test alters NameExpr elements, it is possible for the Assertion to hook into the pre and post methods of the tree walk, and verify that the action performs as expected. The AssertionAction provides a convenient means to access only the elements of an abstract syntax tree that need verification. In other words, there is no need to manually traverse the abstract syntax tree after performing the action in order to verify that the action did what it was supposed to do.

The source directory contains three sub-directories: main, site, and test. The main directory contains all source files used directly in the ProcessJ compiler. The site directory stores all files used to generate the project site. Finally, the test directory contains source files and resources to verify the compiler works as expected.

Target

The final top-level element of the project is the target directory. Generated files are put into the target directory. That includes the site, the generated source for the lexer and parser, and all class files as well as JUnit and static analysis reports.

There is no need to discuss the contents of this folder further. However, it is important to note that files in the `target` directory are not meant for source control. The entire directory is deleted when the clean life cycle phase is run.

Chapter 4

Design & Implementation

Like many compilers, the ProcessJ compiler can be thought of as a pipeline. This chapter will cover the parts of the ProcessJ compiler pipeline, beginning with the Command Line Processor, then Syntax Analysis, Semantic Analysis, and finally Code Generation. Figure 4.1 shows the control flow of the ProcessJ compiler.

In this chapter, reference will be made to many classes. For the sake of brevity, the '∼' symbol will be used when referring to the base package `edu.unlv.cs.processj`. For instance, `edu.unlv.cs.processj.Main` will be referred to as ∼`.Main`.

4.1 Command Line Processor

The ProcessJ compiler pipeline begins at the command line. The Command Line Processor (CLP) portion of the compiler is implemented in the ∼`.Main` class. As with the beginning of the pipeline, the ∼`.Main` class has three main responsibilities: to parse the command line options, to determine the source files, and to hand control off to the syntactic analyzer.

4.1.1 Available Command Line Options

For convenience, there is a script file called `pjc` that takes the arguments specified and hands them to the correct Java command line invocation for ∼`.Main`. Figure 4.2 displays the result of running `pjc` with no command line arguments or with the `--help` option. The arguments are meant to be self explanatory, but there are four options worthy of note:

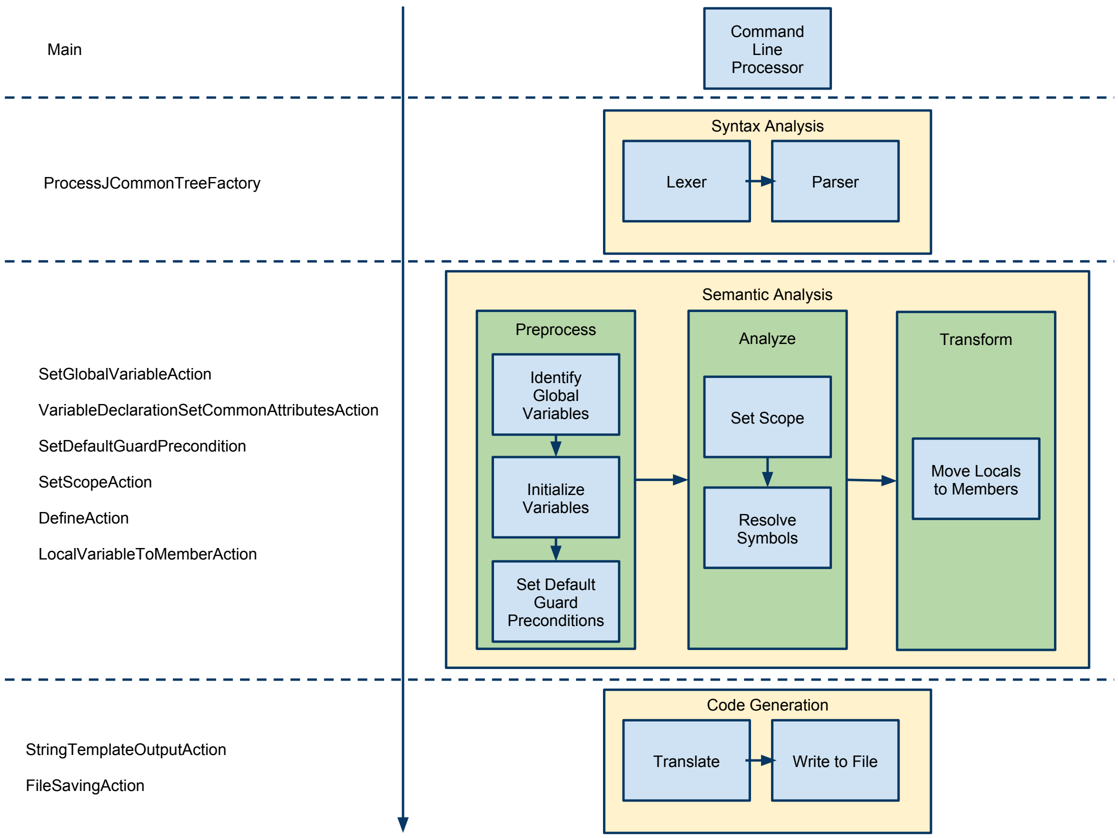

Figure 4.1: ProcessJ Pipeline.

```
usage: pjc [-d <arg>] [-ea] [-h] [-j | -m]  [-nowarn] [-pjpath <arg>]
       [-sourcepath <arg>] [-v]
 -d <arg>                 Set the output directory.
 -ea,--enableassertions   enable assertions
 -h,--help                show the help message
 -j,--java                compile to java
 -m,--mpi                 compile to c and use mpi
 -nowarn                  Disable warning messages.
 -pjpath <arg>            Set the user pj library path, overriding the
                          user pj library path in the PJPATH environment
                          variable. If neither PJPATH or -pjpath is
                          specified, the user class path consists of the
                          current directory
 -sourcepath <arg>        Specify the source code path to search for
                          definitions. As with the user pjpath, source
                          path entries are separated by semicolons (;) and
                          can be directories. If packages are used, the
                          local path name within the directory must
                          reflect the package name.
 -v,--verbose             set log level to debug.
```

Figure 4.2: ProcessJ Compiler Command Line Help Message.

output directory, the target language, source path, and verbose.

To clear the base working directory of generated source files, the output directory is used to place the files in a generated source folder inside the `target` directory. Putting the output files in this source folder guarantees two things: they will be cleaned up when the clean life cycle is executed, and they are not accidentally put into source control.

Although both MPI and Java are listed as options for targets, only Java is currently implemented in order to show how to setup a second target language.

Finally, the verbose option greatly facilitates debugging the compiler. Adding the verbose option at the command line will output the abstract syntax tree, and scopes as they are processed. It is cleaner to print to the debug log than to have print line statements all over the code.

```
final Options options = new Options();
final OptionGroup targetLanguages = new OptionGroup();
targetLanguages.addOption(new Option(”m”, ”mpi”, false,
                ”compile to c and use mpi”));
targetLanguages.addOption(new Option(”j”, ”java”, false,
                ”compile to java”));
options.addOptionGroup(targetLanguages);
```

Figure 4.3: Sample of Command Line Option Declaration Stage.

4.1.2 Command Line Option Parsing

The ProcessJ compiler delegates all command line processing to the Apache Commons Command Line Interface (CLI) [Apa07b]. There are three stages to command line processing with Apache Commons CLI: definition, parsing, and interrogation.

All the available command line options in the ProcessJ compiler are declared in the definition stage. The command line options are declared in the `getCommandLineOptions` method of the ~`.Main` class. Figure 4.3 is a sample of how options are declared for the CLP. In the sample, the target language option group is declared. An option group is a set of mutually exclusive options. For instance, there can only be one target language; therefore, the user can either choose `mpi` or `java`.

Apache Commons CLI parses the command line options in several formats. The format chosen for the ProcessJ compiler was the `BasicParser`, simply because the format seemed cleaner. Command line parsers are interchangeable, so the change is simple if a programmer later decides to use a different format. The command line parser takes the arguments provided to the `main` method and figures out which options were used.

The last stage in command line processing is interrogation. In the interrogation stage, the parsed command line options are queried to find which options were used and get any associated values. For instance, to find which target language was selected, the `CommandLine` is queried with the `hasOption` method for either ‘`mpi`’ or ‘`java`’.

To initiate the compiler pipeline, the CLP has the responsibility to determine the source files to process. Source files are collected and handed to the syntax analysis phase to continue the pipeline.

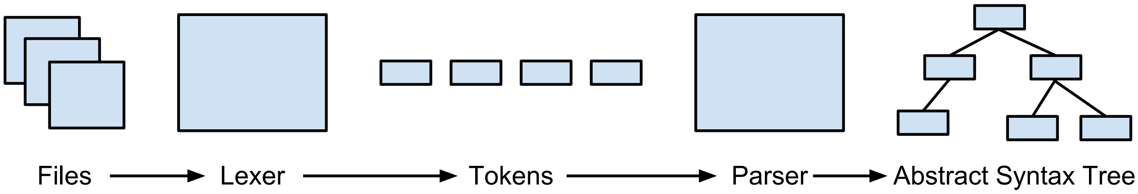

Figure 4.4: Syntax Analysis Phase.

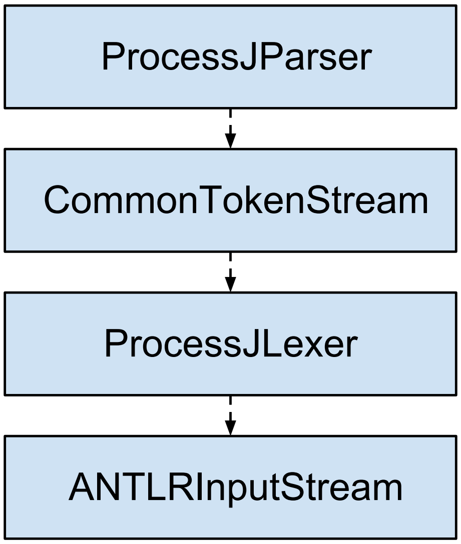

Figure 4.5: Syntax Analysis Interactions.

4.2 Syntax Analysis Phase

Syntax analysis is the second phase of the pipeline; it takes in the files to parse and outputs an abstract syntax tree. The purpose of this phase is to ensure that the input program adheres to the ProcessJ syntax. In other words, the syntax analysis phase ensures the input program has the correct structure. Without the correct structure, there is no way to determine meaning that is required later in the pipeline.

There are four main components to converting from a source file into an abstract syntax tree: an input stream, the lexer, a token stream, and the parser. Each component pulls information from the element below, creating a slightly higher level of abstraction. From the bottom up, the input stream reads data from a file, and the input stream is used by the lexer to create tokens. The token stream reads the tokens from the lexer, which feeds the parser. Finally, a client reads a complete abstract syntax tree from the parser.

The syntax analysis phase is complicated. There are roughly 120 lexical patterns in the ProcessJ grammar and far more syntax rules. Given the complicated nature of this phase, it does not make sense to code the lexer and parser directly in Java. Instead, it is simpler and more natural to use a language that is similar to Extended Backus-Naur Form (EBNF), a language used to describe grammars.

4.2.1 ANTLR

ANother **T**ool for **L**anguage **R**ecognition (ANTLR) [PQ95], generates the lexer and parser for the ProcessJ compiler from a grammar file. Rather than write the lexer and parser by hand, ANTLR allows the programmer to describe the syntax of a language in a domain specific language similar to EBNF and generate a lexer and parser from the grammar.

An ANTLR grammar consists of two logical parts: a header and body. In the header, there is meta-data about the grammar and details that help the code generator. The body consists of the lexical and syntactic rules of the grammar. There is nothing to distinguish the header from the body in the file; it is merely a way to distinguish between the meta-data and the rules.

The meta-data, or header of the grammar, is composed of options, tokens, and a section to directly add user-defined code. First, options are a set of key-value pairs that alter the code generated by ANTLR. After the options, the tokens section, in combination with the lexical rules, define the tokens the lexer will produce. Finally, ANTLR provides a section to add user-defined code, which the code generator copies directly into the generated classes for the lexer and parser. This section will describe the options used in the ProcessJ compiler, give a brief overview of the token section, and explain the purpose of the user-defined code.

As stated previously, ANTLR grammar options are a set of key-value pairs that alter the generated code. The ProcessJ grammar utilizes five grammar options: language, output, backtrack, memoize, and ASTLabelType, as displayed in Figure 4.6.

First, the language option tells the code generator what language to generate the lexer

```
options {
    backtrack = true;
    memoize = true;
    output = AST;
    ASTLabelType = ProcessJTree;
}
```

Figure 4.6: ANTLR Grammar Header Options.

and parser. The default language is Java, so the option is not used explicitly.

Next, the output option defines the data structure that the generated recognizer will generate [Par07]. The ProcessJ grammar specifies that the generated code should produce an abstract syntax tree (AST) as the final product. Using AST also allows the programmer to use tree construction operators in the parser rules; these will be discussed later.

Backtracking allows the generated parser to try alternative rules, should the first rule match fail. Once a rule fails, the input is rewound, and the next alternative is tried.

Backtracking has performance issues, and turns the usually linear time LL(*) algorithm to an exponential time algorithm [Par07]. Of course, the alternative to backtracking is to complicate the grammar by pulling out common sub-expressions or adding explicit syntactic or semantic predicates. Fortunately, there is a way to complement backtracking in order to bring the algorithm back to linear time.

To combat the performance problems with backtracking, the memoization option is enabled. The memoization option enables a dynamic programming technique that achieves linear time by saving partial parsing results. Saving the partial parsing results is not without consequence, however. Memoization increases the total amount of memory necessary to parse the input; however, this is an acceptable trade for the simplified grammar.

The final option used in the ProcessJ grammar is ASTLabelType. Since the ANTLR code generator does not make any assumptions about the type of AST created, ANTLR allows a base type to be specified. The base AST type in ProcessJ is ~`.ast.ProcessJTree`.

After specifying each of the options, the next portion of the grammar header is the tokens section. The tokens section specifies all the keywords, operators, and imaginary tokens, as seen in Figure 4.7.

Imaginary tokens are tokens that are not directly associated to specific text. However,

```
tokens {
    // operators and other special chars
    AND                         = '&'              ;
    AND_ASSIGN                  = '&='             ;
    ASSIGN                      = '='              ;
    BIT_SHIFT_RIGHT             = '>>>'            ;
    ...
    // keywords
    ASSERT                      = 'assert'         ;
    BOOLEAN                     = 'boolean'        ;
    BREAK                       = 'break'          ;
    BYTE                        = 'byte'           ;
    ...
    // tokens for imaginary nodes
    ALT_CASE;
    ARGUMENT_LIST;
    ARRAY_DECLARATOR;
    ARRAY_DECLARATOR_LIST;
    ARRAY_ELEMENT_ACCESS;
    ...
}
```

Figure 4.7: ANTLR Grammar Header Tokens.

```
@lexer::header {
package edu.unlv.cs.processj.antlr;
}

@lexer::members {
public boolean preserveWhitespacesAndComments = false;
}
```

Figure 4.8: ANTLR Grammar Header Lexer Header and Members.

imaginary tokens can indicate to the tree adapter the type of node to create. More details will be given on the tree adapter later in the chapter. ANTLR allows tokens or character literals in parsing rules. Even though it is possible to use character literals for keywords and operators within parser rules, the generated code is much cleaner after declaring all the character literals as tokens.

The last header section is the user defined code section otherwise known as *named global actions* [Par07]. There are four global actions used in the ProcessJ grammar: the lexer header and members, and the parser header and members. Each action is a place holder for code to insert at certain positions in the generated code.

Lexer header and members actions are simple. The lexer header action, as seen in

```
@header {
package edu.unlv.cs.processj.antlr;

import java.util.List;
import java.util.Stack;

import edu.unlv.cs.processj.ast.ProcessJTree;
import edu.unlv.cs.processj.Notification;
}

@members {
private final Notification notification = new Notification();
private final Stack<String> paraphrases = new Stack<String>();
private String filename;

@Override
public void emitErrorMessage(String error){
  StringBuilder message = new StringBuilder(getFilename()).append(‘‘ ‘‘)
      .append(error);
  notification.addError(message.toString());
}
// more methods ...
}
```

Figure 4.9: ANTLR Grammar Header Parser Header and Members.

Figure 4.8, is inserted before the lexer class definition; therefore, the only necessary code is the package declaration. The lexer members action is a place to put lexer field declarations and functions. There is a single boolean field, `preserveWhitespacesAndComments`, which does exactly what the name indicates. When true, the white space and comments are preserved in the token stream; when false, the white space and comments are ignored.

Moving on to more interesting code, we will now look into the parser header and members actions, as shown in Figure 4.9. Similar to the lexer header and members actions, the header action is the placeholder above the class declaration, and the members action is the place to define fields and member functions. The parser members action contains three fields and nine methods.

Inside the parser members action, there are three fields: `notification`, `paraphrases`, and `filename`. The `notification` field is an object that maintains error information. When the parser comes across an input that it cannot recognize, it will file an error in the `notification`. The message the parser will file in the `notification` is determined by the `paraphrase` field. Messages are pushed onto the `paraphrase` stack when the parser can recover from failure. The `paraphrase` stack is popped, and the value is discarded after a rule has been successfully parsed. However, when there is a failure, the top of the `paraphrase` is popped and used to provide a helpful error message. Error handling will be explained in a later section. Finally, the `filename` field is used to store the file name of the input so that it can be passed along to the AST.

ProcessJ parser members action defines several methods. This will not be described in detail because the main purpose of the members action methods support the error handling and recovery. Note, however, that this section is copied into the generated parser code.

After the grammar header, the heart of the grammar – the parser and lexer rules – are defined. Going through each of the rules individually would be a long and arduous task. Instead, what parser and lexer are, their roles and responsibilities, their interface, and how the generated implementation works will be described in detail.

4.2.2 Lexer

So far, the term 'lexer' has been used liberally and without sufficient explanation. This section gives a little background on what a lexer is, its roles and responsibilities in the ProcessJ compiler, and how ANTLR implements the code for the lexer from the grammar.

A lexer, or lexical analyzer, reads input characters of the source program, matches them by patterns, groups them into *lexemes*, and produces as output a sequence of tokens for each lexeme in the source program [ALSU07]. In that description, there are three terms that can be confusing: *token*, *pattern*, and *lexeme*.

- A token is represented in ANTLR by the `Token` interface, and is implemented by the `CommonToken` class. The type of a token is a uniquely generated integer representing the token name. The token also contains information about its position in the source program, such as the line number and position within the line.

- A pattern is a description of the form that the lexemes of a token may take [ALSU07]; it is implemented as a lexical rule in the grammar and similar to a regular expression.

- A lexeme is the string value associated to a token instance. In other words, when

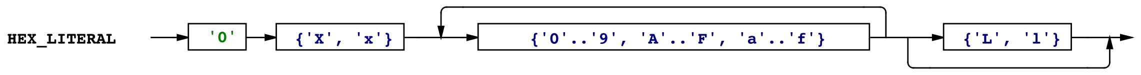

Figure 4.10: Example Lexical Rule DFA.

```
HEX_LITERAL : ’0’ (’x’|’X’) HEX_DIGIT+ INTEGER_TYPE_SUFFIX? ;

fragment
HEX_DIGIT : (’0’..’9’|’a’..’f’|’A’..’F’) ;

fragment
INTEGER_TYPE_SUFFIX : (’l’|’L’) ;
```

Figure 4.11: Example Lexical Rule.

the lexer matches the input to a pattern, it generates a token with the value of the matching lexeme.

In the ProcessJ grammar, the complex tokens are specified using patterns, while simple tokens are specified as the strings they match. Figure 4.11 displays a sample of an ANTLR pattern. In the sample, the `HEX_LITERAL` is a full pattern, while `HEX_DIGIT` and `INTEGER_TYPE_SUFFIX` are fragments. A fragment cannot generate a token by itself, but it allows reuse of a pattern to other patterns. In this case, `HEX_LITERAL` matches a hexadecimal number as accepted by Java. For instance, a value like ‘0x1A’ would be a valid value. The compiled rule is expressed as a deterministic finite automata in figure 4.10.

One of the benefits of the ANTLR generator is its ability to generate readable code for the parser and lexer. When there is an issue with how the tokens or valid input are not parsed correctly, it is nice to step through the code and understand what is going on. Now that we have an idea of how the lexical rules look in the grammar, we can look at the code generated in ~`.ProcessJLexer`.

To start, each token is converted into a field. For instance, the token `AND` generates the following field: `public static final int AND = 4`. These integers are used in later generated methods to set the `_type` field.

After the token integers have been generated, the ANTLR generator defines a method to recognize each token. Figure 4.12 is an example of the generated code for the character

```
// ANTLR start "AND"
public final void mAND() throws RecognitionException {
  try {
    int _type = AND;
    int _channel = DEFAULT_TOKEN_CHANNEL;
    // edu/unlv/cs/processj/antlr/ProcessJ.g:19:5: ( '&' )
    // edu/unlv/cs/processj/antlr/ProcessJ.g:19:7: '&'
    {
      match('&');
    }

    state.type = _type;
    state.channel = _channel;
  }
  finally {
  }
}
```

Figure 4.12: Example Generated AND Method.

literal token `AND`. The methods for character literal tokens are simple. First, the `type` and `channel` are set. The `type` is set to the current token type, in this case the `AND`, and the `channel` is set to default. ANTLR has two channels: `DEFAULT_TOKEN_CHANNEL` and `HIDDEN`. The default token channel is the channel usually used by the parser, and the hidden channel is used to output comments and white space characters. For instance, in the `COMMENT` and `WS` rules, the channel is switched to hidden.

The core of this method is the call to `match`. Inside `match`, the lexer checks that the look ahead is as expected, in this case the '&' character. If it is in the correct state, the input is consumed and control returned. If the lexer is not in the correct state, an exception is thrown so the driver can try to recover or add an error notification.

Finally, the last portion of the generated compiler is the `mTokens` method and its associated deterministic finite automata (DFA). The `mTokens` method uses a table-driven DFA to determine which token is next in the input. The DFA is string encoded during code generation because the resulting table could potentially make the lexer class so big that the Java compiler will not allow it to compile. Instead, the DFA is unpacked into an array of short type. The super class of ∼`.ProcessJLexer` is the lexer provided by the ANTLR run-time. That class calls the `mTokens` method from its `nextToken` method.

Trying to explain which method calls another can get confusing. Instead, examine

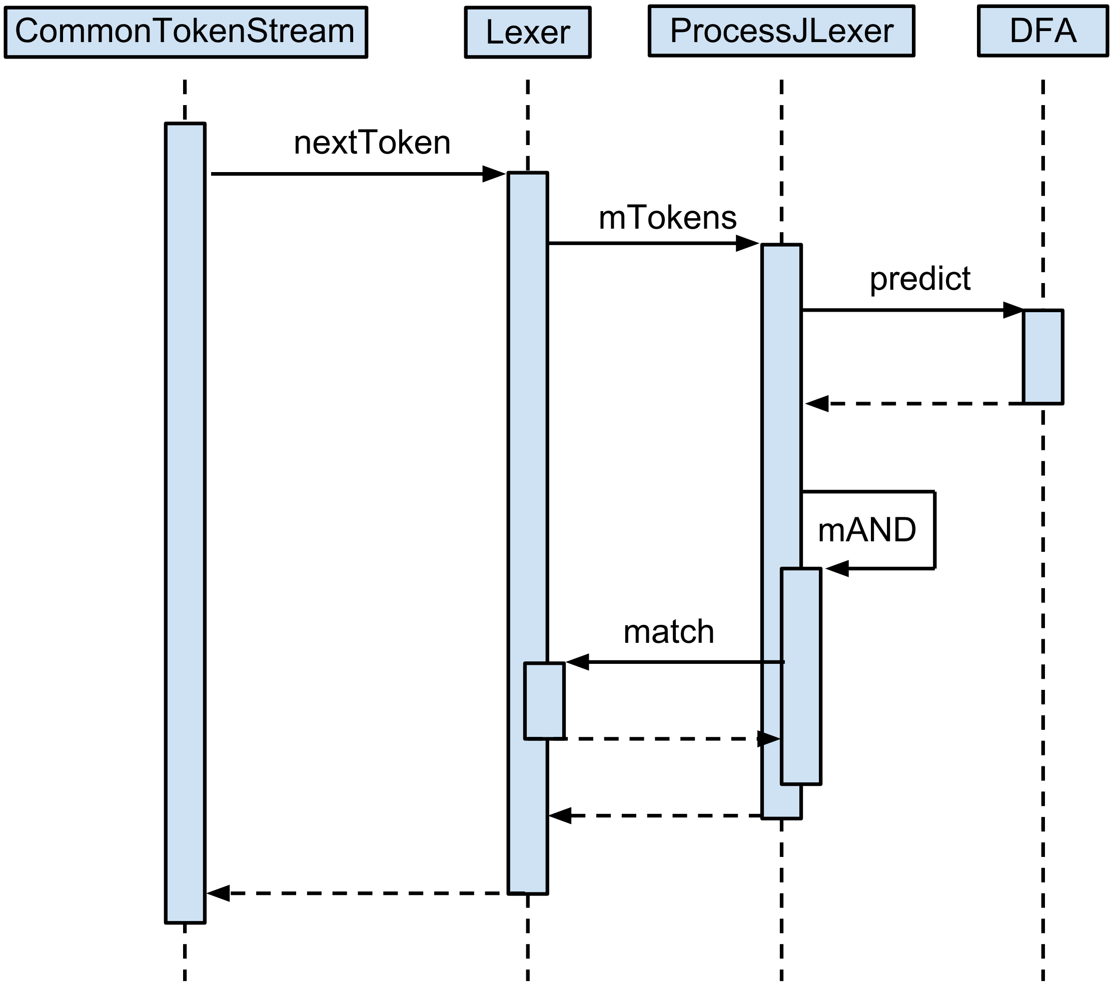

Figure 4.13: Sequence Diagram for `nextToken`.

Figure 4.13 to determine an example flow of control for generating a new token.

4.2.3 Parser

While the lexer uses patterns to generate a stream of tokens, the parser takes that stream of tokens as input; the output is an AST. The role of the parser is to ensure that the syntax of the input matches the rules described in the grammar. In other words, the parser ensures that source files that are input into the compiler are syntactically correct or that they conform to the language. For instance, the following English sentence is syntactically correct, "John dangles the sky." The sentence has a subject, a verb, an object and they are in the correct order of an English sentence. Of course, the sentence does not make sense. That does not matter to the parser; the parser checks that the input has the correct form of a ProcessJ program.

To give some background and understand the underpinnings of the ANTLR language, we will start with context-free grammars (CFG) and EBNF. Digging a little deeper, the LL(*) grammar (pronounced LL-Star), as well as top-down recursive-descent parsing, will be explained. The ANTLR rules in the ProcessJ grammar will be introduced as well as the parser implementation, `~.ProcessJParser`.

The following definition is from Compilers [ALSU07]. A CFG consists of terminals, non-terminals, a start symbol, and productions.

1. *Terminals* are the basic symbols from which strings are formed.

2. *Non-terminals* are syntactic variables that denote sets of strings.

3. In a grammar, one non-terminal is distinguished as the start symbol, and the set of strings it denotes is the language generated by the grammar. Conventionally, the productions for the start symbol are listed first.

4. The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of:

   (a) A non-terminal called the head or left hand side (LHS) of the production; this production defines some of the strings denoted by the head.

   (b) The symbol ':' which separates the left side from the right side.

   (c) A body or right hand side (RHS) consisting of zero or more terminals and non-terminals. The components of the body describe one way in which strings of the non-terminal at the head can be constructed.

To formally describe a context-free language (CFL), a domain specific language is necessary to describe a CFG. This language for describing languages is a meta-language called EBNF. The ANTLR grammar language is based on EBNF, and is derived from YACC [Joh79], where rules begin with a lowercase letter and token types begin with an up-

```
a : b c;
b : ‘b’;
c : ‘c’;
```

Figure 4.14: Sample Parser Rule.

```
a : ‘a’ b
  | ‘a’ c
  ;
b : ‘b’;
c : ‘c’;
```

Figure 4.15: LL(2) Parser Rule.

percase letter [Par07]. Other features of the language include optional elements, repeated elements, and parenthesized groups of grammar elements, called sub-rules.

Grammars defined in ANTLR need to be LL(*) grammars. An LL(*) grammar parses the input from left to right, and always takes the left-most derivation. For instance, consider Figure 4.14, where a production `a` has two non-terminals, `b` and `c`, on its RHS. In an LL grammar, `b` will always be derived before `c` because `b` is the left-most non-terminal.

Now that we know what the LL in LL(*) stands for, let us examine what the ‘*’ means. The most basic form of an LL grammar is LL(1). An LL(1) parser will only look ahead one token before the input is consumed. That means that if a rule has two alternatives, as in Figure 4.15, and the first token is common between both alternatives, the parser cannot tell which alternative is the correct one to take. This grammar is non-deterministic with only one look-ahead. In the case of Figure 4.15, the terminal ‘a’ is the first token; however, without looking at the next token, the parser cannot tell if it should execute `b` or `c`.

In order for the parser to recognize the language described in Figure 4.15, either the grammar needs to be left factored or it needs to be termed as LL(2) instead of LL(1). In the first case, we would need to factor out the common sub-expression ‘a’ into a new rule as in Figure 4.16. The latter case means the parser can use a look-ahead of no more than two tokens.

The general form of this increase in look-ahead is LL(k) where k is some predefined finite number. Utilizing a larger look-ahead can make the grammar simpler while maintaining linear time complexity. Unfortunately, the look-ahead needs to be finite, and it needs to be

```
a : ‘a’ d;
b : ‘b’;
c : ‘c’;
d : b
  | c
  ;
```

Figure 4.16: Left Factored Parser Rule.

```
def : modifier* classDef
    | modifier* interfaceDef
    ;
```

Figure 4.17: Rule That Cannot Be Parsed With `k` Look-Ahead.

known when the compiler is written, not at run time. Consider a grammar example from The Definitive ANTLR Reference [Par07], as shown in Figure 4.17. The `def` rule has two alternatives, and both begin with zero or more modifiers. An LL(k) parser cannot recognize this rule because it is not possible to determine the number of look-aheads the parser will need. Therefore, it is considered non-deterministic for LL(k).

An LL(*) parser generator, like ANTLR, does not need to specify k. It uses an arbitrary look-ahead by generating a DFA for the look-ahead language to predict the correct alternative. It is not without limitation, the look-ahead language cannot have unreachable states, no dangling states, and at least one accept state for each alternative [Par07]. In order to implement the DFA look-ahead, ANTLR generates a syntactic predicate.

Using LL(*) increases the expressiveness of the grammar, but it is no silver bullet. LL(*) grammars can only predict with a look-ahead that can be expressed as a DFA. That means the decision is only as powerful as a regular expression. Consider the grammar in Figure 4.18, which is used in the Definitive ANTLR Reference [Par07]. It is not possible to predict which alternative to choose because it is not possible to construct a DFA that matches parentheses around a recursive rule due to the pumping lemma [Sip05].

In cases where it is not possible to use a DFA to predict, ANTLR can uses a syntactic predicate to gate the alternatives. ANTLR generates a check before entering the alternative, which uses a mini-parser to check if the alternative fits. When backtracking is enabled, as in the ProcessJ grammar, the auto-backtracking feature automatically generates syntactic

```
s  :  e ‘%’
   |  e  ‘!’
   ;

e  :  ‘(’  e  ‘)’
   |  INT
   ;

INT  :  ‘0’..‘9’+  ;
```

Figure 4.18: Not LL(*).

predicates where it is necessary.

It should be noted that even though ANTLR uses many strategies to make the grammar more expressive, and also can use an arbitrary look-ahead, it still cannot handle left recursive grammars. This limitation is determined by the LL nature of the parser. It is not possible to generate an LL parser from a grammar that has left recursion because it would send the parser into an infinite loop.

LL grammars are synonymous with top-down parsers. Top-down parsers begin with the start rule and attempt to predict which subsequent rules to match. There are two variations of the top-down parser: table driven and recursive descent. In table-driven parsing, the parser uses an explicit stack and a look-up table to match input terminals to non-terminals. However, in recursive descent, the parser uses an implicit stack, called the activation stack, and directly matches grammar rules to methods.

In 1980, Sir Anthony Hoare was presented the Turing Award for his fundamental contributions to the definition and design of programming languages. In his acceptance speech, he said of top-down recursive descent parsers [Hoa81]:

> I can still recommend single-pass top-down recursive descent both as an implementation method and as a design principle for a programming language. First, we certainly want programs to be read by *people* and people prefer to read things once in a single pass. Second, for the user of a time-sharing or personal computer system, the interval between typing in a program (or amendment) and starting to run that program is wholly unproductive. It can be minimized by the high speed of a single pass compiler. Finally, to structure a compiler according to the syntax of its input language makes a great contribution to ensuring its correctness. Unless we have absolute confidence in this, we can never have confidence in the results of any of our programs.

Sir Anthony Hoare mentions three distinct advantages to top-down recursive descent parsers in his speech; they are human readable, the operate at high speed, and the match between the parser code and the grammar ensures correctness. Each of these points deserves greater attention.

First, the parser generated by ANTLR is human readable. It is helpful to step through the execution of the compiler to understand exactly what it was doing. It is perfectly possible to look at the generated code and understand what is happening.

Sir Anthony Hoare's second point was regarding performance. This being my initial iteration of the compiler, and the first compiler I have ever written, my focus has not been on performance so much as on getting it to work and to make the implementation as easy to understand as possible. The next iteration of the compiler will have a baseline set and improved performance.

The final point Sir Anthony Hoare made was that structuring the code according the the syntax of the language helps ensure correctness. ANTLR conforms two-fold to this point. First, the code is generated directly from the specification, so the code cannot help but conform to the grammar. Second, there is a one-to-one mapping between the grammar rules and the parser methods. Given these two features of ANTLR, the ProcessJ parser conforms to Sir Anthony Hoare's recommendations.

4.2.4 Error Reporting & Recovery

In addition to the benefits of using a top-down recursive descent parser, as mentioned previously, the parser also has excellent error diagnostics [FCL09]. Before work began on the ProcessJ grammar, my adviser, Dr. Pedersen had an implementation of the ProcessJ grammar, using the CUP[Hud05] LALR parser generator. He lamented the difficulty of providing meaningful error messages.

According to Parr [Par07], "The quality of a language application's error messages and recovery strategy often makes the difference between a professional application and an amateurish application." Error reporting and recovery was a major benefit in ANTLR's favor when deciding to switch from CUP.

Since error reporting is so crucial to a language's success, the ProcessJ compiler had

```
<<rule-name>>
@init{ setCurrentParaphrase("<<message to indicate rule>>");}
@after{ removeCurrentParaphrase();}
    : << RHS >>
    ;
    catch [RecognitionException e] {
      reportError(e);
      <<RECOVER>>
    }
```

Figure 4.19: General Form of Rule with Error Reporting.

to have the capability to give meaningful feedback to the user. Figure 4.19 indicates the general form of an ANTLR rule supporting error reporting.

The rule begins with an `init` annotation. Inside is a block of code that ANTLR will execute before matching the rule. A stack was maintained for error reporting. Before each rule begins, a phrase is pushed onto the stack that indicates what type of rule the parser is in. In this way, an intuitive error is generated that can express what the parser is attempting to match. For instance, it could give an error that looks like the following, "`line 1:12 mismatched input ';' expecting ')' in expression`."

If the input does throw a `RecognitionException`, the rule will report the error by means of the `reportError` method. Taking the advice of Fowler [FP10], the Notification pattern, described in Section 5.8, was used to separate the error collection logic from the parser. Separating the error collection also has the benefit of code reuse, because the notification is used in the semantic analysis part of the pipeline as well.

Regarding ANTLR's error recovery strategies, ANTLR has automatic error recovery based on ideas from Wirth [Wir78], Topor [Top82], and Grosch [Gro90]. There are three strategies: single token insertion, single token deletion, and scanning for the following symbols.

Single token insertion involves inserting a symbol where it makes sense so the parser can continue. Think of entering a complex mathematical formula, one involving several nested parentheses. If you make a mistake and enter one too few closing parentheses, the parser can report an error saying you are missing a parenthesis, and will specify the line and position where it should be entered.

```
int x = (m * (x + b)));
```

Figure 4.20: Single Token Deletion.

```
statement
@init{ setCurrentParaphrase(”in statement”);}
@after{ removeCurrentParaphrase();}
    :   channelCommWriteStatement
    |   ...
    |   whileStatement
    ;
    catch [RecognitionException e] {
      reportError(e);
      consumeUntil(input, SEMI);
      input.consume();
    }
```

Figure 4.21: Panic Mode Recovery.

On the other hand, single token deletion is useful if you enter too many closing parentheses. Consider the input in Figure 4.20. In this case, the parser recognized that there was an extra parenthesis, removed it, and moved on.

The last strategy used in error recovery is to scan the input for following symbols. This could be seen as a form of panic mode [ALSU07]. In Figure 4.21, for example, if there is an error recognizing the input, then the error is reported, and tokens are consumed until the parser sees a semicolon indicating the end of the statement.

Given the lexer is able to successfully generate each of the tokens, and the parser is able to recognize the input without error, the final result of the syntax analysis phase is an AST.

4.2.5 Abstract Syntax Tree

The previous section did not mention one of the most important responsibilities of the parser. It mentions the theory of a parser, how to implement that theory, and how the parser reports and recovers from errors. However, one of the primary responsibilities of a parser is to take a one-dimensional stream of tokens and convert it into a hierarchical syntactic structure of the source program [ALSU07], called an AST.

An AST represents the source program during semantic analysis; it represents the rela-

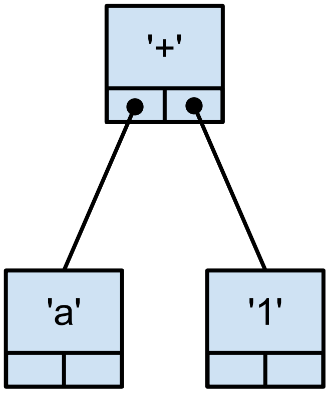

Figure 4.22: Homogeneous AST.

tionships between tokens, or nodes. The AST also is a repository for program meta-data later in the pipeline.

This section will first discuss tree structure patterns, how the compiler represents a tree in memory, and the structure of the tree. Once the tree is structured, the tree can be built from the token stream. Finally, this section will describe the AST class implementation in ProcessJ.

Tree Structure Patterns

There are three tree structure patterns that were considered when designing the AST: Homogeneous AST (HAST), Normalized Heterogeneous AST (NHAST), and Irregular Heterogeneous AST (IHAST). Each pattern is slightly different from the next, and each pattern has its pros and cons.

In a HAST, all nodes are of the same type. The only difference between instances of a node is the token they encapsulate. Using only one node type has the benefit of uniformity, which makes walking the nodes easy. Each node has a collection of children as well as a token. This is easy to learn and easy to use.

From Figure 4.22, it is possible to visualize a HAST. All elements have the same shape, and all elements have a list of children. The only way to distinguish one from the other is by means of the token.

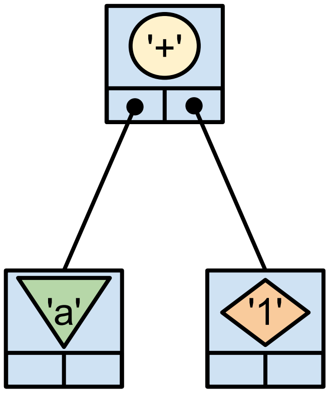

Figure 4.23: Normalized Heterogeneous AST.

The disadvantage of using a HAST comes with accessing a specific child or storing information about a node. There is one uniform way to access children, as a collection; after that, you need to search the collection to find a specific child. As an alternative to searching for a child, you can access it from a specific index. However, this approach is also flawed. Code throughout the project needs to know the location of the child. Similar situations happen to data regarding a node: spreading the logic around like that breaks encapsulation.

A HAST might be acceptable for simple languages, but ProcessJ needs something a little more powerful. The Normalized Heterogeneous AST is similar to the HAST, in that it implements a common interface. As a result, a client can treat the tree as a HAST, and it can encapsulate node-specific data members by using different node types. Each node type is implemented by its own class. In this way, it is possible to centralize access to node specific data.

Figure 4.23 attempts to visualize a NHAST. Each element is implemented with its own type, depicted by varying internal shapes and colors. The external of each shape is still the same. Every element has a list of children, a token and so forth.

Like the HAST, the NHAST also uses a uniform method for child access. However, unlike the two previous tree patterns, the IHAST uses an irregular child list representation. In other words, in an Irregular Heterogeneous AST a different class is used for each node

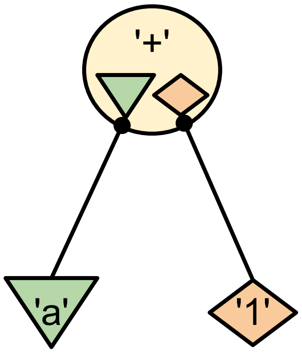

Figure 4.24: Irregular Heterogeneous AST.

type, similar to the NHAST; however, there is no uniform method for child access. Children are accessed by way of accessors and mutators, which encapsulates child access. This pattern also encapsulates node specific data by using a separate class for each node type.

Figure 4.24 depicts an IHAST. Each element has its own shape and color, and none have a uniform appearance. The parent element, a binary expression, has exactly two slots for children, each with an expected shape and color. The parent knows exactly how many children it has, and any client that needs to use that element needs to know the application programming interface (API) of that element.

The lack of any uniform treatment of AST nodes makes an IHAST difficult to use and hard to learn. Since no single pattern has good encapsulation nor uniform access, the ProcessJ compiler uses a combination of NHAST and IHAST. Each node type has its own class implementation, and they all implement the same interface, just as in the NHAST. However, in addition to the uniform method of child access, accessors and mutators were added for child-specific access. In this way, it allows clients to access all children in a uniform manner, and allow clients that need access to specific children to do so without spreading child access logic all over the place. Now that we know how to structure the AST, let us look at how the tree is built.

```
variableDeclaratorId
    :   IDENT^  arrayDeclaratorList?
    ;
```

Figure 4.25: Example of Inline Tree Rule.

```
variableDeclaration
    : modifierList  type  variableDeclaratorList  SEMI
      -> ^(VAR_DECLARATION  modifierList  type  variableDeclaratorList  )
    ;
```

Figure 4.26: Example of Tree Rewriting Rule.

Tree Building

Before delving into the specifics of how nodes are created and related to their children, this section starts with a general overview of the tree building and rewriting aspects of the ANTLR grammar. After describing the grammar, the `TreeAdaptor` will be explained. Finally, the ~`.ProcessJParser` implementation will be described.

ANTLR offers great flexibility for tree construction, and is defined directly in the grammar. Each grammar rule has several options on how to structure its result. As an introduction, in Figure 4.25, the `variableDeclaratorId` rule has two components, `IDENT` and `arrayDeclaratorList`. In this rule, the `IDENT` token becomes the root of the tree through the use of the '`^`' operator; the `arrayDeclaratorList` is made optional by the '`?`' operator. When the term 'the token becomes the root of the tree' is used, what is meant is that the '`^`' operator indicates that ANTLR is to create a node, and place the other elements of the rule as children of the new node. The result of running the `variableDeclaratorId` rule is a new `ProcessJTree` that has zero or one of the results of the rule `arrayDeclaratorList`.

In order to understand what is going on, let us look at another example in Figure 4.26 that is more explicit. This example shows a list of rules and tokens that it expects to match. After the parser has matched the input, it does a tree rewrite, as indicated by the '`->`' operator. Everything to the right of the '`->`' operator indicates how to restructure the input. This rule indicates that a new node is created with the `VAR_DECLARATION` token as the root. The `modifierList`, `type`, and `variableDeclaratorList` elements are added as

```
variableDeclaratorList
    : variableDeclarator (COMMA variableDeclarator)*
      -> variableDeclarator+
    ;
```

Figure 4.27: Example of Exclusion and Collecting Input Elements.

children, and the `SEMI` token is not included in the tree. The tree construction in the rule was written this way because it clearly shows how the tree structure will look after the rule completes.

There are many more ways to rewrite the input tokens into trees in ANTLR. However, I would like to give an example of consolidating a complex statement into something that is simpler, as in Figure 4.27. In this case, the only thing that is important is the `variableDeclarator`s in the rule. The `COMMA` is just syntactic sugar, and has no semantic meaning. The `COMMA` helps separate the `variableDeclarator`s, but after that point, they are no longer necessary. All that is necessary is a list of these tokens; ANTLR allows us to covert that complex set of tokens into a simple list of one or more `variableDeclarator`s.

Now that we have a basic understanding of how the grammar describes tree construction, let us look at how the nodes are created.

Creating nodes is best viewed by thinking of the structure first. Since the chosen structure of the AST is a hybrid NHAST / IHAST, we can look at it from the perspective of a NHAST. In a Normalized Heterogeneous AST, each node is implemented by different classes, but all the children implement a common interface. It also is important to keep the tokens decoupled from the implementation class and centralize the node creation logic. The solution is to create a new class, with the responsibility of knowing the conversion from the token type to node implementation and returning the normalized type, just as in the Factory pattern described in Section 5.4.

The implementation of the factory is the ~`.ProcessJTreeAdaptor`. The main method is the `create` method which takes a `Token` and returns a `ProcessJTree`. The method uses the token type to determine which class to create. This is one of the cleverer ways how ANTLR builds ASTs. The `create` method needs only declare the type of node to create.

The other methods of tree building are also delegated to the tree adapter, although there is a default implementation in the base class provided by ANTLR.

```
modifierList
    :   modifier*
        ->  ^(MODIFIER_LIST modifier*)
    ;
```

Figure 4.28: Example of Imaginary Token.

```
procedureTypeDeclaration
@init{ setCurrentParaphrase(”in process declaration”);}
@after{ removeCurrentParaphrase();}
    :  modifierList PROCESS voidableType
         IDENT formalParameterList (block | SEMI)
       ->  ^(PROCESS[\$IDENT] modifierList
              voidableType IDENT
              formalParameterList block?)
    ;
```

Figure 4.29: Example of a Rule With Multiple Variable Length Lists.

The lack of control in the node creation can be frustrating. There is no context attached to the token, and the tree adapter does not know what rule it is in. However, by using imaginary tokens, the context is not necessary. Imaginary tokens are the tokens that are defined in the ‘tokens’ section of the header, and that do not have an associated pattern or character literal. They are used to indicate something that the concrete tokens cannot indicate.

Consider the rule in Figure 4.28. In rule `modifierList`, a new node is created for `MODIFIER_LIST` and each of the specified modifiers are added to it as children. When there are a variable number of elements, such as the modifier list, a new node is created to hold them. By always creating a node that holds modifiers, the node is able to encapsulate how the modifiers are accessed.

Another benefit to encapsulating a variable length list in its own node, such as the modifier list, is that any element that contains a list will be able to know where the next element is without searching. Searching would get particularly problematic with rules like `procedureTypeDeclaration` as seen in Figure 4.29. In the `procedureTypeDeclaration`, there are two variable length lists, each of different types. It is much easier to implement them as their own node to allow the `PROCESS` node to know where it can access each child.

Now that we know what the tree construction looks like from the grammar side, and

```
sync : SYNC^ parenthesizedExpression
     ;
```

Figure 4.30: Grammar rule for `sync`.

```
root_0 = (ProcessJTree)adaptor.nil();
SYNC425=(Token)match(input,SYNC,FOLLOW_SYNC_in_sync9737);

if (state.failed) return retval;
if ( state.backtracking==0 ) {
  SYNC425_tree = (ProcessJTree)adaptor.create(SYNC425);
  root_0 = (ProcessJTree)adaptor.becomeRoot(SYNC425_tree, root_0);
}
pushFollow(FOLLOW_parenthesizedExpression_in_sync9740);
parenthesizedExpression426=parenthesizedExpression();

state._fsp--;
if (state.failed) return retval;
if ( state.backtracking==0 ) {
  adaptor.addChild(root_0, parenthesizedExpression426.getTree());
}
```

Figure 4.31: Partial Implementation of the `sync` Rule.

how it knows what type of node to create, let us look at the simple rule `sync`, shown in Figure 4.30, and a portion of the code generated from that rule, as seen in Figure 4.31. The `sync` rule has only two elements, the `SYNC` token and the `parenthesizedExpression` element.

To start, a nil root is created; this is a non-null node that is empty, but can hold children. The implementation then matches the `SYNC` token, checks if it was successful, and has the adapter create a new node. Next, it makes the `SYNC` token into the root node, matches the `parenthesizedExpression`, again checks for failure, and finally adds it as a child to the root.

We now know how ANTLR defines tree rewrite rules, how nodes are created, and how the generated implementation code works. The next section will cover the AST implementation.

ProcessJTree

Finally, we come to the implementation of the AST. The `~.ast.ProcessJTree` is the interface that all AST nodes implement. A class diagram of `ProcessJTree` is displayed in

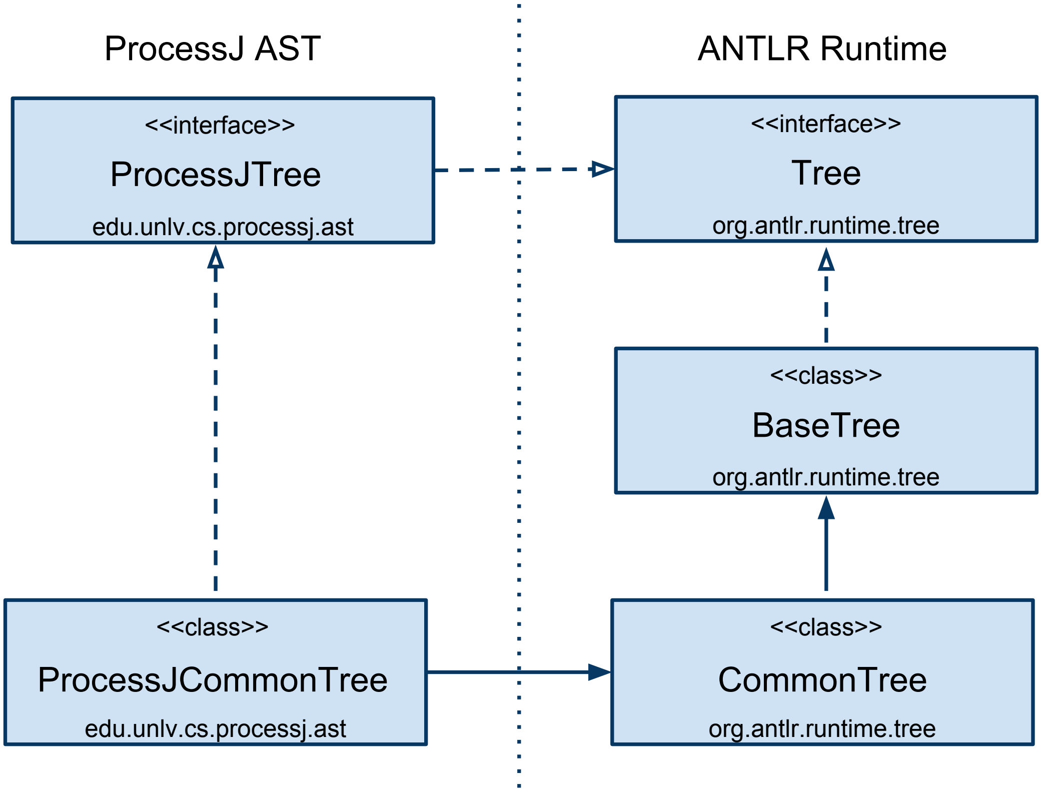

Figure 4.32: ProcessJ AST Class Diagram.

Figure 4.2.5. Here we can see the static relationships between the ProcessJ implementation and the ANTLR run-time.

From the diagram, it is easy to see how `ProcessJTree` extends the ANTLR run-time. The `ProcessJTree` interface extends two interfaces: `Visitable` and the ANTLR run-time interface `Tree`. Also in the diagram are the implementation classes, `ProcessJCommonTree` and its ancestors `CommonTree` and `BaseTree`. The important thing to take away from these relationships is that the interface within the project to use is the `ProcessJTree`, and the base implementation is the `ProcessJCommonTree`.

Throughout the project, the impact from the ANTLR run-time was minimized. The team had already switched the lexer and parser generator once. The `ProcessJTree` interface was used in order to isolate that design decision, and the `ProcessJCommonTree` is extended as a base implementation. None of the code uses the ANTLR run-time classes, with the exception of the tokens. The ANTLR tokens are a convenient implementation, and never needed to be extended.

Examining the AST class hierarchy, the `Visitable` interface defines only a single method with signature *void acceptVisitor(Visitor v).* Extending this interface allows a node to participate in the Visitor Pattern, as described in Section 5.3. Since the `Visitor` type is an interface, nodes that are `Visitable` and `Visitors` are loosely coupled. In order to have multiple strategies for implementing visitors, the AST was designed to know as little about them as possible.

To extend the functionality that already exists within the ANTLR run-time, `ProcessJTree` extends the `Tree` interface. Although the `ProcessJTree` extends the ANTLR run-time `Tree`, using `Tree` directly in the code was avoided to minimize the ANTLR dependency. In this way, the front-end strategy of the compiler can be changed at a later date. It may be wise to further decouple the `ProcessJTree` from the ANTLR run-time by using the Adapter Pattern [GHJV95], where `Tree` is needed instead of extending `Tree`.

Having all nodes in the AST implement `ProcessJTree` has two key benefits. First, it allowed implementation of either a HAST (Section 5.5) or a NHAST (Section 5.6). Second, parents of the tree are decoupled from their children.

The implementation of the AST is a good closing point to this section on syntax analysis,

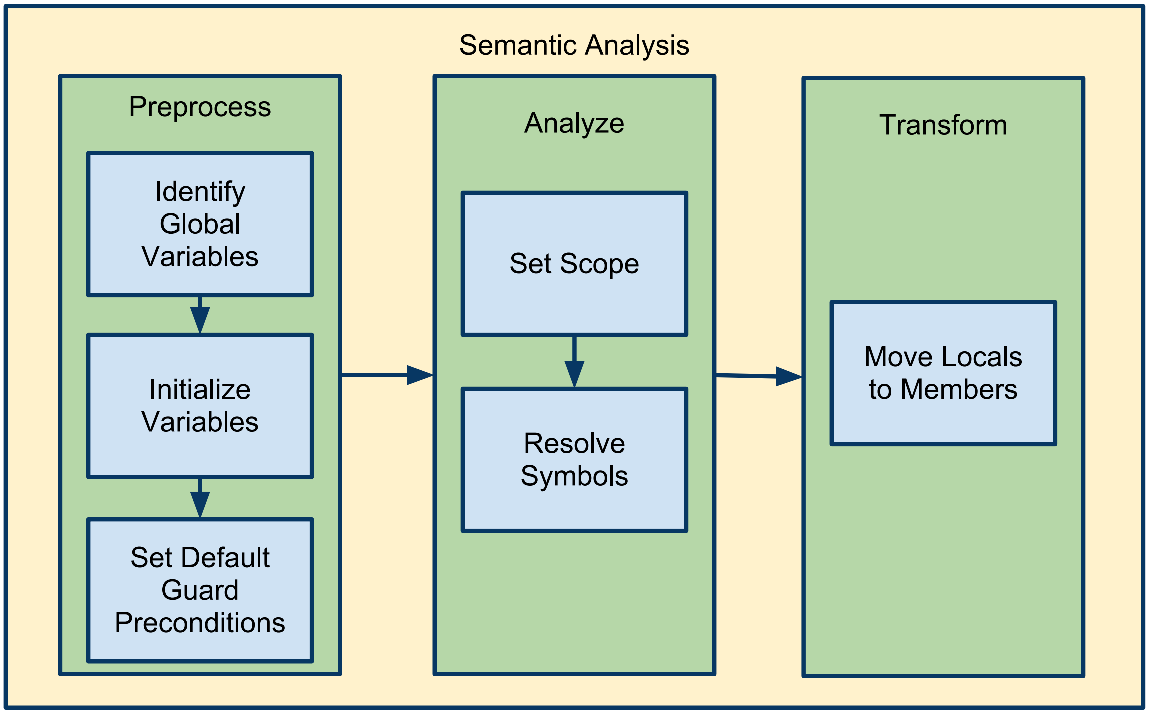

Figure 4.33: Semantic Analysis Pipeline Phase.

since the AST is the result of the syntax analysis phase and is used throughout the semantic analysis phase.

4.3 Semantic Analysis Phase

Semantic Analysis is the third phase of the ProcessJ compiler pipeline. In the previous stage of the pipeline, syntax analysis, the compiler checks that the input program has the correct grammatical structure. However, it does not check that the input program makes any sense. The syntax analysis phase also builds an AST.

It is the responsibility of the semantic analysis phase to ensure, to the best ability of the compiler, that the input program 'makes sense,' and to prepare the AST for output. This section will first go over each of the sub-phases of the semantic analysis phase, then turn to the implementation.

The semantic analysis phase is for all the checks of things that 'make sense,' and the phase that prepares the AST for output. The semantic analysis phase was split into three sub-phases: preprocess, analyze, and transform.

Important architectural decisions are also covered in this section. There are three main topics for implementation of the semantic analysis phase: the `Visitor`, tree walking, and actions.

4.3.1 Preprocessing

In the preprocessing sub-phase, the AST is prepared for the analysis phase. By ‘prepared’, is meant that the abstract syntax tree is made more uniform, and certain features of the tree are identified. Three tasks of this sub-phase are identifying global variables, initializing variables, and setting default values.

Identifying global variables is the first task of the preprocessing sub-phase. The term ‘global variable’ is kind of a misnomer. ProcessJ allows for package-visible constant values. Since ProcessJ is process-oriented, the team wanted a way for processes to refer to constant values across multiple process without having to send the value back and forth.

The process of identifying these elements of the AST is rather easy. Since ProcessJ has a limited number of top-level elements, as the program explores the AST, the variable declarations at the top-level are marked as global. Any variable declaration that is not explicitly set as global is implicitly local; since the global variable declarations are at the top of the AST, the sub-phase need only look at the top-level elements. Looking at only top-level elements saves the phase a great number of elements in the whole input source.

The second sub-phase of the preprocessing phase initializes variables. ProcessJ does not require initialization of some types; the compiler automatically initializes variables for these types. For instance, variables of barrier, channel, and timer types do not need initialization. However, for the sake of uniformity in the analysis phase, this sub-phase initializes these variables.

Finally, default values are set for certain elements of the AST. For now, the only instance where this need happen are ∼`.ast.Guard` nodes. Guard elements have preconditions; if no precondition is set in the source, a default of `true` is set. Although this is a simple task, other language features might be adopted later that may need default values set in the AST. After the AST has been preprocessed, it is ready to be analyzed.

4.3.2 Analyze

Analysis is what most people think of when it comes to the semantic analysis phase. This phase has three main tasks during this early stage in the development of the ProcessJ compiler: scope definition, symbol definition, and type checking. These three tasks are the quintessential semantic analysis tasks for a statically typed language like ProcessJ.

The scope of a variable is the range of statements in which the variable is visible; in other words, the range from which the variable can be referenced [Seb07]. ProcessJ is a statically scoped language. In a statically scoped language, the compiler is able to figure out where it is legal to reference a variable. Static scoping allows the compiler to find logical errors in source programs. For instance, if a variable `x` is used in some block, but `x` is never given a value or it is declared later in the code, it is ambiguous what `x` means at that point.

By tracking scopes, it is possible to resolve symbols. A symbol is an identifier. A symbol could be anything that is named, such as a label, process, record, variable, or the type of a variable. It is necessary to find where the symbols are declared or defined in order to link the use of elements with the declarations and to link the declarations to the definitions.

The difference between a symbol declaration, definition, and reference could be confusing; however, it is important to understand the different forms a symbol can take.

A symbol *reference* is used when a symbol is referenced in an expression. For instance, in the expression '`x / y`', `x` and `y` are both variables that are symbol references. To know what this expression means, we need to look at the *declaration* of each variable.

That same expression, '`x / y`', could have different results, depending on how the variables were declared. The expression would be different if only one variable were declared as an integer than if both variables were integers. The declaration assigns a type to a variable.

ProcessJ has user-defined types. For instance, a record is a user-defined data type. A record is given a name, and there are several variables that could be associated to that data type, as in Figure 4.34. For example, the symbol `Point` is defined as a pair of integers, `x` and `y`.

Ultimately, symbol references are resolved so they can be type checked. ProcessJ is a statically typed language, so type checking is done during compile time. When a symbol is used in an expression along with other symbols, the compiler checks if the combined

```
record Point {
  int x;
  int y;
}
```

Figure 4.34: Example Record Definition.

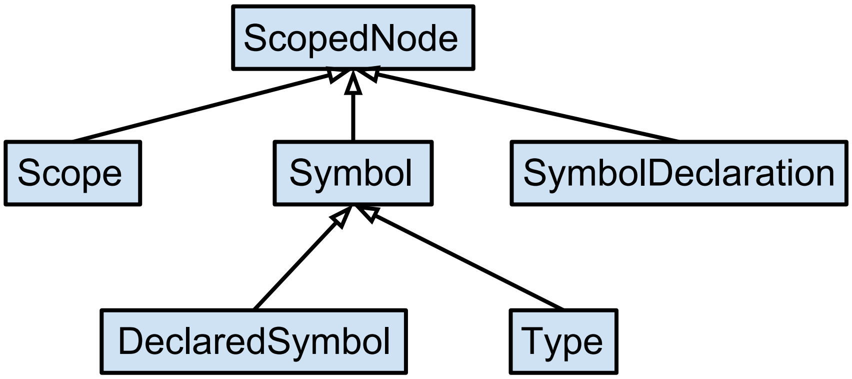

Figure 4.35: `ScopedNode` Class Diagram.

expression is allowed. After a reference is associated to its declaration, and all declarations are associated to definitions, it is possible to resolve a reference to a definition or type. At that point, it is possible to determine if the combinations of expressions used in a source program are legal. For instance, an expression '`p() + x`', could be legal or illegal, depending on how `p` is defined and how `x` is declared. If `p` were defined as `void`, the expression would not make any sense and so it would be illegal.

To understand the implementation of this phase, abstractions and how they are used will be described. Abstractions are how the semantic constructs of scope, symbol, declaration, and definition are represented in the abstract syntax tree. Then, we will look at how each of these abstractions are used to analyze the AST.

Five interfaces in the ~`.ast` package abstract the semantic constructs for the analysis sub-phase: `Scope`, `ScopedNode`, `SymbolDeclaration`, `Type`, and `Symbol`.

The `Scope` interface represents a static scope. Packages, records, protocols, protocol cases, processes, blocks, for loops, etc., are examples of static scopes. Each are a place where symbols can be referenced, defined, or declared. The `Scope` has methods for defining and resolving symbols as well as importing symbols from other packages. Although there are many implementations of Scope, they inevitably delegate the definition and resolution of symbols to the SymbolTable class.

The ScopedNode represents any element that is within a scope and might need to resolve a symbol. For instance, all Symbols are ScopedNodes because they always need to be resolved. Other examples are symbol declarations, type definitions, and expressions. ScopedNode ensures two methods, getEnclosingScope and setEnclosingScope. It is convenient for elements to have uniform access to their enclosing scope.

A SymbolDeclaration declares a symbol. There are several places where a symbol can be declared: formal parameters, local variables, protocol members, and record members.

Next, a Type represents a symbol definition. There are two sorts of Type: built-in and user-defined. User-defined types are records, protocols, protocol cases, and procedures. In the case of user-defined types, the type is an explicit symbol definition. In the case of a built-in type, the symbol is defined in the language. Barriers and timers are two examples of built-in types. There is no code to look at and see the definition; however, their semantics are built into the language.

Finally, the Symbol represents a name. A symbol just abstracts a string that can be used to define or resolve something.

Since Scope is the center of symbol resolution, the following is a brief overview of how the Scope works with the other abstracts to provide symbol resolution. Figure 4.36 depicts the collaborations of the Scope class.

A Scope is a ScopedNode, which is to say, a scope has an enclosing scope. The only scope without an enclosing scope is the global scope. From its definition, a ScopedNode has an enclosing scope. Symbols are used to define and resolve Types and SymbolDeclarations within a scope. For instance, while tree walking, a NameExpr is visited, and the compiler needs to determine its type. The current scope is queried through the resolve method, and the SymbolDeclaration is found. With the SymbolDeclaration, the type symbol can then be queried to find the declared type from the scope.

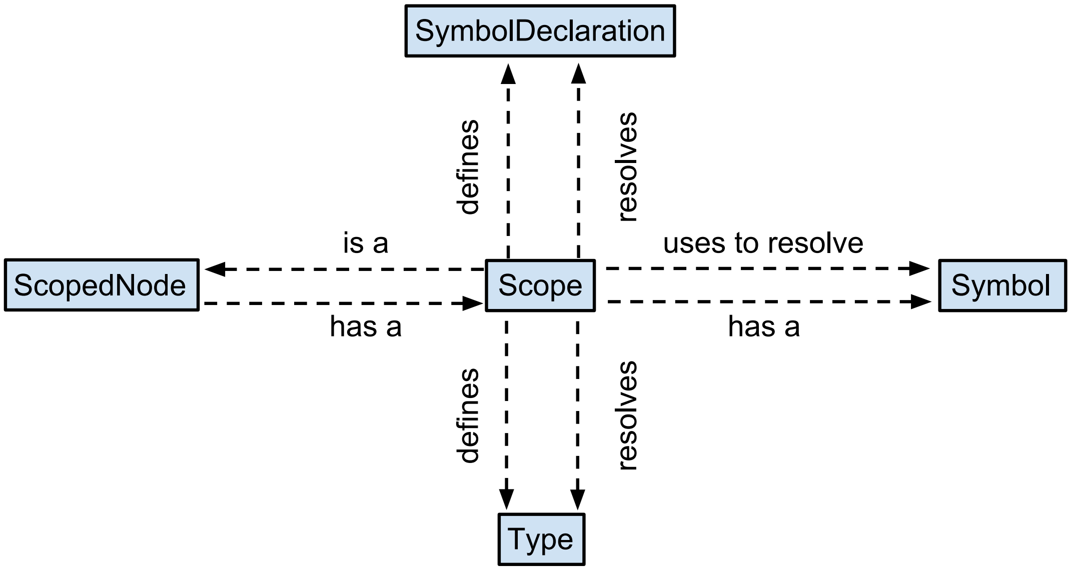

Figure 4.36: Collaborations of `Scope`.

4.3.3 Transform

The final semantic analysis sub-phase transforms the AST. After the analysis phase, we can do what we like to the AST to make it useful for output, more efficient, or to implement some feature of the language. Currently, only one transformation of this sort exists; however, there are many transformations required to support mobile processes and polymorphic resumption interfaces.

The article Mobile Process Resumption in Java Without Bytecode Rewriting [SP11], describes a method to allow mobile processes to transparently suspend and resume in the same control and variable state by manipulating the AST. Only one of the required transformations is currently implemented. The transform takes all local variables and makes them fields in the resulting Java source.

Semantic Analysis is composed the three sub-phases: preprocessing, analysis, and transformation. This next section will explore how these tasks are accomplished.

4.3.4 Actions

There is a great deal of logic in the semantic analysis phase. Separation of concerns is a major concern for tasks during semantic analysis. It is all too easy to start adding responsibilities to a class simply because the data needed is at hand in the current method. It is also difficult to centralize logic when it can be distributed across all the classes that comprise the AST.

Luckily for compiler writers, the Gang of Four defined a suitable solution to the problems that plague semantic analysis in their book Design Patterns [GHJV95]. In the Visitor pattern (Section 5.3), each bit of logic can be centralized into highly specialized classes. The Visitor then traverses the AST and executes that logic across it.

Although the ProcessJ compiler does use the Visitor pattern, it implemented in the usual manner. The `Visitor` interface defines the contract required to iterate the AST. This was left open to implement the visitor pattern differently in the future. For now, there is one implementation, `ActionVisitor`.

The `ActionVisitor` encapsulates the tree walking, and takes a collection of `Actions` to execute as it traverses the AST. An `Action` represents some logic that is executed as a node is visited. Tree walking is the process of traversing the AST. The actions are clear and their executions are uniform. It is easier to understand how all the actions work once familiar with how one of them work.

There are two methods for each element in every action: `pre` and `post`. The `pre` method is called before the element is considered visited, and `post` is called after. In the base action, `AbstractAction`, every method calls one of two methods, `preCatchAll` and `postCatchAll`. The 'catch all' methods are there for convenience.

Sometimes the same exact action needs to be performed on all elements. For instance, in the `LogAction`, as shown in in Figure 4.37, a message is logged and the indent is increased before an element is visited. After the element is visited, the indent is decreased. However, it is possible to write element-specific code, as in the `Source` element in the `LogAction`. The `Source` element needs special treatment in the log action because the current filename is stored.

It may be confusing when to choose between `pre`, `post` and the catch all methods. The

```
public final class LogAction
  extends AbstractAction {
  ...
  @Override
  public void postCatchAll(final ProcessJTree tree) {
    indent -= 2;
  }

  @Override
  public boolean preCatchAll(final ProcessJTree tree) {
    final StringBuffer message = new StringBuffer()
      .append(getIndent())
      .append(getInfo(tree));

    LOG.info(message);

    return VISIT_CHILDREN;
  }

  @Override
  public boolean pre(final Source tree) {
    final StringBuffer message = new StringBuffer()
      .append(getIndent())
      .append(tree.getFileName())
      .append(' ')
      .append(getInfo(tree));

    LOG.info(message);
    this.filename = tree.getFileName();
    return VISIT_CHILDREN;
  }
  ...
}
```

Figure 4.37: Sample of `LogAction`.

`pre` method has the benefit of telling the action visitor it is no longer necessary to visit descendents of the current node. Consider a case, such as the `SetGlobalVariableAction`, where the maximum depth of the tree traversal is known; the number of elements processed can be decreased, which will speed the compilation. On the other hand, the `post` method is simpler because no return value is necessary. In general, when choosing between implementing a method in `pre` or `post`, it is recommended to only use `pre` in situations where something *needs* to happen before visiting children. If it is possible to implement using `post`, then do so.

Along the same lines as choosing between `post` over `pre`, it is better to use the `preCatchAll` and `postCatchAll` methods where possible. Since the 'catch all' methods are applicable to all methods, there is less code to write; also, it is more uniform in application. However, just as in the `LogAction`, sometimes it is necessary to do some special processing for certain nodes.

In summary, for the semantic analysis phase, the main purpose is to check, as far as is statically possible, that the source program makes sense. The three sub-phases ultimately lead up to the last section of implementation, which is code generation.

4.4 Code Generation Phase

The final phase of the compiler takes the AST and writes Java code. The code generation phase has two sub-phases, translation and output. In the first sub-phase, the AST is used to generate strings of Java code. The translated strings are associated to the elements that they represent. After the source AST is translated, Java source files are generated.

4.4.1 Translation

After the syntax and semantics of a source program are verified, it is finally time to translate the source to the implementation language. The translation sub-phase walks the AST; after visiting all children, it applies a *template* to each element in order to translate it. The translated values are stored on the element so they are accessible by the elements parents.

The first iteration of the ProcessJ compiler is the simplest implementation that will work.

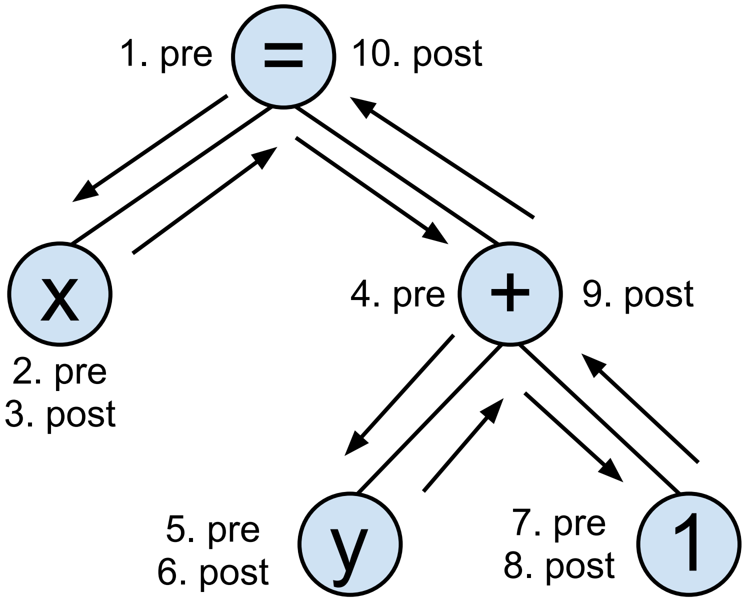

Figure 4.38: AST for Assignment.

Eventually, ProcessJ will be multi-targeted, although Java is the only target supported at present. Java was chosen as the first target, because the syntax of ProcessJ is based on the syntax of Java. There is almost a one-to-one correlation between ProcessJ and Java.

Since ProcessJ and Java are similar languages, the translation process usually is simple. For instance, variable references are output as their input values as well as the binary operators like '`+`' and '`-`'. A ProcessJ statement – such as '`x = y + 1`' – is a good place to start understanding how the translation phase works. A graphic representation of the AST for the assignment statement is depicted in Figure 4.38. The figure also shows the visitation order.

For a brief explanation of the AST, the LHS of the statement is a name; on the RHS is a binary expression of operator plus. The binary expression has a left operand and a right operand. The left operand is a name expression, and the right operand is a literal.

Translation happens during the `post` method; therefore, leaves are always translated before their parents. Leaves are always translated first for the sake of simplicity. In this case, the 'x' element is translated, first producing 'x;' then, the 'y' and '1' are translated.

```
binaryExpression(lhs,op,rhs) ::= <<
  $lhs$ $op$ $rhs$
>>
```

Figure 4.39: Binary Expression String Template.

```
@Override
public void post(final BinaryExpr tree) {
  final StringTemplate template = templates.getInstanceOf(
                ”binaryExpression”, ATTRIBUTES);
  ATTRIBUTES.clear();
  ATTRIBUTES.put(”lhs”, tree.getLeftOperand().getOutput());
  ATTRIBUTES.put(”op”, tree.getText());
  ATTRIBUTES.put(”rhs”, tree.getRightOperand().getOutput());
  tree.setOutput(template.toString());
}
```

Figure 4.40: Code to Translate a Binary Expression.

Since its leaves are already translated, the binary expression only needs to put them in the right place along with its own operator. Templates are used to put the output children in the correct order.

Templates are used to declare how the translated values are formatted. Rather than use Java strings, or string builders, to specify the output, a template engine called StringTemplate [Par04] was chosen. The StringTemplate template engine promotes a strict, model-view separation [Par09], which keeps the view and model are loosely coupled.

StringTemplate has a domain specific language for describing how to output data; a sample can be found in Figure 4.39. In the binary expression example, a template called ‘binaryExpression’ takes three parameters: the LHS, the operator, and the RHS. The code to use the template is depicted in Figure 4.40. The translator populates a map called `ATTRIBUTES` with the required parameters and their values, and tells the template to construct the string.

In terms of model and view, the AST is the model and the template is the view. By keeping the model and view loosely coupled, multiple target languages could eventually be supported. The only changes necessary would be to specify all the required templates in the new target.

After building the translated values from leaves up to the root of the AST, the compiler is ready to save the completed Java source to disk.

4.4.2 Output

The final task of the ProcessJ compiler pipeline is to save the translated values as Java source files. Two tasks required in the output sub-phase are to save global variables to a module level source file and to save each top-level element to its own file.

Since ProcessJ allows multiple, top-level elements in a single file, each record, process, and protocol is split into its own class file. The module, or original ProcessJ source file, has no bearing on the final location of the Java source file. It is the package declaration that determines the location of the generated source files.

Global variables are placed into a package visible class called `Constants`, with the ProcessJ file, or module name, appended to it. For example, if the source file was `MyModule.pj`, then the resulting constants file is saved as `ConstantsMyModule`. Although carrying the element's visibility into the target language is not strictly necessary, the global variables are set as default visibility static final variables.

The ProcessJ compiler pipeline is now complete. The pipeline starts at command line processing; then, the source files are given to the syntax analysis phase. Source files are converted from plain text input stream into a token stream by the lexer. A token stream is converted into an AST and given to the semantic analysis phase; during this process, the structure of the source program is verified. Semantic analysis takes the AST and performs three sub-phases: preprocessing, analysis, and transformation. Finally, in the code generation phase, the AST is converted into the target language and saved to files.

The next chapter will describe each design pattern used in the usual manner for design patterns.

Chapter 5

Patterns

Since their use in the book Design Patterns [GHJV95], written by the authors commonly referred to as the 'Gang of Four,', patterns have become an essential way to express solutions to common problems found during software development.

The patterns used throughout the development of the ProcessJ compiler are enumerated in this chapter in the same pattern format used in Object Design [WBM03]. This format is preferable to the original Gang of Four because it is more concise. This chapter will discuss the composite, double dispatch, visitor, factory method, homogeneous AST, normalized heterogeneous AST, irregular heterogeneous AST, and the notification patters.

In this chapter, there are six subsections in each section: problem, forces, context, solution, consequences, and an example. The 'problem' subsection gives a general overview of the problem that the pattern addresses. The 'forces' subsection describes the varying concerns that are taken into consideration. The 'context' subsection describes when the solution is appropriate. The 'solution' subsection determines how the pattern resolves the forces, and the 'consequences' subsection describes the positive and negative effects that the pattern has on the implementation. Finally, the 'example' subsection gives a concrete example as to how the pattern might be used.

5.1 Composite

5.1.1 Problem

A flexible tree structure is needed where nodes and leaves are treated uniformly [GHJV95].

5.1.2 Forces

Tree data structures contain leaf nodes and aggregate nodes. By unifying the interface of leaf nodes and aggregate nodes, it is easier on collaborators to interact with the tree.

For instance, take the component used to build the tree. It is easier for the builder when every component in the tree has a `addChild` method. All the builder needs to do is create children then add the children to a particular node. The builder need not know what type of node it is attaching the children.

5.1.3 Context

The following quoted is directly from the Design Patterns book [GHJV95]:

Use the Composite Pattern when

- You want to represent par-whole hierarchies of objects.

- You want clients to be able to ignore the differences between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

5.1.4 Solution

Create a common base object for both aggregates and leaf nodes, as in Figure 5.1.4, from Design Patterns [GHJV95]. The `Client` is able to treat both `Leaf` and `Composite` nodes uniformly because they both implement the same interface. The only difference between a `Leaf` and a `Composite` is how it internally handles child-related methods.

The `Leaf` nodes would either have a no-operation method for the child classes or, if necessary, could throw an exception if the client tried to add a child to a true `Leaf`. `Composite`

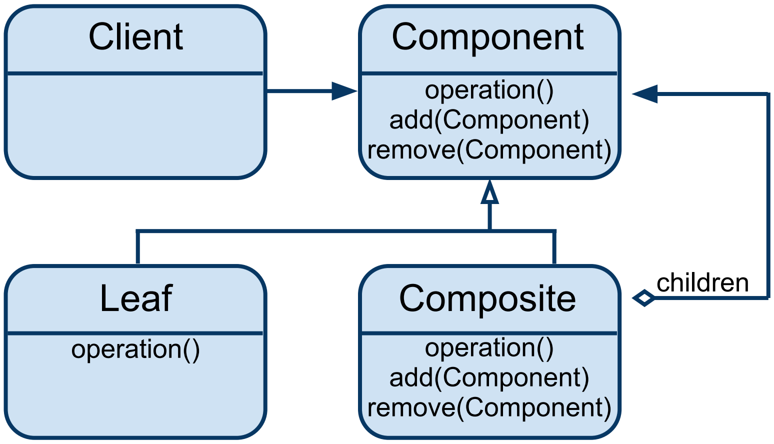

Figure 5.1: Composite Class Diagram.

nodes, on the other hand, implement all operations and contain a recursive structure. The children of a `Composite` are `Components`.

5.1.5 Consequences

Consequences, as quoted directly from Design Patterns [GHJV95] for the Composite Pattern are as follows:

- Defines class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively. Whatever client code expects a primitive object, it can also take a composite object.

- Makes the client simple. Clients can treat composite structures and individual objects uniformly. Clients normally do not know (and should not care) whether they are dealing with a leaf or a composite component. This simplifies client code, because it avoids having to write tag-and -case-statement-style functions over the classes that define the composition.

- Makes it easier to add new kinds of components. Newly defined Composite or Leaf sub-classes work automatically with existing structures and client code. Clients do not have to be changed for new Component classes.

- Can make your design overly general. The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you cannot rely on the type system to enforce those constraints for you. You will have to use run-time checks instead.

5.1.6 Example

The AST of ProcessJ is implemented with the Composite Pattern. The base class `ProcessJTree` implements the necessary functions that act as either a `Leaf` or a `Composite`. In that way, it acts as both the `Component` and the `Composite`.

Implementing the AST as a Composite allows a client to navigate and rewrite the tree with ease. Since all nodes can have children that are `ProcessJTree`s, removing a node from its parent and reattaching it in another node can be accomplished without knowing they type of the node being moved, or of its new parent.

5.2 Double Dispatch

5.2.1 Problem

A function is required to behave differently based on the run-time class of an argument [Mey95, WBM03].

5.2.2 Forces

Conditional statements that rely on the run-time class of an object are difficult to maintain because every time a new class is added, existing code needs to be modified. Modifying the existing code breaks the open/closed principle [Mey88], which states that software entities should be open for extension, but closed for modification.

Object-oriented systems use polymorphism to allow each implementation to define its behavior. When there is a need for an object to act differently, polymorphism allows the object to define its behavior.

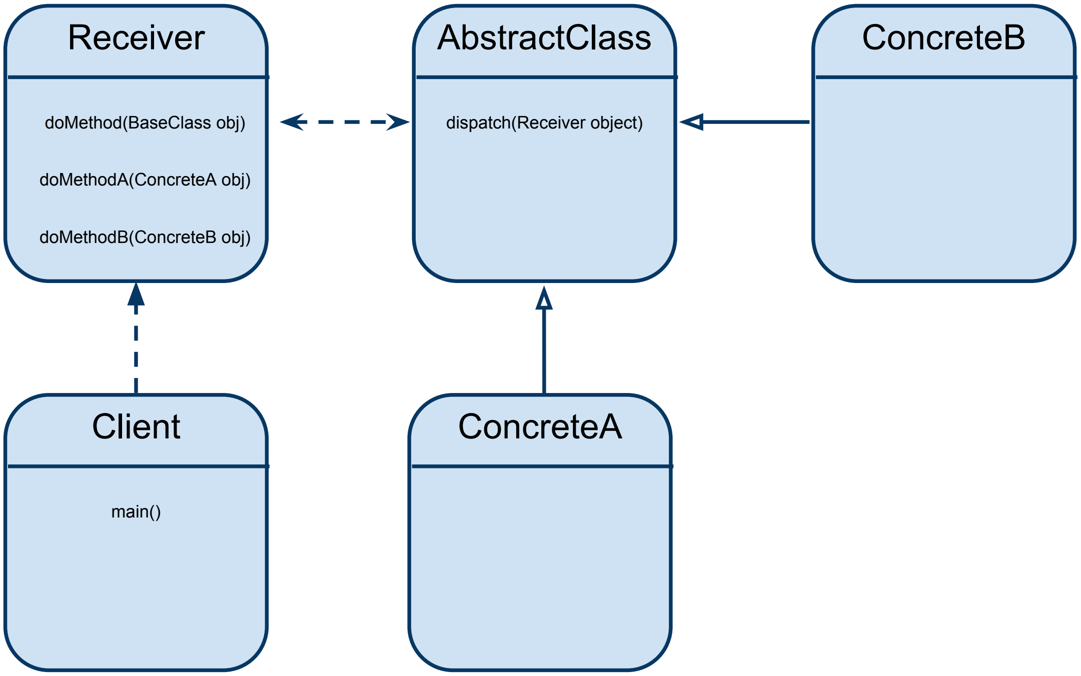

Figure 5.2: Double Dispatch Class Diagram.

5.2.3 Context

A function should act differently, depending on the run-time value of an argument.

5.2.4 Solution

The solution is to call a method on the argument that is run-time dependent. The argument then turns around and calls a specific method on the original receiver. Let us look at a specific example in Figures 5.2 and 5.3.

In the case of Figure 5.3, the `Client` calls `doMethod` on `Receiver`, passing an instance of `ConcreteA`. The `doMethod` method is the function that needs to act differently based on the run-time type of the `AbstractClass` argument. The `Receiver` calls the `dispatch` method of the argument, which immediately calls the `doMethodA` method in the `Receiver`.

The next time, the client calls `doMethod` with a `ConcreteB` type argument. The same thing happens, only this time, the `dispatch` method implementation of `ConcreteB` calls

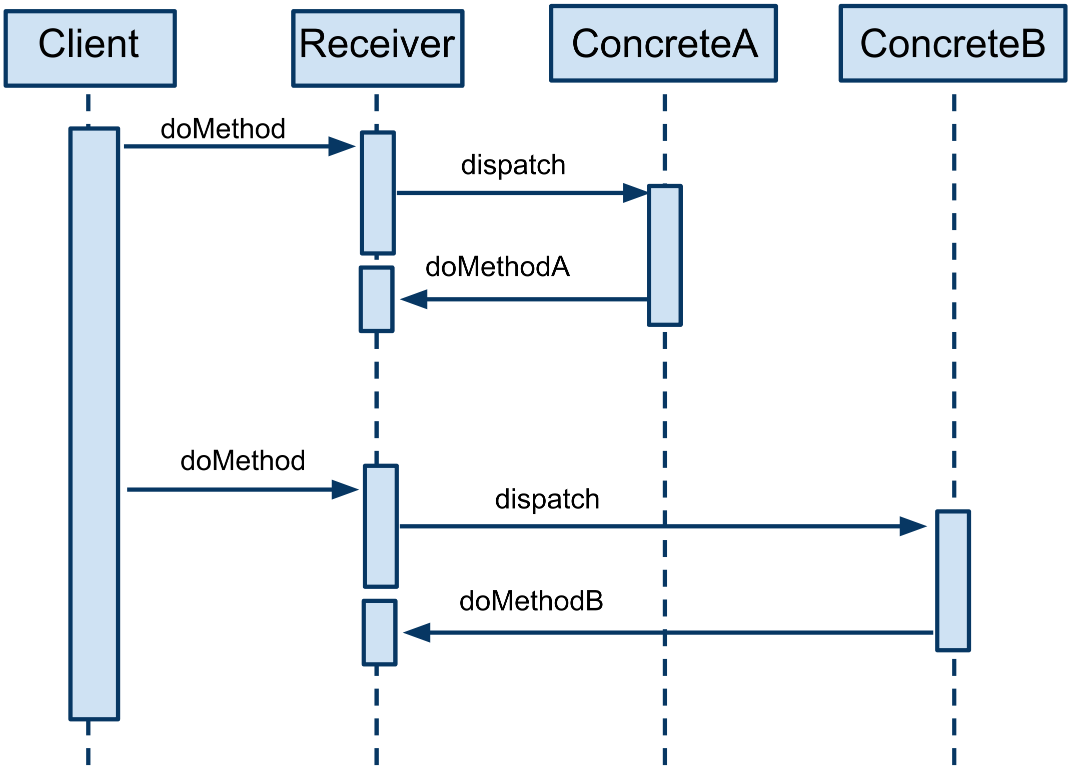

Figure 5.3: Double Dispatch Sequence Diagram.

the `doMethodB` on the `Receiver`.

This solution appears more complicated than a `switch` statement or adding `if` statements. What does this added complexity provide to the system? Consider the two solutions. The first solution utilizes a control block, like a `switch`, to provide the different functionality based on the run-time type of the argument. The second solution uses double dispatch.

Now, let us see what changes to existing code when we add another possible argument type in each solution. In the `switch` solution, the original code needs to be edited in order to add an extra case; however, changing existing functionality potentially creates bugs and breaks the open/closed principle [Mey88]. With the second solution, the original `Receiver` may not even need to be changed at all. Another possibility is that a new method, `doMethodC`, can be created in `Receiver`; also, a new extension of `Receiver` and the new type can call that method. The difference between the first solution and the second, is the second adds functionality while the first edits existing functionality.

5.2.5 Consequences

Double dispatch creates an extension point in the *Receiver* removing the need for a `switch` statement based on an argument. It could require the addition of a class-specific method, but the necessity of modifying existing code is removed.

5.2.6 Example

Rock-Paper-Scissors is a simple game that can be used as an example of Double Dispatch in Object Design [WBM03]. In Rock-Paper-Scissors, there is a `GameObject`, which has three boolean methods: `beatsRock`, `beatsPaper`, and `beatsScissors`. A class diagram is depicted in Figure 5.2.6.

To correspond the original solution to this example, the `GameController` acts as the `Client`. The `GameObject` acts as both the `Receiver` and the `BaseClass`; the `Rock`, `Paper`, and `Scissors` each act as one of the `Concrete` classes.

An example execution is depicted in Figure 5.2.6. The `GameController` creates two `GameObject` objects, a `Rock` and a `Paper`. The `GameController` then calls the `beats` method on the `Rock` object passing the `Paper` object as an argument. The `beats` method

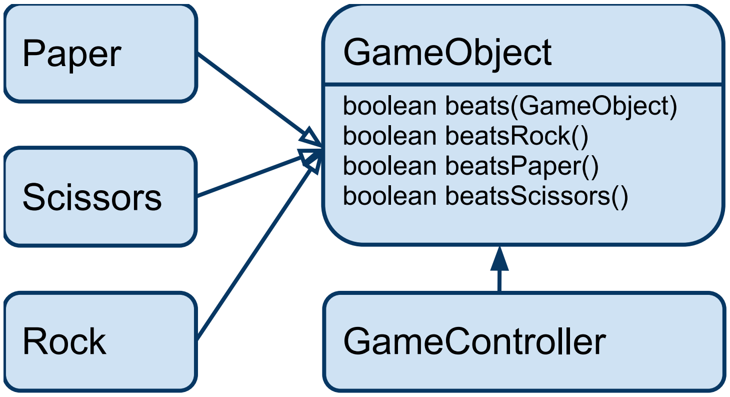

Figure 5.4: Rock Paper Scissors Class Diagram.

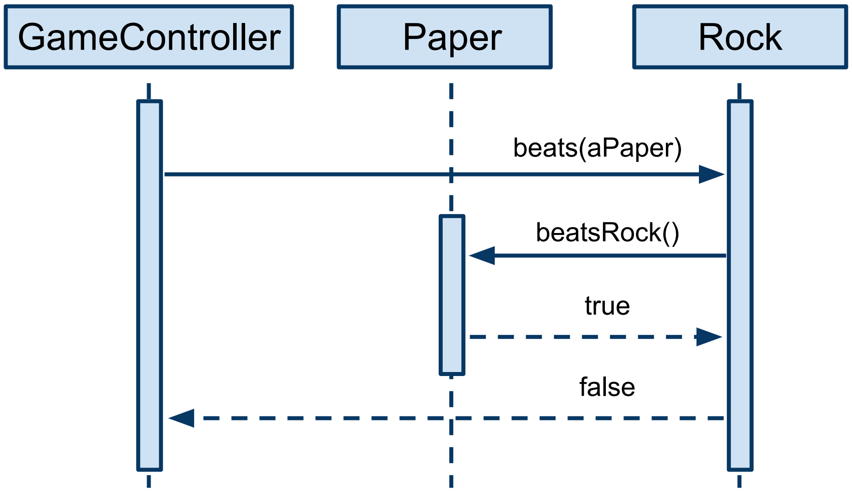

Figure 5.5: Rock Paper Scissors Sequence Diagram.

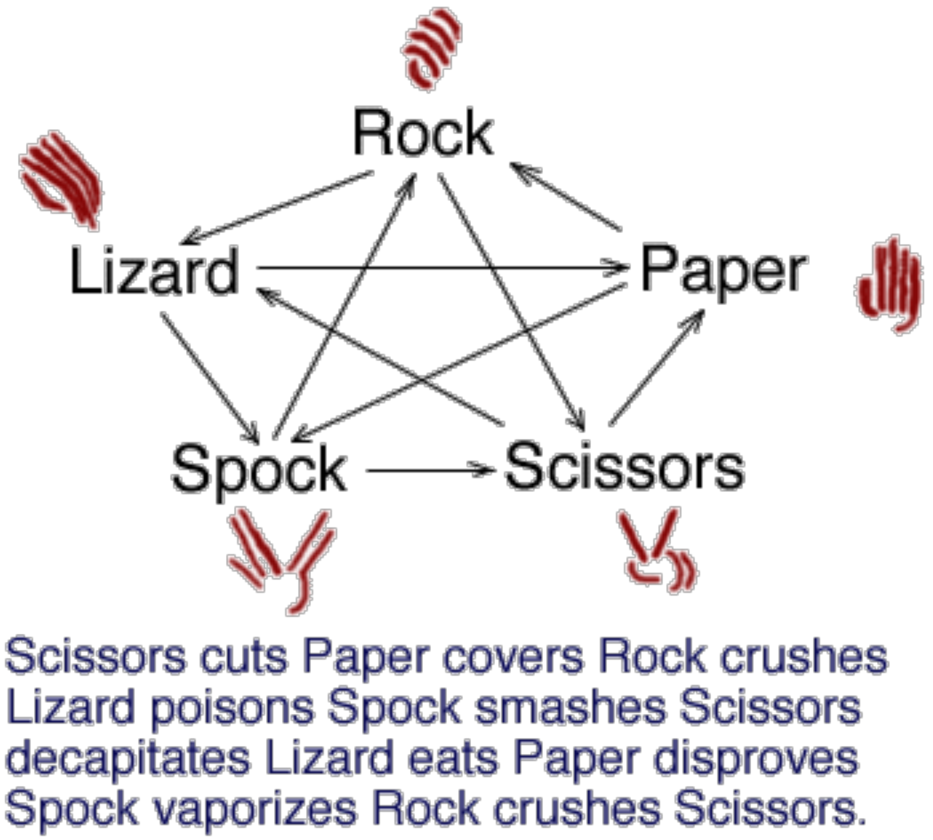
Rock
Lizard
Paper
Spock
Scissors
Scissors cuts Paper covers Rock crushes Lizard poisons Spock smashes Scissors decapitates Lizard eats Paper disproves Spock vaporizes Rock crushes Scissors.

Figure 5.6: Rock Paper Scissors Lizard Spock.

then calls the `beatsRock` method on the `Paper` object. The `Paper` object returns `true` so the `beats` method of the `Rock` object returns `false`.

To show the power of Double Dispatch, let us make Rock-Paper-Scissors more interesting. The problem with Rock-Paper-Scissors is that there are not many options, so the probability of getting a tie is one in three. To reduce the chance of a tie, we can implement Rock-Paper-Scissors-Lizard-Spock [KB98], as depicted in Figure 5.2.6.

To implement Rock-Paper-Scissors-Lizard-Spock, each of the `GameObject` classes now need an additional two methods `beatsLizard` and `beatsSpock`, plus the addition of two new `GameObject`, classes `Lizard` and `Spock`. That is it! No existing code needs modification.

The power of Double Dispatch comes from the ability to extend functionality without modifying existing code. By adding two `GameObject` classes, and by adding a few methods to existing `GameObject` classes, the game can be changed without modifying any of the original `beats` methods.

5.3 Visitor

5.3.1 Problem

Many operations need to be performed on a relatively static set of classes that compose an object structure. Adding new operations needs to have a minimal impact on existing code [GHJV95].

5.3.2 Forces

Having a number of disparate operations distributed across each of the classes of an object structure decreases the cohesion of the classes while increasing the coupling. Each time you add a new operation, every class in the object structure needs to be edited.

It would be better to centralize the logic of each operation into a single class while retaining the ability to perform type-specific operations on the object structure.

It also may be necessary to extend functionality, with minimal impact to the existing code-line. In order to minimize the impact, the object structure needs to be decoupled from the algorithms performed against that object structure.

5.3.3 Context

The Visitor is used when there is a relatively static set of classes that compose an object structure. For example, the set of classes that compose the AST in a compiler. An abstract syntax tree is a good example of an object structure that works well with the Visitor Pattern, because there rarely are additions or subtractions to the set of nodes.

Sometimes it is necessary to extend functionality without modifying the object structure. A compiler is the perfect example of this need, because there is always the need to extend functionality without modifying the AST. It is always possible to improve a compiler which is the basis for the Full Employment Theorem for Compiler Writers.

The Full Employment Theorem says that a perfect size-optimizing compiler would simplify an infinite loop program to the same size as the optimum program. Thus, it would provide a method of detecting an infinite loop program, which, due to the halting problem [Tur37], is known to be undecidable. Since the best compiler is impossible, it is always possible to write a better compiler!

5.3.4 Solution

The Visitor Pattern uses Double Dispatch (Section 5.2) to apply the correct method, depending on the current node. Each `Visitor` performs a single duty and is able to contain type-specific methods so that no switch is needed.

To implement the Visitor, each AST node needs to be broken into a different class. In each class, implement an `acceptVisitor(Visitor)` method. Then each operation that need to happen to the object structure becomes a new `Visitor` class, and each `Visitor` class implements a a `visit` method for each node type.

5.3.5 Consequences

Below are listed the benefits and liabilities of using the Visitor Pattern, quoted directly from Design Patterns [GHJV95]:

- Visitor makes adding new operations easy.
- A visitor gathers related operations and separates unrelated ones.
- Adding new `ConcreteElement` classes is hard.
- Visiting across class hierarchies.
- Accumulating state.

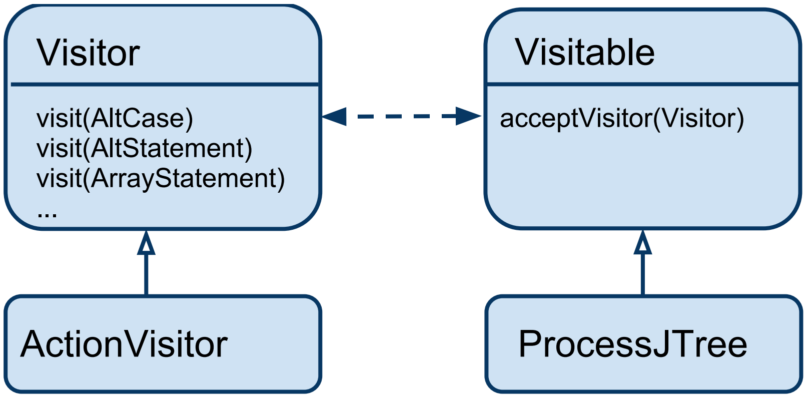

Figure 5.7: Visitor Class Diagram.

5.3.6 Example

In the ProcessJ compiler, the Visitor Pattern is used to isolate operations from the AST. There are four main classes that are involved in the Visitor Pattern in ProcessJ: `ProcessJTree`, `Visitor`, `ActionVisitor`, and `ProcessJCompiler`.

All classes that are `Visitable` need to implement the `acceptVisitor(Visitor)` method. The `acceptVisitor` method then calls the `visit` method on the `Visitor`, just as in the Double Dispatch Pattern (Section 5.2). The `Visitable` interface is not strictly needed, since a `Visitor` needs to implement a specific `visit` method for each node type. However, it is important every node needs to implement that method.

The interface that abstracts a particular operation is the `Visitor`. In ProcessJ, the `Visitor` is an interface because it then has the ability to utilize multiple methods for applying an operation to an AST. For instance, ANTLR has a domain specific language that allows manipulation of abstract syntax trees. It would be useful to allow the operation implementer to determine if it would be easier to write an operation in ANTLR or in Java.

An `ActionVisitor` is a concrete implementation of `Visitor`, although it does not do any operation other than apply `Actions` to each node in the tree. The `ActionVisitor` separates tree navigation from the details of an operation. The `ActionVisitor` and `Actions` are described in further detail in Chapter 4. Suffice to say that the `ActionVisitor` is an example of a concrete implementation of a `Visitor`.

The `ProcessJCompiler` class acts as a client to the Visitor Pattern. It instantiates

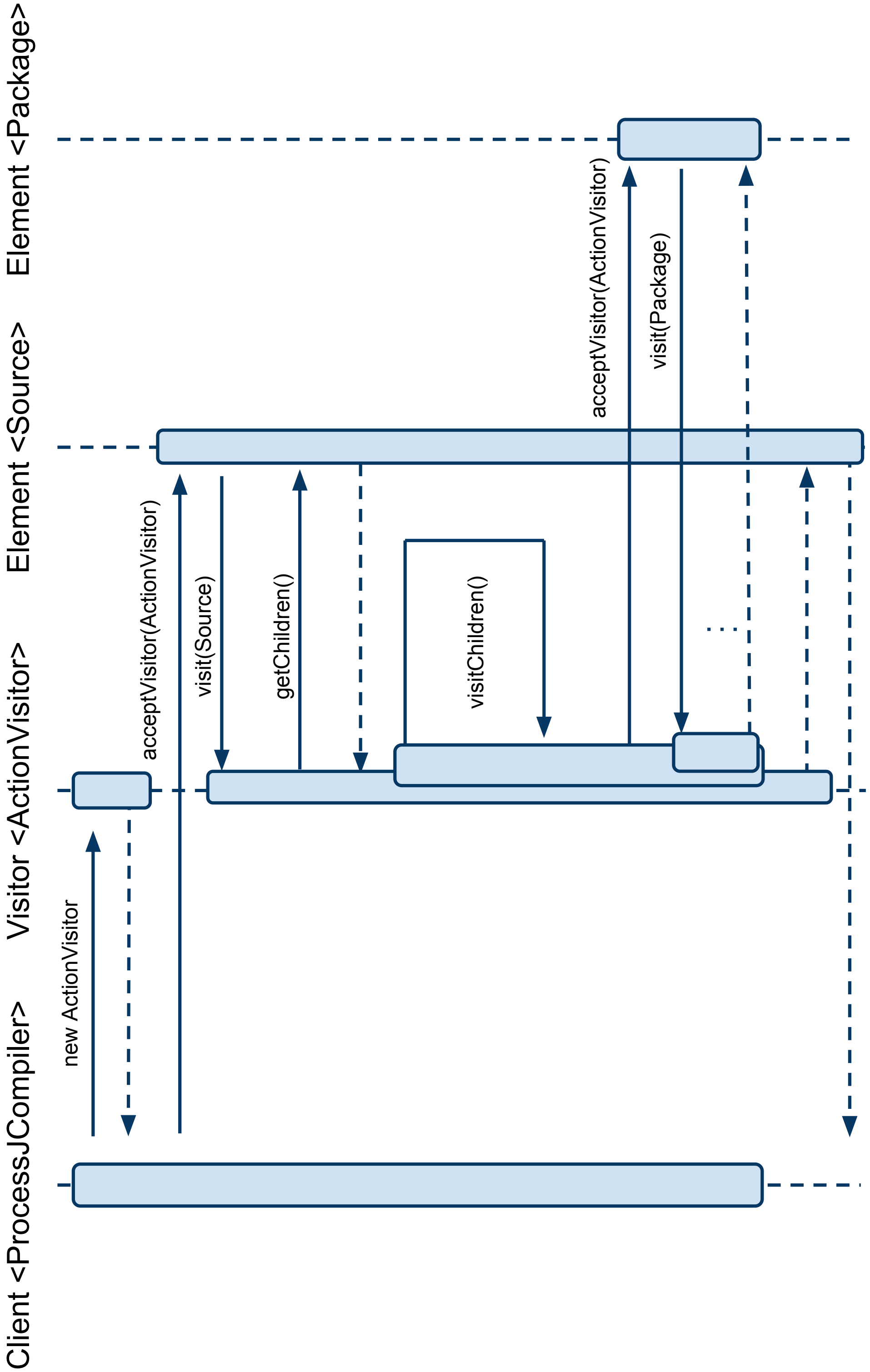

Figure 5.8: Visitor Sequence Diagram.

`Visitor`s and initiates a tree walk from the root `Source` node. It also acts as a map of the pipeline because it applies each operation in order.

5.4 Factory Method

The Factory Method pattern is from Design Patterns [GHJV95].

5.4.1 Problem

A framework must instantiate classes, but it only knows about abstract classes, which it cannot instantiate.

5.4.2 Forces

Frameworks use abstract classes to define and maintain relationships between objects. A framework often is responsible for creating these objects as well.

The Factory Method pattern encapsulates the knowledge of `ConcreteProduct`, and creates and moves this knowledge out of the framework.

5.4.3 Context

Use the Factory Method Pattern when:

- A class cannot anticipate the class of objects it must create.
- A class wants its sub-classes to specify the objects it creates.
- Classes delegate responsibility to one of several helper sub-classes, and localization of the knowledge of which helper subclass is the delegate is needed.

5.4.4 Solution

The structure is depicted in Figure 5.9. A `Creator` has a method `FactoryMethod` which is called when it needs an implementation of some abstract type `Product`. The `ConcreteCreator` knows which implementation to use, which in this case is `ConcreteProduct`.

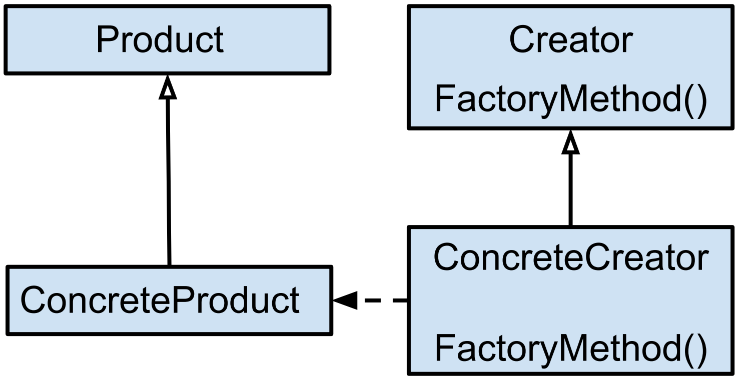

Figure 5.9: Structure of Factory Method.

5.4.5 Consequences

Factory methods eliminate the need to bind application-specific classes into the code. The code only deals with the Product Interface; therefore, it can work with any user-defined `ConcreteProduct` classes.

A potential disadvantage of factory methods is that clients might have to subclass the `Creator` class just to create a particular `ConcreteProduct` object. Sub-classing is fine when the client has to subclass the `Creator` class anyway, but otherwise the client now must deal with another point of evolution.

Two additional consequences of the Factory Method pattern are:

1. *It provides hooks for sub-classes.* Creating objects inside a class with a factory method is always more flexible than creating an object directly. Factory Method gives subclasses a hook for providing an extended version of an object.

2. *It connects parallel class hierarchies.* Clients can find factory methods useful, especially in the case of parallel class hierarchies. Parallel class hierarchies result when a class delegates some of its responsibilities to a separate class.

5.4.6 Example

ANTLR uses the `TreeAdaptor` interface to determine which class to create for a given token. To relate ANTLR to the given solution, the `TreeAdaptor` interface is the `Creator`, and in the ProcessJ compiler, the `ProcessJCommonTreeAdaptor` is the `ConcreateCreator`. The default implementation can be overridden to define ProcessJ specific classes for the AST.

5.5 Homogeneous AST

The Homogeneous AST (HAST) pattern is taken from the book, Language Implementation Patterns [Par09].

5.5.1 Problem

An AST needs to be represented with a simple and uniform interface with a normalized child list.

5.5.2 Forces

Tree walking is easy if every element of the tree has a uniform interface and a normalized child list. All the tree walker needs to do is process the token at the current element and iterate over its children.

Having only one interface makes it easy for developers to learn. It is much easier to figure out what to do when there is only one option.

5.5.3 Context

Use HAST when:

- The AST is simple.
- Behavior does not change across node types.
- A external tree visitor is required.
- Implementing in a non-object-oriented language like C.

5.5.4 Solution

The solution for this pattern is simple. There is one class; the AST node. The node holds a token, and has a normalized list of its children.

5.5.5 Consequences

HAST has the benefit of an easy-to-learn API. Developers are sure to understand a data structure when there is only one type.

It is easy to write external tree visitors because of the uniform API. It is possible to switch on the token type to execute different actions, and there is uniform access to child elements.

The downside is that there is no way to encapsulate data for a particular node or have polymorphism. In short, this is not an object-oriented approach.

5.6 Normalized Heterogeneous AST

The Normalized Heterogeneous AST (NHAST) is a pattern described in Language Implementation Patterns [Par09].

5.6.1 Problem

An AST with a simple and uniform interface and a normalized child list is needed, but more than one data type needs to be represented.

5.6.2 Forces

HAST is good for simple applications or to implement in languages that are not object-oriented. However, when there is a need to encapsulate data within an element, or have nodes act polymorphic, the NHAST can be used. It is possible to have all the benefits of a simple API and normalized child list by having all elements of the AST implement the same interface. There is no need to limit the element data types to a single data type in an object oriented language.

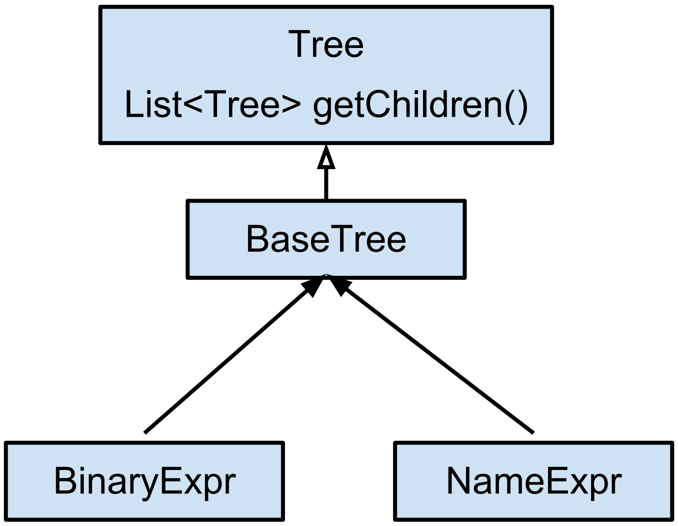

Figure 5.10: Normalized Heterogeneous AST.

5.6.3 Context

A Normalized Heterogeneous AST (NHAST) is used when:

- There is a need to store node-specific data, and
- The programmer plans to use external tree visitor.

5.6.4 Solution

Figure 5.10 depicts a NHAST. The base interface `Tree` is implemented by an abstract base class, `BaseTree`, which is extended by specific node types, such as `BinaryExpr` and `NameExpr`. All classes in the AST provide access to a normalized list of children.

5.6.5 Consequences

Like HAST, NHAST has the benefit of an easy-to-learn API because all nodes have a the same interface. External tree visitors are also simple because of the uniform interface and uniform child access. It also has the same downside, child access logic can be spread across the entire application.

Unlike HAST, NHAST is able to encapsulate node specific information and has the ability to act polymorphicly.

5.7 Irregular Heterogeneous AST

The Irregular Heterogeneous AST (IHAST) pattern is from Language Implementation Patterns [Par09].

5.7.1 Problem

This pattern is used when there is a need to implement an AST using more than a single node data type and with an irregular child list representation.

5.7.2 Forces

Accessing child elements through a list breaks encapsulation. If the grammar changes the order in which child nodes are placed into the child list, all the code used to access those child nodes need to change. Optional children also can change order so nodes need to be found. Code to find a child could also end up being duplicated.

Having accessor methods for each child encapsulates the child location information. There is always a single point of access to child elements; if the storage mechanism or location of a child changes, code is only needs to change in one place.

5.7.3 Context

Use an IHAST when:

- Readability of the AST is important.
- The project is small.
- It is worth the extra effort involved in writing external visitors.
- Encapsulation is important to the AST.

5.7.4 Solution

The IHAST is difficult to depict because it is *irregular*, by definition. Every element has its own type, and access to its children are through accessor methods. There is no uniform method for child access.

5.7.5 Consequences

Readability and encapsulation of node-specific data are two benefits of using IHAST. Every node can store node-specific information, and access to child elements is controlled by each element. This pattern can be considered in terms of a domain model, in most general applications. The domain is queried for what is needed, and all logic is stored within the domain model.

The disadvantage to the domain model approach are the ones that the Visitor Pattern is meant to fix. The only problem is that it is difficult to write a visitor in which the elements do not provide a uniform interface and normalized access to child elements.

5.8 Notification

The Notification pattern is from Domain Specific Languages [FP10].

5.8.1 Problem

There is a need to collect errors and other messages to report back to the caller.

5.8.2 Forces

When validating an object model, it is beneficial to report all errors rather than stopping validation after the first error. Reporting a collection of errors potentially allows users to fix more in a single iteration; otherwise, they can glean more from related errors.

5.8.3 Context

Use the Notification pattern when:

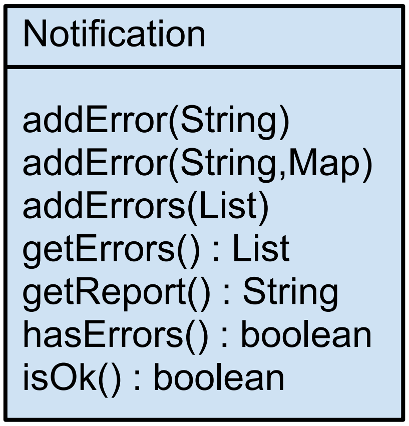

Figure 5.11: Notification Solution.

- There is a complicated operation that may trigger multiple errors.
- Failure at the first error is not wanted.
- Lower layers of an application need to interact with the user interface directly.
- Error collection and reporting needs to be centralized across multiple operations.

5.8.4 Solution

The Notification solution can have a separate class that contains the error reporting logic. Figure 5.11 depicts such a class. Adding an error to the notification and getting a final report are goals of the Notification pattern.

5.8.5 Consequences

Using a Notification centralizes error reporting code. Since errors can be reported from the parser and lexer during syntax analysis, and any of the actions during semantic analysis, it is good to have a single method for determining errors during the compilation process.

The Notification also allows more than one error to be reported. Since the parser supports error recovery, the parser can figure out where it is and then report more errors as they are found. Reporting multiple errors allows the ProcessJ developer to fix multiple errors for each compile.

Chapter 6

Related Work

The ProcessJ compiler is currently implemented to run on the Java Virtual Machine (JVM), however, a few other languages that have taken a similar approach. Although the JVM originally was developed as a run-time for Java, it has become common for other languages to compile to byte code. For example, Scala [Ode07] is an original language that mixes the object-oriented approach with functional programming. Another example is Erjang [Tho10], which compiles Erlang programs into byte code so it can be run on the JVM.

There are several libraries for using a CSP-based process-oriented programming model. Communicating Haskell Processes [Bro08], CCSP [Moo99], C++CSP [BW03], Communicating Scala Objects [Suf08], and JCSP [W^{+}03] each allow the programmer to write process-oriented code in Haskell, C++, Scala, and Java respectively. As mentioned previously, the major drawback to this approach is that it requires self control on the part of the programmer. The compiler cannot catch semantic errors through the use of a library, whereas a natively process-oriented language can do that.

Taking the library approach a step further, some have developed domain specific languages for programming CSP. A domain specific language (DSL) is a computer programming language of limited expressiveness that is focused on a particular domain [FP10]. More specifically, an internal DSL is a particular way of using a general-purpose language that handles one small aspect of the overall system in a manner that feels like a custom language [FP10]. For example, two libraries that provide internal DSL for CSP programming are `python-csp` [MHWN09] and Groovy Parallel [KBS05], through Python and Groovy

```
@process
def producer(channel):
  x = 0
  while True:
    x = x + 1
    channel.write(x)

@process
def consumer(channel):
  while True:
    print channel.read()

chan = Channel()
Par(consumer(chan), producer(chan)).start()
```

Figure 6.1: Producer Consumer With `python-csp`.

respectively. Figure 6.1 illustrates a producer consumer example using `python-csp`.

Extending existing languages is another approach to implementing CSP. Petitpierre [Pet02], used an active-object approach where Java is extended to support CSP. Classes declared as active execute in their own thread. When a method is called on an instance of one of these active classes, the call blocks until the active object reaches an accept statement, allowing both threads to continue. This strategy uses method calls in order to communicate rather than by means of channels.

To take CSP implementation a step further, Go [Inc09], Honeysuckle [Eas02], and Rain [Bro06] are all programming languages offering features from process-oriented programming. The concurrency model of Go is based on CSP, but can be thought of as a type safe generalization of Unix pipes [Inc09]. Go provides a construct called goroutines which are functions that execute in their own thread of control and communicate through channels. Honeysuckle is a language that seeks to improve on occam-π by offering source-code modularity, object encapsulation, and recursive definition of both objects and processes. Finally, of the three programming languages mentioned, Rain is nearest to ProcessJ. Rain is to occam-π and C++ as ProcessJ is to occam-π and Java because it implements almost the same feature set.

CSP can be thought of as a network of processes in which each node or component can be a sub-network or sub-component. Because of this network metaphor, it is natural to model them by using graphs. A Graphical Modeling Language for Specifying Concurrency based on CSP [Hil02] defines a diagram language for CSP systems, similar to UML for object-oriented systems. In Visual Occam [Slo10], a development environment was developed that describes occam-π processes as component networks. Finally, SystemCSP [OB06] is a CSP-based visual notation used for formal specification of formally verifiable, component-based designs of distributed real-time systems.

One final alternative approach to consider is the direct use of CSP as a formal language. There are several tools that use a machine readable CSP, such as Failures-Divergence Refinement (FDR) [FW09] and ProBE [LF08]. FDR is a model checker for CSP, capable of checking for livelock and deadlock. ProBE is an animation and model checking tool. CSP has been compiled into CTJ, JCSP, and CCSP [RRS03] as well as directly executable code [Bar06].

Chapter 7

Conclusion and Future Work

ProcessJ is a general purpose process-oriented programming language meant to bridge the gap between Java and `occam-π`. It is built on the solid algebraic principals of the CSP concurrency model. The compiler and language still are in the infancy stage of development, although much of the compiler pipeline is now in place.

There is still much work to do to make ProcessJ a successful language. Three immediate points of extension that will improve on the work already done to ProcessJ are: a system library, implementation of some more CSP features, and integrated development environment support.

Above all ProcessJ needs a system library akin to Java's `java.lang` package. A general purpose programming language is not worth much if you cannot read in data, output data, open sockets, and interact with the world in general. This would need to be a deviation point from Java, since we would want to design a process-oriented library with these facilities.

One feature that is currently being developed by Sean Kau, a graduate student at UNLV, is parallel usage checking. The compiler will check that a variable can be read in parallel, but never written in parallel.

In addition, some of the features described in Sir Anthony Hoare's CSP are not yet implemented in ProcessJ. Interrupts, catastrophe, restart, checkpoints, and traces seem to be interesting language features [Hoa85]. These features are quoted directly from Communicating Sequential Processes [Hoa85].

- *Trace* A trace of the behaviour of a process is a finite sequence of symbols recording the events in which the process has engaged up to some moment in time. It would be nice to define a trace for a process as a means for testing or verification.

- *Interrupts* Define a kind of sequential composition (PˆQ) which does not depend on successful termination of P. Instead, the progress of P is just interrupted on occurrence of the first event of Q; and P is never resumed. It follows that a trace of (PˆQ) is just a trace of P up to an arbitrary point when the interrupt occurs, followed by a trace of Q.

- *Catastrophe* Let ↯ be a symbol standing for a catastrophic interrupt event, which it is reasonable to suppose would not be caused by P. A process behaves like P up to a catastrophe and thereafter like Q.

- *Restart* One possible response to catastrophe is to restart the original process again.

- *Checkpoints* Let process P be a process which describes the behavior of a long-lasting data base system. When lightning strikes, one of the worst responses would be to restart P in its initial state, losing all the laboriously accumulated data in the system. I would be much better to return to some recent state of the system which is known to be satisfactory much like transactions in modern database systems. Such a state is known as a checkpoint.

Finally, as with all programming languages, it is nice to work in an integrated development environment (IDE). ProcessJ could use an Eclipse plugin that provides all the features that a Java programmer is used to. For instance, code assist, cross referencing, and code highlighting are some of the features of IDEs. There are also plenty of possibilities for static analysis and visual programming techniques that should work nicely with ProcessJ.

Appendix A

The Santa Clause Problem

The Santa Clause Problem, defined by Trono [Tro94], is an exercise in concurrency that involves several process types and limited resources. There are three process types: the elves, the reindeer, and Santa.

There are ten elves which make toys. Occasionally, the elves find they need to consult with Santa. Santa cannot be bothered to meet with one elf at a time so they need to queue in groups of three. After three elves are ready, Santa will greet each of them, consult with them, and finally say goodbye to each elf.

There are nine reindeer which spend their time on holiday except Christmas. On Christmas, all nine come back to the north pole, and the last one back tells Santa they are ready to be harnessed. Santa harnesses each of the reindeer and goes off to deliver presents. When Santa gets back, he unharnesses each of the reindeer, and goes back to his house while the reindeer go back on vacation.

Finally, if the reindeer and elves are ready for Santa at the same time, Santa chooses to work with the reindeer.

```
package edu.unlv.cs.santa;

const int N_REINDEER = 9;
const int G_REINDEER = N_REINDEER;

const int N_ELVES = 10;
const int G_ELVES = 3;

const int HOLIDAY_TIME = 100000;
const int WORKING_TIME = 200000;
const int DELIVERY_TIME = 100000;
const int CONSULTATION_TIME = 200000;

protocol ReindeerMessage {
 holiday:        { int id; }
 deerReady:      { int id; }
 deliver:        { int id; }
 deerDone:       { int id; }
}

protocol ElfMessage {
 working:        { int id; }
 elfReady:       { int id; }
 consult:        { int id; }
 elfDone:        { int id; }
}

protocol SantaMessage {
 reindeerReady:  { }
 harness:        { int id; }
 mushMush:       { }
 woah:           { }
 unharness:      { int id; }
 elvesReady:     { }
 greet:          { int id; }
 consulting:     { }
 santaDone:      { }
 goodbye:        { int id; }
}

protocol Message extends ReindeerMessage, ElfMessage, SantaMessage;

native proc void println(String msg);

proc void display (chan<Message>.read in) {
  while(true){
    Message msg = in.read();
    switch(msg.tag){
    case holiday:
         println(”Reindeer ”+msg.id+” is on holiday.”);
         break;
    case deerReady:
         println(”Reindeer ”+msg.id+” is on ready.”);
         break;
    case deliver:
         println(”Reindeer ”+msg.id+” is delivering.”);
         break;
    case deerDone:
         println(”Reindeer ”+msg.id+” is done.”);
         break;
    case working:
         println(”Elf ”+msg.id+” is working.”);
         break;
    case elfReady:
         println(”Elf ”+msg.id+” is ready.”);
         break;
    case consult:
         println(”Elf ”+msg.id+” is consulting with Santa.”);
```

```
      break;
 case elfDone:
      println(”Elf ”+msg.id+” is done.”);
break;
 case reindeerReady:
    println(”Santa and the reindeer are ready.”);
      break;
 case harness:
    println(”Santa is harnessing ”+msg.id);
      break;
 case mushMush:
    println(”Mush! Mush!”);
      break;
 case woah:
      println(”Woah!”);
      break;
 case unharness:
      println(”Santa is unharnessing ”+msg.id);
      break;
 case elvesReady:
      println(”Santa, the elves are ready!”);
      break;
 case greet:
      println(”Santa greets elf ”+msg.id);
      break;
 case consulting:
      println(”Santa is consulting with elves.”);
      break;
 case santaDone:
      println(”Santa is done.”);
      break;
 case goodbye:
      println(”Santa says goodbye to ”+msg.id);
      break;
    }
  }
}

proc void partialBarrierKnock (const int n,
                               chan<boolean>.read a,
                               chan<boolean>.read b,
                               chan<boolean>.write knock) {
 while (true) {
   for (int i=0; i<n; i++) {
     boolean any;
     any = a.read();
   }
   knock.write (true);
   for (int i=0; i<n; i++) {
     boolean any;
     any = b.read();
   }
 }
}

proc void partialBarrier (const int n,
                          chan<boolean>.read a,
                          chan<boolean>.read b) {
 while (true) {
   for (int i=0; i<n; i++) {
     boolean any;
     any = a.read();
   }
   for (int i=0; i<n; i++) {
     boolean any;
     any = b.read();
   }
 }
```

```
}

proc void synchronize (shared chan<boolean>.write a,
                      shared chan<boolean>.write b) {
 claim (a) {
   a.write(true);
 }
 claim (b) {
   b.write(true);
 }
}

proc void reindeer (const int id,
                   const int seed,
                   barrier justReindeer,
                   barrier santaReindeer,
                   shared chan<int>.write toSanta,
                   shared chan<ReindeerMessage>.write report) {
 int mySeed = seed;
 timer tim;
 long t, wait;
 for (int i=0; i<1000; i++) {
   skip;   // the reindeer takes a breather
 }
 while (true) {
   claim (report) {
     report.write(new ReindeerMessage.holiday(id));
   }
   // wait, mySeed := random (...)
   t = tim.read();    // dies in the checker for some strange reason
   tim.timeout(t + wait);
   claim (report){
     report.write(new ReindeerMessage.deerReady(id));
   }
```

```
    sync (justReindeer);
    claim (toSanta){
      toSanta.write(id);
    }
    sync (santaReindeer);
    claim (report){
      report.write(new ReindeerMessage.deliver(id));
    }
    sync (santaReindeer);
    claim (report){
      report.write(new ReindeerMessage.deerDone(id));
    }
    claim (toSanta){
      toSanta.write(id);
    }
  }
}

proc void elf (const int id,
               const int seed,
               shared chan<boolean>.write elvesA,
               shared chan<boolean>.write elvesB,
               shared chan<boolean>.write santaElvesA,
               shared chan<boolean>.write santaElvesB,
               shared chan<int>.write toSanta,
               shared chan<ElfMessage>.write report) {
  int mySeed = seed;
  timer tim;
  long t, wait;
  for (int i=0; i<1000; i++)
    skip; // wait, mySeed := random (workingTime, mySeed);
  while (true) {
    claim (report){
      report.write(new ElfMessage.working(id));
    }
    // wait, mySeed := random (workingTime, mySeed);
    t = tim.read();
    tim.timeout (t + wait);
    claim (report){
      report.write(new ElfMessage.elfReady(id));
    }
    synchronize (elvesA, elvesB);
    claim (toSanta) {
      toSanta.write(id);
    }
    synchronize (santaElvesA, santaElvesB);
    claim (report){
      report.write(new ElfMessage.consult(id));
    }
    synchronize (santaElvesA, santaElvesB);
    claim ( (report){
          report.write(new SantaMessage.mushMush());
        }
        sync (santaReindeer);
        t = tim.read();
        tim.timeout (t + wait);
        claim (report){
          report.write (new SantaMessage.woah());
        }
        sync (santaReindeer);
        for (int i=0; i< G_REINDEER; i++) {
          id = fromReindeer.read({
            claim (report){
              report.write (new SantaMessage.unharness(id));
            }
          });
        }
      }
```

```
     any = knock.read (): { // 3 Elves ready
       claim (report){
         report.write (new SantaMessage.elvesReady());
       }
       for (int i=0; i<G_ELVES; i++) {
         id = fromElf.read();
         claim (report){
           report.write (new SantaMessage.greet(id));
         }
       }
       claim (report){
         report.write (new SantaMessage.consulting());
       }
       synchronize (santaElvesA, santaElvesB);
       t = tim.read();
       tim.timeout (t + wait);
       claim (report){
         report.write (new SantaMessage.santaDone());
       }
       synchronize (santaElvesA, santaElvesB);
       for (int i=0; i<G_ELVES; i++) {
         id = fromElf.read ({
           claim (report){
             report.write( new SantaMessage.goodbye(id));
           }
         });
       }
     }
   }
 }
}

proc void main() {
 timer tim;
 long seed;
 seed = tim.read();
 seed = (seed >> 2) + 42;
 barrier justReindeer, santaReindeer;
 shared write chan<boolean> elvesA, elvesB;
 chan<boolean> knock;
 shared write chan<boolean> santaElvesA, santaElvesB;
 shared write chan<int> reindeerSanta, elfSanta;
 shared write chan<Message> report;

 par {
   par enroll (santaReindeer) {
     santa (seed + (N_REINDEER + N_ELVES), knock.read,
            reindeerSanta.read, elfSanta.read, santaReindeer,
            santaElvesA.write, santaElvesB.write, report.write);
     par for (int i=0; i<N_REINDEER; i++) {
       par enroll (justReindder, santaReindeer) {
         reindeer (i, seed + i, justReindeer, santaReindeer,
            reindeerSanta.write, report.write);
       }
     }
   }
   par for (int i=0; i<N_ELVES; i++){
     elf (i, N_REINDEER + (seed + i),
          elvesA.write, elvesB.write,
          santaElvesA.write, santaElvesB.write,
          elfSanta.write, report.write);
   }
   display (report.read);
   partialBarrierKnock (G_ELVES, elvesA.read, elvesB.read, knock.write);
   partialBarrier (G_ELVES + 1, santaElvesA.read, santaElvesB.read);
 }
}
```

Bibliography

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, & tools.* Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007.

[AO07] Greg Wilson Andy Oram, editor. *Beautiful Code: Leading Programmers Explain How They Think.* O'Reilly Media, first edition, 2007.

[Apa03] Apache Software Foundation. Apache maven. `http://maven.apache.org/index.html`, April 2003.

[Apa07a] Apache Software Foundation. The apt format. `http://maven.apache.org/doxia/references/apt-format.html`, October 2007.

[Apa07b] Apache Software Foundation. Commons cli - home. `http://commons.apache.org/cli/`, August 2007.

[Bar06] Frederick R. M. Barnes. Compiling CSP. In Peter H. Welch, Jon Kerridge, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2006*, pages 377–388, September 2006.

[Bro06] Neil C.C. Brown. Rain: A New Concurrent Process-Oriented Programming Language. In Peter H. Welch, Jon Kerridge, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2006*, pages 237–251, September 2006.

[Bro08] Neil C.C. Brown. Communicating Haskell Processes: Composable Explicit Concurrency Using Monads. In Peter H. Welch, Susan Stepney, Fiona A.C Polack, Frederick R. M. Barnes, Alistair A. McEwan, Gardiner S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, pages 67–83, September 2008.

[BW03] Neil C.C. Brown and Peter H. Welch. An Introduction to the Kent C++CSP Library. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 139–156, September 2003.

[BW09] Federico Biancuzzi and Shane Warden. *Masterminds of Programming: Conversations with the Creators of Major Programming Languages*. O'Reilly Media, Inc., March 2009.

[Eas02] Ian R. East. The 'Honeysuckle' Programming Language: Event and Process. In James S. Pascoe, Roger J. Loader, and Vaidy S. Sunderam, editors, *Communicating Process Architectures 2002*, pages 285–300, September 2002.

[FCL09] Charles N. Fischer, Ronald K. Cytron, and Richard J. LeBlanc. *Crafting A Compiler*. Addison Wesley, November 2009.

[FP10] M. Fowler and R. Parsons. *Domain-Specific Languages*. The Addison-Wesley Signature Series. Addison Wesley Professional, 2010.

[FW09] Leo Freitas and Jim Woodcock. FDR Explorer. *Formal Aspects of Computing*, 21(1-2):133–154, February 2009.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, first edition, 1995.

[GPB$^+$06] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, 2006.

[Gro90] Josef Grosch. Efficient and comfortable error recovery in recursive descent parsers. *Structured Programming*, 11:129–140, 1990.

[Hil02] Gerald H. Hilderink. A Graphical Modeling Language for Specifying Concurrency based on CSP. In James S. Pascoe, Roger J. Loader, and Vaidy S. Sunderam, editors, *Communicating Process Architectures 2002*, pages 255–284, September 2002.

[Hoa81] Sir Charles Anthony Richard Hoare. The emperor's old clothes. *Commun. ACM*, 24(2):75–83, 1981.

[Hoa85] Sir Charles Anthony Richard Hoare. *Communicating Sequential Processes (Prentice Hall International Series in Computing Science)*. Prentice Hall, April 1985.

[Hud05] Scott Hudson. Lalr parser generator in java. `http://www2.cs.tum.edu/projects/cup/`, June 2005.

[Inc09] Google Inc. The go programming language. `http://golang.org/`, August 2009.

[Joh79] Steven C. Johnson. Yacc: Yet another compiler compiler. In *Unix Programmer's Manual*, volume 2, pages 354–387. Holt, Rinehart, and Winston, New Your, NY, USA, 1979.

[JYSP07] Leon Jen-Yuan Su and Terence Parr. gunit - grammar unit testing - antlr 3 - antlr project. `http://www.antlr.org/wiki/display/ANTLR3/gUnit+-+Grammar+Unit+Testing`, August 2007.

[KB98] Sam Kass and Karen Bryla. Rock paper scissors spock lizard. `http://www.samkass.com/theories/RPSSL.html`, January 1998.

[KBS05] Jon Kerridge, Ken Barclay, and John Savage. Groovy Parallel! A Return to the Spirit of occam? In Jan F. Broenink, Herman Roebbers, Johan P. E. Sunter, Peter H. Welch, and David C. Wood, editors, *Communicating Process Architectures 2005*, pages 13–28, September 2005.

[Lea99] Doug Lea. *Concurrent Programming in Java(TM): Design Principles and Pattern (2nd Edition).* The Java Series ... from the Source. Addison-Wesley Professional, second edition, 1999.

[LF08] Michael Leuschel and Marc Fontaine. Probing the depths of CSP-M: A new FDR-compliant validation tool. In *Formal Methods and Software Engineering*, volume 5256 of *Lecture Notes in Computer Science*, pages 278–297. Springer Berlin / Heidelberg, 2008.

[Mar08] Robert Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall, first edition, 2008.

[MD06] Jacob R. Matijevic and Elizabeth A. Dewell. Anomaly recovery and the mars exploration rovers. AIAA, 2006.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*, volume 0 of *Prentice-Hall International Series in Computer Science.* Prentice-Hall, 22, illustrated edition, 1988.

[Mey95] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[MHWN09] Sarah Mount, Mohammad Hammoudeh, Sam Wilson, and Robert Newman. CSP as a Domain-Specific Language Embedded in Python and Jython. In Peter H. Welch, Herman Roebbers, Jan F. Broenink, Frederick R. M. Barnes, Carl G. Ritson, Adam T. Sampson, Gardiner S. Stiles, and Brian Vinter, editors, *Communicating Process Architectures 2009*, pages 293–309, November 2009.

[Moo99] James Moores. CCSP - A Portable CSP-Based Run-Time System Supporting C and occam. In Barry M. Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 147–169, mar 1999.

[OB06] Bojan Orlic and Jan F. Broenink. SystemCSP - Visual Notation. In Peter H. Welch, Jon Kerridge, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2006*, pages 151–177, September 2006.

[Ode07] Martin Odersky. The scala programming language. `http://www.scala-lang.org/`, January 2007.

[Par04] Terence Parr. Stringtemplate template engine. `http://www.stringtemplate.org/`, February 2004.

[Par07] Terrence Parr. *The definitive ANTLR reference: building domain-specific languages.* Pragmatic Bookshelf Series. Pragmatic, 2007.

[Par09] Terrence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages.* Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2009.

[Pet02] Claude Petitpierre. Synchronous Active Objects Introduce CSP's Primitives in Java. In James S. Pascoe, Roger J. Loader, and Vaidy S. Sunderam, editors, *Communicating Process Architectures 2002*, pages 109–122, September 2002.

[PQ95] Terrence Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[RRS03] Varsha Raju, Luming Rong, and Gardiner S. Stiles. Automatic Conversion of CSP to CTJ, JCSP, and CCSP. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 63–81, September 2003.

[Seb07] Robert Sebesta. *Concepts of Programming Languages (8th Edition).* Addison Wesley, 8 edition, 2007.

[Sip05] Michael Sipser. *Introduction to the Theory of Computation.* Course Technology, second edition, 2005.

[Slo10] Mikolaj Slomka. Visual occam: High level visualiztion and design of process networks. Master's thesis, University of Nevada, Las Vegas, August 2010.

[Son08] Sonatype Company. *Maven: The Definitive Guide.* O'Reilly Media, Inc., first edition, 2008.

[SP11] Matthew Sowders and Jan Pedersen. Mobile process resumption in java without bytecode rewriting. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 2, pages 499–505. WORLDCOP, CSREA Press, July 2011.

[Suf08] Bernard Sufrin. Communicating Scala Objects. In Peter H. Welch, Susan Stepney, Fiona A.C. Polack, Frederick R. M. Barnes, Alistair A. McEwan, Gardiner S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, pages 35–54, September 2008.

[Tho10] Kresten Krab Thorup. Welcome to erjang! `https://github.com/trifork/erjang/wiki`, September 2010.

[Top82] Rodney W. Topor. A note on error recovery in recursive descent parsers. *SIGPLAN Not.*, 17:37–40, February 1982.

[Tro94] John A. Trono. A new exercise in concurrency. *SIGCSE Bull.*, 26:8–10, September 1994.

[Tur37] Allen Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, s2-42(1):230–265, 1937.

[W$^+$03] Peter Welch et al. Communicating sequential processes for java (jcsp). `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`, June 2003.

[WB05] Peter Welch and Frederick R.M. Barnes. Communicating Mobile Processes: introducing occam-π. In Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[WBM03] Rebecca Wirfs-Brock and Alan McKean. *Object design: roles, responsibilities, and collaborations.* Addison-Wesley Object Technology Series. Addison-Wesley Professional, illustrated edition, 2003.

[Wir78] Niklaus Wirth. *Algorithms + Data Structures = Programs.* Prentice Hall PTR, 1978.

Vita

Graduate College
University of Nevada, Las Vegas

Matthew Sowders

Degrees:

Bachelor of Science, Computer Science 2009
University of Nevada, Las Vegas

Thesis Title: ProcessJ: A Process-Oriented Programming Language

Thesis Examination Committee:

Chairperson, Dr. Jan Pedersen, Ph.D.
Committee Member, Dr. Laxmi Gewali, Ph.D.
Committee Member, Dr. Evangelos Yfantis, Ph.D.
Graduate Faculty Representative, Dr. Aly Said, Ph.D.