Lee Benfield

jOcc - An Occam to Java (Source & Bytecode) Compiler

Examination:	Computer Science Tripos (CST) Part II
College:	Robinson
Year:	2000

Proforma

Name:	Lee Benfield
College:	Robinson
Examination:	CST pt II
Year:	2000
Approximate word count:	10,500
Project Originator:	Dr Alan Mycroft
Project Supervisor:	Dr Martin Richards

Original Aims

The original aims of this project were to implement a compiler for the core features of the Occam language (based on Occam version 2.1). The targets of the compiler were to be both Java source code and Java Virtual Machine Bytecode.

Work Completed

A functional subset of the language was selected; a working compiler as described above was constructed for this subset. Modules for Lexing, Parsing and typechecking were written, and two seperate back ends were written for the compiler, one to construct Java source code output, the other to construct Java class files.

Special Difficulties

None

Declaration of Originality

I, Lee Benfield of Robinson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

Chapter 1

Introduction

1.1 Project Overview

The intention of this project was to create a compiler for the language Occam 2.1 which would produce both Java source code and Java Byte code. The compiler (for both targets) supports all of the primitive features of the language as well as additional features such as named data types, records, procedures etc.

Java makes a particularly interesting target, due to its cross platform nature. Occam, by virtue of its close relationship with the Transputer, is not considered a particularly portable language.[1]

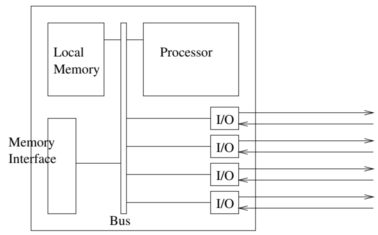
1.2 The Inmos Transputer

The Inmos Transputer (from *trans*istor com*puter*) has a single chip microcomputer architecture, designed for parallel use in Multiple Instruction, Multiple Data (MIMD) configurations. The T414, Inmos's first production transputer had

- a RISC architecture
- 6 registers (3 stack, 1 Program Counter, 1 Stack Pointer and 1 Operand register)
- 4 bi-directional memory mapped serial I/O ports, capable of 10 Mbits/s.
- Instruction level support for multi threading.

The link interface between two transputer processors handles all communication and synchronisation. When a task is waiting for I/O, it is placed in the idle queue, and only woken on I/O completion. All 4 links can operate concurrently.

[1] Occam compilers targeting Unixen do exist and a windows implementation of KrOC is underway, but nothing as cross platform as Java.



The links enable any number of transputers to be joined together to form a network which can operate in parallel. Therefore a process containing concurrent subprocesses can utilise either 1 or many transputers.

1.3 The Language Occam

> The Transputer and Occam were designed together. All transputers include special instructions and hardware to provide maximum performance and optimal implementations of the occam model of concurrency and communications.[2]

Occam is a High Level Language which was designed by INMOS and Oxford University in parallel with the Transputer. The development of both the Transputer and proto-Occam (Occam 1) were greatly influenced by C. Hoare's Communicating Sequential Processes (CSP). [9]

Occam 2, (upon which this project is based) also allows complex data types using the RECORD keyword.

Due to their closely linked development, Occam primitives map extremely well to Transputer instructions. As will be discussed, Occam also maps well to CSP primitives, which allows CSP specifications to be transformed into Occam programs, as well as providing Occam with a strong formal basis.

A program in Occam can be represented as a collection of Processes, all of which can be regarded as black boxes which start, perform a number of actions and terminate. Processes can be combined using concatenation and parallelism.

Ultimately, all Occam processes are constructed from three primitive processes - assignment, input and output.

An assignment computes the value of an expression and sets a variable to that value.

Input and Output are used for communicating between two processes. A pair of concurrent processes communicate using a one way channel connecting the two. One process outputs a message to the channel, and the other inputs it from the channel. (Here we can see the strong resemblance to CSP[3])

[2] INMOS Ltd, Transputer Development System (Prentice Hall) p 6.
[3] CSP P. 134

1.4 Previous Work

Previous work has been made on compiling Occam for use in systems other than transputers. The Southampton Portable Occam Compiler, SPOC [2] compiles Occam into ANSI C which can then be compiled using native tools.

Also available is KROC [3], the Kent Retargettable Occam Compiler. KROC works by translating code produced by an INMOS OCCAM Toolset compiler into the native assembler for the target architecture and linking in a small (< 2k) kernel that provider the process scheduling and message passing functionality of the transputer micro-code.

JavaPP[8] is a project which introduces the CSP/Occam thread model over Java threads.

Chapter 2

Preparation

2.1 Basic Occam Language Constructs & Semantics

I was not acquainted with Occam prior to this project. As it is (now) considered a fairly esoteric language, it is valuable to discuss the languages' basic constructs, and give examples of their use where relevant.

It should be noted that a strong similarity exists between Occam constructs and the CSP semantics which reflect them. This is due to the large influence of CSP over the design of Occam. In some circumstances, CSP specifications can be translated into Occam with very little alteration.

2.1.1 SKIP and STOP

2.1.2 Sequential Composition

```
SEQ
  A
  B
```

Semantics

(A ; B)

is the process which behaves like `A`, and on termination of `A`, behaves like `B`. If `A` never terminates, `(A ; B)` never does.

Replicated SEQ

```
SEQ i = base FOR length
  P
```

A replicated SEQ construct as given here will perform process `P` `length` times, with `i` iterating from `base` to `base+length-1`. This can be compared with the FOR construct which is provided in many imperative languages (ie C, Modula-3).

Semantically, the above statement is identical to

```
SEQ
  VAL i IS base:
  P
  VAL i IS (base+1):
  P
  ...  etc
```

however, it is decidedly more convenient, and since *length* can be a variable, allows non-fixed length "loops".

2.1.3 Assignment & Parallel Assignment.

Occam has the familiar form of assignment found in many languages, namely

```
a := e
```

which evaluates the expression '`e`', and then stores the result in '`a`'.

Also allowed is multiple (also called parallel) assignment, where the expressions on the right of the := are evaluated, then assigned to the lValues in parallel.

```
a, b := b, a
```

has the effect of swapping the values of `a` & `b`.

Occam allows more interesting forms of parallel assignments; if f(x) is a function (sec. 2.2.3) which returns two values, it can be used in the place of two rValues in a parallel assignment.

Semantics

```
(a := e; P)
```

is a process which behaves like `P`, except that the initial value of `a` is defined to be the value of expression `e`.

This generalizes to multiple assignments by allowing `x` to stand for a list of distinct variables

$x = x_0, x_1, x_2, ..., x_{n-1}$

and e stands for a list of expressions

$e = e_0, e_1, e_2, ..., e_{n-1}$

x := e

assigns e_i to x_i for all i. Note that all e_i are evaluated before any of the assignments are made.

2.1.4 Concurrent Composition

```
PAR
  A
  B
```

has the semantics

(A || B)

ie the processes `A` and `B` are concurrently executed.

`||` is both symmetric and associative.

Note that the above process only terminates when both processes `A` and `B` have terminated. If either of `A` or `B` was the simple process `STOP`, the `PAR` construct would not terminate.

Replicated PAR

As with replicated SEQ, it is possible to execute P n times in parallel by issuing

```
PAR i = 1 FOR n
  P
```

Unlike the replicated SEQ instruction, n is required to be a constant by the specification. However, see section 3.7.4 for implementation details.

2.1.5 Communication

When two processes are executing in parallel, communication is achieved by using the *input* and *output* processes. Two processes can only communicate if they share a channel over which to do so.

Output

```
c1 ! x
```

is a process which outputs the variable x over channel c1.

Input

```
c1 ? y
```

is a process which reads a value from the channel c1 and assigns it to y

Hoare makes the definition that communication on a channel should only occur between two processes, and should be unidirectional. Occam enforces this.

(c1 ! $x \rightarrow$ P)

defines a process which outputs x to channel c1and then behaves like P. It should be noted that Input and Output are both synchronous; an output will not complete until a corresponding input has accepted its data. Hence the process

```
PAR
  c1 ! x
  SKIP
```

will behave identically to the process `STOP`.

2.1.6 ALTing

An ALT(ernation) construct combines a number of processes, only one of which is executed. Each of the processes is guarded by an optional boolean expression and a channel input (a *guard*). The ALT performs the first process whose guard becomes ready. If two processes are both ready simultaneously, one is chosen non-deterministically.

```
ALT
  BoolA & Channel1 ? x
    P
  BoolB & Channel2 ? x
    Q
```

The ALT consstruct is needed in order for a "Farmer" process to combine results from several workers. (See Appendix A.3)

2.2 Additional Occam Constructs

2.2.1 Variable Abbreviations

Variable abbreviations allow renaming of variables - as in the Procedure example (see 2.2.4).

An example of variable abbreviation is

```
INT x:
VAR y IS x:
SEQ
  y := 3
```

This has the effect of setting x to 3. Note that once a variable has been abbreviated for, it cannot be assigned to, and once an array element has been abbreviated for, elements in the indexing expression cannot be assigned to.

eg:

```
VAR z IS p[i]:
```

neither p[...] nor i can be assigned to after this, though further abbreviations to other elements of p can be made.

2.2.2 Value Abbreviations

Value abbreviations are defined similarly to variable abbreviations:

```
VAL x IS p+q:
```

The expression 'p+q' is evaluated at the time the abbreviation is encountered, and the result is assigned to x. x cannot be subsequently assigned to.

2.2.3 Functions

Occam allows definition of functions, named blocks of code that can be called from anywhere after their definition (Forward references are not supported by the language, therefore neither is recursion).

A function can take 1 or more arguments under a Call-by-value scheme, and can return 1 or more results (which can be used in a multiple assignment).

Functions are subject to several rules:

- Must be side effect free
  - Cannot use any channels
  - Cannot assign to any variables not defined inside function
  - Cannot assign to any of the function arguments

  (This is made explicit, as all function arguments have to be declared as VALs).

- Cannot contain any PARallel constructs

2.2.4 Procedures

Procedures in Occam are named processes, and hence have no return value(s). Procedures are evaluated by aliasing the formal parameter(s) to the named parameter(s) and executing the procedure body in-place.

Hence

```
PROC square(INT x)
  x := x*x
:

INT y:
SEQ
  y := 3
  square(y)
```

Is semantically equivalent (and could be transformed to)

```
INT y:
SEQ
  y := 3
  VAR x IS y:
  x := x*x
```

2.2.5 Named Data Types

Occam allows new data types to be derived from the primitive data types. A derived type has the same properties as the original. Consider:

```
DATA TYPE apple IS REAL32:
DATA TYPE orange IS REAL32:
```

A value of type “apple” could not now be assigned to a variable of type “orange”, without an explicit cast. This allows additional type constraints to be imposed by a user to be sure an apple is not confused with an orange.

2.2.6 Record Data Types

Record data types can be used to construct more complicated types (as in C structs) from defined types (whether named types, primitive types or other record types). An example is

```
DATA TYPE myType
  RECORD
    [5]INT x:
    INT y:
:
```

To refer to a field inside a record, an expression identical to array addressing is used. *ie* to refer to the integer field y of a myType record p, p[y] would be used.

This is somewhat awkward for a compiler writer - consider the following code:

```
myType p:
INT x:
SEQ
  x := 3
  p[x][x] := 2
```

In the expression p[x][x], the first x refers to the field 'x' of p, which is a 5 element INT array. The second x is the variable defined above it. This syntax can prove quite cumbersome, and it would perhaps have been simpler to define a record indexing operator, such as C's . or -> operators.

Despite this, jOcc supports the syntax defined above.

2.3 Usage Rules for Concurrent processes

Occam has several rules for concurrent use of variables and channels, which are either identical to or minor variants on the original CSP rules.

2.3.1 Parallel Assignment

As described in section 2.1.3, CSP states (page 186) that in a parallel assignment, the list x has to be composed of unique variables. Occam requires this rule be enforced, i.e.

```
x, x := y, z
```

is invalid.

2.3.2 Use of variables in Parallel processes

The Occam reference manual states that

> A variable ... which is assigned to in a component of a parallel, may not appear in any other component of the parallel.

This is formalized by Hoare in (CSP page 188). If we define:
var(P) to be the set of variables that may be assigned within P.
acc(P) to be the set of variables that may be accessed in expressions within P.

Hoare states that if `P` and `Q` are joined by `||`, we should stipulate that:
$var(P) \bigcap acc(Q) = var(Q) \bigcap acc(P) = \{\}$
By enforcing these rules, we remove the need to worry about concurrent access to variables for updating, and hence do not need to place any protection mechanisms inside concurrent processes.

2.3.3 Use of Channels

Again, the Occam reference states:

> A Channel implements a point-to-point communication between two parallel processes... A Channel may only be used in one component of a parallel for input and in one *other* component for output. (my italics)

Clearly, using a channel in the same component of a parallel for both input and output would be nonsensical, as deadlock would be created.

2.4 Java

Java was designed as a "object oriented, portable, multi-threaded language"[1]. This makes it a natural target for compilation to, given that there are few platforms capable of using Occam, but a wide user base for Java; Java's native support for multi threading allows a clean implementation of Occams' parallel processes.

Java's class files are quite compact; given this and the widespread acceptance of Java within the computing community, which has led to the creation of hardware Java VMs such as the javaButton (IBM), it is a clear advantage to be able to retarget Occam code to Java, allowing Occam programs to be executed on devices which would not traditionally be considered viable, such as the Psion 5MX!

2.4.1 The Object Paradigm

Java was designed as an Object Oriented language, and Occam does not map particularly well onto this paradigm. However, we can represent an entire Occam program as one class, and Occam processes as methods of that class. Processes which execute sequentially can be combined into the same method. (See Section 3.7.4 for a full explanation).

2.4.2 Concurrency under Java

Concurrency under Java is based on Hoare's concept of Monitors[2], an idea which preceded his introduction of CSP.

Monitors allow an object to declare methods as synchronised, such that only one thread can be active in those methods at any one time. This is needed to protect resources shared between two threads, as this example demonstrates:

```
1: public class counter {
2:   int myValue;
3:   void increment () {
4:     int oldVal = myValue;
5:     oldVal++;
6:     myValue = oldVal;
7:   }
8: }
```

Consider two Threads running in parallel which use the same "counter" object. If they both call its increment method, the counters value should increase by two. However, if thread A calls increment, which gets to line 5, then thread

[1] A good overview of Java can be found at http://java.sun.com/docs/overviews/java/java-overview-1.html

[2] 1974. Hoare, C.A.R. Monitors: An Operating System Structuring Concept, Comm. ACM 17, 10:549-557 (October)

B becomes active and calls increment, which completes, then myValue only increases by 1.

It is therefore neccessary for Java to have the 'synchronized' keyword, which forces only one thread to be active inside certain methods at any one time.

The example above, though slightly contrived, shows how easy it is to make a mistake when updating variables concurrently. Fortunately, we do not have to consider concurrent writing to variables when mapping Occam processes onto threads, due to the rules given in section 2.3.

By adhering to these rules, an implementation of Occam can use Java's threading model without any concern over concurrent access, and hence there is no need to use Synchronised methods, with the exception of for simulating CSP for channels and ALTing.

2.4.3 Multiple methods, multiple buffers

Consider the Occam process given in section 3.7.4. As we represent each component of a PAR as a seperate method in Java, it is neccessary at line 4 to stop writing method process_1 and switch to writing method process_2 (and then write method process_3). Once this is finished however, we need to resume writing process_1.

It was decided to achieve this by holding multiple text buffers in memory, one for each method. Then a PAR node in the AST could be processed thus:

```
oldbuffer = currentbuffer
for each component of PAR
  currentbuffer = getNewBuffer()
  translate component
currentbuffer = oldbuffer
```

This method provides a clean way of handling arbitrary PAR constructs without the untidiness of creating temporary output files for each component. It does require that all output is first stored in memory, however this is not an unfeasible requirement.

2.5 The Java Virtual Machine

The Java Virtual Machine (JVM) is an abstract specification of a stack based computing machine with a variable instruction length; it knows nothing of the Java language *per se*, but does share common concepts and vocabulary. An excellent reference for the JVM can be found at [1]

2.5.1 Justification

Adding a Bytecode target for jOcc could seem an extravagance - javac will create bytecode anyway! However, I believe there are benefits to be had from having an (albeit semi-redundant) second target for jOcc. jOcc can be demonstrated to be flexible enough that two (quite different) backends can be added, demonstrating the benefits of a modular design.

We also have the option to improve on the bytecode that javac can create. By investigating the class files produced from the compilation (with javac) of jOcc's Java output, it was noticed that a substantial portion of the class file ($\approx 15\%$) was taken up with information which was not neccessary for execution. (*ie* lists of source code line to byte code address mappings).

Furthermore, domain specific knowledge can be used to remove redundant exception handlers. An example is a call 'join' at the end of a PAR block; we **know** that there will be no exceptions thrown, but the java spec states that a Java compiler should bracket 'join' with try...catch. It is neccessary to do this when jOcc outputs Java, but we don't have to have the baggage when we output Bytecode directly.

2.5.2 Bytecode instructions

As mentioned, the JVM instruction size is variable. This is particularly notable in that there are special instructions used for loading small, frequently used integer constants. Consider these two, semantically identical instructions:

```
iconst_0
iconst      0
```

The first is 1 byte in size, the second two. The JVM is peppered with such instructions.

2.5.3 Threads under the JVM

The JVM has low level support for Threads; each thread of execution has its own working memory with copies of global variables (though global variables are not used in this compiler) and stack.

2.5.4 Frames

A JVM frame is used to store local variables and an operand stack. Each time a thread of execution invokes a method, a new frame is allocated with enough space for all local variables and the maximum size to which the operand stack can grow. (This size can be calculated at compile time).

When a method is invoked, a new frame is created and becomes the current frame - when that method terminates, the frame is discarded, and the previous frame becomes current.

2.5.5 Class files

Each Java Bytecode class is stored in a Class file (standard extension ".class"). These files have a clearly defined format which details member variables, inheritance information, and method code (plus other miscellaneous information).

The JVM reference mentioned earlier has a good discussion of the Class file format.

2.5.6 Class File Reader (CFR)

The standard distribution of Java comes with a utility **javap**, which displays the contents of Class files as human readable opcodes. Unfortunately, javap requires the Class file to be verified[3] prior to displaying it.

Since javap cannot be used to display erroneous class files (and hence was not able to display output from developmental stages of this compiler), it proved neccessary to construct a similar utility which could cope with partial class files. **CFR** is a standalone utility which was written in the course of this project. Its construction has served as a useful teaching aid to the class file format. (See Appendix C)

2.5.7 The ConstPool

One interesting feature of a class file with which it is important to be familiar is the ConstPool. All references to class names, method names, exceptions etc, are actually given as references to the ConstPool. This saves substantially on space, but increases the complexity of outputting Bytecode to a class file.

A detailed discussion of the ConstPool structure is available in [5].

2.5.8 Percieved difficulties at time of Preparation

It has already been mentioned that the JVM is stack based. It was realised that this could pose an interesting challenge; there is a substantial difference between writing

```
x = new jObj(3)
```

and

```
new 5             // 5 would refer to the constpool entry for jObj
dup
iconst_3
invokespecial 8   // 8 - <init> for constpool #5
astore_1          // variable #1 on frame.
```

It was decided to implement a pseudo JVM instruction set (*ie* an intermediate language) which had *very little* difference from the real instruction set, save for variable naming, and ConstPool management. The details of this are discussed in Section 3.8; however it should be noted that prior investigation saved a great deal of time when it came to implementing the Bytecode backend.

[3] For security purposes, java VMs perform a verification check on all class files as they are loaded.

Chapter 3

Implementation

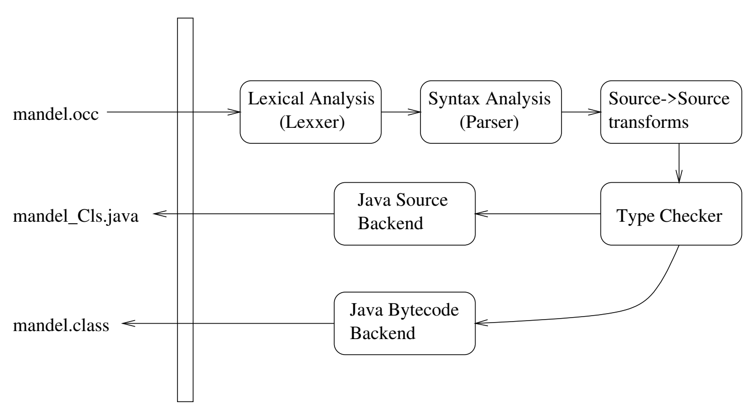
3.1 Overview

This project can, broadly speaking, be split into five distinct modules.

- Tokenizing input stream (Lexical Analysis).
- Constructing an Abstract Syntax Tree (AST) from the token stream (Syntax Analysis).
- Optimizing Source → Source transforms.
- Performing Type checking of the AST.

  At this stage, it is also convenient to

  – Tag AST nodes with extra information.
- Conversion of AST to Java Source Code.
- Conversion of AST to Java Byte Code.



The compiler was written entirely in C (using gcc, yacc & lex), except for a few support modules (regression tester, header file generators) which were written in Perl.

3.2 Lexical Analysis

The initial input to the compiler is an Occam source code program containing freeform ASCII text. This is converted by the Lexical Analysis module into a stream of *tokens*, which can then be passed to the Syntax Analysis stage.

A standard tool, Flex[1], was used to construct the Lexical Analysis Module.

Comments are removed from the Occam code at this stage, and each token is returned with a line and character number, in order to allow informative error messages.

Token insertion

As mentioned, Occam relies on whitespace to delimit functional blocks. It is convenient to disregard whitespace information and instead output PASCAL style BEGIN/END tokens to the Syntax Analysis module.

To this end, instead of calling the standard yylex() function, a wrapper function, yyylex() was added. This function notes changes in indentation level, and inserts BEGINBLK and ENDBLK tokens appropriately.

Occam		Token Stream
`SEQ`		`SEQ BEGINBLK`
`  SEQ`		`SEQ BEGINBLK`
`    x := 2`	⇒	`NAME (x) ASSIGN DIGITS (2) EOL`
`  x := 3`		`ENDBLK`
		`NAME (x) ASSIGN DIGITS (3) EOL`
		`ENDBLK`
		`EOL`

In order to help locate errors, the Lexing module keeps track of the line number of the source file being processed.

3.3 Syntax Analysis

Given a stream of tokens, the Syntax Analysis Module (Parser) constructs an AST which represents the syntax of the original Occam program.

At this stage it is possible to detect a program which is entirely composed of valid tokens, but which is not syntactically valid in the language for some reason, *eg.*using a keyword as a variable name.

[1]Fast Lexical Analyser Generator, http://www.gnu.org/manual/flex-2.5.4/html_mono/flex.html

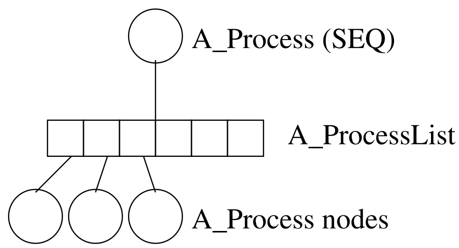
The previous insertion of BEGINBLK and ENDBLK tokens allow a context free grammar to be created for Occam, based on the syntax given in Appendix H of[2].Another standard tool, yacc[3] was used to create the Syntax Analysis module.

Yacc has the facility to attatch semantic actions to grammar rules. In this way, AST nodes can be created upon reduction of a grammar rule.

An example of this is:

```
SEQ separ BEGINBLK proclist ENDBLK {$$=(A_process) A_SeqProcess(EM_linePos,$4);}
```

(A proclist is a linked list of `A_process` nodes). This creates a portion of an AST as follows:



`EM_linePos` is the current line number obtained from the Lexer, as mentioned above.

3.3.1 AST Node types

The parser makes use of four types of AST nodes, classified to present a clear correspondance to the Occam code being parsed.

- A_process

  Operations such as assignment, input and output are represented using A_process nodes.

- A_spec

  Specifications such as declarations of variables, abbreviations, functions & procedures are represented using A_spec nodes.

- A_exp

  All AST nodes which contain an Lvalue or an Rvalue are of type A_exp. Each of these nodes has an associated A_dataType.

- A_dataType

  A_dataType nodes are used to tag A_exp nodes with the data type of that expression.

[2]SGS-Thompson Microelectronics Ltd, Occam 2.1 reference manual.
[3]Yet Another Compiler Compiler

These nodes are internally held as C structs; in order to allow a node to have a sub type (so an A_process PAR node is different from an A_process SEQ node), each struct has a 'kind' tag. A union contains all the data for the node. See Appendix D for example implementation.

3.4 Source → Source Translations

3.4.1 Flattening PARs

Translating as described in 3.7.4, the following code would be written a four java methods (and hence require four threads to be forked as described).

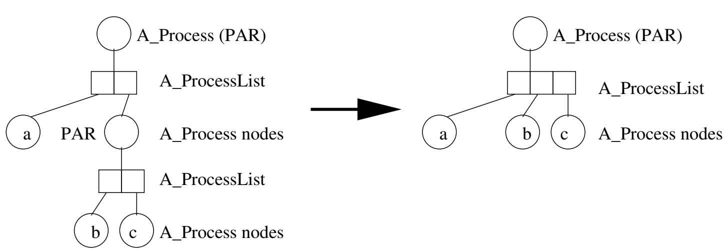
```
PAR
  a
  PAR
    b
    c
```

However, this is semantically equivalent to

```
PAR
  a
  b
  c
```

which requires only three methods and threads.

In order to perform this transformation, the AST is stepped over, until a PAR node is encountered. This PAR node is examined, and, if any PAR nodes exist as children, their A_ProcessList lists are combined into its list, and those (children) nodes are removed. (This is performed recursively, so a tree of PAR nodes would collapse to one node).



3.5 Type Checker Module

Unlike weakly typed languages *(cf* Lisp or Prolog), Occam is a strongly typed language; *ie* every variable has to have a data type associated with it[4], which

[4] With the exception of CHAN OF ANY

is (except sometimes in the case of Function, Procedure, VAL and VAR declarations, where the type is implicit) explicitly declared at the same time as the variable.

Because we do not permit forward declarations (ie something can only be used once it has been fully defined), it has been possible to use the type checking pass of the AST to perform additional work, such as scope flattening and, to a limited extent, re-writing of the AST to minimize the work which has to be done by the translation phase. (since there are two translation back ends, this was required to avoid duplication of code).

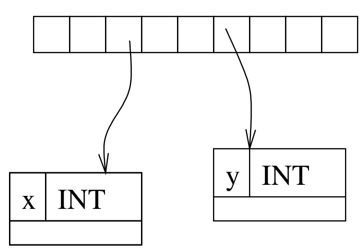
3.5.1 The Environment

The compiler keeps track of which Variables, Functions and Procedures are in scope using a Hash table and a Stack.

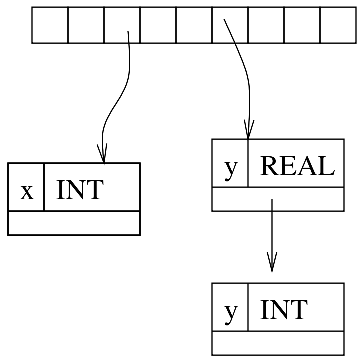
When a variable is declared, its name is pushed onto the "scope stack" and an entry is inserted into the hash table. By using an externally chained hash table, variable hiding is naturally implemented. Consider the following example:

		Scope Stack
1	`INT y:`	{}
2	`INT x:`	{y}
3	`SEQ`	{x,y}
4	`  x, y := 3, 2`	{x,y}
5	`  REAL32 y:`	{x,y}
6	`  y := 1.34`	{y,x,y}
7	`  y := 12`	{x,y}

The environment at lines 4 & 7 is represented by the Hashtable



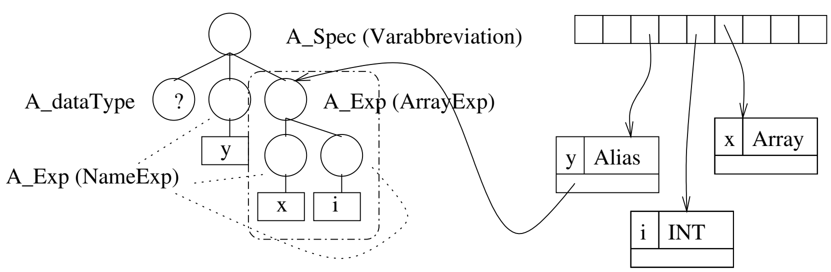
The declaration of `REAL32 y` at line 4 is valid for the immediately following process, namely line 5. Thus the scope Hash at line 5 is represented by



looking up the data type for y in the hash will now return A_realDataType, since the initial declaration of y is hidden. Since a declaration is only valid for the immediately following process, the `REAL32 y` goes out of scope after line 6, and is removed from the hash. The `INT y` is again in scope.

VAR (alias) declarations

A special case of the scheme above are Abbreviations, (see section 2.2.1). In this case, we want to expand the Alias into whatever is being aliased for, allowing us to 'undo' its use. In order to do this, consider a portion of the AST representing `VAR y IS x[i]`.



A "Varabbreviation" specification contains an (inferrable, therefore optional) datatype and two Expressions: the abbreviation and what is being abbreviated. An entry for the alias is added to the Scope hash, with a pointer to the sub-tree of the AST containing the Expression being abbreviated. It is now possible, each time an `A_NameExp("y")` expression occurs, to replace it with `A_ArrayExp(A_NameExp("x"),A_NameExp("i"))`.[5]

Named Type declarations

(As defined in section 2.2.5).

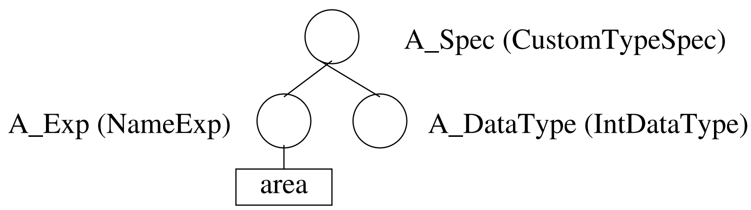
The type checker handles named data types as follows:

[5]This does not pose problems if x or i are re-defined, since the nodes in the AST are tagged with their "unique" names. ('scope flattening')

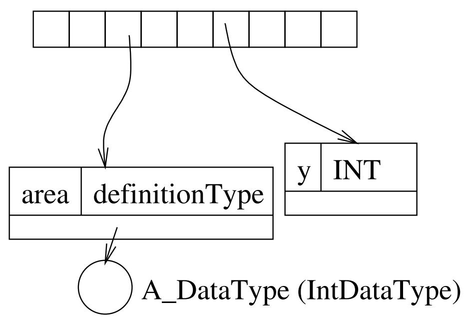
If the base type is a primitive type or a record type (or an array of either), then an entry is placed in the symbol table for the new type name, with a data type of "`DefinitionType`", and a pointer to the base type. Consider this example:

```
DATA TYPE area IS INT
```

would be represented in the AST by



and the environment would be modified to give:



If the base type is itself a Named Type, then the base type of that is looked up, to give a true base type, which is treated as above.

By treating Named types as described (adding entries to the symbol table), we create a perhaps undesirable effect of pushing a variable out of scope by defining a new type. However, it should be noted that this is the behaviour seen in the INMOS reference compiler, so it was deemed to be the proper solution.[6]

Record declarations

Record declarations are handled much in the same way as named type declarations, except that a hash table is constructed for each record definition mapping each record field to its type. (The below diagram shows the symbol table containing the record definition in Section 2.2.6)

[6] No reference manual actually states how this should be implemented, however a conversation with the author of KROC (see 'Previous Work') indicates that the solution taken is the correct one.

3.5.2 Scope Flattening

In order to simplify the code which will be generated, the typechecker performs scope flattening on variables. When a variable is declared, a unique name is constructed for it (by appending `_x`, where `x` is a monotonically increasing integer). This unique name is held in the scope hash, and every time the variable name occurs in the AST, it can be tagged with the unique name.

3.5.3 Type Checking

As mentioned above, Occam does not support any form of forward declaration,

As it is possible to determine the type of all variables during the first pass, it is possible to check the validity of all assignments, inputs, outputs, function and procedure calls in a single pass over the AST.

During the traversal of the AST, each time an Expression node is encountered, its type is determined by calling `T_descendExp`. If the expression is simply a variable, the type is looked up in the environment; in the case of a complex expression, the sub-expressions are typechecked, and the type of the complex expression is returned.

Consider the below examples:

Assignments

```
a, b, c := doubleArgs(x), 3
```

When `T_descendProcess` is called with the AST node for the above assignment, an `A_dataTypeList` (linked list of `A_dataType` elements) is constructed for the LHS and RHS, by calling `T_descendExp` for each of the expressions in turn, and appending the results. We also check that none of the variables being assigned to are VAL abbreviations.

By allowing `T_descendExp` to return an `A_dataTypeList` containing more than one element, it is possible for the typechecker to handle assignments like the above.

Function Calls

Calling `T_descendExp` on the function call above would involve searching for the function "`doubleArgs`" in the functionTable hashtable, then comparing the types of each argument with the `A_dataTypeList` constructed when the function was defined.

This example also serves to show how the typechecker can be used to perform a level of semantic checking, since function calls which return two or more results are legal in Occam, but cannot be used to provide arguments to functions. Such a use of a function is trapped by the typechecker.

3.5.4 Functions and Procedures

Once a unique name has been assigned to Occam procedures and functions and their code has been typechecked, the definitions can be removed from the AST. They are stored in a hash table which maps each unique name to the portion of the original AST which defined it. Two hash tables are used for this purpose, functionTable and procedureTable. This allows us to convert them separately to the main Occam process.

3.6 Translation Modules

The compiler as written has two separate back ends, one which compiles to Java source code (hereafter “Java” for simplicity in naming) and a further one, which compiles to Java Bytecode (“Bytecode”).

The Java backend was created first; due to the human readable nature of the output it allowed relatively quick prototyping and error checking of the output.

Once the Java backend was considered complete, the Bytecode backend was constructed. The ideal output for the bytecode backend was considered to be (initially) an identical replica of **javac**'s output, when the Java backends' output was compiled. Subsequent optimizations to the output were made to reduce the size of the class file in comparison with **javac**'s effort.

3.6.1 Input to the Translation modules

The translation modules take a modified AST, in which each `A_NameExp` node has been given the correct unique identifier (see section 3.5.2), each PAR or replicated PAR node has been tagged with the list of variables in scope, and other modifications mentioned previously.

3.7 Translation to Java source code.

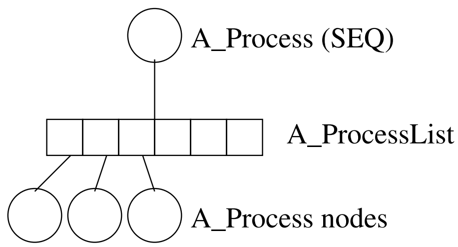
Below I describe how Occam processes can be represented as Java source code.

3.7.1 SKIP

Since `SKIP` performs no function, `A_Process` nodes of type skip `SKIP` can be safely ignored. For the purpose of readability, the Java back end will output ' `// SKIP` '. (The bytecode back end outputs a '`nop`'.)

3.7.2 Sequential processes

Occam's `SEQ` construct is represented in an AST by an `A_Process` node, which contains a list of processes to be executed, in order.



So the back end can treat SEQ processes as having a list of further nodes which need to be examined, in order.

3.7.3 Assignment

Simple assignment, eg `x := y` can be translated to a single Java statement, `x.set(y)` - note that use of encapsulation is neccessary as described in Section 3.7.5.

Multiple (parallel) assignment requires a little more thought; the statement `x,y := y,x` cannot be naively translated as two assignments, or a value will be lost. Instead, a Vector object is used to store a **clone** of each of the variables on the RHS of the assignment, and once all elements on the RHS have been evaluated, they are assigned to destinations:

```
assignVector = new Vector();
assignVector.addElement(y_1.clone());
assignVector.addElement(x_0.clone());
x_0.set((jObj) assignVector.elementAt(0));
y_1.set((jObj) assignVector.elementAt(1));
```

3.7.4 Parallel processes

Since there is no need to consider concurrent access to variables (Section 2.3.2), Occam processes can be mapped onto Java Threads neatly.

In order to minimize the complexity of the generated code, it was decided to declare all Threads in a single java Class. This is achieved by having an Object field which defines the Occam process to be executed, and for each process which needs to execute concurrently, creating a Java function which contains that process.

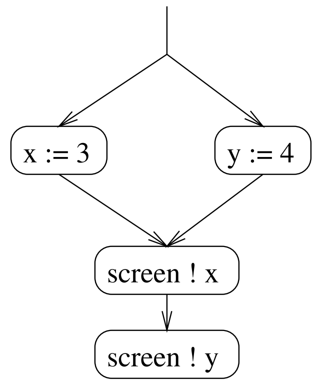
Each time a new Occam process needs to be forked, it is only neccessary to create a new instance of the Object, and specify which Occam process it is to run when the Object starts.

Consider the following Occam program

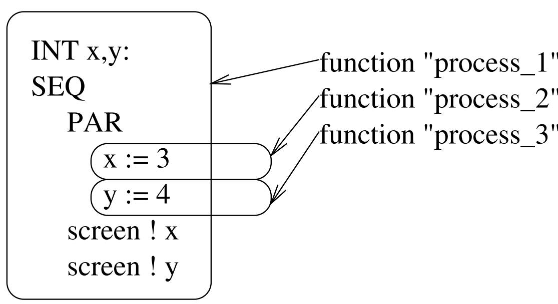
```
INT x,y:
SEQ
  PAR
    x := 3
```

```
  y := 4
screen ! x
screen ! y
```

which can be represented as



This would be compiled into three Java functions, delineated as shown:



`process_1` would create Objects to run `process_2` and `process_3`, when these terminated `process_1` would continue its execution (see below).

To fork a new Thread, it is only neccessary to create a new instance of the program Object, and specify which function it should run. eg:

```
new partest_Cls(3,display_0);
```

Displays

In order to execute two Occam processes concurrently, it has been neccessary to create a new Thread for each process, and call the Java function in which that process is defined.

However, this means that concurrent processes no longer have access to variables which were in scope when they were created, as these variables are on the local variable stack of the process which forked the Thread.

In order to overcome this, each time a PAR statement is encountered, the compiler tags it with a list of all variables currently in scope. In this way, when the translation module comes to write output, it can fill a Vector[7] (in the example above, display_0) with references to all the variables currently in scope. This is passed to the new process, which re-assigns elements from the Vector to variables, once again making them accessible.

In order to assign variables to a Vector, it is neccessary to wrapper them inside an Object, since a Vector cannot hold Javas' primitive datatypes.

Termination of Parallel processes

As mentioned 2.1.4, a PAR process only terminates when all processes defined inside it have terminated. Therefore, in the above example, no process defined lexically after `process_1` could start until `process_2` and `process_3` had terminated.

```
partest_Cls thread_0 = new partest_Cls(3,display_0);
partest_Cls thread_1 = new partest_Cls(4,display_0);
try {
  thread_1.join();
  thread_0.join();
} catch (Exception e) { }
```

This is acheived by assigning each new Thread to an Object as above, and then using Java's `Thread.join` method once all processes have been created. The join method has the effect of blocking until the object which is the subject of the join has terminated.

Replicated PARs

Replicated PARs can be translated in much the same manner; instead of creating a distinct object for each thread as above, an array is constructed for the threads. The requirement still exists for all threads to be *join*ed before the replicated PAR terminates, again, this can be achieved similarly to above.

i.e.

```
PAR i = 2 FOR 5
  P
```

would translate to the following Java source

```
partest_Cls [] parArray = new partest_Cls[5];
for (int parIterator=2; parIterator < (2+5); parIterator++) {
  parArray[parIterator-2] = new partest_Cls(3[8],display);
```

[7] from java.util.Vector - A Vector is somewhat like an array, except that its size can be easily altered.

[8] Here,the process P has been translated to sub par_thread_3,so the dispatcher needs to be passed that number.

```
}
try {
  for (int parIterator=2; parIterator < (2+5); parIterator++) {
    parArray[parIterator-2].join();
  }
}
```

A point worth noticing is how, by using an array and a for loop in this manner, a replicated PAR construct can fork as many processes as desired. A limitation of Occam[9] is that the `count` (amount of threads) be a fixed number. Since we do not have any hardware constraints, it is trivial to extend the specification to allow for variable sized replicated PARs.

3.7.5 jObj objects

The problem with representing parallel Occam processes in Java using the method described in 3.7.4 is that Javas' wrapper classes (java.lang.Integer, .Float, etc) are immutable.

This means that updating an Object 'a' can only be performed by creating a new Object, and assigning it to 'a'. Consequently, the original Object that 'a' pointed to has not been modified; the effect is that updates to any Object which has been passed to a Thread in a display will not be visible outside of that Thread; clearly unacceptable behaviour.

In order to get around this problem, we create a new class, which can be used to wrapper all the neccessary primitive types, called jObj. Data is encapsulated using traditional get and set methods. This allows several Threads to refer to the same variable, and for changes in it to be reflected.

(Since CSP forbids concurrent threads using a variable if one of the threads is to update it, this is only of use when a variable is updated in a component of a PARallel, then subsequently accessed after that PARallel has terminated.)

3.7.6 Channels (input and output)

Because of the restrictions placed in Section 2.3, channels provide the only way to communicate between two components of a PARallel. Recall that channels are defined to block when they are waiting to perform either input or output; therefore in order to implement a channel in Java we need only to implement an object which is capable of holding a single value (channels can not be passed over channels, unlike in π-calculus.)

We use Javas' polymorphism to implement a single class of channel, `Channel_Of_Object`, which has two methods, `put(x)` and `get()`, corresponding to the ! and ? operators; only objects of class java.lang.Object are used, explicit casting is output by the compiler where neccessary.

So to implement

[9] Which is of course perfectly valid, given that Occam code *should* execute on a transputer network.

```
PAR
  cx ! x
  cx ? y
```

we would produce the Java (called from a PAR construct as described above)

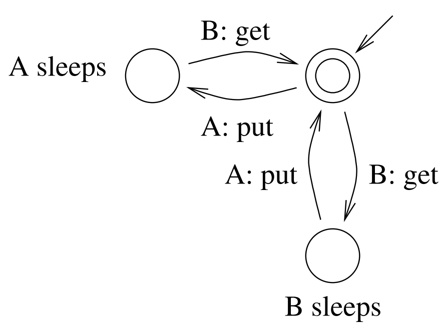
```
public void par_thread_3 () {
Channel_of_Object cx_2 = (Channel_of_Object) display.elementAt(1);
jObj x_0               = (jObj) display.elementAt(3);
cx_2.put(x_0);
}
public void par_thread_4 () {
Channel_of_Object cx_2 = (Channel_of_Object) display.elementAt(1);
jObj y_1               = (jObj) display.elementAt(2);
y_1.set( (jObj) cx_2.get());
}
```

(note the need for explicit casting to a jObj)

By handling channels this way, it is possible to create the Channel_Of_Object class seperately, and provide it for run-time linking.

How does the Channel Object work?

The Channel object performs all the neccessary synchronization of threads involved in communication. If we consider channel communication between process A (writer) and B (reader), the Channel Object has to exhibit the following behaviour:



Behaviour is complicated by addition of the ALT construct, see 2.1.6.

This can be constructed an a SRSW queue using Javas' 'synchronized' methods for concurrent processes as in Appendix E.

Reference was made to JavaPP [8] mailing lists when designing the Alternative and Channel objects.

3.7.7 Records

In order to handle records, it was decided to utilize Javas' facilities for creation of objects. Each record definition is translated into an appropriate Class definition, (once the record has been given the usual unique name)

```
DATA TYPE bob
  RECORD
    [5]INT x:
    REAL32 y:
:
```

would result in the (slightly cumbersome) definition of the class bob_x (here x=0)

```
public class bob_0 implements Cloneable {
  jObj[] x;
  jObj y;
  bob_0 () {
    x = new jObj[5];
    for (int zz_0 = 0; zz_0 < 5; zz_0++) {
      x[zz_0] = new jObj(0);
    }
    y = new jObj(0.0F);
  }
}
```

It is perhaps interesting to note that (despite the neccessarily complicated constructor) this provides a particularly elegant addition to the Java representation of Occam jOcc uses. An Occam expression

```
p[y] := 3
```

can be translated to

```
p->y = new jObj(3);
```

(given the record/array indexing problem, which will be further discussed)

Record accesses

Translation of Record accesses presents an interesting challenge (Section 2.2.6); all it is possible to determine in the parser is that an array access is made.

Given an expression of the form `p[y]`, the type of p is checked. If it is a record, then that record is checked for the presence of a field 'y'. (An error is flagged if the field does not exist). Otherwise a check is made that p is an array.

Therefore in the example given in Section 2.2.6,
p is a variable of type myType
p[x] is a field in p which is an INT array
p[x][1] is an INT

3.8 Translation to Bytecode

Much of the discussion above (compiling to Java source) was re-applied in the Bytecode backend of jOcc. In much the same was as outlined above, multiple output buffers were used, to allow PARallel components to be written to different methods.

Perhaps one of the most interesting features of the Bytecode back end is the internal representation of a JVM instruction. Since these instructions are variable length, it was decided to use a structure which was a superset of instruction formats. As mentioned in Section 2.5.8, it was also decided to create a slightly different form of instruction which would allow the backend to be as clean as possible, but be very easy to convert to a proper JVM instruction for output.

3.8.1 Internal instruction format (Intermediate Language)

A JVM instruction would be something like

```
invokespecial 8
```

Which is both difficult to write and difficult to read.

Since 8 is just a references to the ConstPool specifiying the class, name of method and the signature of the method, we do not resolve them until later. When issued, jOcc would contain the line

```
OutputCodeSimple(JVM_INSTR_INVOKESPECIAL,''jObj'',''<init>'',''()V'');
```

By calling this, a JV_lineOfMethod structure is filled, and appended to the current output buffer. The JV_lineOfMethod structure looks like this:

```
struct JV_LineOfMethod_ {
 int instruction;
 char * string1; char * string2; char * string3;
 float fArg;
 unsigned int Arg1; unsigned int Arg2;
 void * extraArgs; // for tableswitch etc.
 int posnInMethod;
};
```

The strings (string1, 2 & 3) contain text strings instead of references to the ConstPool. These strings are transformed into ConstPool references at a later pass.

The strings can also represent labels, or variable names. (see later).

To demonstrate how powerful this is, consider the line of java code

```
int zzx = -1;
```

In order to compile this into bytecode, it is neccessary to

- allocate a frame location for zzx.
- create an integer -1 on the operand stack
- store this in the location for zzx.

If all this was done directly, the backend code would quickly become unreadable. By adding the extra passes, it is possible to create the above instructions using the extremely readable code below:

```
OutputMethodCodeSimple (JVM_INSTR_iconstm1, NULL)[10];
OutputLocalVariableDec (''zzx'',''I'');
OutputMethodCodeSimple (JVM_INSTR_istore, ''zzx'');
```

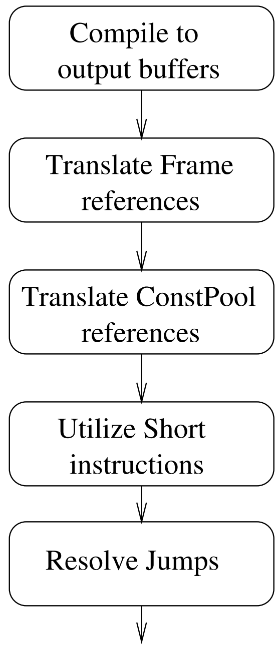
3.8.2 Output buffers

Instead of plain text output buffers as in the Java backend, the Bytecode back end uses linked lists of JV_lineOfMethod structs. An advantage of this is that a special item can be placed in the list to represent a label, and then painlessly removed at a later pass.

3.8.3 Passes involved in the ByteCode backend.

Because the internal representation of a JVM instruction differs from the correct representation, it is neccessary to have a few extra passes after the translation is complete. By not resolving any ConstPool entries until these passes, we can defer constructing the ConstPool until after most of the compilation is already complete, saving repeated lookups of entries.

[10] In order to abstract this further,a function OutputMethodCodeIval was created,which takes a single int,and creates an iconstx instruction,or an iconst instruction with an argument,depending on the integer (see section 2.5.2)



3.8.4 Compiling to Output Buffers

Though this represents the lions share of the work for the backend, the same techniques are used as in the Java backend (though obviously targetted to Bytecode). Hence this pass is not covered in detail.

3.8.5 Translate Frame references

By way of simplifying the backend, references to local (Frame) variables in the intermediate language are made using their symbolic names, instead of an index into the Frame. It is possible to go through all instructions in each buffer in a separate pass, and transform these references into Frame references by assigning each variable a Frame location.

Hence, loading 'a' from the frame, if a is the third variable encountered for that frame when translating references, the instruction that was created with

```
OutputCodeSimple(JVM_INSTR_ALOAD,''a'',NULL);
```

would become

```
aload 3
```

It is also possible to minimize the size of frame which needs to be allocated by performing live variable analysis on frame variables.

3.8.6 Translating ConstPool references

As mentioned (Section 2.5.7), ConstPool references are not used when outputting JVM psuedo-code. This requires an extra pass to be added to resolve all references. However, this is not particularly complicated, since a module for the maintenance of the ConstPool has been created.

In the style of the example given above

```
OutputCodeSimple(JVM_INSTR_INVOKEVIRTUAL,''jObj'',''<init>'',''()V'');
```

would be replaced with

```
invokevirtual 4
```

and the ConstPool would have the relevant entries inserted, unless they were already present.

3.8.7 Utilizing short instructions

As mentioned in Section 2.5.2, the JVM has instructions for special cases which are smaller than their common counterparts.

This pass goes through the constructed code looking for instructions of the form

`aload x` $x \in \{0,1,2,3,4,5\}$
`astore x` $x \in \{0,1,2,3,4,5\}$
`iconst x` $x \in \{-1,0,1,2,3,4,5\}$

and replacing them with

```
aload_0
astore_0
iconst_m1
```

etc.

This pass provides both a space and time improvement, since JVM implementations are typically optimized to better execute these special cases.

3.8.8 Resolving Jumps

Since Bytecode has no higher level constructs such as while, it is neccessary to represent them in terms of jumps. JVM Bytecode only uses relative jumps, so in order to make creating Bytecode simpler, the pseudo Bytecode allows labels to be used. In this pass, the location of each instruction is calculated, (simply by iterating through all instructions and adding their lengths), and stored in the posnInMethod field.

When a label is encountered, its location is set to that of the next valid instruction.

When a jump to a label is encountered, it is therefore simple to transform this to a relative jump, since each instruction (and label) has a location. The label elements in the linked list can be removed or ignored when the final code is output.

Chapter 4

Evaluation

The initial project proposal (Appendix F) called for compilation of Occam to Java source code, with a second aim of targetting the compiler to produce Java Bytecode. Both of these aims have been comprehensively met - the compiler successfully compiles the example programs given to both target languages, and the results of the compiled programs match those of the reference compiler (Kroc [3]).

4.1 Regression Testing

An important Software Engineering aspect of any project is testing of the finished product and, to as large an extent as possible, testing of the product prior to completion.

Since the design of the Compiler is inherently modular, we can construct a regression tester which tests each module and therefore provides more useful information than a simple pass/fail.

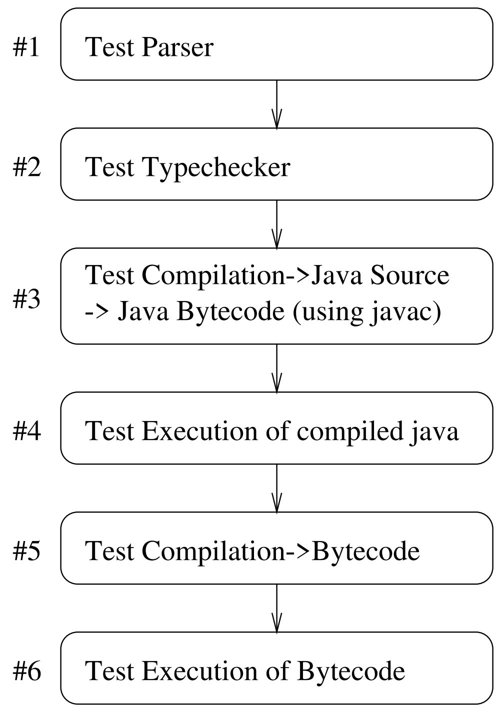
By using a modular approach, it allows us to clearly monitor the effect of adding optimization passes; in the unlikely event that a pass has an unforseen side effect, a well designed, modular approach to testing can save a great deal of time in tracking down the error.

4.1.1 Difficulties due to parallelism

Consider an Occam program which outputs the squares of 1 to 10, by using a PAR construct. Clearly, the 10 results could be in any order - Hence it is not possible for the regression tester to assume that a certain output is *the* correct output. One solution which was considered was to allow the regression tester to re-order output, however this would have the undesirable effect of potentially allowing failures to pass. Therefore, a category of pass was created which simply requires output, though this is weaker than testing for a specific output, it does solve the problem.

4.1.2 Stages of the Regression Tester

The stages of the regression tester closely follow those of the compiler itself, as shown in Section 3.1. They are:



4.1.3 Construction of the Regression Tester

For the purposes of this project, a custom Regression Tester was constructed using Perl 5.005_03, with a Berkeley DB backend for persistant storage of results.

Each program in the test suite has a corresponding entry in the database, giving the expected results. This is stored in the form of a 1-5 character string, such as "`.....`", where the digits can be one of

`.` Passes module test.

`X` Fails module test.

`d` Produces some output, does not fail test.

`!` Dumps core.[1]

[1] Clearly, this is not desirable behaviour, however when a program is blocked on all threads and cannot continue, this is the behaviour of the reference compiler.

The amount of times each file has been tested is also kept in the dB; this is displayed on testing.

4.1.4 Running the Regression Tester

The tester goes through the stages outlined above until there is a failure, at which point it stops the test for that file. The results of the test, which is a 1-5 character string as described above, are compared to the expected results and the program test is classified as a pass or a fail.

Consider a program which has a syntactical error in it (ie should not parse), which somehow manages to parse, typecheck and compile[2]. (but subsequently dumps core when the java source and bytecode are executed). This program would have an expected result of

"X"

however, given the occurence above, the actual result string would be

"...!.!"

This would be flagged a failure, and the compiler could be investigated to find the problem.

4.1.5 Example Regression Tester Output

```
25 files to check
1 alttest.occ                  #70 : ...... [OK]
2 arrayTest.occ                #70 : .X     [OK]
3 assign1.occ                  #70 : ...... [OK]
4 assign2.occ                  #70 : ...... [OK]
5 assign3.occ                  #70 : ....X  [OK]
6 assignment.occ               #70 : ...... [OK]
7 break.occ                    #70 : .X     [OK]
8 bubblesort.occ               #70 : ...... [OK]
9 complexDataType.occ          #70 : .X     [OK]
10 conv.occ                    #70 : ...... [OK]
11 fnassignment.occ            #70 : ...... [OK]
12 fnassignmentBadlyTyped.occ  #70 : .X     [OK]
13 invalidtyping.occ           #65 : .X     [Should pass stage 2. ]
```

.... etc

4.2 Using the compiler

What follows is an example of using the compiler. The program to be compiled is presented in Appendix A.3.

[2]This is unlikely...

4.2.1 Command Line Interface

In order to simplify interaction, a simple Perl script was created which serves as a command line interface to the compiler, much in the manner of the CLI for SML[3]. This command line allows a user to test parsing, test typechecking, compile to Java source code, or compile to Java bytecode.

```
[lab27@lab27 cli]$ ./jocli
JOC - Occam to Java (Source and Bytecode) Compiler V1.0
    (C) 1999-2000 Lee Benfield, University of Cambridge.
Type help for help.
>
```

as with most CLIs of this nature, there is a minimal online help facility, describing the available commands

```
> help
help                -- this text.
file filename       -- sets input file.
parse               -- parses input file (to test).
tcheck              -- typechecks input file (to test).
joc                 -- compiles to java source code.
jvc                 -- compiles to java bytecode.
list                -- display Occam code.
exec                -- execute compiled code.
quit                -- exit program.
>
```

The user can now specify which file is to be processed

```
> file farmer_giles.occ
Ok.
>
```

and choose to compile it into Java source. (jocli takes care of compiling using javac)

```
> joc
Compiled to Java source Ok.
Compiling farmer_giles_Cls.java using javac.
Compiled OK. Type exec to execute
>
```

or compile directly to bytecode..

```
> jvc
Compiled to farmer_giles.class Ok.
Compiled OK. Type exec to execute
>
```

[3] Standard ML of New Jersey.

Finally, the program can be executed:

```
> exec
Executing farmer_giles
385
>
```

4.2.2 Error Messages

There are two categories of error which can be flagged by the compiler, **parse** errors and **typecheck** errors (which are errors detected at the semantic analysis phase).

Parse errors

Consider a modified version of the Farmer program presented in Appendix A.3, with the definition of 'square' changed to (note the missing O in 'VALOF')

```
INT FUNCTION square(VAL INT x)
  VALF
    SKIP
    RESULT x*x
:
```

When we attempt to parse this, the compiler detects a syntax error, and reports it (along with some local context) with the offending line highlighted:

```
> file fg.occ
Ok.
> parse
There were errors:
7:  INT FUNCTION square(VAL INT x)
8:    VALF
    ^^^^^^^^
9:      SKIP
10:     RESULT (x*(x*x))
>
```

By clearly highlighting errors, the parser makes it easier to correct such problems.

Semantic Errors

Some errors cannot be picked up at the syntactic checking stage. Consider the farmer program again, with the farmer process altered to read:

```
    SEQ tmp = 0 FOR 10
      toWorkers[tmp] ! (tmp+1)
    REAL32 y:
    SEQ
      SEQ tmp = 0 FOR 10
        ALT tmp2 = 0 FOR 10
          fromWorkers[tmp2] ? y
            tot := tot + y
      screen ! tot
```

Notice how y has been declared as type 'REAL32'. This will obviously parse, but must be picked up by the typechecker.

When the program is typechecked, several problems are now highlighted:

```
> tcheck
There were errors:
Error: dataTypeLists contain different types at line 9
Error: Receiving wrong type of data from channel at line 26
Error: dataTypeLists contain different types at line 7
Error: dyop with different data types at line 27
>
```

Line 26, (`fromWorkers[tmp2] ?  y`) should get a value from channel from `Workers[tmp2]`, which has been declared as a `CHAN OF INT`. As y is a `REAL32` value, an error is flagged.

More complicated semantic errors

As a demonstration of the ability of the typechecker to catch subtle semantic errors, consider the following program:

```
DATA TYPE currency IS INT:
currency myBalance:
DATA TYPE currency IS INT:
currency hisBalance:
SEQ
  hisBalance := CONVERT currency 2400
  myBalance := hisBalance
```

The error here is that myBalance is of type 'currency' (declared at line 1), wheras hisBalance is of type 'currency' (declared at line 3)! These data types have the same name, but are different! (recall Section 2.2.5)

```
> tcheck
There were errors:
Not the same dataType: currency_0 currency_2
Error: dataTypeLists contain different types at line 2
Error: Assignment LHS dataTypeList doesnt match RHS at line 7
>
```

4.3 Size of Output files

As stated in Section 2.4, one of the advantages of using Java is that relatively small output files can result.

Consider the mandelbrot example (Appendix A.2). The java class file sizes produced for this by javac (of the Java source output), the Bytecode output, and KrOC can be compared below. This is not strictly a like for like comparison, but an impressive size difference can nonetheless be observed, which is always valuable, especially when considering the recent increase in the use of portable hand-held computers which have again required programmers to consider available memory.

Compiler	Size of output (in bytes)
KrOC	77381
JOC (Java Source Backend)	3294 (when compiled by javac)
JOC (ByteCode Backend)	2728

The 17% reduction in size that can be obtained when using JOC's Bytecode backend is explained in Section 2.5.1 - admittedly this is at the cost of removing unused symbolic information from the Class file.

Even if this advantange is ignored, there is obviously a further benifit from being able to target bytecode directly - the compilation is substantially faster, and also there is no need for Java support on the system on which the compiler runs.

4.4 Execution Speed

Code generated by JOC is substantially (orders of magnitude) slower than that generated by KrOC. Consider the Mandelbrot program again.

Language	Execution time for 30*20 Mandelbrot
C	0.02 s
Occam (KrOC)	0.02 s
Java	0.74 s
Java produced by jOcc	7.14 s

It is perhaps surprising to note the difference in execution time between the Java and jOcc Java results. This can be attributed to the fact that all variable manipulations in jOcc Java require the use of a jObj Object (Section 3.7.5) instead of primitive Java types.

Chapter 5

Conclusion

The project met all of the goals set; it was a success. I feel that the time spent in the planning stage and experimenting with KrOC payed dividends - a hallmark of a good Software Engineering project is that there was no need to redesign any of the compiler code, or to 'retrofit' features which should have been thought of at the planning stages. The modular nature of the project meant that it was possible to create 'clean', distinct modules, which increased the readability of the compiler and allowed what was a fairly complex piece of code to be constructed without losing sight of its workings.

As demonstrated in the evaluation, the performance of the generated code is substantially below that of code generated by industrial strength Occam compilers. A portion of this can be attributed to the choice of Java as a target language, the majority due to encapsulating all variables in Objects. Nevertheless, the compiler works well, and implements a great deal of the Occam language specification. Given more time, it would be an interesting excercise to add a further backend to compile to threaded C, for greater performance. It would alse provide performance benefits if jObj objects could be used as little as possible.

In total the compiler contains approx 12,000 lines of C as well as associated yacc and lex specifications, a fairly substantial project given the time constraint of 6 months.

The use of pre-existing tools such as lex and yacc saved a substantial amount of time during the implementation, gdb (a C debugger) also proved to be an invaluable asset.

Occam is a language of which I had not heard prior to choosing a project; I found it to be an instructive departure from the current methods of looking at parallelism in programming and also enjoyed being introduced to CSP. Occams basis in CSP is fascinating and has proved an interesting background study.

In summary, the original goals of the project were met within the timescale. The compiler works as specified (as demonstrated by Appendix A), and the output will permit execution on any system capable of running a Java Virtual machine.

Bibliography

[1] Occam 2.1 Reference Manual, SGS-Thompson Microelectronics LTD. http://www.hensa.ac.uk/parallel/occam/documentation/occ21refman.ps.gz

[2] http://unix.hensa.ac.uk/parallel/occam/compilers/spoc/index.html

[3] http://www.hensa.ac.uk/parallel/occam/projects/occam-for-all/index.html

[4] Internet Parallel Computing Archive, http://wotug.ukc.ac.uk/parallel

[5] Tim Lindholm & Frank Yellin, The Java Virtual Machine Specification, Addison Wesley. (1997)

[6] D. Mitchell, Inside the Transputer, Blackwell Scientific Publications. (1990)

[7] Inmos LTD, Transputer Reference Manual, Prentice Hall.

[8] http://www.cs.bris.ac.uk/~alan/javapp.html

[9] C.A.R. Hoare, Communicating Sequential Processes (Prentice Hall) (1985)

Appendix A

Example Occam Programs

A.1 BubbleSort

This program presents a peculiar approach to bubblesort - to sort n numbers, n PARallel components are used, with a channel between each successive component.

Each component holds one number. When it is passed a new number, it emits the larger of the two to the next component.

-1 is used to terminate the stream.

A.1.1 Source Code

```
VAL INT flush IS -1:
VAL INT N IS 10:
[11] CHAN OF INT CX:
PAR
  SEQ
    CX[0] ! 12
    CX[0] ! 7
    CX[0] ! 2
    CX[0] ! 123
    CX[0] ! 14
    CX[0] ! 11
    CX[0] ! 21
    CX[0] ! 31
    CX[0] ! 64
    CX[0] ! 45
    CX[0] ! -1
  PAR i=0 FOR N
    INT next,current:
    SEQ -- start of code for ith cell
      CX[i] ? current
```

```
      WHILE current <> flush -- terminate with -1
        SEQ
          CX[i] ? next
          IF
            current > next -- output largest element
              SEQ
                CX[i+1] ! current
                current := next
            TRUE
                CX[i+1] ! next
      CX[i+1] ! flush -- pass on termination element
  SEQ
    INT vl:
    SEQ i=0 FOR N
      SEQ CX[10] ? vl
      screen ! vl
      screen ! '*n'
    INT pointless:
    CX[10] ? pointless
```

A.1.2 Example Output

```
[lab27@lab27 generate1]$ ./jvc bubblesort.occ
Typechecking...
JV_CONVERT start.
Examining main process.
Examining Functions, Procedures, Dispatcher, Main.
Writing class file `‘¨bubblesort.class`‘¨
JV_CONVERT end.
[lab27@lab27 generate1]$ java bubblesort
123
64
45
31
21
14
12
11
7
2
```

A.2 Mandelbrot Generator

Massively parallel systems lend themselves to programs where one part of the calculation is utterly independent of another. A classic example of this is the mandelbrot set.

In the program given below, each pixel of the mandelbrot is calculated in a seperate thread, though this could easily be altered to be 1 row/thread by changing one PAR to a SEQ.

A.2.1 Source Code

```
REAL64, REAL64 FUNCTION complexsquare (VAL REAL64 a, b)
  REAL64 aout, bout:
  VALOF
    SEQ
      REAL64 FUNCTION complexsquare[1] (VAL REAL64 a)
        VALOF
          SKIP
          RESULT (a*a)
      :
      aout,bout := ((complexsquare(a)) - (b*b)), (2.0 * (a * b))
    RESULT aout, bout
:

BYTE FUNCTION mandelpoint (VAL REAL64 a,b)
  REAL64 inita, initb:
  REAL64 x,y:
  INT cycles:
  BYTE retval:
  VALOF
    SEQ
      x,y,inita,initb,cycles := a,b,a,b,100
      WHILE ( (((x*x) + (y*y)) < 100.0 ) AND (cycles > 0))
        SEQ
          x,y := complexsquare (x,y)
          x,y,cycles := x+inita , y+initb, cycles-1
      IF
        ( cycles = 0 )
          retval := '**'
        TRUE
          retval := ' '
    RESULT retval
:

[40][40]BYTE mdata:
SEQ
  PAR x = 0 FOR 40
```

[1]This clumsy multiplication is unneccessary,but serves to demonstrate that the compiler can cope with awkward nesting of functions of the same name...

```
    PAR y = 0 FOR 40
      mdata[x][y] := mandelpoint ( (ROUND REAL64 (x-32)) / 20.0,(ROUND REAL64 (y-2
  SEQ y = 0 FOR 40
    SEQ
      SEQ x = 0 FOR 40
        screen ! mdata[x][y]
      screen ! '*c'
      screen ! '*n'
```

A.2.2 Example Output

```
[lab27@lab27 generate1]$ java mandel
                              *
                            ****
                            ****
                            ****
                           **  *
                      *  **********
                     ******************
                      *****************
                   * *****************
                    *******************
                  **********************
                   *********************
         * ***    **********************
         *******  **********************
        ********* **********************
        ********* **********************
     ** ********* *********************
**************************************
     ** ********* *********************
        ********* **********************
        ********* **********************
         *******  **********************
         * ***    **********************
                   *********************
                  **********************
                    *******************
                   * *****************
                      *****************
                     ******************
                      *  **********
                          **   *
                            ****
                            ****
                            ****
```

*

A.3 Farmer & Workers

This program demonstrates how one process can act as a "farmer" and give out work to other processes, collating the results. Two arrays of channels are used - one to pass information to the workers, one to receive the results.

This small example creates ten workers, each of which is given a number, which it squares, then returns. The farmer adds the results. The net effect is to sum the squares from 1 to 10.

Note that by doing this, we adher to the rules of CSP, as if each of the workers independently added their result to tot, there would be a potential resource conflict.

A.3.1 Source Code

```
INT FUNCTION square(VAL INT x)
  VALOF
    SKIP
    RESULT x*x
:

INT tot:
[11]CHAN OF INT toWorkers, fromWorkers:
SEQ
  tot := 0
  PAR
    INT x:
    PAR y = 0 FOR 10              -- kick off ten workers.
      SEQ                         -- this is one worker.
        toWorkers[y] ? x
        x := square(x)
        fromWorkers[y] ! x
    SEQ                           -- Farmer process.
      SEQ tmp = 0 FOR 10          -- For each of 10 workers
        toWorkers[tmp] ! (tmp+1)  -- Give them their value
      INT y:
      SEQ tmp = 0 FOR 10
        ALT tmp2 = 0 FOR 10
          fromWorkers[tmp2] ? y
            tot := tot + y
      screen ! tot
```

A.3.2 Example Output

As expected, the program returns 385!

```
[lab27@lab27 generate1]$ java farmer_giles
385
```

Appendix B

Generated Java Example

From the farmer/worker example given in Appendix A.3, the portion to generate ten workers translates to:

```
public void par_thread_3 () {
  Vector assignVector;
  Channel_of_Object screen_8 = (Channel_of_Object) display.elementAt(0);
  Channel_of_Object[] fromWorkers_7 = (Channel_of_Object[]) display.elementAt(1);
  Channel_of_Object[] toWorkers_6 = (Channel_of_Object[]) display.elementAt(2);
  jObj tot_5 = (jObj) display.elementAt(3);
  jObj x_4 = (jObj) display.elementAt(4);
  jObj x_9 = new jObj(0);
  example3_Cls[] threada_0 = new example3_Cls[(new jObj(10)).intVal()];
  for (int parIterator = ((new jObj(0))).intVal();
                       parIterator < ((new jObj(0)).add((new jObj(10)))).intVal();
    jObj y_10 = new jObj(parIterator);
    Vector display_1 = new Vector(); display_1.addElement(y_10);
    display_1.addElement(x_9); display_1.addElement(fromWorkers_7);
    display_1.addElement(toWorkers_6);
    threada_0[parIterator-((new jObj(0))).intVal()] = new example3_Cls(4,display_1);
  }
  try {
    for (int parIteratorWt = 0;parIteratorWt < ((new jObj(10))).intVal();
         parIteratorWt++) {
      threada_0[parIteratorWt].join();
    }
  } catch (Exception e) { }
}

public void par_thread_4 () {
  Vector assignVector;
  jObj y_10 = (jObj) display.elementAt(0);
```

```
    jObj x_9 = (jObj) display.elementAt(1);
    Channel_of_Object[] fromWorkers_7 = (Channel_of_Object[]) display.elementAt(3);
    Channel_of_Object[] toWorkers_6 = (Channel_of_Object[]) display.elementAt(4);
    jObj tot_5 = (jObj) display.elementAt(5);
    x_9.set( (jObj) toWorkers_6[y_10.intVal()].get());
    x_9.set(((jObj) (square_1(x_9).elementAt(0))));
    fromWorkers_7[(new jObj(0)).intVal()].put(x_9);
  }
```

Appendix C

Generated Java Bytecode

The bytecode given here is a direct output of this compiler. The tool **cfr** (see section 2.5.6) was used to display it.

This snippet of bytecode corresponds to the java source code "par_thread_4" in the above example.

```
par_thread_4 ()V     Access_flags = 0002 (Private )
max_stack    : 20
max_locals   : 20
0       aload_0
1       getfield        display Ljava/util/Vector;
4       iconst_0
5       invokevirtual   java/util/Vector elementAt (I)Ljava/lang/Object;
8       checkcast       jObj
11      astore_2
12      aload_0
13      getfield        display Ljava/util/Vector;
16      iconst_1
17      invokevirtual   java/util/Vector elementAt (I)Ljava/lang/Object;
20      checkcast       jObj
23      astore_3
24      aload_0
25      getfield        display Ljava/util/Vector;
28      iconst_2
29      invokevirtual   java/util/Vector elementAt (I)Ljava/lang/Object;
32      checkcast       Channel_of_Object
35      astore          4
37      aload_0
38      getfield        display Ljava/util/Vector;
41      iconst_3
```

```
42      invokevirtual   java/util/Vector elementAt (I)Ljava/lang/Object;
45      checkcast       [LChannel_of_Object;
48      astore          5
50      aload_0
51      getfield        display Ljava/util/Vector;
54      iconst_4
55      invokevirtual   java/util/Vector elementAt (I)Ljava/lang/Object;
58      checkcast       [LChannel_of_Object;
61      astore          6
63      aload_0
64      getfield        display Ljava/util/Vector;
67      iconst_5
68      invokevirtual   java/util/Vector elementAt (I)Ljava/lang/Object;
71      checkcast       jObj
74      astore          7
76      aload_3
77      aload           6
79      aload_2
80      invokevirtual   jObj intVal ()I
83      aaload
84      invokevirtual   Channel_of_Object get ()Ljava/lang/Object;
87      checkcast       jObj
90      invokevirtual   jObj set (LjObj;)V
93      aload_3
94      aload_0
95      aload_3
96      invokevirtual   farmer_giles square_1 (LjObj;)Ljava/util/Vector;
99      iconst_0
100     invokevirtual   java/util/Vector elementAt (I)Ljava/lang/Object;
103     invokevirtual   jObj set (LjObj;)V
106     aload           5
108     new             jObj
111     dup
112     iconst_0
113     invokespecial   jObj <init> (I)V
116     invokevirtual   jObj intVal ()I
119     aaload
120     aload_3
121     invokevirtual   Channel_of_Object put (Ljava/lang/Object;)V
124     return
```

Appendix D

AST Node Structure Example

```
/* A_spec should be used when dynamically creating specification
   nodes */
typedef struct A_spec_ * A_spec;
/* For when lists of specifications are needed, such as a
   custom record datatype */
typedef struct A_specList_ * A_specList;
struct A_specList_ {A_spec head; A_specList tail;};

/* By setting the 'kind' flag, it is possible for a case statement
   to choose how to interpret the data in the union */
struct A_spec_ {
  A_pos pos;
  enum {
    A_declaration,
    A_valAbbreviation,
    A_varAbbreviation,
    A_procedureSpec,
    A_functionSpec,
    A_valueProcSpec,
    A_fparmSpec,
    A_tagProtoSpec,
    A_customTypeSpec,
    A_customRecordTypeSpec,
  } kind;
  union {
    struct {A_dataType type;A_expList names; } declaration;
    struct {A_dataType type;A_exp name;A_exp exp1; } valAbbreviation;
    struct {A_dataType type;A_exp name;A_exp exp1; } varAbbreviation;
    struct {A_exp name;A_specList sl1;A_process proc; } procedureSpec;
    struct {A_dataTypeList dtl1;A_exp name;A_specList sl1;A_process proc; } functionSpec;
```

```
    struct {A_process proc; A_expList el1; } valueProcSpec;
    struct {A_dataType dt1; A_exp name; } fparmSpec;
    struct {A_exp tag; A_dataTypeList dtl1; } tagProtoSpec;
    struct {A_exp name; A_dataType dt1; } customTypeSpec;
    struct {A_exp name; A_specList sl1; } customRecordTypeSpec;
  } u;
};
```

Appendix E

Implementation of Channel Object

```
public class Channel_of_Object {
   // Value waiting to be got from channel
   private Object      channel_hold;
   // is the channel empty?
   private boolean     channel_empty = true;
   // if an Alternative is
   private Alternative alt;

   public synchronized Object get() {
      try {
         if (channel_empty) {
            channel_empty = false;
            wait ();
            notify ();
         } else {
            channel_empty = true;
            notify ();
         }
         return channel_hold;
      } catch (Exception e) {
         System.out.println(`‘¨Error\n`‘¨+e);
      }
      return null;
   }

   public synchronized void put (Object n) {
      try {
         // Use tmp_alt to avoid race conditions
```

```
         Alternative tmp_alt = alt;
         channel_hold = n;
         if (tmp_alt != null && tmp_alt.state == 3) {
            // Poss A, Alt is waiting.  Schedule Alt.
            channel_empty = false;
            tmp_alt.schedule();
            wait();
         } else if (channel_empty) {
            // Poss B. No value in channel.  Put one in, and wait.
            channel_empty = false;
            wait();
         } else {
            // Poss C. get() is blocking somewhere.
            channel_empty = true;
            notify ();
            wait ();
         }
      } catch (Exception e) {
         System.out.println(`‘¨Channel Error\n`‘¨+e);
      }
   }

   // If an Alternative wants to claim this object, it has to call
   // this.  Only succeeds if false returned.
   public synchronized boolean registerAltInterest(Alternative alt) {
      if (channel_empty) {
         this.alt = alt;
         return false;
      } else {
         return true;
      }

   }

   // Alternatives no longer wish to claim this object.
   public synchronized boolean unregisterAltInterest() {
      alt = null;
      return (!channel_empty);
   }
}
```

Appendix F

Project Proposal

F.1 Introduction

F.1.1 Occam

Occam is a parallel computing language based on the CSP (Communicating Sequential Processes) programming model by C. A. R. Hoare. It was designed to express concurrent algorithms, and their implementation on a network of processing components.

Occam has the particularly interesting feature of having parallelism built into the language at its core.

An application (in Occam) is described as a collection of processes - these processes execute serially or concurrently and communicate with each other through channels. Each process gives the behaviour of a particular part of the application, and each channel describes a communication between two processes. This has the effect of giving Occam programs clearly defined structure, as well as allowing an Occam program to exploit a system made of many parts.

> ... The development of the INMOS Transputer, a family of devices which place one or more microprocessors on a single chip, has been closely related to Occam, its design and implementation... [1]

As stated, each statement in Occam is a process, and these processes can be performed in Sequence or Parallel through the use of the SEQ and PAR constructs. Eg:

```
SEQ
  keyboard ? message
  user[3] ! message
  user[4] ! message
  user[5] ! message
```

will perform four sequential processes,whereas:

```
PAR
  user[3] ! message
  user[4] ! message
  user[5] ! message
```

will perform three outputs in parallel. (The example given has no guarantee of order of completion or priority).

Occam provides for interprocess communication by the use of 'Channels', which are unidirectional synchronous paths between two processes; they are global objects which can be seen by all processes. By restricting IPC to this 'channel' paradigm, Occam removes the problems associated with the implementation of Semaphores or Monitors to protect shared variables - variables are never shared!

A rendezvous mechanism is used to perform message passing.

The above SEQ example also demonstrate the use of message passing with the ? and ! operators. 'keyboard ? message' requests a value from the 'keyboard' channel - this is then passed to the three channels 'user{3.4.5}'.

F.2 Project Description

It would be interesting to compile Occam to Java, which provides concurrency constructs - hence allowing use of Occam on a wide range of target systems, most of which will be single processor machines.

This would be implemented with two distinct targets - initially the target language would be Java source code which can be compiled using 'Javac'. This aim will allow the compiler to output human readable code as a proof of concept prior to moving on to the second, more ambitious aim of re-targeting the compiler to produce Java Byte code directly, permitting the compiler output to be executed as-is by a Java Virtual Machine (JVM).

F.2.1 Proposed Omissions

The project will not, (unless they are implemented as extensions) support several non-key features of the Occam 2.1 specification, such as:

- Placement of Variables and Channels.
- Variable sized INTs.
- Ported output. (The implementation will include only minimal I/O facilities, such as text from keyboard and to screen).

F.2.2 Work Planned

- Learn Occam, using the Southampton Portable Occam Compiler (SPOC) on Thor.

- Implement trial of rendezvous message passing using Java.
- Decide overall structure of Java implementation, plan Class definitions etc.
- Create Lexer for Occam 2.1 specification using Lex. (C version of Lex).
- Create Recursive Descent Parser for Occam 2.1 specification in C.[1]
- Create module to perform semantic analysis on abstract syntax tree from parser; produce an intermediate representation of the code.
- Intermediate representation translation to Java Source
- Intermediate representation translation to Java Byte Code.

F.2.3 Extensions

Possible extensions include further implementation of Occam implementation specific features, such as

- Compiler Directives (#pragma, etc)
- Priority Execution of processes.
- Assertions.
- Rescheduling.

F.2.4 Testing Strategy

Testing will be an ongoing process throughout this project. As far as possible, units will be tested on their own prior to integration with those already completed. The project also lends itself to an incremental testing strategy, as the parser will require the output of the lexer, etc. Modules can therefore be tested alongside those already working to ensure correct behaviour.

F.3 Resources

The bulk of all programming and dissertation work will be done on my local Machine, (AMD K6/2, 128 Mb RAM, 24Gb HDD, running Linux), and saved as a CVS[2] repository in order to ensure that the current version may be rolled-back, should any problem occur or any changes need to be undone. The CVS repository will be mirrored on Pelican.

[1] Because Occam uses lexical indentation of constructs from the left hand edge of text to delimit scope, it is not suited to parsing using Yacc. Initial investigation will focus on creating a recursive descent parser, (as Occam is a relatively simple language, this is not an infeasible task.)

[2] Concurrent Versions System.

A weekly backup of all code and dissertation work will saved on Pelican. This will probably be automated using a Cron[3] job.

No special resources will be required.

F.4 Starting Point

- 1B Java Programming course.
- 1B Compilers course.
- Prior knowledge of C, though no previous experience with Lex.
- No current experience with Occam.

F.5 Timetable and Milestones

22/10	Proposal Deadline
29/10	Become familiar with Occam through use of another implementation, SPOC[2] or KrOC[3]. Collect / create several sample programs which can be tested on the compiler.
05/11	Write lexer specification for Occam 2.1 (using lex)
19/11	Write parser, initially just checking syntax.
03/12	Have parser producing abstract syntax tree for Occam 2.1, design intermediate code.
Christmas	Consider translation from intermediate code to Java source. Study JVM specification. Further testing for modules already completed.
21/01	Translate from intermediate code to Java source.
04/02	Midterm report deadline.
18/02	Translate from intermediate code to JVM byte code.
10/03	Final system testing, consider writing extensions.
Easter	Begin writing dissertation. (Using LyX)
14/04	Finish dissertation first draft.
19/05	Dissertation deadline.

[3]Cron is a daemon that is used to execute scheduled commands.

Bibliography

[1] Occam 2.1 reference manual SGS-THOMPSON Microelectronics Limited

[2] http://unix.hensa.ac.uk/parallel/occam/compilers/spoc/index.html

[3] http://www.hensa.ac.uk/parallel/occam/projects/occam-for-all/index.html