µOCCAM

⟨Copyright message 2a⟩ ≡

```
/* Copyright K. Mitchell 1995.
   This file has been generated by a program and is not intended to be read
   by a human.  Run the file occam.w through nuweb.  This produces a file
   called occam.tex.  Typeset this file using LaTeX and then preview the
   resulting .dvi file, or just print it out.
*/
```

Macro referenced in scraps 2b, 10b, 23a, 77e.

⟨Standard μOCCAM includes 2b⟩ ≡

```
⟨Copyright message 2a⟩
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

typedef int bool;

#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
#define TRUE 1
#endif
```

Macro referenced in scraps 10e, 23d, 29a, 78c, 86b.

Here is the makefile to build the system. The dependencies are specified in each web file to help keep them consistent.

"code/Makefile" 3 ≡

```
.SUFFIXES:
.SUFFIXES: .c .o .h

CFLAGS= -O
CC= gcc

OBJECTS= ⟨Occam object files 10d, ... ⟩

occam:  $(OBJECTS)
        $(LINK.c) -o occam $(OBJECTS)

⟨Makefile dependencies 10c, ... ⟩

clean:
        -rm *.o *~ occam
```

Chapter 1

The Grammar for μOCCAM

μOCCAM is essentially a subset of the OCCAM programming language, as described in the occam 2.1 reference manual, SGS-Thomson 1995. This is available electronically from the course web pages, and copies are distributed at the introductory lecture. For further copies, ask at the Informatics Teaching Office, JCMB room 1502. There are also books on occam in the JCM library and in the machine halls.

We assume that the reader has already glanced at a book on OCCAM, particularly at the examples, and is familiar with the general approach followed by this language. The simplifications in μOCCAM are primarily in the area of types. The only types supported are integers, channels, and one-dimensional arrays of these. Furthermore, channels can only carry integers. We treat booleans in the same way as C, i.e. as integers with 0 denoting **false**, and any other value denoting **true**. In some cases we have altered the syntax of the language where this aids syntactic or semantic analysis.

The language μOCCAM is described by the following grammar, where we have used the same indentation and repetition conventions as in Appendix H of the reference manual.

```
process       =     STOP  |  SKIP  |  action  |  construction  |  instance
                 |  specification
                    process

action        =     assignment  |  input  |  output

assignment    =     variable := expression

input         =     channel ? variable
output        =     channel ! expression

construction  =     sequence  |  conditional  |  loop  |  parallel  |  alternation

sequence      =     SEQ
                      { process }
                 |  SEQ replicator
                      process

conditional   =     IF
                      { choice }
                 |  IF replicator
                      choice
choice        =     boolean
                      process
boolean       =     expression
```

```
loop            =     WHILE boolean
                        process

parallel        =     PAR
                       { process }
                |     PAR replicator
                        process

alternation     =     ALT
                       { alternative }
                |     ALT replicator
                        alternative

alternative     =     guard
                        process
guard           =     input  |  boolean & input

replicator      =     name = base FOR count
base            =     expression
count           =     expression

type            =     primitive.type
                |     [ expression ] primitive.type

primitive.type  =     INT  |  CHAN

literal         =     integer

element         =     name  |  name [ subscript ]

subscript       =     expression
variable        =     element
channel         =     element

operand         =     variable  |  literal  |  ( expression )

expression      =     monadic.operator operand
                |     operand dyadic.operator operand
                |     operand

specification   =     declaration  |  definition

declaration     =     type name :
                |     INT name = expression :
                |     VAL name IS expression :

definition      =     PROC name ( {0, formal } )
                        procedure.body
                      :

formal          =     primitive.type name
procedure.body  =     process

instance        =     name ( {0, actual } )
```

actual	=	element
monadic.operator	=	`-` | `NOT`
dyadic.operator	=	`+` | `-` | `*` | `/` | `\` | `=` | `<` | `>` | `<=` | `>=` | `<>` | `AND` | `OR`

Notes:

The following items clarify certain points about the language, and highlight differences from the language as given in the occam 2.1 reference manual.

- Integer literals consist of one or more digits, and a name consists of a sequence of alphanumeric characters and dots starting with an alphabetic character (reference manual page 105).
- Indentation carries significant meaning in μOCCAM. See pages 3 and 4 of the reference manual.
- TAB characters. Any line may begin with some TAB characters, each counting for four levels of indentation, followed by some spaces, two spaces per indentation level. Once you are on to spaces, no more TAB characters are allowed. TAB characters are not legal in the middle of a line.
- Comments. There may be comments anywhere within a program except within a continuation line.
- Blank lines. There may be blank lines anywhere within a program except within a continuation line.
- End of file. The last statement of a program need not have a terminating newline character. That is, the end of a file may appear either at the end of the last program line, or on a line of its own. This is necessary because not all editors allow the user easy control of this.
- Procedure parameters are passed by *value*, just as in C. This is different from the reference manual.
- Invalid processes should be treated like `STOP` (reference manual page 101).
- The count in a replicated `ALT`, the expression in a `VAL` declaration, and the array bounds in an array declaration must all be constant expressions, *i.e.* expressions whose value can be calculated at compile time. The function `constant_expression` in Section 5.8.2 can help to check this.
- The base and count of a replicated `PAR` do not have to be constant expressions as we are not running on a real distributed system.
- Two channels, `stdin` and `stdout`, are predefined and connected to the standard input and output at the start of execution.

When you find an illegal program (e.g. with lexical, syntactic or semantic errors) you should call the error function (see Section 6.3). This will stop the program and set the return code appropriately. You do not need to spend time adding error recovery mechanisms to the parser. Obviously this is something you would like in a real compiler, but there is insufficient time in this practical for such niceties. When completed, the compiler should check for type errors such as integers used as channels, procedures called with the wrong number of arguments *etc.* However, you do not need to check that variables are only updated by one process, or that only one process inputs or outputs on a channel. As the language has arrays, such constraints can only be partially checked at compile time anyway. However, we will assume that all programs passed to the compiler satisfy these constraints.

Hopefully the grammar, in addition to the `OCCAM` manual, will answer most of your questions about μOCCAM. When in doubt, ask the project organiser rather than just guessing.

Chapter 2

The Symbol Table

2.1 Pools

It's not always clear what a C compiler does when a program calls `free`. A compiler may create lots of temporary storage and we would like a reasonably efficient way of reclaiming it. If most of the temporary objects have the same size then we can just maintain a *free list* to hold the disposed objects. An extension of this idea, useful when the objects cover a range of sizes, is to use an array of free lists. When we want to create storage for an object we index into the free-list array using the size of the object as index. Clearly this involves placing a bound on the size of object we can allocate using this method. The bound is defined by the constant `MAX_POOL_SIZE`. For larger objects we just call `malloc` directly. The intention is that we will use pools to allocate structures, where the size is known at compile time. Other dynamically allocated objects, such as strings, will use `malloc`. One of the big advantages of using pools, rather than a direct call to `malloc`, is that we can detect storage leaks more readily.

⟨Pool definitions 7a⟩ ≡

```
#define MAX_POOL_SIZE 100
```

Macro defined by scraps 7ab, 8ad.
Macro referenced in scrap 10b.

Each entry in a free-list is chained to the next one. We use the type `link` for such chains.

⟨Pool definitions 7b⟩ ≡

```
typedef struct link { struct link* next; } *link;
```

Macro defined by scraps 7ab, 8ad.
Macro referenced in scrap 10b.

The roots of the free lists are held in the array `pool`.

⟨Pool externs 7c⟩ ≡

```
extern link pool[];
```

Macro defined by scraps 7c, 8b, 9ad.
Macro referenced in scrap 10b.

⟨Pool data 7d⟩ ≡

```
link pool[MAX_POOL_SIZE+1];
```

Macro defined by scraps 7d, 8c, 9c.
Macro referenced in scrap 10e.

All of the entries in the free-list attached to `pool[`i`]` are of size i. We attach the type `link` to them to allow easy chaining, but their actual size will usually be bigger than this. Of course this raises the question of what happens when the actual size is smaller than the size of a `link` structure. We take steps to ensure this cannot happen.

To create some storage we use the `new` macro. This takes the type of the object to create as argument and returns a pointer to some new storage. It does this by checking the appropriate pool to see if it is non-empty. If it is it just has to unlink the first entry in the list and return this. If it is empty then it calls the function `extend_pool` to create some more storage. The macro is a bit messy because we have to use a temporary variable to hold the head of the list while we unlink it. We can't declare temporaries local to an expression, and so we use the `pool_temp` global for this purpose.

⟨Pool definitions 8a⟩ ≡

```
#define new(type_name) \
  ((pool[sizeof(type_name)]) \
   ? (pool_temp = pool[sizeof(type_name)], \
      pool[sizeof(type_name)] = pool[sizeof(type_name)]->next, \
      (type_name *) pool_temp) \
   :  (type_name *) extend_pool(sizeof(type_name)))
```

Macro defined by scraps 7ab, 8ad.
Macro referenced in scrap 10b.

⟨Pool externs 8b⟩ ≡

```
extern link pool_temp;
```

Macro defined by scraps 7c, 8b, 9ad.
Macro referenced in scrap 10b.

⟨Pool data 8c⟩ ≡

```
link pool_temp;
```

Macro defined by scraps 7d, 8c, 9c.
Macro referenced in scrap 10e.

To free the storage associated with an object pointer we use the macro `dispose`. This just chains the object to the appropriate free list.

⟨Pool definitions 8d⟩ ≡

```
#define dispose(object) \
  { ((link)object)->next = pool[sizeof(*object)]; \
    pool[sizeof(*object)] = (link)object; }
```

Macro defined by scraps 7ab, 8ad.
Macro referenced in scrap 10b.

Suppose we want some storage and the appropriate free list is empty. We could just call `malloc` but this would be inefficient. It is better to allocate storage for a whole bunch of these objects in one go, on the assumption that more will be needed later. But how many to allocate? The choice could depend on the size of the object, but for simplicity we just use the constant `NELEM` for all sizes.

⟨Pool code 8e⟩ ≡

```
#define NELEM 100
```

Macro defined by scraps 8e, 9be.
Macro referenced in scrap 10e.

We allocate storage for `NELEM` objects of the required size and link them together before returning the first one to the caller. We have to be a bit careful if the size of the object is smaller than a `link` object (i.e. less than four bytes). This is unlikely to occur very frequently, and in the rare cases where it does we just allocate storage as if the object was four bytes long. This wastes space, but the objects couldn't be chained together otherwise.

⟨Pool externs 9a⟩ ≡
```
extern void *extend_pool(int n);
```
Macro defined by scraps 7c, 8b, 9ad.
Macro referenced in scrap 10b.

⟨Pool code 9b⟩ ≡
```

void *extend_pool(int n)
{ int esize; char *start, *last, *p;
  esize = (n < 4) ? 4 : n;
  if ((pool[n] = malloc(NELEM * esize)) == 0) {
    ⟨Report memory exhausted and exit 10a⟩ }
  allocated_pool[n] += NELEM * esize;
  start = (char *)pool[n];
  last  = &start[(NELEM-1)*esize];
  for (p = start; p<last; p+=esize) ((link)p)->next = (link)(p+esize);
  ((link)last)->next = NULL;
  pool[n] = pool[n]->next;
  return(start);
}
```
Macro defined by scraps 8e, 9be.
Macro referenced in scrap 10e.

It's sometimes useful to get a feel for how much storage is being allocated and recycled. We keep track of the amount of pool data allocated using the array `allocated_pool`.

⟨Pool data 9c⟩ ≡
```
int allocated_pool[MAX_POOL_SIZE+1];
```
Macro defined by scraps 7d, 8c, 9c.
Macro referenced in scrap 10e.

The function `pool_statistics` prints out a summary of the total amount of storage allocated, and the amount currently in use, for each pool.

⟨Pool externs 9d⟩ ≡
```
extern void pool_statistics();
```
Macro defined by scraps 7c, 8b, 9ad.
Macro referenced in scrap 10b.

⟨Pool code 9e⟩ ≡
```

void pool_statistics()
{
  int i, on_list, in_use; link p;
  printf("Pool statistics: in use / total (%%)\n");
  for (i=1; i <= MAX_POOL_SIZE; i++) {
    if (allocated_pool[i] == 0) continue; /* Skip unused entries */
    p = pool[i]; on_list = 0;
    while (p) { on_list += i; p = p->next; }
    in_use = allocated_pool[i] - on_list;
    printf("%3d: %5d /%5d (%2d%%)\n", i, in_use, allocated_pool[i],
           100 * in_use / allocated_pool[i]);
  }
}
```
Macro defined by scraps 8e, 9be.
Macro referenced in scrap 10e.

If the program runs out of memory we just stop.

⟨Report memory exhausted and exit 10a⟩ ≡

```
error("Fatal error", "Memory exhausted\n");
```

Macro referenced in scraps 9b, 12c.

The implementation of pools is held in the files `pool.h` and `pool.c`.

`"code/pool.h"` 10b ≡

```
⟨Copyright message 2a⟩
#ifndef _POOL_H
#define _POOL_H
⟨Pool definitions 7a, ... ⟩
⟨Pool externs 7c, ... ⟩
#endif
```

⟨Makefile dependencies 10c⟩ ≡

```
pool.c: occam.h pool.h
```

Macro defined by scraps 10c, 23c, 25c, 28c, 78b, 85f.
Macro referenced in scrap 3.

⟨Occam object files 10d⟩ ≡

```
 pool.o
```

Macro defined by scraps 10d, 23b, 25d, 28d, 78a, 86a.
Macro referenced in scrap 3.

`"code/pool.c"` 10e ≡

```
⟨Standard μOCCAM includes 2b⟩
#include "occam.h"
#include "pool.h"
⟨Pool data 7d, ... ⟩
⟨Pool code 8e, ... ⟩
```

2.2 The Symbol Table

The same name may appear many times in a program. We use a hash table so that only one copy of the name needs to be saved. These strings are not accessed directly, but via structures that collect further information about the names. The `name_info` structures are linked together into (hopefully) short lists sharing the same hash key by means of the `hash_link` field.

⟨Symbol table typedef declarations 10f⟩ ≡

```
typedef struct name_info
{ char *name;
  ⟨Additional information about the name 13a⟩
  struct name_info *hash_link;
} name_info, *name, *names;
```

Macro defined by scraps 10f, 12d, 16c, 18a.
Macro referenced in scrap 23a.

The pointers to the heads of the hash lists are stored in the array `hash`. Identifiers can be found by computing a hash code `h` and then looking at the identifiers represented by the names `hash[h]`, `hash[h]->hash_link`, `hash[h]->hash_link->hash_link`, ..., until either finding the desired name or encountering the null pointer. The table is maintained by the function `find_name`, which finds a given identifier and returns the appropriate `name`. If there is no match for the identifier, it is inserted into the table. The `hash` table has size `hash_size`, which should be a prime number. Our hashing scheme is represented by Figure 2.2.

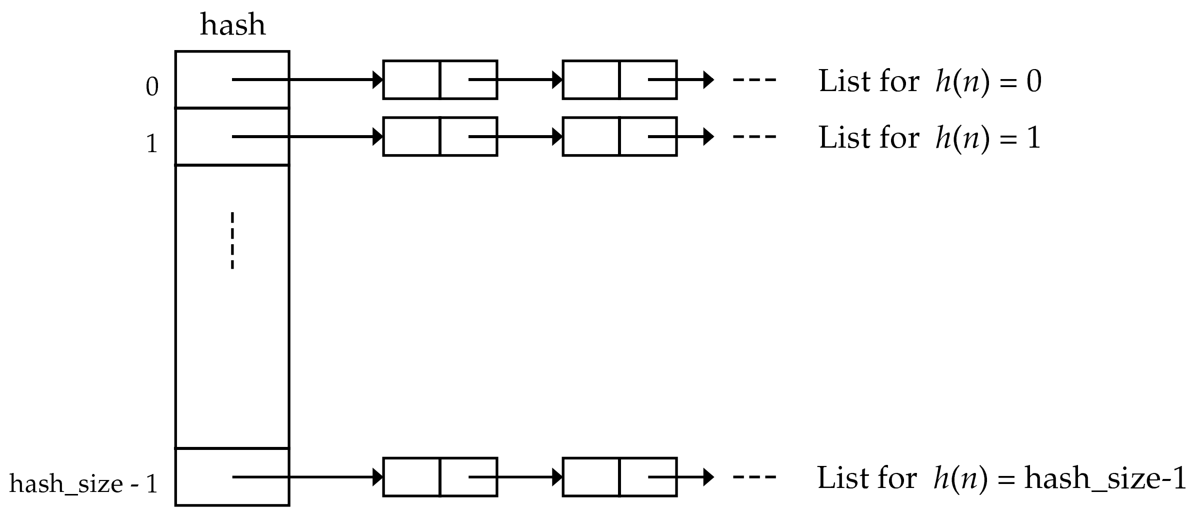

Figure 2.1: A hashing scheme

⟨Symbol table global variables 11a⟩ ≡

```
#define hash_size 353  /* should be prime */
names hash[hash_size]; /* heads of hash lists */
```

Macro defined by scraps 11a, 13g, 14c, 18bd, 21b.
Macro referenced in scrap 23d.

Initially all the hash lists are empty. Although strictly speaking the array `hash` should be initialised to null pointers by the compiler, we don't count on this.

⟨Initialise symbol table globals 11b⟩ ≡

```
{ int i=hash_size; do hash[--i]=NULL; while(i>0); }
```

Macro defined by scraps 11b, 13h, 14d, 18ce, 21c.
Macro referenced in scrap 23d.

Here is the main function for finding names in the hash table. The parameter `ident` points to a null-terminated string.

⟨Symbol table prototypes 11c⟩ ≡

```
extern name find_name (char *ident);
```

Macro defined by scraps 11c, 13e, 14bf, 15a, 16ae, 17ac, 20ac, 21ad, 22a.
Macro referenced in scrap 23a.

⟨Symbol table functions 11d⟩ ≡

```
name find_name (char *ident)
{ int h;
  ⟨Compute the hash code h of the string ident 12a⟩
  ⟨Find and return the associated name, possibly by entering a new identifier into the table 12b⟩
}
```

Macro defined by scraps 11d, 14g, 15b, 16bf, 17bd, 20bd, 21e, 22b.
Macro referenced in scrap 23d.

A simple hash code is used: If the sequence of character codes is $c_1c_2\ldots c_n$, its hash value will be

$$(2^{n-1}c_1 + 2^{n-2}c_2 + \cdots + c_n) \bmod \texttt{hash\_size}.$$

This isn't a particularly inspired choice of hash function, but it is adequate for the practical.

⟨Compute the hash code `h` of the string `ident` 12a⟩ ≡

```
{ char *p = ident;
  h=*p; while (*(++p) != 0) h=((h<<1)+*p)%hash_size;
}
```

Macro referenced in scrap 11d.

Here we either find an existing node for the identifier, or create one, inserting it into the name table and the hash list. The maximal average length of hash lists should be sufficiently small that linearly searching them is quite fast.

⟨Find and `return` the associated name, possibly by entering a new identifier into the table 12b⟩ ≡

```
{ name p=hash[h]; /* the head of the hash list */
  while (p!=NULL && strcmp(p->name,ident)) p=p->hash_link;
  if (p==NULL) /* we haven't seen this identifier before */
    ⟨Make p point to a fresh name_node and link the node to hash list h 12c⟩
  return p;
}
```

Macro referenced in scrap 11d.

Here is the code to create an entry for a new identifier. We just use `malloc` to create the storage for the name. If the allocation fails we stop the program.

⟨Make `p` point to a fresh `name_node` and link the node to hash list `h` 12c⟩ ≡

```
{ p = new(name_info);
  if ((p->name = malloc(strlen(ident)+1)) == 0)
    ⟨Report memory exhausted and exit 10a⟩

  p->hash_link = hash[h]; hash[h] = p; /* insert p at beginning of hash list */
  strcpy(p->name, ident);
  p->table = NULL;
}
```

Macro referenced in scrap 12b.

2.3 Handling Scopes

The purpose of a symbol table is not just to share the storage for identical names. It is also used to store information associated with each name, such as its size, type and location in the current scope. Attached to each name in the hash tabel is a symbol table entry. The intention is that the lexer converts a string of characters into a name and passes this to the parser. The parser then hangs the semantic information off of this name. The information we need to store for procedures is slightly different from other names, and so we use a union for these fields.

⟨Symbol table typedef declarations 12d⟩ ≡

```
typedef union symbol_table_info {
  struct {
    ⟨Symbol table shared fields 13d, ... ⟩
    ⟨Symbol table name fields 13b, ... ⟩
  } name;
  struct {
    ⟨Symbol table shared fields 13d, ... ⟩
    ⟨Symbol table proc fields 16d⟩
  } proc;
} symbol_table_info, *symbol_table_entry;
```

Macro defined by scraps 10f, 12d, 16c, 18a.
Macro referenced in scrap 23a.

⟨Additional information about the name 13a⟩ ≡

```
union symbol_table_info *table;
```

Macro referenced in scrap 10f.

The simplest thing we might record about a name is its size.

⟨Symbol table name fields 13b⟩ ≡

```
int size;
```

Macro defined by scraps 13b, 14a.
Macro referenced in scrap 12d.

Names in μOCCAM can denote integers and channels, and arrays of these. They can also be used as procedure names, and formal parameters to procedures. VAL declarations introduce constant integers. A channel will be represented by a pointer to a block of storage holding information about the channel, such as whether a process is waiting on the channel. These blocks will also be held on the stack, and hence in the symbol table. Procedures will use displays (see Section 5.2) to access non-local names and these will also be held on the stack. This discussion motivates the following definition.

⟨Symbol table type typedef 13c⟩ ≡

```
typedef enum name_type { INT_T,        CHAN_T      ,
                         INT_ARRAY_T,  CHAN_ARRAY_T,
                         PROC_T,       VAL_T,
                         CHAN_BLOCK_T, DISPLAY_T } type;
```

Macro referenced in scrap 23a.

⟨Symbol table shared fields 13d⟩ ≡

```
type type;
```

Macro defined by scraps 13df, 14e, 19b.
Macro referenced in scrap 12d.

The symbol table also keeps track of the location of a name. We associate a *nesting level* with each part of a program. The nesting level is initally one, and is incremented whenever we enter the scope of a procedure body and also when we enter the scope of a PAR.

⟨Symbol table prototypes 13e⟩ ≡

```
extern int nesting_level;
```

Macro defined by scraps 11c, 13e, 14bf, 15a, 16ae, 17ac, 20ac, 21ad, 22a.
Macro referenced in scrap 23a.

⟨Symbol table shared fields 13f⟩ ≡

```
int nesting_level;
```

Macro defined by scraps 13df, 14e, 19b.
Macro referenced in scrap 12d.

The current nesting level is held in a global variable of that name.

⟨Symbol table global variables 13g⟩ ≡

```
int nesting_level;
```

Macro defined by scraps 11a, 13g, 14c, 18bd, 21b.
Macro referenced in scrap 23d.

⟨Initialise symbol table globals 13h⟩ ≡

```
nesting_level = 1;
```

Macro defined by scraps 11b, 13h, 14d, 18ce, 21c.
Macro referenced in scrap 23d.

To access a name we need to know its offset in the stack (once we have used the nesting level to find the appropriate stack frame). The `offset` field holds this value. Offsets are calculated from the base of the stack frame, *not* relative to the current stack pointer.

⟨Symbol table name fields 14a⟩ ≡
```
int offset;
```
Macro defined by scraps 13b, 14a.
Macro referenced in scrap 12d.

We keep track of the current stack offset in the global variable `stack_offset`.

⟨Symbol table prototypes 14b⟩ ≡
```
extern int stack_offset;
```
Macro defined by scraps 11c, 13e, 14bf, 15a, 16ae, 17ac, 20ac, 21ad, 22a.
Macro referenced in scrap 23a.

⟨Symbol table global variables 14c⟩ ≡
```
int stack_offset;
```
Macro defined by scraps 11a, 13g, 14c, 18bd, 21b.
Macro referenced in scrap 23d.

⟨Initialise symbol table globals 14d⟩ ≡
```
stack_offset = 1;
```
Macro defined by scraps 11b, 13h, 14d, 18ce, 21c.
Macro referenced in scrap 23d.

What happens when we declare a name in a scope where the name already has a declaration? We add a new symbol table entry, but keep hold of the old one using the `prev_defn` link.

⟨Symbol table shared fields 14e⟩ ≡
```
union symbol_table_info *prev_defn;
```
Macro defined by scraps 13df, 14e, 19b.
Macro referenced in scrap 12d.

To add an entry to the symbol table for a name we must supply its size, type and stack offset.

⟨Symbol table prototypes 14f⟩ ≡
```

extern void insert_name_with_offset(name name, int offset, int size, type type);
```
Macro defined by scraps 11c, 13e, 14bf, 15a, 16ae, 17ac, 20ac, 21ad, 22a.
Macro referenced in scrap 23a.

⟨Symbol table functions 14g⟩ ≡
```

void insert_name_with_offset(name name, int offset, int size, type type)
{ symbol_table_entry s;
  s = new(symbol_table_info);
  s->name.prev_defn = name->table; name->table = s;
  s->name.nesting_level = nesting_level;
  s->name.size = size; s->name.type = type;
  ⟨Add name to current scope 19a⟩
  s->name.offset = offset;
}
```
Macro defined by scraps 11d, 14g, 15b, 16bf, 17bd, 20bd, 21e, 22b.
Macro referenced in scrap 23d.

It becomes tedious to have to explicitly manipulate the `stack_offset` field and so we provide a convenient abbreviation. Stacks grow downwards in the interpreter, and so stack offsets are usually negative (remember the offset is relative to the base of the stack frame). If the `type` is `VAL_T` we treat the `size` argument as the value of the name. The stack offset allocated for this name is returned as result. Note that if the object represented by the name occupies more than one stack slot, e.g. it is an array, then the offset returned will be to the beginning of the object (see Figure 2.3).

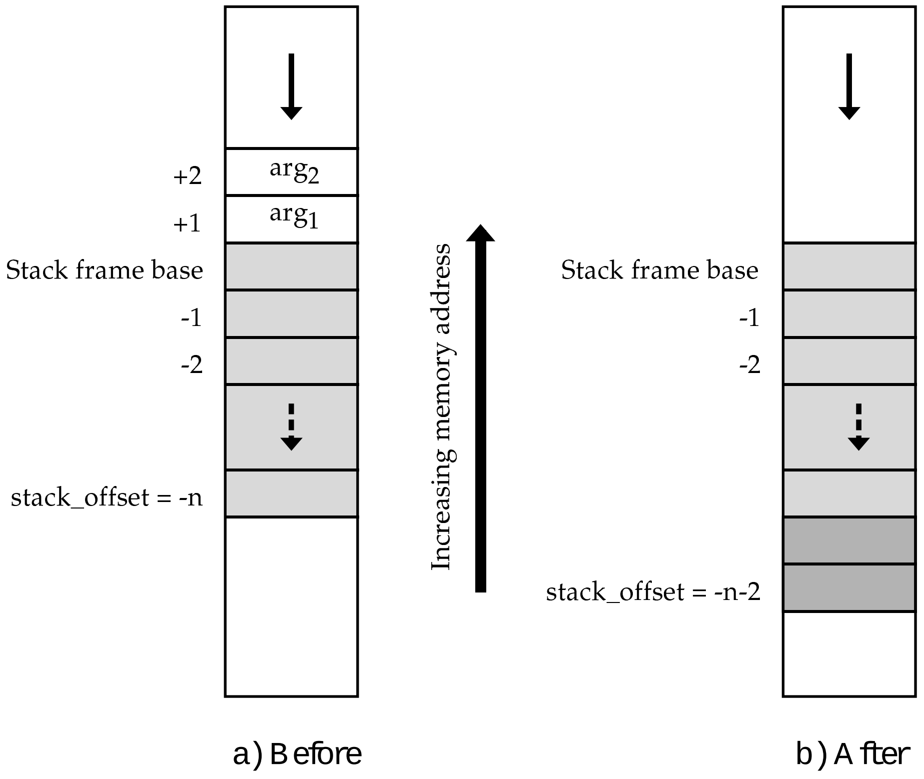

Figure 2.2: Effect of the call `insert_name(..., 2, ...)`

⟨Symbol table prototypes 15a⟩ ≡

```
extern int insert_name(name name, int size, type type);
```

Macro defined by scraps 11c, 13e, 14bf, 15a, 16ae, 17ac, 20ac, 21ad, 22a.
Macro referenced in scrap 23a.

⟨Symbol table functions 15b⟩ ≡

```
int insert_name(name name, int size, type type)
{ symbol_table_entry s;
  if (stack_offset > 1) stack_offset = 1; /* In case we have inserted args */
  if (type != VAL_T) stack_offset -= size;
  insert_name_with_offset(name, stack_offset, size, type);
  return stack_offset;
}
```

Macro defined by scraps 11d, 14g, 15b, 16bf, 17bd, 20bd, 21e, 22b.
Macro referenced in scrap 23d.

Arguments to procedures will be accessed with positive offsets from the base of the stack frame (see Section 5.2) and so we provide a variant of the `insert_name` function that adds `size` to `stack_offset` at each call. In this case we know that `type` is restricted by the language to `INT_T` or `CHAN_T`, and hence `size` is always 1.

⟨Symbol table prototypes 16a⟩ ≡

```
extern int insert_arg(name name, type type);
```

Macro defined by scraps 11c, 13e, 14bf, 15a, 16ae, 17ac, 20ac, 21ad, 22a.
Macro referenced in scrap 23a.

⟨Symbol table functions 16b⟩ ≡

```
int insert_arg(name name, type type)
{ symbol_table_entry s;
  insert_name_with_offset(name, stack_offset, 1, type);
  return(stack_offset++);
}
```

Macro defined by scraps 11d, 14g, 15b, 16bf, 17bd, 20bd, 21e, 22b.
Macro referenced in scrap 23d.

We treat inserting a procedure specially. The important things we need to know about a procedure are its starting location in memory (a label – see Section 5.6), the maximum amount of stack space it requires at runtime, and the types of its arguments.

⟨Symbol table typedef declarations 16c⟩ ≡

```
typedef struct arg {type type; struct arg *next; } *arg, *args;
```

Macro defined by scraps 10f, 12d, 16c, 18a.
Macro referenced in scrap 23a.

⟨Symbol table proc fields 16d⟩ ≡

```
int start_loc;
int stack_required;
struct arg *args;
```

Macro referenced in scrap 12d.

⟨Symbol table prototypes 16e⟩ ≡

```
extern void insert_proc(name name, int label, args a, int stack_required);
```

Macro defined by scraps 11c, 13e, 14bf, 15a, 16ae, 17ac, 20ac, 21ad, 22a.
Macro referenced in scrap 23a.

⟨Symbol table functions 16f⟩ ≡

```
void insert_proc(name name, int label, args a, int stack_required)
{ symbol_table_entry s;
  s = new(symbol_table_info);
  s->proc.nesting_level = nesting_level+1;
  s->proc.stack_required = stack_required; s->proc.type = PROC_T;
  s->proc.start_loc = label;
  s->proc.args = a;
  s->proc.prev_defn = name->table; name->table = s;
  ⟨Add name to current scope 19a⟩
}
```

Macro defined by scraps 11d, 14g, 15b, 16bf, 17bd, 20bd, 21e, 22b.
Macro referenced in scrap 23d.

Because we store different information for procedures and other names, we also need different lookup functions. We use the function `lookup_name` to retrieve the information currently associated with a non-procedure name. If no entry exists for the name we just give an error and stop. If the name represents a procedure at this point in the program we also complain and stop. The only non-obvious feature is that we return 0 for the nesting depth if the nesting depth of the name is the same as the current nesting depth (i.e. the name represents a local variable).

⟨Symbol table prototypes 17a⟩ ≡

```
extern void lookup_name(name name, int *nesting_level, int *stack_offset,
                        int *size, type *type);
```

Macro defined by scraps 11c, 13e, 14bf, 15a, 16ae, 17ac, 20ac, 21ad, 22a.
Macro referenced in scrap 23a.

⟨Symbol table functions 17b⟩ ≡

```
void lookup_name(name name, int *nld, int *so, int *sz, type *type)
{ extern int line_number;
  if (name->table == NULL)
    error("Syntax error",
          "Identifier %s has been used before it was declared.\n", name->name);
  else if (name->table->name.type == PROC_T)
    error("Syntax error",
          "You cannot use procedure name %s in this context.\n", name->name);
  else {
    if (nesting_level == name->table->name.nesting_level) *nld = 0;
    else *nld = name->table->name.nesting_level;
    *type = name->table->name.type; *sz = name->table->name.size;
    *so   = name->table->name.offset; }
}
```

Macro defined by scraps 11d, 14g, 15b, 16bf, 17bd, 20bd, 21e, 22b.
Macro referenced in scrap 23d.

Looking up a procedure name is similar; we just return different information.

⟨Symbol table prototypes 17c⟩ ≡

```
extern void lookup_proc(name name, int *nesting_level,
                        int *start_label, args *a, int *stack_required);
```

Macro defined by scraps 11c, 13e, 14bf, 15a, 16ae, 17ac, 20ac, 21ad, 22a.
Macro referenced in scrap 23a.

⟨Symbol table functions 17d⟩ ≡

```
void lookup_proc(name name, int *nld, int *sl, args *a, int *sr)
{ extern int line_number;
  if (name->table == NULL)
    error("Syntax error",
          "Identifier %s has been used before it was declared.\n", name->name);
  else if (name->table->proc.type != PROC_T)
    error("Syntax error",
          "The name %s was encountered where a procedure was expected.\n",
          name->name);
  else {
    *nld = name->table->proc.nesting_level;
    *sl  = name->table->proc.start_loc;
    *a   = name->table->proc.args;
    *sr  = name->table->proc.stack_required; }
}
```

Macro defined by scraps 11d, 14g, 15b, 16bf, 17bd, 20bd, 21e, 22b.
Macro referenced in scrap 23d.

What happens when we leave the scope of a declaration? We need to remove the declaration entry from the symbol table. But how do we find those entries that have been added during the current scope? A simple solution is to link together all names that have been given declarations during the current scope. We can't just link the table entries themselves as otherwise when we remove an entry we won't have access to the the pointer in the name structure and so cannot update it to point to the old table entry. This is a pity, as otherwise we could just add another link to the symbol table structure which would avoid having to define the following structure.

⟨Symbol table typedef declarations 18a⟩ ≡

```
typedef struct scope_table_entry {
  struct name_info *name;
  struct scope_table_entry *next;
} scope_table_entry, *scope_list;
```

Macro defined by scraps 10f, 12d, 16c, 18a.
Macro referenced in scrap 23a.

Scopes nest, and so we can use a stack to hold these lists. We keep track of the current scope level in the global variable `scope_level`.

⟨Symbol table global variables 18b⟩ ≡

```
int scope_level;
```

Macro defined by scraps 11a, 13g, 14c, 18bd, 21b.
Macro referenced in scrap 23d.

⟨Initialise symbol table globals 18c⟩ ≡

```
scope_level = 1;
```

Macro defined by scraps 11b, 13h, 14d, 18ce, 21c.
Macro referenced in scrap 23d.

For each scope level we keep track of the associated nesting level, the stack offset at the start of the scope. and the list of names declared during this scope. We have to decide how big to make this stack. We choose fifty, which should be more than adequate.

⟨Symbol table global variables 18d⟩ ≡

```
#define MAX_SCOPE_STACK 50
struct {
  scope_list scope_entries;
  int nesting_level;
  int stack_offset;
} scope_stack[MAX_SCOPE_STACK+1];
```

Macro defined by scraps 11a, 13g, 14c, 18bd, 21b.
Macro referenced in scrap 23d.

⟨Initialise symbol table globals 18e⟩ ≡

```
scope_stack[1].nesting_level = 1;
scope_stack[1].stack_offset =  1;
scope_stack[1].scope_entries = NULL;
```

Macro defined by scraps 11b, 13h, 14d, 18ce, 21c.
Macro referenced in scrap 23d.

I'd like to give a picture here illustrating the scope stacks. However, I fear that it will just look like a plate of spaghetti and so I'll just rely on the textual description. There may be some confusion due to the use of scope levels and nesting levels. The nesting level changes whenever we change stack frame, either by spawing a new process via `PAR` or by entering a procedure body. In the latter case the stack can change because the stack in use at the time the procedure is called may not be the same as the stack in effect when the procedure was declared. The scope level changes whenever we enter a new scope. Thus the nesting level will never exceed the scope level. An example may be useful here. Each variable declaration in the following program is indexed by three integers; the scope level, the nesting level and the offset. Some of the offsets might seem surprising. However, as explained in Section 5.2, the interpreter stores some information, the display, on the stack when a procedure is called, or a process created. The size of this information depends on the nesting depth of the procedure. A process at nesting depth n will have $n + 1$ such entries. A procedure at this depth will have $n + 2$ entries because the return `PC` also needs to be saved. The offsets also assume that a channel block occupies just one stack slot (true in the interpreter, but possibly not in a compiler for this language) and that a channel is represented by a pointer to this block. Thus each channel created will take up two stack slots.

```
CHAN C$_{1,1,-3}$:
VAL a$_{1,1,\_}$ IS 1:
PROC P(INT b$_{2,2,1}$)
  PROC Q(INT c$_{4,3,1}$)
    SEQ
      INT d$_{6,3,-5}$ = a+b:
      C ! (d+c)
      INT e$_{6,3,-5}$ = a:
      C ! (e+c)
  :
  Q(a)
:
INT a$_{1,1,-4}$ = 3:
PAR
  P(a)
  INT a$_{2,2,-3}$:
  SEQ
    C ? a
    INT a$_{3,2,-4}$:
    C ? a
```

Although there is no choice in the allocation of nesting level, there is some choice in the allocation of scope level. In the above example we enter a new scope (and nesting level) when we start to process the arguments to a procedure. For convenience, we also enter a new scope when we process the procedure body. However, we could process the body in the same scope as the arguments if we wanted to. The choice is often influenced by the code generation strategy.

When we add a new entry to the symbol table we must add the entry to the appropriate scope list.

⟨Add `name` to current scope 19a⟩ ≡

```
{ scope_list n;
  name->table->name.scope_level = scope_level;
  n = new(scope_table_entry);
  n->name = name;
  n->next = scope_stack[scope_level].scope_entries;
  scope_stack[scope_level].scope_entries = n;
}
```

Macro referenced in scraps 14g, 16f.

Note that we store the scope level with the name. This is to help us print out the symbol table in an informative fashion when debugging, but it is not essential.

⟨Symbol table shared fields 19b⟩ ≡

```
int scope_level;
```

Macro defined by scraps 13df, 14e, 19b.
Macro referenced in scrap 12d.

When we enter a new scope we must call the `enter_scope` function. This takes as argument a boolean indicating whether a change in nesting level is associated with the change of scope.

⟨Symbol table prototypes 20a⟩ ≡
```
extern void enter_scope(bool new_nesting_level);
```
Macro defined by scraps 11c, 13e, 14bf, 15a, 16ae, 17ac, 20ac, 21ad, 22a.
Macro referenced in scrap 23a.

⟨Symbol table functions 20b⟩ ≡
```

void enter_scope(bool new_nesting_level)
{ scope_stack[scope_level++].stack_offset = stack_offset;
  scope_stack[scope_level].scope_entries = NULL;
  if (new_nesting_level) {
    scope_stack[scope_level].nesting_level = ++nesting_level;
    stack_offset = 1; }
  else scope_stack[scope_level].nesting_level = nesting_level;
}
```
Macro defined by scraps 11d, 14g, 15b, 16bf, 17bd, 20bd, 21e, 22b.
Macro referenced in scrap 23d.

Note that the stack offset is set to 1 initially. We assume that the parser will set the stack offset to take into account the display at the current nesting level. We could do this in the `enter_scope` function, but if we are entering the scope of a procedure with arguments, then we want to start with a stack offset of 1. The arguments are then loaded into the symbol table with positive offsets. Once all the arguments are loaded we should then set the stack offset to `-nesting_level` (assuming we are using displays), by inserting a dummy name of size `nesting_level+1` into the table (of type `DISPLAY_T`).

When we exit a scope we call the `exit_scope` function. This removes the current symbol table entries for names introduced during this scope. If a name had a definition before this scope then that definition will be reinstated when we leave the scope (remember we saved the old entries using the `prev_defn` field). We return the amount of stack space occupied by the freed names as result. Channels complicate the situation. If it wasn't for these we could just generate code to do one big pop of all the stack space occupied by names in the current scope. However, because we have to pop channel blocks explicitly, we must process the popped names one at a time. When `exit_scope` is called it pops one name. If this is a value or a procedure then `exit_scope` is called recursively. If a channel block is encountered then the size is returned as a negative value. Otherwise the size of the name is returned. Thus the caller has a simple mechanism for distinguishing between popped channel blocks and other values. After repeated calls to `exit_scope` we will eventually reach the stage where there are no names left in the current scope. At this point the previous scope is reinstated and 0 returned to indicate that we have exited from the scope.

⟨Symbol table prototypes 20c⟩ ≡
```
extern int exit_scope();
```
Macro defined by scraps 11c, 13e, 14bf, 15a, 16ae, 17ac, 20ac, 21ad, 22a.
Macro referenced in scrap 23a.

⟨Symbol table functions 20d⟩ ≡
```

int exit_scope()
{ int size;
  scope_list entry = scope_stack[scope_level].scope_entries;
  if (entry == NULL) {
    --scope_level;
    nesting_level = scope_stack[scope_level].nesting_level;
    stack_offset  = scope_stack[scope_level].stack_offset;
    return 0; }

  else {
    name name = entry->name;
```

```
        union symbol_table_info *table = name->table;

        scope_stack[scope_level].scope_entries = entry->next;
        name->table = name->table->name.prev_defn;

        switch (table->name.type) {
          case VAL_T: size = exit_scope(); break;
          case PROC_T: {
            args o, args = table->proc.args;
            while (args != NULL) { o = args; args = args->next; dispose(o); }
            size = exit_scope(); break; }
          case CHAN_BLOCK_T:
            size = -table->name.size; break;
          default:
            size = table->name.size; if (size == 0) size = exit_scope(); break; }

        dispose(table); dispose(entry); return size;
      }
    }
```

Macro defined by scraps 11d, 14g, 15b, 16bf, 17bd, 20bd, 21e, 22b.
Macro referenced in scrap 23d.

The stack is used to hold values without an associated name. Examples include the display, channel blocks, saved program counters and counters for replicated constructs. We could just manipulate the stack offset directly when we need to. However, it seems neater to introduce dummy names for these objects so that a dump of the symbol table (see function `print_symbol_table` below) produces more informative output when debugging. We prefix these names with a space to avoid possible conflicts with 'real' names.

⟨Symbol table prototypes 21a⟩ ≡

```
extern name display, pc, chanblock, repl;
```

Macro defined by scraps 11c, 13e, 14bf, 15a, 16ae, 17ac, 20ac, 21ad, 22a.
Macro referenced in scrap 23a.

⟨Symbol table global variables 21b⟩ ≡

```
name display, pc, chanblock, repl;
```

Macro defined by scraps 11a, 13g, 14c, 18bd, 21b.
Macro referenced in scrap 23d.

⟨Initialise symbol table globals 21c⟩ ≡

```
display=find_name(" display"); pc=find_name(" PC");
chanblock=find_name(" chanblock"); repl=find_name(" repl");
```

Macro defined by scraps 11b, 13h, 14d, 18ce, 21c.
Macro referenced in scrap 23d.

It's sometimes useful, for debugging purposes, to print out the current state of the symbol table. Here is a function that does just that. If there are two declarations with the same name in the same scope then only the details of the later one will be reported, but twice. It would be messy to detect this case, and it's not too important, so we don't bother.

⟨Symbol table prototypes 21d⟩ ≡

```
extern void print_symbol_table();
```

Macro defined by scraps 11c, 13e, 14bf, 15a, 16ae, 17ac, 20ac, 21ad, 22a.
Macro referenced in scrap 23a.

⟨Symbol table functions 21e⟩ ≡

```
void print_symbol_table()
```

```
{ int sl = scope_level;
  scope_list entry;
  symbol_table_entry tabent;
  printf("Symbol table entries...\n");
  for (sl = scope_level; sl >= 1; sl--) {
    entry = scope_stack[sl].scope_entries;
    if (entry == NULL) continue;
    printf("%2d: nesting level = %d.\n", sl, scope_stack[sl].nesting_level);
    while (entry != NULL) {
      printf("    ");
      tabent = entry->name->table;
      while (tabent->name.scope_level != sl) tabent = tabent->name.prev_defn;
      switch (tabent->name.type) {
        case INT_T       : printf("INT "); break;
        case CHAN_T      : printf("CHAN "); break;
        case CHAN_BLOCK_T: printf("CHAN BLOCK"); break;
        case PROC_T      : printf("PROC "); break;
        case DISPLAY_T   : printf("DISPLAY "); break;
        case INT_ARRAY_T :
          printf("[%d]INT ", entry->name->table->name.size); break;
        case CHAN_ARRAY_T:
          printf("[%d]CHAN ", entry->name->table->name.size); break; }
      printf("%s: nesting level = %d, ",
        entry->name->name, tabent->name.nesting_level);
      if (tabent->proc.type == PROC_T)
        printf("start label = %d\n", tabent->proc.start_loc);
      else printf("offset = %d\n", tabent->name.offset);
      entry = entry->next;
    }
  }
}
```

Macro defined by scraps 11d, 14g, 15b, 16bf, 17bd, 20bd, 21e, 22b.
Macro referenced in scrap 23d.

Names are never discarded from the hash table. This makes it difficult to detect storage leaks in other parts of the program. The following function can be called at the end of compilation to return this storage. It's not essential that it is called as the storage will be returned anyway when the program terminates. However, it might be useful for debugging purposes.

⟨Symbol table prototypes 22a⟩ ≡

```
extern void clear_symbol_table();
```

Macro defined by scraps 11c, 13e, 14bf, 15a, 16ae, 17ac, 20ac, 21ad, 22a.
Macro referenced in scrap 23a.

⟨Symbol table functions 22b⟩ ≡

```
void clear_symbol_table()
{ int i;
  for (i=0; i < hash_size; i++) {
    name p = hash[i]; name o;
    while (p != NULL) {
      if (p->table != NULL)
        error("Internal error",
          "You should leave the outermost scope before calling "
          "clear_symbol_table.\n" );
      o = p; p=p->hash_link; free(o->name); dispose(o); }
    hash[i] = NULL;
  }
}
```

Macro defined by scraps 11d, 14g, 15b, 16bf, 17bd, 20bd, 21e, 22b.
Macro referenced in scrap 23d.

Finally, here are the files for the symbol table.

```
"code/symbol_table.h" 23a ≡
      ⟨Copyright message 2a⟩
      #ifndef _SYMBOL_TABLE_H
      #define _SYMBOL_TABLE_H
      ⟨Symbol table type typedef 13c⟩
      ⟨Symbol table typedef declarations 10f, ... ⟩
      ⟨Symbol table prototypes 11c, ... ⟩

      extern void initialise_symbol_table();

      #endif
```

```
⟨Occam object files 23b⟩ ≡
       symbol_table.o
```

Macro defined by scraps 10d, 23b, 25d, 28d, 78a, 86a.
Macro referenced in scrap 3.

```
⟨Makefile dependencies 23c⟩ ≡

      symbol_table.c: occam.h pool.h symbol_table.h
```

Macro defined by scraps 10c, 23c, 25c, 28c, 78b, 85f.
Macro referenced in scrap 3.

```
"code/symbol_table.c" 23d ≡
      ⟨Standard μOCCAM includes 2b⟩
      #include <string.h>
      #include "occam.h"
      #include "pool.h"
      #include "symbol_table.h"

      ⟨Symbol table global variables 11a, ... ⟩
      ⟨Symbol table functions 11d, ... ⟩

      void initialise_symbol_table()
      { ⟨Initialise symbol table globals 11b, ... ⟩ }
```

Function name	Return value type	Argument types	Page No.
extend_pool	void *	int	9
pool_statistics	void	void	9
find_name	name	char *	11
insert_name_with_offset	void	name, int, int, type	14
insert_name	int	name, int, type	15
insert_arg	int	name, type	16
insert_proc	void	name, int, args, int	16
lookup_name	void	name, int *, int *, int *, type *	17
lookup_proc	void	name, int *, int *, args *, int *	17
enter_scope	void	bool	20
exit_scope	int	void	20
print_symbol_table	void	void	21
clear_symbol_table	void	void	22

This table shows all the functions defined in the preceeding chapter, their argument and return parameter types and the page on which the function is defined.

Chapter 3

The Lexer

This chapter contains only a minimal lexer, which ignores many aspects of the language.

3.1 Lexer files

The header file declares functions and variables exported by the lexer.

"code/lexer.h" 25a ≡

```
#ifndef _LEXER_H
#define _LEXER_H
extern void initialise_lexer(FILE* input);
extern int yylex(void);
extern int line_number;
#endif
```

The actual lexer description follows the standard lex layout.

"code/lexer.l" 25b ≡

```
%{
⟨Lexer C declarations 26a, ... ⟩
%}
⟨Lexer options 26c⟩
%%
⟨Lexer rules 26d⟩
%%
⟨Lexer user code 27⟩
```

The lexer requires its own section in the Makefile.

⟨Makefile dependencies 25c⟩ ≡

```
lexer.o: parser.tab.h lexer.h
lexer.c: lexer.l
        flex -olexer.c lexer.l
```

Macro defined by scraps 10c, 23c, 25c, 28c, 78b, 85f.
Macro referenced in scrap 3.

It will also give rise to an object file.

⟨Occam object files 25d⟩ ≡

```
lexer.o
```

Macro defined by scraps 10d, 23b, 25d, 28d, 78a, 86a.
Macro referenced in scrap 3.

3.2 Definitions

We need our own header file, and one generated by Bison from `parser.h` that will include information about the tokens to generate.

⟨Lexer C declarations 26a⟩ ≡

```
#include "lexer.h"
#include "parser.tab.h"
```

Macro defined by scraps 26ab.
Macro referenced in scrap 25b.

We keep track of the current line number in a global variable.

⟨Lexer C declarations 26b⟩ ≡

```
int line_number;
```

Macro defined by scraps 26ab.
Macro referenced in scrap 25b.

We don't want to read more than one input file.

⟨Lexer options 26c⟩ ≡

```
%option noyywrap
```

Macro referenced in scrap 25b.

3.3 Rules

As stated earlier, this is a minimal lexer. It recognises only two keywords, newlines, and some comments. There is no attempt to measure indentation.

⟨Lexer rules 26d⟩ ≡

```
\n        { line_number++; return(NL); }

STOP      { return(STOP); }
SEQ       { return(SEQ); }

[ ]*                          /* Ignore spaces */

--.*\n    { line_number++; }  /* Pass over comments */

.         error("Lexical error", "Unrecognised input.\n");
```

Macro referenced in scrap 25b.

3.4 User code

The main function the lexer provides is `yylex`, which flex will write for us. We still need to establish where to take input from, and for this we export the following function.

⟨Lexer user code 27⟩ ≡

```
void initialise_lexer(FILE* input)
{
  yyin = input;
  line_number = 1;
}
```

Macro referenced in scrap 25b.

This is called later from the file handling code in Section 6.2.

Chapter 4

The Parser and Code Generator

This chapter contains only a minimal parser. Like the lexer, it ignores almost every part of the language; it also generates no meaningful code. It does however interact properly with the rest of the μOCCAM compiler, so makes a suitable basis for improvement.

4.1 Parser files

Only one function is exported by the parser.

"code/parser.h" 28a ≡

```
#ifndef _PARSER_H
#define _PARSER_H
extern int yyparse(void);
#endif
```

The parser description is in the standard format.

"code/parser.y" 28b ≡

```
%{
⟨Parser C declarations 29a⟩
%}
⟨Parser Bison declarations 29b⟩
%%
⟨Parser grammar rules 29c, ... ⟩
%%
⟨Parser user code 30c⟩
```

We put a suitable note of dependencies in the Makefile, and record the object file it will generate.

⟨Makefile dependencies 28c⟩ ≡

```
parser.tab.o: occam.h pool.h interpreter.h symbol_table.h lexer.h parser.h
parser.tab.c parser.tab.h : parser.y
        bison --defines parser.y
```

Macro defined by scraps 10c, 23c, 25c, 28c, 78b, 85f.
Macro referenced in scrap 3.

⟨Occam object files 28d⟩ ≡

```
parser.tab.o
```

Macro defined by scraps 10d, 23b, 25d, 28d, 78a, 86a.
Macro referenced in scrap 3.

4.2 Declarations

The only declarations required by C are the included header files.

⟨Parser C declarations 29a⟩ ≡

```
⟨Standard μOCCAM includes 2b⟩

#include "pool.h"
#include "symbol_table.h"
#include "occam.h"
#include "interpreter.h"
#include "parser.h"
```

Macro referenced in scrap 28b.

Bison needs to know what tokens to expect.

⟨Parser Bison declarations 29b⟩ ≡

```
%token NL
%token STOP
%token SEQ
```

Macro referenced in scrap 28b.

4.3 Grammar

The start symbol is `program`, which consists of a single process.

⟨Parser grammar rules 29c⟩ ≡

```
program :
⟨Initialise symbol table for parser 30a⟩
  process
⟨Build code for a trivial program 30b⟩
```

Macro defined by scraps 29cd.
Macro referenced in scrap 28b.

A process can either be a command or some more complex construction. In this tiny parser, `STOP` is the only command and `SEQ` the only constructor.

⟨Parser grammar rules 29d⟩ ≡

```
process: STOP NL | sequence;

sequence: SEQ NL processes;

processes:                   /* no processes at all */
         | process processes; /* one or more processes */
```

Macro defined by scraps 29cd.
Macro referenced in scrap 28b.

Notice the blank entry representing ε; a list of processes might include none at all.

Before the program is even started, the interpreter initialises processes to handle `stdin` and `stdout`. So we have to tell the symbol table that these things are already on the stack.

⟨Initialise symbol table for parser 30a⟩ ≡

```
{ insert_name (display, 2, DISPLAY_T);
  insert_name (chanblock,  CHANNEL_BLOCK_SIZE, CHAN_BLOCK_T);
  insert_name (find_name("stdin" ), 1, CHAN_T);
  insert_name (chanblock,  CHANNEL_BLOCK_SIZE, CHAN_BLOCK_T);
  insert_name (find_name("stdout"), 1, CHAN_T); }
```

Macro referenced in scrap 29c.

Rather than actually generate real code while parsing, this minimal solution writes only the "stop now" code to go at the end.

⟨Build code for a trivial program 30b⟩ ≡

```
{ program_code = build_ccode(exit_scope_code(), Stop, END); }
```

Macro referenced in scrap 29c.

The call to `exit_scope_code` generates appropriate code to deallocate one level of scoped values from the stack — these will be the channels for `stdin` and `stdout` mentioned above.

The variable `program_code` is where the parser must put the code sequence for the whole program. See Section 5.6 for some information on how to construct code sequences.

4.4 User code

The only additional user code required is that for error reporting.

⟨Parser user code 30c⟩ ≡

```
yyerror (char* s)
{
  error ("Parse error", "Unrecognized input.\n");
}
```

Macro referenced in scrap 28b.

Chapter 5

The μOCCAM Abstract Machine

The μOCCAM abstract machine is basically a simple stack machine extended with instructions to support processes and communication. We start by describing the sequential aspects of the machine and then deal with the complications introduced by concurrency. The machine implementation errs on the side of clarity rather than efficiency.

Note: The machine doesn't make many safety checks. It is up to the code generator to ensure that stacks don't overflow, jumps are in range etc.

5.1 The Sequential Machine

The machine uses a stack to hold the program variables and intermediate values. The code for the program is stored in a separate array. Two machine registers, `SP` and `PC` are used to index into these components, as illustrated in Figure 5.1.

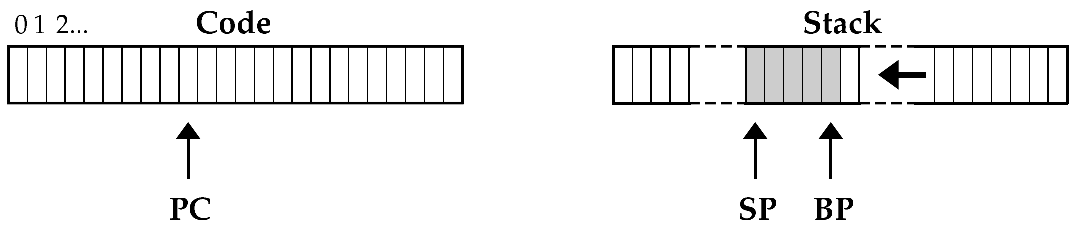

Figure 5.1: The sequential machine state

An instruction consists of two components. The `op` field determines the instruction type. The `arg` field contains an argument for the operation. Only some of the operations take arguments. The value in the `arg` field is ignored in the other cases. We assume that integers occupy thirty-two bits. It turns out to be convenient if instructions can occupy the same amount of space as ints, and so we restrict the size of the argument field to twenty-four bits. This should be more than adequate for our needs. Table 5.1 on page 33 lists the complete instruction set of the abstract machine.

⟨Machine opcodes `typedef` 31⟩ ≡

```
    typedef enum opcode
    { ⟨Machine opcodes 32e, ... ⟩ } opcode;
```

Macro referenced in scrap 77e.

⟨Machine `typedefs` 32a⟩ ≡

```
typedef struct instruction
{ opcode op :  8;
  int arg   : 24;
} instruction;
```

Macro defined by scraps 32ac, 57cd, 72d.
Macro referenced in scrap 77e.

The instructions to be executed by the machine are held in an array of instructions called `Code`. The program counter, `PC`, indexes into this array.

⟨Machine state 32b⟩ ≡

```
register int PC; /* Indexes into Code array */
```

Macro defined by scraps 32bd.
Macro referenced in scrap 71b.

Each stack element is thirty-two bits long. Most of the time the stack is used to hold integers. However, occasionally we have to store other kinds of value on the stack. These will be introduced as the need arises. We therefore use a `union` to avoid lots of arbitrary casts. The variable `BP` points into this stack; initially it points to the end of the stack, but it is changed whenever a procedure call is made (see Section 5.2). The stack pointer, `SP`, points to the top of the stack. Stacks grow downwards, and so usually `SP` $\leq$ `BP` except when the stack is empty.

⟨Machine `typedefs` 32c⟩ ≡

```
typedef union stack_slot
{ int as_int;
  ⟨Other stack_slot options 34e, ... ⟩
} stack_slot;
```

Macro defined by scraps 32ac, 57cd, 72d.
Macro referenced in scrap 77e.

⟨Machine state 32d⟩ ≡

```
stack_slot *BP = NULL, *SP = NULL;
```

Macro defined by scraps 32bd.
Macro referenced in scrap 71b.

5.1.1 Load and Store Instructions

A variety of instructions are provided to load values onto the stack, and to store values.

The `Ldc` instruction takes a constant as argument and pushes it onto the top of the stack.

⟨Machine opcodes 32e⟩ ≡

```
 Ldc = 1,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 32f⟩ ≡

```
case Ldc : { (--SP)->as_int = curr_inst.arg; break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 32g⟩ ≡

```
case Ldc : return(1);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

Mnemonic	Operation Performed	Page
Ldc	Load constant value onto top of stack	32
Ldo	Load existing stack element onto top of stack	34
Lda	Load address of element onto top of stack	34
Ind	Perform indirections to retrieve data from stack	34
Sto	Write top value to stack location with constant address	35
Sta	Write top value to stack location with variable address	35
Ccb	Create new channel	35
Dcb	Dispose of channel control block	36
Swap	Swap topmost two elements on stack	36
Pop	Discard element from top of stack	37
Ldn	Reserve stack space for array	37
Idx	Access element of an array	37
Ida	Calculate address of array element	38
Add	Add two numbers	38
Sub	Subtract one number from another	38
Mul	Multiply two numbers	38
Div	Divide one number by another	38
Mod	Modulus operator - find remainder of division	38
Lt	Less than relational comparison	39
Leq	Less than or equal to relational comparison	39
Gt	Greater than relational comparison	39
Geq	Greater than or equal to relational comparison	39
Eq	Equal to relational comparison	39
Neq	Not equal relational comparison	39
Ujp	Unconditional jump instruction	39
Fjp	Conditional jump instruction	39
Enter	Setup display for procedure call	40
Leave	Restore previous activation record	41
Call	Transfer control to procedure	41
Ret	Return from procedure	42
Spawn	Spawn new process	43
Spawnv	Spawn new process with specified initial value	43
Stop	Terminate current process	44
Wait	Wait until all child processes have terminated before continuing	44
Out	Output a value onto channel	47
In	Take input value from channel	46
Alt	Choose value from set of alternatives	46
Input	Read integer from stdin and push onto stack	46
Output	Pop integer from stack and write to stdout	47
Trace	Switch on/off machine state tracing in the debugger	48
Lab	Label pseudo-instruction	49

Table 5.1: Instruction set for the μOCCAM abstract machine.

To access existing stack elements we use the `Ldo` instruction. Its argument is the stack offset (relative to `BP`).

⟨Machine opcodes 34a⟩ ≡

```
Ldo,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 34b⟩ ≡

```
case Ldo : { (--SP)->as_int = BP[curr_inst.arg].as_int;  break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 34c⟩ ≡

```
case Ldo : return(1);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

To access the *address* of a stack element we use the `Lda` instruction.

⟨Machine opcodes 34d⟩ ≡

```
Lda,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

We extend the kinds of values that can be stored on the stack to include addresses.

⟨Other `stack_slot` options 34e⟩ ≡

```
union stack_slot *as_addr;
```

Macro defined by scraps 34e, 45g, 54c.
Macro referenced in scrap 32c.

⟨Execute instruction `curr_inst` 34f⟩ ≡

```
case Lda : { (--SP)->as_addr = &BP[curr_inst.arg];  break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 34g⟩ ≡

```
case Lda : return(1);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

We can perform repeated indirections via an address on the top of the stack using the `Ind` instruction. The number of indirections to be performed is passed as the argument to the instruction.

⟨Machine opcodes 34h⟩ ≡

```
Ind,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 35a⟩ ≡

```
case Ind : { int n = curr_inst.arg;
             while (n--) *SP = *(SP->as_addr);  break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 35b⟩ ≡

```
case Ind : return(0);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

We can overwrite an arbitrary stack location with the contents of the top of stack using the `Sto` instruction. The stack top is popped as a by-product of this operation.

⟨Machine opcodes 35c⟩ ≡

```
Sto,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 35d⟩ ≡

```
case Sto : { BP[curr_inst.arg] = *(SP++); break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 35e⟩ ≡

```
case Sto : return(-1);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

Sometimes we don't know the location to write to at compile time (e.g. an array access) and so we provide a variant, called `Sta`, that takes the destination address off the stack. The value is on the top of the stack, with the destination below it.

⟨Machine opcodes 35f⟩ ≡

```
Sta,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 35g⟩ ≡

```
case Sta : { *((SP+1)->as_addr) = *SP; SP += 2; break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 35h⟩ ≡

```
case Sta : return(-2);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

To create a new channel we use the `Ccb` instruction. Channels in the interpreter occupy two stack slots. The first one, the channel control block, contains a pointer to a waiting process, or `NULL` if no process is blocked. The second slot contains the address of the channel control block. The argument to the instruction contains the number of channels to create. If the argument is > 1 then all the channel control blocks are allocated first, followed by the pointers to these blocks. This allows channel arrays to be indexed like integer arrays.

⟨Machine opcodes 36a⟩ ≡

```
Ccb,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 36b⟩ ≡

```
case Ccb : {
  int i = curr_inst.arg; stack_slot *start;
  while (i-- > 0) (--SP)->as_addr = NULL;
  i = curr_inst.arg; start = SP;
  while (i-- > 0) (--SP)->as_addr = start++;
  break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 36c⟩ ≡

```
case Ccb : return(2*curr_inst.arg);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

We also provide an instruction to dispose of the channel control blocks. This is not really necessary in the interpreter as we could just pop the stack. However, in the runtime system for a compiler we may have to tidy up system data associated with a channel control block (e.g. mutexes) and so we make channel block destruction explicit. The number of channel control blocks to dispose of is passed as argument to the instruction. Note: the `Ccb` instruction creates the channel control blocks *and* the pointers to them. The `Dcb` instruction just disposes of the channel control blocks.

⟨Machine opcodes 36d⟩ ≡

```
Dcb,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 36e⟩ ≡

```
case Dcb: { SP += curr_inst.arg; break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 36f⟩ ≡

```
case Dcb : return(-curr_inst.arg);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

It is sometimes difficult to produce the arguments to some of these instructions in the correct order. The `Swap` instruction allows us to exchange the top two elements on the stack.

⟨Machine opcodes 37a⟩ ≡

```
Swap,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 37b⟩ ≡

```
case Swap : { stack_slot s = *SP; *SP = *(SP+1); *(SP+1) = s; break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 37c⟩ ≡

```
case Swap : return(0);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

We can also discard elements off the top of the stack using the `Pop` instruction.

⟨Machine opcodes 37d⟩ ≡

```
Pop,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 37e⟩ ≡

```
case Pop : { SP += curr_inst.arg; break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 37f⟩ ≡

```
case Pop : return(-curr_inst.arg);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

Arrays are stored on the stack just like any other values. To reserve n slots on the stack, each initialised to 0, the `Ldn` instruction is used.

⟨Machine opcodes 37g⟩ ≡

```
Ldn,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 37h⟩ ≡

```
case Ldn : { int i = curr_inst.arg; while (i-- > 0) (--SP)->as_int = 0;
             break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 38a⟩ ≡

```
case Ldn : return(curr_inst.arg);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

To allow array elements to be accessed the machine provides an index instruction, `Idx`. Both the array base and index may be unknown at compile time and so these arguments are passed on the stack rather than one or other of them being arguments of the instruction itself. The index is on the top of the stack, and the base one down. The base is represented by an address rather than as a stack offset. The `Lda` instruction is useful for setting up an appropriate base.

⟨Machine opcodes 38b⟩ ≡
```
    Idx,
```
Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 38c⟩ ≡
```
    case Idx : { *(SP+1) = ((SP+1)->as_addr)[SP->as_int]; SP++; break; }
```
Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 38d⟩ ≡
```
    case Idx : return(-1);
```
Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

To get the address of an indexed expression, rather than its value, use the `Ida` instruction.

⟨Machine opcodes 38e⟩ ≡
```
    Ida,
```
Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 38f⟩ ≡
```
    case Ida : { (SP+1)->as_addr += SP->as_int; SP++; break; }
```
Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 38g⟩ ≡
```
    case Ida : return(-1);
```
Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

5.1.2 Arithmetic and Relational Instructions

A variety of arithmetic and relational operations are provided. These act on the top element(s) of the stack. In the case of binary operators be careful of the argument order; the left argument of the operator is one down in the stack and the right argument is at the stack top. The argument component of each of these instructions is ignored.

⟨Machine opcodes 39a⟩ ≡
```
    Add, Sub, Mul, Div, Mod,
```
Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 39b⟩ ≡
```
    case Add : { (SP+1)->as_int += SP->as_int; SP++; break; }
    case Sub : { (SP+1)->as_int -= SP->as_int; SP++; break; }
    case Mul : { (SP+1)->as_int *= SP->as_int; SP++; break; }
    case Div : { (SP+1)->as_int /= SP->as_int; SP++; break; }
    case Mod : { (SP+1)->as_int %= SP->as_int; SP++; break; }
```
Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 39c⟩ ≡

```
case Add : case Sub: case Mul: case Div: case Mod: return(-1);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

The relational operators treat zero as representing false, and anything else as representing true.

⟨Machine opcodes 39d⟩ ≡

```
Lt, Leq, Gt, Geq, Eq, Neq,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 39e⟩ ≡

```
case Lt  : { (SP+1)->as_int = (SP+1)->as_int <  SP->as_int; SP++; break; }
case Leq : { (SP+1)->as_int = (SP+1)->as_int <= SP->as_int; SP++; break; }
case Gt  : { (SP+1)->as_int = (SP+1)->as_int >  SP->as_int; SP++; break; }
case Geq : { (SP+1)->as_int = (SP+1)->as_int >= SP->as_int; SP++; break; }
case Eq  : { (SP+1)->as_int = (SP+1)->as_int == SP->as_int; SP++; break; }
case Neq : { (SP+1)->as_int = (SP+1)->as_int != SP->as_int; SP++; break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 39f⟩ ≡

```
case Lt : case Leq: case Gt: case Geq: case Eq: case Neq: return(-1);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

5.1.3 Control Flow Instructions

In the sequential fragment of the language there are two instructions for altering the control flow. The `Ujp` instruction is used to perform unconditional jumps. The argument indicates the new value of the program counter.

⟨Machine opcodes 39g⟩ ≡

```
Ujp,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 40a⟩ ≡

```
case Ujp : { PC = curr_inst.arg; break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 40b⟩ ≡

```
case Ujp : return(0);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

We can also perform a conditional jump using the `Fjp` instruction. The jump is only performed if the top of stack contains the value `false`, i.e. is zero. The top of stack is popped by this instruction.

⟨Machine opcodes 40c⟩ ≡
```
        Fjp,
```
Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 40d⟩ ≡
```
      case Fjp : { if (!(SP++)->as_int) PC = curr_inst.arg; break; }
```
Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 40e⟩ ≡
```
      case Fjp : return(-1);
```
Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

5.2 Procedure Calls

The space occupied on the stack by each procedure invocation is called its *activation record.* This includes information such as the procedure arguments, the return address and the space required for declarations introduced in the procedure body. If we maintain a pointer, `BP`, to a fixed point in this activation record then the procedure can access the fields of the record via offsets from this pointer. Whenever we make a new procedure call, and hence a new activation record, we must adjust `BP` and `SP` accordingly.

Procedures in μ`OCCAM` are statically scoped, i.e. if the procedure body refers to a name not declared in the procedure then this *free* name is looked up in the environment that existed at the time the procedure was declared, *not* the environment that existed at the time the procedure was called. Thus μ`OCCAM` is similar to languages such as Pascal and, to some extent, C. Procedures can be nested, but they cannot be passed as arguments to other procedures. To allow access to free names we could either use *access links*, or *displays* (see the "Dragon" book for more details). We use displays in this interpreter. The general idea is quite simple. If we are currently executing the body of a procedure at nesting depth n, then then code can access the activation record at nesting depth $i < n$, and hence the local variables at this depth, by following the pointer `BP[`$-i$`]`. Remember that stacks grow downwards, hence the negative offset. Before calling a procedure we must set up a display appropriate for the nesting depth of the new procedure. The `Enter` instruction does this. It takes as argument the new nesting depth.

⟨Machine opcodes 41a⟩ ≡
```
        Enter,
```
Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 41b⟩ ≡
```
      case Enter : initialise_display(curr_inst.arg, &SP, &BP); break;
```
Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Machine utility functions 41c⟩ ≡

```
void initialise_display(int nesting_level,
                        stack_slot **the_SP, stack_slot **the_BP)
{ stack_slot *SP = *the_SP, *BP = *the_BP;
  (--SP)->as_addr = BP;
  *the_BP = SP;
  while (--nesting_level > 0) *(--SP) = *(--BP);
  (--SP)->as_addr = *the_BP;
  *the_SP = SP;
}
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

⟨Stack effect of executing instruction `curr_inst` 41d⟩ ≡

```
case Enter : return(curr_inst.arg+1);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

The `Enter` instruction updates the `BP` and `SP` registers to point to the new display, but the old values are saved on the stack. The previous activation record can be restored using the `Leave` instruction.

⟨Machine opcodes 41e⟩ ≡

```
 Leave,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 41f⟩ ≡

```
case Leave : { SP = BP; BP = (SP++)->as_addr; break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

The argument to `Leave` is the nesting level of the called procedure. It is ignored when evaluating the instruction, but allows the stack effect of the instruction to be easily computed.

⟨Stack effect of executing instruction `curr_inst` 41g⟩ ≡

```
case Leave : return(-(curr_inst.arg+2));
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

To transfer control to a procedure, after the display has been set up, we use the `Call` instruction which saves the current PC on the stack and then jumps to the start of the procedure.

⟨Machine opcodes 42a⟩ ≡

```
 Call,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 42b⟩ ≡

```
case Call : { (--SP)->as_int = PC; PC = curr_inst.arg; break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 42c⟩ ≡

```
case Call : return(1);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

To return we need to pop the stack, if necessary, until the return address is on top, and then execute the `Ret` instruction.

⟨Machine opcodes 42d⟩ ≡

```
 Ret,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 42e⟩ ≡

```
case Ret : { PC = (SP++)->as_int; break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

The stack effect might seem a bit odd. However, the effect is from the view of the procedure currently executing, not the procedure being returned to.

⟨Stack effect of executing instruction `curr_inst` 42f⟩ ≡

```
case Ret : return(0);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

Arguments to a procedure should be placed on the stack *before* the `Enter` instruction is evaluated. They can then be accessed by the called procedure using positive offsets from `BP`. Thus a typical calling sequence looks like

Evaluate m arguments

Enter n

Call L /* Call procedure at nesting depth n */

Leave n

Pop m

The stack during the execution of the procedure is illustrated in Figure 5.2.

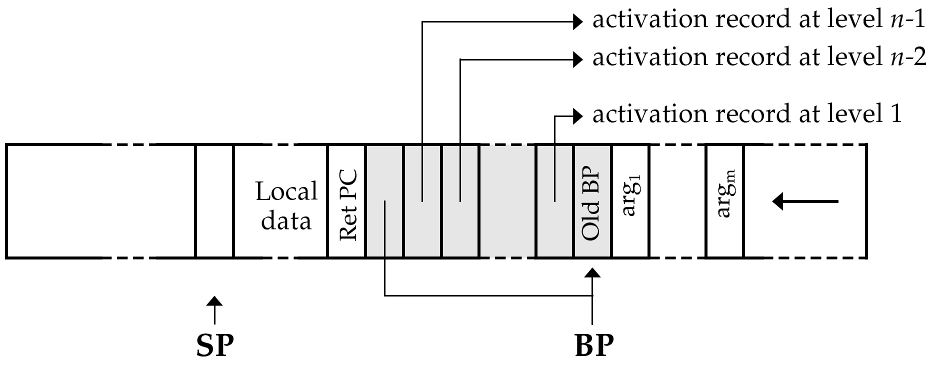
activation record at level n-1
activation record at level n-2
activation record at level 1
Local data
Ret PC
Old BP
arg_1
arg_m
SP
BP

Figure 5.2: A typical activation record at nesting level n

5.3 The Parallel Machine

A process may spawn many subprocesses. Each of these must be allocated its own stack, otherwise chaos reigns. However, each subprocess must be able to access (but not update) variables in its parent's stack if they are in scope. We must therefore provide a mechanism for accessing the stacks of our ancestors. We can use the display mechanism for this purpose. We treat the spawning of a process as a degenerate kind of procedure call. The display at the beginning of the new stack contains a pointer to the activation record of the spawner. However, unlike a procedure call, we don't need to save the PC as processes don't return to the caller.

The state of the abstract machine, from the viewpoint of a single process, now looks like Figure 5.3.

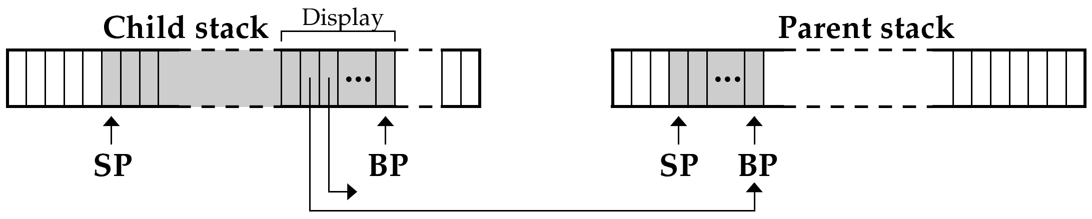

Figure 5.3: The parallel machine state

In order to create a process we must specify the stack size to be used for the new process, and the initial value of its program counter. Note that all processes share the same code array. The compiler needs to calculate the maximum stack size required by the process (including the display at the base of the stack). The language is constrained enough that the maximum can be calculated at compile time. The `Spawn` instruction can be used to spawn a new process. This creates the data structures required by the new process and schedules it for execution. The process counter is supplied as an argument to the instruction and the stack size is passed on the top of the stack. The nesting level of the new process is supplied as the next element on the stack.

⟨Machine opcodes 43a⟩ ≡
```
    Spawn,
```
Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 43b⟩ ≡
```
    case Spawn : {
      process p = create_process(curr_inst.arg, (SP++)->as_int);
      p->BP = BP;
      initialise_display((SP++)->as_int, &p->SP, &p->BP);
      ⟨Schedule process p for execution 71c⟩
      break; }
```
Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 44a⟩ ≡
```
    case Spawn : return(-2);
```
Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

Sometimes it is useful to be able to pass an initial value to the spawned process (e.g. in the case of a replicated `PAR`). The `Spawnv` instruction allows you to do this.

⟨Machine opcodes 44b⟩ ≡

```
Spawnv,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 44c⟩ ≡

```
case Spawnv: {
  process p = create_process(curr_inst.arg, (SP++)->as_int);
  p->BP = BP;
  initialise_display((SP++)->as_int, &p->SP, &p->BP);
  *(--(p->SP)) = *(SP++);
  ⟨Schedule process p for execution 71c⟩
  break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 44d⟩ ≡

```
case Spawnv : return(-3);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

To halt a process we use the `Stop` instruction. This terminates the current process and schedules the next process currently awaiting execution. If all other processes have terminated, or are blocked, then the interpreter Stops.

⟨Machine opcodes 44e⟩ ≡

```
Stop,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 44f⟩ ≡

```
case Stop: { ⟨Terminate current process 52b⟩
             ⟨Schedule next process or return if none 72c⟩ }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 44g⟩ ≡

```
case Stop : return(0);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

When we spawn processes in a `PAR` construct we must wait until the spawned processes have terminated before we can proceed further. The `Wait` instruction performs this task.

⟨Machine opcodes 45a⟩ ≡

```
Wait,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 45b⟩ ≡

```
case Wait: { ⟨Suspend if children and schedule next process 72b⟩ }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 45c⟩ ≡

```
case Wait : return(0);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

5.3.1 Interprocess Communication

μOCCAM treats input and output asymmetrically. Inputs can appear in alternatives, but outputs cannot. Although this is often a pain from the programmer's viewpoint, it simplifies the implementation somewhat.

The `Out` instruction tries to output a value on a channel, and suspends the current process if no partner is waiting for input on this channel. The address of the channel to use for output is assumed to be on the top of the stack with the value to be transmitted just below it. Both of these are popped when the instruction completes.

⟨Machine opcodes 45d⟩ ≡

```
Out,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 45e⟩ ≡

```
case Out : { ⟨Communicate with waiting input; suspend process if none 54d⟩; break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 45f⟩ ≡

```
case Out : return(-2);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

For simplicity, we treat a single input on a channel the same as an `ALT` with one alternative. To process an `ALT` we first load all the alternatives on the stack using the `In` instruction. This takes as argument the instruction to jump to if the input is successful. The top of the stack is assumed to contain the address of the channel to use for the input. The instruction just places itself on the stack for use by the `Alt` instruction.

⟨Other `stack_slot` options 45g⟩ ≡

```
instruction as_inst;
```

Macro defined by scraps 34e, 45g, 54c.
Macro referenced in scrap 32c.

⟨Machine opcodes 46a⟩ ≡

```
In,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 46b⟩ ≡

```
case In  : { (--SP)->as_inst = curr_inst; break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 46c⟩ ≡

```
case In : return(1);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

The argument of the `Alt` instruction indicates how many alternatives there are. It assumes that the stack has been set up appropriately using `curr_inst.arg` input instructions. It looks for a matching partner for one of the alteratives. If it finds one then all the alternatives are popped from the stack and the index of the chosen alternative and the received value then pushed in their place. Alternatives are numbered from 0. The program counter is set to the location indicated by the successful alternative. The partner's stack is tidied up and then scheduled for execution.

⟨Machine opcodes 46d⟩ ≡

```
 Alt,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 46e⟩ ≡

```
case Alt : { ⟨Execute Alt instruction 55b⟩; break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 46f⟩ ≡

```
case Alt : return(1);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

5.4 Input/Output

The interpreter provides two instructions for the input and output of integers. The user program doesn't have direct access to these, and accesses them via the channels `stdin` and `stdout`.

The `Input` instruction reads an integer from stdin and pushes the result on the stack, or -1 if EOF. We use a non-blocking read if the input is connected to the terminal so the system will not hang while waiting for input. If there is no input currently available we reset the PC to point to this instruction again and reschedule the process.

⟨Machine opcodes 46g⟩ ≡

```
 Input,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Machine globals 47a⟩ ≡

```
char input_buffer[80];  /* Input characters are buffered here. */
int bufptr = 0;         /* A pointer into the above buffer.    */
int eof_flag = FALSE;   /* Have we exhausted the input yet?    */
```

Macro defined by scraps 47a, 49b, 51c, 53be, 64b, 65b, 67b, 70a, 72e.
Macro referenced in scrap 78c.

⟨Execute instruction `curr_inst` 47b⟩ ≡

```
case Input : {
  int i; char c;
  if (active_process->next == active_process
  &&  BP[-1].as_addr[-2].as_addr == NULL)
    return; /* Only process running and no-one waiting for input */
  if (eof_flag) { (--SP)->as_int = -1; break; }
  i = read(input_fd, &c, 1);
  if (i == 0 && input_buffered) { i++; c = '\04'; }
  if (i == 1) {
    if (c == '\04') eof_flag = TRUE;
    if (isdigit(c) || (c == '-' && bufptr == 0)) input_buffer[bufptr++] = c;
    else if (bufptr > 0) {
      input_buffer[bufptr] = 0;
      (--SP)->as_int = atoi(input_buffer); bufptr = 0; break; } }
  PC--; goto save_state;
}
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 47c⟩ ≡

```
case Input : return(1);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

The `Output` instruction pops an integer from the stack and outputs it on stdout followed be a newline.

⟨Machine opcodes 47d⟩ ≡

```
Output,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 47e⟩ ≡

```
case Output : { printf("==> %d\n", (SP++)->as_int); break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 47f⟩ ≡

```
case Output : return(-1);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

Before the interpreter starts to run a user's program, it creates the initial display and two channels for `stdin` and `stdout`. A user program should therefore be compiled as if it were prefixed with the declaration

```
CHAN stdin:
CHAN stdout:
...
```

The following sequence of statements can perform this task:

```
insert_name (display, 2, DISPLAY_T);
insert_name (chanblock,  CHANNEL_BLOCK_SIZE, CHAN_BLOCK_T);
```

```
    insert_name (find_name("stdin" ), 1, CHAN_T);
    insert_name (chanblock,  CHANNEL_BLOCK_SIZE, CHAN_BLOCK_T);
    insert_name (find_name("stdout"), 1, CHAN_T);
```

The interpreter also automatically starts up two processes to read and write the standard input and output to these channels.

5.5 Printing and Debugging

The function `print_instruction` can be used to print individual instructions on `std_out`.

⟨Machine prototypes 48a⟩ ≡

```
    extern void print_instruction(instruction i);
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

Tracing of the machine state can be switched on and off by executing the `Trace` instruction. The intention is that such instructions can be added into the program code when debugging the compiler. The `-t` command line option can be used to enable tracing for the whole program, not just for the current process.

⟨Other PCB fields 48b⟩ ≡

```
    bool trace_process;
```

Macro defined by scraps 48b, 51b, 52a, 77a.
Macro referenced in scrap 50e.

⟨Machine opcodes 48c⟩ ≡

```
    Trace,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 48d⟩ ≡

```
    case Trace:{ active_process->trace_process = curr_inst.arg; break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 48e⟩ ≡

```
    case Trace : return(0);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

5.6 Manipulating Code

It can be tedious building instruction sequences. If they are stored in arrays then we have to do lots of shuffling when building them. Some instructions contain code offsets as arguments and these have to be updated whenever we move code around. It would be simpler if we could store instructions in a linked list and refer to jump destinations via labels. This section describes such an interface. The interpreter uses the function `load_code` (see Section 5.8.5) to build an array version of the instruction sequence, replacing labels by their true offsets in the process.

Labels will just be integers. Any PC offsets refered to by instructions in `code_entry` instructions will be assumed to refer to a label rather than an actual offset.

We provide two functions to assist in the construction of code sequences. The functions `build_icode` and `build_ccode` take a variable number of arguments terminating with the distinguished argument `END`. Each argument can either be an instruction or a (pointer to a) code sequence; for build_icode the first argument must be an instruction, for build_ccode a code pointer. These functions have no way of knowing how many arguments were passed: it is therefore most important that the END is not omitted from the end of the argument list, or all sorts of nasty things are likely to occur. Similarly, if an instruction takes an argument, but this is omitted, then the function will get very confused. In other words, this function allows complex code sequences to be constructed simply, but has to be used carefully. There are two variants of the function, depending on whether the first argument is an instruction or a code sequence. See Section 5.9.1 for an example of their use.

⟨Machine prototypes 49a⟩ ≡

```
extern code build_icode(int op, ...);
extern code build_ccode(code c, ...);
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

We use a global variable called label_counter to keep track of the labels already allocated. This allows us to create fresh labels when required. We add a label to a code sequence by introducing a new pseudo-instruction, Lab, that is ignored by the interpreter. In fact the interpreter will never see such instructions as they are removed during the code loading phase.

⟨Machine globals 49b⟩ ≡

```
int label_counter = 1;
```

Macro defined by scraps 47a, 49b, 51c, 53be, 64b, 65b, 67b, 70a, 72e.
Macro referenced in scrap 78c.

⟨Machine opcodes 49c⟩ ≡

```
Lab,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Stack effect of executing instruction curr_inst 49d⟩ ≡

```
case Lab : return(0);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

To obtain a fresh label we use the new_label macro.

⟨Machine prototypes 49e⟩ ≡

```
extern int label_counter;
#define new_label() (label_counter++)
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

The following function generates the correct code sequence to remove stack items when leaving a level of scope.

⟨Machine prototypes 50a⟩ ≡

```
code exit_scope_code();
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

⟨Machine utility functions 50b⟩ ≡

```
code exit_scope_code()
{ int i = exit_scope(); code c = NULL;
  while (i != 0) {
    if (i > 0) c = build_ccode( c, Pop, i, END);
    else c = build_ccode( c, Dcb, -i / CHANNEL_BLOCK_SIZE, END);
    i = exit_scope(); }
  return(c);
}
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

We can print out a code sequence using the `print_code` function. Remember that jump destinations in such a printout refer to symbolic labels rather than actual offsets.

⟨Machine prototypes 50c⟩ ≡

```
extern void print_code(code c);
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

Finally, to execute a code sequence we use the `interpret` function.

⟨Machine prototypes 50d⟩ ≡

```
extern void interpret(code c);
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

5.7 Implementation Details

This section describes the remaining implementation details of the interpreter. Such details are probably unimportant if you just want to use the machine.

We need to switch between processes to give the impression of parallel execution. This implies that we must be able to save the state of a process in a *process control block* and restore it later. We frequently need to build queues of processes, for example the queue of processes waiting to execute, or the queue of processes waiting for input on a particular channel. Each process can only be in one queue at a time, and so we add the queue links to the PCB.

The PCB must allow us to determine the stack associated with this process. We must also be able to access the current activation record of the parent process (if any) so that we can look up non-local variables. When a PCB is allocated (as part of creating a new process) we allocate sufficient memory to hold the process block plus the stack. In other words, the stack for this process is assumed to start immediately after the block finishes. The last field of the PCB, the `stack_overflow_check`, is used to detect stack overflows (see later).

⟨Process block `typedef` 50e⟩ ≡

```
typedef struct process_block
{ int PC;
  union stack_slot *BP, *SP;
  int stack_size;
  ⟨Other PCB fields 48b, ... ⟩
  struct process_block *next, *parent;
  int stack_overflow_check;
} process_block, *process_queue, *process, *processes;
```

Macro referenced in scrap 77e.

A process can be in one of a number of states. When `ACTIVE` the process is either currently running or waiting to run. If it is blocked waiting for a communication from another process then it is in either the SUSPENDED_IN or SUSPENDED_OUT states. A process that is waiting for its children to terminate before continuing execution is in the WAITING state. When a process terminates we cannot immediately recycle its storage as it may have spawned some children who have access to its stack. In such a case we must keep the process until its children have terminated. Processes in this position are in the RETIRED state. Finally, retired processes with no living children are put into the REINCARNATED state where they wait until the system needs to create another process requiring the same size of stack.

⟨Process status typedef 51a⟩ ≡

```
typedef enum process_status {
  ACTIVE,
  SUSPENDED_IN,
  SUSPENDED_OUT,
  WAITING,
  RETIRED,
  REINCARNATED
} process_status;
```

Macro referenced in scrap 77e.

The process state is stored in the PCB.

⟨Other PCB fields 51b⟩ ≡

```
 process_status status;
```

Macro defined by scraps 48b, 51b, 52a, 77a.
Macro referenced in scrap 50e.

The processes currently awaiting execution are stored in a circularly-linked list. This allows the next process to be scheduled in a round-robin fashion by just following the next pointers. We use the interpreter variable active_process to point to this queue, where the process pointed at is the currently running process. We need to be able to remove the active process from this queue when it terminates or is suspended. To enable this to be done efficiently we also keep track of the process that points to the active process so that the queue can be respliced when the process is removed.

⟨Machine globals 51c⟩ ≡

```
process_queue active_process = NULL;
process_queue prev_process = NULL;
```

Macro defined by scraps 47a, 49b, 51c, 53be, 64b, 65b, 67b, 70a, 72e.
Macro referenced in scrap 78c.

The prev_process pointer does not necessarily point to the process that last ran. Processes that become scheduled for execution get added between the prev_process and active_process pointers, and the prev_process pointer updated, as illustrated in Figure 5.4.

To create a process we need to know the stack size and the starting value of the PC. We allocate the appropriate storage, initialise the fields and return the result. The stack_overflow_check field is initialised to a distinctive bit pattern. We check that the field still has this bit pattern whenever an instruction is about to be executed.

⟨Machine prototypes 51d⟩ ≡

```
extern process create_process(int PC, int stacksize);
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

⟨Machine utility functions 51e⟩ ≡

```
#define OVERFLOW_CHECKSUM 0x55555555
```

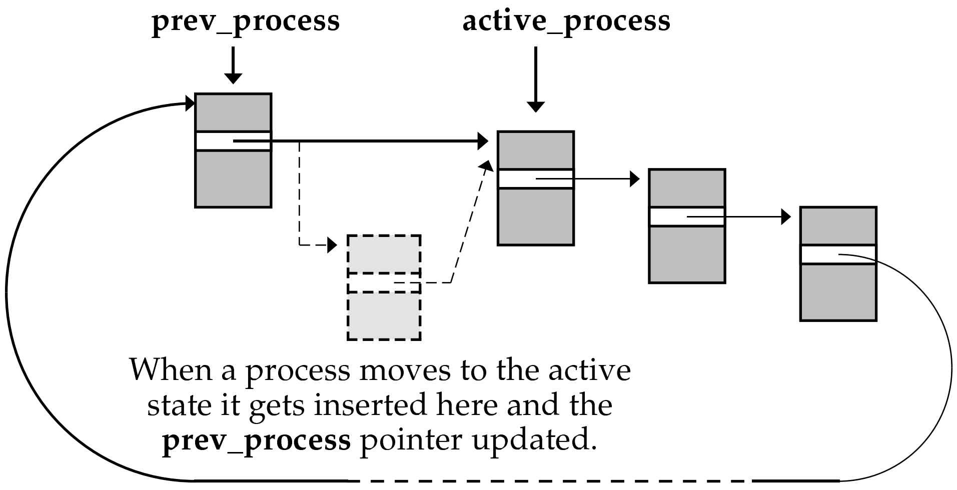
prev_process
active_process
When a process moves to the active state it gets inserted here and the prev_process pointer updated.

Figure 5.4: The active process queue

```
process create_process(int PC, int stacksize)
{ process p;
  ⟨Allocate storage for process p with a stack of size stacksize 53f⟩
  p->PC = PC;
  p->next = NULL;
  p->children = 0;
  p->parent = active_process;
  if (active_process) active_process->children++;
  p->stack_size = stacksize;
  p->status = ACTIVE;
  p->SP = (stack_slot *)((char *)p + sizeof(process_block)) + stacksize;
  p->BP = NULL;
  p->stack_overflow_check = OVERFLOW_CHECKSUM;
  p->step_process = p->trace_process = FALSE;
  return p;
}
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

When a process dies by executing a `Stop` instruction we must decide whether to retire it or reincarnate it straight away. This requires knowing how many spawned children are still alive. We therefore add a `children` field to the PCB to keep track of this.

⟨Other PCB fields 52a⟩ ≡

```
    int children;
```

Macro defined by scraps 48b, 51b, 52a, 77a.
Macro referenced in scrap 50e.

⟨Terminate current process 52b⟩ ≡

```
{ process p = active_process;
  ⟨Remove p from active process queue 53a⟩
  if (p->children) { /* Retire as there are still children alive */
    p->status = RETIRED; }
  else ⟨Reincarnate process p 53c⟩
}
```

Macro referenced in scraps 44f, 55b, 56a.

When we remove a process from the active process queue we must be careful to deal with the cases where there is only one process.

⟨Remove p from active process queue 53a⟩ ≡

```
if (p == prev_process) active_process = prev_process = NULL;
else prev_process->next = active_process = p->next;
```

Macro referenced in scraps 52b, 72a.

Reincarnated processes are chained together in a queue accessed via the `reincarnated_processes` variable.

⟨Machine globals 53b⟩ ≡

```
process_queue reincarnated_processes = NULL;
```

Macro defined by scraps 47a, 49b, 51c, 53be, 64b, 65b, 67b, 70a, 72e.
Macro referenced in scrap 78c.

To reincarnate a process we must add it to this queue. However, we must also decrement the `children` count of the parent if there is one. If this count goes to zero, and the parent is retired, then it should be reincarnated as well. Thus reincarnating one process may trigger a whole bunch of ancestors being reincarnated as well. If the parent is in the `WAITING` state when its count reaches 0 then we schedule it for execution again.

⟨Reincarnate process p 53c⟩ ≡

```
do {
  ⟨Add process p to reincarnated list 53d⟩
  p->status = REINCARNATED;
  if (!p->parent) break;
  p = p->parent; }
while ((!--p->children) && (p->status == RETIRED));

if ((p->status == WAITING) && (!p->children)) {
  ⟨Schedule process p for execution 71c⟩ }
```

Macro referenced in scrap 52b.

We add the reincarnated process to the head of the reincarnated list. It would be more efficient to keep this list sorted by process size (i.e. stack size), but it is not worth worrying about for an interpreter.

⟨Add process p to reincarnated list 53d⟩ ≡

```
p->next = reincarnated_processes; reincarnated_processes = p;
```

Macro referenced in scrap 53c.

When we allocate storage for a process we first check to see if there is a reincarnated process of the appropriate size. If not, or the reincarnated list is empty, then we must use `malloc` to allocate some storage. For diagnostic purposes we keep track of the total amount of storage allocated using the `memory_used` variable.

⟨Machine globals 53e⟩ ≡

```
int memory_used = 0;
```

Macro defined by scraps 47a, 49b, 51c, 53be, 64b, 65b, 67b, 70a, 72e.
Macro referenced in scrap 78c.

⟨Allocate storage for process p with a stack of size stacksize 53f⟩ ≡

```
{ process q = NULL;
  p = reincarnated_processes;
  while ((p != NULL) && (p->stack_size != stacksize)) { q = p; p = p->next; }

  if (p) /* Unlink it from the reincarnated_processes queue */
```

```
        if (p == reincarnated_processes) reincarnated_processes = p->next;
        else q->next = p->next;

      else { /* No suitable reincarnated process */
        int size = sizeof(struct process_block) + stacksize*sizeof(stack_slot);
        p = malloc(size); memory_used += size; }
    }
```

Macro referenced in scrap 51e.

Here is a simple function that gives us some statistics on process memory usage.

⟨Machine prototypes 54a⟩ ≡

```
    extern void process_statistics();
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

⟨Machine utility functions 54b⟩ ≡

```
    void process_statistics()
    { int free_space = 0; process p = reincarnated_processes;
      while (p != NULL)
        { free_space += sizeof(struct process_block) + p->stack_size*sizeof(stack_slot);
          p = p->next; }
      printf("Total process space allocated = %d, of which %d still in use at exit.\n",
             memory_used, memory_used - free_space);
    }
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

5.7.1 Establishing a Communication

Each channel control block will either contain `NULL` if there is no process waiting, or will contain the pointer to a waiting process. We talk about channel queues, but the language is defined so that there can be at most one process waiting on each channel. Thus it is a bounded queue, with the bound equal to one.

To output on a channel we must first check whether there is a waiting process. If there is then we must remove the partner from any other channels it might be waiting on. We then transmit the output value by pushing it onto the partner's stack. Finally we resume both processes. If there is no matching partner at this point in time we add the process to the channel queue and suspend.

⟨Other `stack_slot` options 54c⟩ ≡

```
     process as_process;
```

Macro defined by scraps 34e, 45g, 54c.
Macro referenced in scrap 32c.

⟨Communicate with waiting input; suspend process if none 54d⟩ ≡

```
    { stack_slot *the_chan = SP->as_addr;
      if (the_chan->as_process != NULL) { /* We have a match */
        ⟨Establish output communication on channel the_chan 56c⟩
        goto save_state; /* Save state and relinquish control (for fairness ) */ }
      else ⟨Suspend ouput process until communication partner 55a⟩
    }
```

Macro referenced in scrap 45e.

⟨Suspend ouput process until communication partner 55a⟩ ≡

```
{ process p = active_process;
  the_chan->as_process = p;    /* Add the process to the channel queue */
  p->status = SUSPENDED_OUT;
  ⟨Suspend process p 72a⟩
}
```

Macro referenced in scrap 54d.

Consider the state of the stack when an `Alt` i instruction is encountered. Assuming the compiler hasn't goofed, there should be i groups of input channel requests stacked up for us, each one consisting of a channel address followed by the corresponding `In` instruction.

The first step is to traverse these entries looking for a matching partner. If we find a non-empty channel then we establish a communication. Otherwise we must suspend until another process wishes to communicate with us.

⟨Execute `Alt` instruction 55b⟩ ≡

```
{ stack_slot *the_chan = NULL;
  int index;

  if (curr_inst.arg == 0) {
    ⟨Terminate current process 52b⟩
    ⟨Schedule next process or return if none 72c⟩ }

  ⟨Look for partner; if found place matching channel in the_chan and set PC & index 55c⟩

  if (the_chan == NULL) /* We must wait */
    ⟨Suspend input process until communication partner 56a⟩
  else {
    SP += 2*curr_inst.arg; /* Pop alternatives off the stack */
    ⟨Establish input communication on channel the_chan 56b⟩
    goto save_state; /* Save state and relinquish control (for fairness ) */ }
}
```

Macro referenced in scrap 46e.

To find a match we just look through the stack entries that have been set up for the `Alt`. If we find one whose channel has a waiting process we save the channel in the variable `the_chan`, set up the new PC and break. We don't check whether any partner we find is of the opposite polarity. It should be in a valid μ`OCCAM`program, but we check later anyway. We check the alternatives in a semi-random order to make things interesting.

⟨Look for partner; if found place matching channel in `the_chan` and set PC & `index` 55c⟩ ≡

```
{ int i = curr_inst.arg-1;
  stack_slot *sp, *a;
  if (counter%2)
    for (sp = SP;       i>=0; i--, sp += 2) {
      if ((a = (sp+1)->as_addr) && a->as_process) { /* We have a match */
        PC = sp->as_inst.arg; the_chan = a; index = i; break; } }
  else
    for (sp = SP + 2*i; i>=0; i--, sp -= 2) {
      if ((a = (sp+1)->as_addr) && a->as_process) { /* We have a match */
        PC = sp->as_inst.arg; the_chan = a;
        index = curr_inst.arg-i-1; break; } }
}
```

Macro referenced in scrap 55b.

When we suspend an input process we must put the address of the process in all the channels mentioned in the alternative. We save the number of alternatives on the top of the stack so our eventual partner will know how many stack frames to pop. If the same channel is mentioned in more than one branch of the alternative only one of them will be queued (the choice being made randomly).

⟨Suspend input process until communication partner 56a⟩ ≡

```
{ process p = active_process;
  stack_slot *a, *sp;
  int i = curr_inst.arg-1;
  int alts = 0;
  if (counter%2)
    for (sp = SP;     i>=0; i--, sp += 2) {
      if (a = (sp+1)->as_addr) { a->as_process = p; alts++; } }
  else
    for (sp = SP+2*i; i>=0; i--, sp -= 2) {
      if (a = (sp+1)->as_addr) { a->as_process = p; alts++; } }
  if (alts) {
    (--SP)->as_int = curr_inst.arg; /* Remember the alt count */
    p->status = SUSPENDED_IN;
    ⟨Suspend process p 72a⟩ }
  else {
    ⟨Terminate current process 52b⟩
    ⟨Schedule next process or return if none 72c⟩ }
}
```

Macro referenced in scrap 55b.

To establish a communication we must either transfer a value from the current process to the partner, or vice versa, put the partner into the active state, and fix up the stacks. The μOCCAM semantics states that the name of a channel may only be used in one component of a parallel for input, and in one other component of the parallel for output. This should ensure that there can be at most one process waiting on a channel at any one time. Furthermore, if another process wishes to communicate on the channel then they must be of opposite polarity, i.e. they can communicate. However, it is difficult for the compiler to always check this statically and so we check at run-time as well. After all, interpreters aren't supposed to be efficient anyway ...

⟨Establish input communication on channel the_chan 56b⟩ ≡

```
{ process p = the_chan->as_process; /* The partner */
  stack_slot *PSP = p->SP;  /* The partner's stack pointer   */

  ⟨Remove output process p from the queue for the_chan 57b⟩
  *(--SP) = *(++PSP); PSP++; /* Copy across value */
  (--SP)->as_int = index;
  p->SP = PSP;
  ⟨Schedule process p for execution 71c⟩
}
```

Macro referenced in scrap 55b.

⟨Establish output communication on channel the_chan 56c⟩ ≡

```
{ process p = the_chan->as_process; /* The partner */
  stack_slot *PSP = p->SP;  /* The partner's stack pointer   */
  int index;

  ⟨Remove input process p from all its channel queues and pop PSP; set index 57a⟩
  *(--PSP) = *(++SP);  SP++; /* Copy across value */
  (--PSP)->as_int = index;
  p->SP = PSP;
  ⟨Schedule process p for execution 71c⟩
}
```

Macro referenced in scrap 54d.

When we remove a process from any queues we check its polarity to detect cases of two inputs on a channel, or two outputs.

⟨Remove input process `p` from all its channel queues and pop PSP; set `index` 57a⟩ ≡

```
{ int i;
  if (p->status != SUSPENDED_IN)
    error("Interpreter error",
          "Two processes trying to output on the same channel.\n");

  for (i = (PSP++)->as_int; i>0; i--) {
    stack_slot *pchan = (PSP+1)->as_addr;
    if (pchan == the_chan) { p->PC = PSP->as_inst.arg; index = i-1; }
    if (pchan) pchan->as_process = NULL;
    PSP += 2;
  }
}
```

Macro referenced in scrap 56c.

⟨Remove output process `p` from the queue for `the_chan` 57b⟩ ≡

```
if (p->status != SUSPENDED_OUT)
  error("Interpreter error",
        "Two processes trying to input on the same channel.\n");
the_chan->as_process = NULL;
```

Macro referenced in scrap 56b.

5.8 Code Manipulation Functions

5.8.1 Code construction

We start by defining the support code. First, the type declarations.

⟨Machine `typedef`s 57c⟩ ≡

```
typedef struct code_entry
{ instruction inst;
  struct code_entry *next;
} code_entry;
```

Macro defined by scraps 32ac, 57cd, 72d.
Macro referenced in scrap 77e.

⟨Machine `typedef`s 57d⟩ ≡

```
typedef struct instruction_sequence
{ code_entry *first, *last; } *code;
```

Macro defined by scraps 32ac, 57cd, 72d.
Macro referenced in scrap 77e.

Now the functions themselves. We factor out the common code into the function `code_args`. Note that the function needs to know which instructions take arguments. This is an obvious source of error – it might be better if this information was provided as each instruction was introduced.

⟨Machine prototypes 57e⟩ ≡

```
extern code code_args(int op, va_list args);
#define END -1
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

⟨Machine utility functions 58a⟩ ≡

```
code code_args(int op, va_list args)
{ code c, r; code_entry *n; int arg = 0; int next_op;
  if (op == END) return NULL;

  if ((void *)op == NULL) { /* Assume we have found null code pointer */
    next_op = va_arg(args,int); c = code_args(next_op, args); }

  else if (op > 255) { /* Assume we have found some code */
    c = (code) op;
    next_op = va_arg(args,int);
    if (r = code_args(next_op, args)) {
      c->last->next = r->first; c->last = r->last; dispose(r); } }

  else {
    switch (op) {
      case Stop: case Wait: case Add : case Sub : case Mul : case Div : case Mod :
      case Lt  : case Leq : case Gt  : case Geq : case Eq  : case Neq :
      case Idx : case Ida : case Sta : case Ret : case Out : case Swap:
      case Input:case Output:
        { next_op = va_arg(args,int); break; }

      default: { arg = va_arg(args,int); next_op = va_arg(args,int); break; }
    }

    n = new(struct code_entry); n->inst.op = op; n->inst.arg = arg;

    if (c = code_args(next_op, args)) {
      n->next = c->first; c->first = n; }
    else {
      c = new(struct instruction_sequence);
      c->first = c->last = n; n->next = NULL; } }
  return c;
}
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

⟨Machine utility functions 58b⟩ ≡

```
code build_icode(int op, ...)
{ code c; va_list args;
  va_start(args,op); c = code_args(op, args); va_end(args); return c;
}

code build_ccode(code c, ...)
{ code c1; va_list args;
  va_start(args,c); c1 = code_args((int) c, args); va_end(args); return c1;
}
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

Next, a utility function for disposing of code sequences.

⟨Machine prototypes 58c⟩ ≡
```
extern void dispose_code(code c);
```
Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

⟨Machine utility functions 59a⟩ ≡

```
void dispose_code(code c)
{ if (c) {
    code_entry *e = c->first;
    while (e) { code_entry *n = e->next; dispose(e); e = n; }
    dispose(c); }
}
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

5.8.2 Constant expressions

There are various places in the μOCCAM language where a constant expression is required. To simplify the task of checking for such an expression, and to calculate the value returned by such an expression, the following function can be used. It takes a code sequence corresponding to an expression and the address of an integer as arguments. If the effect of evaluating the code sequence can be computed at compile time then the function returns `TRUE` and the value of the expression is passed back in the integer argument. Otherwise `FALSE` is returned. The function may return `FALSE` even when an expression does denote a constant. An obvious example would be an expression involving array accesses. However, no function is going to be perfect in this regard, and the function provided is adequate for our purposes.

⟨Machine prototypes 59b⟩ ≡

```
extern int constant_expression(code c, int *v);
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

⟨Machine utility functions 59c⟩ ≡

```
int constant_expression(code c, int *v)
{ code_entry *e = c ? c->first : NULL;
  int S[20]; int sp = -1; instruction i;

  while(e) {
    i = e->inst; e = e->next;
    switch (i.op) {
      case Ldc: S[++sp] = i.arg; continue;
      case Add: if (sp > 0) S[sp-1] += S[sp]; break;
      case Sub: if (sp > 0) S[sp-1] -= S[sp]; break;
      case Mul: if (sp > 0) S[sp-1] *= S[sp]; break;
      case Div: if (sp > 0) S[sp-1] /= S[sp]; break;
      case Mod: if (sp > 0) S[sp-1] %= S[sp]; break;
      case Lt : if (sp > 0) S[sp-1] =  S[sp-1] <  S[sp]; break;
      case Leq: if (sp > 0) S[sp-1] =  S[sp-1] <= S[sp]; break;
      case Gt : if (sp > 0) S[sp-1] =  S[sp-1] >  S[sp]; break;
      case Geq: if (sp > 0) S[sp-1] =  S[sp-1] >= S[sp]; break;
      case Eq : if (sp > 0) S[sp-1] =  S[sp-1] == S[sp]; break;
      case Neq: if (sp > 0) S[sp-1] =  S[sp-1] != S[sp]; break;
      default: return(FALSE); }
    sp--;
  }
  if (sp == 0) { *v = S[0]; return(TRUE); }
  return(FALSE);
}
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

5.8.3 Printing code

Printing a code sequence is easy. We just traverse the list calling `print_instruction` at each node.

⟨Machine utility functions 60a⟩ ≡

```
void print_code(code c)
{ code_entry *e = c ? c->first : NULL;
  while (e != NULL) {
    if (e->inst.op == Lab) {
      printf("%3d : ", e->inst.arg);
      e = e->next;
      if (e->inst.op == Lab) { printf("\n"); continue; } }
    else printf("      ");
    print_instruction(e->inst); printf("\n");
    e = e->next; }
}
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

Printing an instruction is even more tedious; it's just a big switch.

⟨Machine utility functions 60b⟩ ≡

```
void print_instruction(instruction i)
{ switch (i.op)
  { case Stop: { printf("Stop     ");break; }
    case Wait: { printf("Wait     ");break; }
    case Add : { printf("Add      "); break; }
    case Sub : { printf("Sub      "); break; }
    case Mul : { printf("Mul      "); break; }
    case Div : { printf("Div      "); break; }
    case Mod : { printf("Mod      "); break; }
    case Lt  : { printf("Lt       "); break; }
    case Leq : { printf("Leq      "); break; }
    case Gt  : { printf("Gt       "); break; }
    case Geq : { printf("Geq      "); break; }
    case Eq  : { printf("Eq       "); break; }
    case Neq : { printf("Neq      "); break; }
    case Idx : { printf("Idx      "); break; }
    case Ida : { printf("Ida      "); break; }
    case Sta : { printf("Sta      "); break; }
    case Ret : { printf("Ret      "); break; }
    case Out : { printf("Out      "); break; }
    case Swap: { printf("Swap     "); break; }
    case Input:{ printf("Input    "); break; }
    case Output:{printf("Output   "); break; }
    case In  : { printf("In %6d",  i.arg); break; }
    case Ind : { printf("Ind %5d", i.arg); break; }
    case Ldc : { printf("Ldc %5d", i.arg); break; }
    case Ldo : { printf("Ldo %5d", i.arg); break; }
    case Lda : { printf("Lda %5d", i.arg); break; }
    case Ldn : { printf("Ldn %5d", i.arg); break; }
    case Sto : { printf("Sto %5d", i.arg); break; }
    case Pop : { printf("Pop %5d", i.arg); break; }
    case Ujp : { printf("Ujp %5d", i.arg); break; }
    case Fjp : { printf("Fjp %5d", i.arg); break; }
    case Alt : { printf("Alt %5d", i.arg); break; }
    case Ccb : { printf("Ccb %5d", i.arg); break; }
    case Dcb : { printf("Dcb %5d", i.arg); break; }
    case Pstk: { printf("Pstk %4d",i.arg); break; }
```

```
        case Call: { printf("Call %4d",i.arg); break; }
        case Spawn:{ printf("Spawn %3d",i.arg);break; }
        case Trace:{ printf("Trace %3d",i.arg);break; }
        case Enter:{ printf("Enter %3d",i.arg);break; }
        case Leave:{ printf("Leave %3d",i.arg);break; }
        case Spawnv:{printf("Spawnv %2d",i.arg); break; }
        default:   { printf("?%d",i.op);         break; }
      }
    }
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

5.8.4 Stack Utilisation

For maximum efficiency and flexibility, we should really calculate the stack movements as we construct the program code. However, this can be error prone, and so we provide a simpler alternative. The function `compute_max_stack` takes a code sequence as argument and computes the maximum amount of stack space it requires. To do this, without knowing anything about the code sequence, is obviously tricky. We therefore make a simplifying assumption. If the code sequence has a loop in it then the size of the stack at the end of the loop must be the same as at the start. This rules out code that loops n times pushing a value onto the stack each time around the loop. This is an inconvenience, but not a major problem. For example, we can allocate n elements on the stack before starting the loop , using the `Ldn` instruction. We can then update each element of this block in turn in a loop. The stack no longer grows each time around the loop and the `compute_max_stack` function is then happy. Probably the only place in the compiler where this is an issue is in the replicated `ALT` construct.

Before presenting the code for the `compute_max_stack` we need a bit of infrastructure. We obviously need to take into account the stack requirements of any procedure calls in the code sequence. However, the code for the procedure body may not be contained within the code sequence. We therefore assume that a `Pstk` instruction is placed immediately after the `Call` instruction. The argument to this instruction is the amount of stack space required by the procedure body. The instruction is ignored by the interpreter (in fact it is removed when converting the sequence to an array of instructions).

⟨Machine opcodes 61a⟩ ≡

```
    Pstk
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

The following function computes the effect on the size of the stack of executing a single instruction. Most of the effects were defined at the point each instruction was introduced.

⟨Machine prototypes 61b⟩ ≡

```
    extern int stack_effect(instruction curr_inst);
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

⟨Machine utility functions 61c⟩ ≡

```
  int stack_effect(instruction curr_inst)
  { switch (curr_inst.op) {
      ⟨Stack effect of executing instruction curr_inst 32g, ... ⟩
      case Pstk: return(0); /* Will be handled specially in compute_max_stack */
      default:
        print_instruction(curr_inst); printf("\n");
        error("Fatal error", "Missing case in stack_effect\n");
    }
  }
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

As mentioned above, we need to check that the stack pointer at the beginning and end of a loop is the same. While traversing the code we keep track of any labels encountered along the path from the start of the code sequence to the current position. With each label we store the associated value of the stack pointer. Because we are essentially performing a depth-first traversal a stack is suitable for this task. We should really check for stack overflow here, but it's not worth it on such a simple practical.

⟨Machine prototypes 62a⟩ ≡

```
struct {
  int label;
  int SP;
} path_stack[50];
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

We make two traversals of the code. The first builds a simple table associating labels with their associated code pointers. This makes it easier to follow jumps in the second pass. For simplicity we represent the table as a linked list; there is obvious scope for improvement here...

⟨Machine prototypes 62b⟩ ≡

```
typedef struct jump_entry
{ int label; code_entry *dest;
  struct jump_entry *next;
} jump_entry, *jump_table;
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

We build a jump table by making a linear traversal of the code.

⟨Machine prototypes 62c⟩ ≡

```
extern jump_table build_jump_table(code c);
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

⟨Machine utility functions 62d⟩ ≡

```
jump_table build_jump_table(code c)
{ code_entry *p = c ? c->first : NULL;
  jump_table jt = NULL;
  while (p != NULL) {
    if (p->inst.op == Lab) { /* Add the entry (p->inst.arg, p) to jump table */
      jump_table l; l = new(jump_entry);
      l->next = jt; l->label = p->inst.arg; l->dest = p; jt = l; }
    p = p->next;
  }
  return(jt);
}
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

To be neat and tidy, we also provide a function to dispose of these tables.

⟨Machine prototypes 62e⟩ ≡

```
extern void dispose_jump_table(jump_table jt);
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

⟨Machine utility functions 63a⟩ ≡

```
void dispose_jump_table(jump_table jt)
{ jump_table n; while (jt != NULL) { n = jt->next; dispose(jt); jt = n; } }
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

Looking up a label is simple giving our dubious choice of data structure.

⟨Machine prototypes 63b⟩ ≡

```
extern code_entry *lookup_jt(int label, jump_table jt);
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

⟨Machine utility functions 63c⟩ ≡

```
code_entry *lookup_jt(int label, jump_table jt)
{ while (jt != NULL && jt->label != label) jt = jt->next;
  if (jt == NULL)
    error("Fatal error", "Jumping to a non-existent label\n");
  return(jt->dest);
}
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

Now for the second phase of the function. We use an auxiliary function, `max_stack`, to perform this task. It takes as argument the code sequence to analyse, a jump table, the path to the starting point and the initial value of the stack pointer. These last two arguments are used when the function calls itself recursively. We also pass the starting point to the whole program fragment to aid debugging, but this is not strictly necessary.

⟨Machine utility functions 63d⟩ ≡

```
int max_stack(code c, code_entry *e,
              int initial_SP, int path_SP, jump_table jt)
{ instruction curr;
  int SP = initial_SP, MAX_SP = initial_SP, i;

  while (e != NULL) {
    curr = e->inst; e = e->next;
    i = stack_effect(curr);

    if (i < 0 && SP < i) {
      print_code(c);
      error("Fatal error", "Stack underflow in function max_stack\n"); }

    SP += i; MAX_SP = (SP > MAX_SP) ? SP : MAX_SP;

    switch (curr.op)
    { case Lab: ⟨Compute stack effect of label 64a⟩

      case Ujp : e = lookup_jt(curr.arg, jt); break;

      case Pstk: { /* Make sure there is enough space for procedure call */
        int newSP = SP + curr.arg;
        MAX_SP = (newSP > MAX_SP) ? newSP : MAX_SP; break; }

      case Stop :
      case Ret  : return(MAX_SP);
```

```
          case Fjp : ⟨Compute stack effect of Fjp 66a⟩
          case In  : ⟨Compute stack effect of In 64c⟩
          case Alt : ⟨Compute stack effect of Alt 65a⟩
        }
      }
      return(MAX_SP);
    }
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

Processing a label is simple, given the restrictions we have placed on the input code to the function. We just have to make sure we don't keep looping for ever...

⟨Compute stack effect of label 64a⟩ ≡

```
    { /* First check to see if we have already seen this label on current path */
      int i = 0;
      while (path_stack[i].label != curr.arg && i < path_SP) i++;

      if (i == path_SP) { /* i.e. label not seen so update path stack */
        path_stack[path_SP].label = curr.arg;
        path_stack[path_SP++].SP = SP;
        break; }

      else if (path_stack[i].SP != SP) { /* Loop encountered with changing SP */
        print_code(c);
        error("Fatal error", "Loop encountered with stack %s in "
              "function max_stack.\n",
              (SP > path_stack[i].SP) ? "increasing" : "decreasing"); }
      else
        /* We have found a well behaved loop. Stop this particular branch of the
           traversal and return the maximum value of SP encountered. */
        return(MAX_SP);
    }
```

Macro referenced in scrap 63d.

Now comes the tricky bit. The complication arises due to the `In` and `Alt` instructions. These act like conditional jumps. When processing an occurrence of an `In` instruction we should record, in a list, the destination of the instruction. When the corresponding `Alt` instruction is encountered we should adjust the stack pointer to the value it would have immediately after the `Alt` instruction has finished. We must then recursively call the function on all the code sequences queued up as a result of processing the `In` instructions, and find out which one of these has the biggest stack requirements. We use a global variable, `in_jt`, to store the jump table constructed by the `In` instructions. The structure of the language insures that such constructs cannot be nested, and so this is safe.

⟨Machine globals 64b⟩ ≡

```
    jump_table in_jt = NULL;
```

Macro defined by scraps 47a, 49b, 51c, 53be, 64b, 65b, 67b, 70a, 72e.
Macro referenced in scrap 78c.

⟨Compute stack effect of In 64c⟩ ≡

```
    { jump_table l; l = new(jump_entry);
      l->next = in_jt; l->dest = lookup_jt(curr.arg, jt); in_jt = l; break;
    }
```

Macro referenced in scrap 63d.

⟨Compute stack effect of Alt 65a⟩ ≡

```
{ jump_table bodies = in_jt;
  if ((SP -= (2*curr.arg - 1)) < 1) {
    print_code(c);
    error("Fatal error", "Stack underflow in function max_stack\n"); }

  if (in_fjp) return(MAX_SP); /* see following paragraphs for explanation */

  in_jt = NULL;
  while (bodies != NULL) {
    int fj_max; jump_table n;
    fj_max = max_stack(c,bodies->dest, SP, path_SP, jt);
    MAX_SP = (fj_max > MAX_SP) ? fj_max : MAX_SP;
    n = bodies->next; dispose(bodies); bodies = n;
  }
  return(MAX_SP);
}
```

Macro referenced in scrap 63d.

The only instruction we haven't considered yet is `Fjp`. We treat this like the alternatives above, i.e. we try each alternative, and return the maximum stack required. However, in this case, the order in which we try the alternatives is important. Consider how we might compile a replicated `ALT`. The code will look something like

```
L: ...

   Fjp M

   ...

   In N

   ...

   Ujp L

M: ...

   Alt n
```

If we process the jump to `M` *before* processing the `In` instruction then `in_jt` will still be `NULL` when we reach the `Alt` instruction. In this case it seems better to process the jump *after* we have processed the code containing the `In` instruction. However, the problem is more subtle than this. Suppose one of the alternatives is guarded, and the guard expression generates code involving `Fjp`. A conjunction of expressions would be an obvious example. This would generate two traversals leading to the `Alt` instruction. The first would result in `in_jt` being cleared, and the second traversal would therefore produce wrong results. If we make the assumption that the stack size will be identical when the `Alt` instruction is reached, no matter which path is taken, then we can fix the problem by making sure that the first traversal just returns when the `Alt` instruction is reached, leaving `in_jt` unchanged. We introduce a counter, `in_fjp`, initially 0.

⟨Machine globals 65b⟩ ≡

```
int in_fjp = 0;
```

Macro defined by scraps 47a, 49b, 51c, 53be, 64b, 65b, 67b, 70a, 72e.
Macro referenced in scrap 78c.

Suppose we are in the middle of processing some inputs, i.e. `in_jt` is not `NULL`, and we encounter a conditional jump. We then have two paths leading to the `Alt`. We increment `in_jt` while we process one of them, and decrement it before processing the other. The `Alt` instruction then only processes the contents of `in_jt` when the count is equal to zero. This ensures that we only process the contents of `in_jt` once. This is a bit of a hack, but it's probably the best I can do given the restrictions a function like `compute_max_stack` is working under.

⟨Compute stack effect of Fjp 66a⟩ ≡

```
{ int fj_max; jump_table old_in_jt = in_jt;
  if (old_in_jt) in_fjp++;
  fj_max = max_stack(c, e, SP, path_SP, jt); /* Process code after jump */
  if (old_in_jt) in_fjp--;
  MAX_SP = (fj_max > MAX_SP) ? fj_max : MAX_SP;
  e = lookup_jt(curr.arg, jt); /* Process the jump */
  break;
}
```

Macro referenced in scrap 63d.

This whole argument sounds plausible, but I'd like a slightly more systematic argument about its correctness, and a more rigorous description of the assumptions being made about the input code for the correctness to hold – any suggestions?

The main procedure to compute stack sizes is now simple to state.

⟨Machine prototypes 66b⟩ ≡

```
extern int compute_max_stack(code c);
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

⟨Machine utility functions 66c⟩ ≡

```
int compute_max_stack(code c)
{ int m; jump_table jt;
  jt = build_jump_table(c);
  m = max_stack(c, c ? c->first : NULL, 0, 0, jt);
  dispose_jump_table(jt);
  return(m);
}
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

5.8.5 The Loader

The compiler builds the abstract machine code program in the form of a sequence with symbolic labels. However, the interpreter requires the instructions to be in an array with all labels resolved to their associated locations. The `load_code` function performs this transformation.

⟨Machine prototypes 66d⟩ ≡

```
extern instruction *load_code(code c);
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

The function needs to keep track of the association between labels and their locations in the code array. We use the `label_entry` structure to build a simple list of such associations. We don't attempt to sort it as there shouldn't be a vast number of labels in the programs we are likely to encounter. We traverse the code list twice. The first time we build the table and count how many instructions there are. We then allocate storage for the instruction array and then traverse the code again stuffing the instructions into the array and resolving jump destinations. We dispose the storage for the code sequence along the way, so the argument `c` shouldn't be used after the call without assigning something to it first.

⟨Machine utility functions 67a⟩ ≡

```
instruction *load_code(code c)
{ typedef struct label_entry
  { int label, code_location;
    struct label_entry *next;
  } label_entry, *label_table;

  int n;
  label_table labels = NULL;
  instruction *instruction_array;
  code_entry *s = c ? c->first : NULL;
  ⟨Compute n, the length of code sequence s, building label table labels in the process 67c⟩
  instruction_array = malloc(n * sizeof(instruction));
  ⟨Pack code sequence s into instruction_array using labels table 68a⟩
  ⟨Dispose of labels table 68b⟩
  dispose(c);
  ⟨Create and initialise breakpoint array of size n 72f⟩
  code_size = n;
  return(instruction_array);
}
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

⟨Machine globals 67b⟩ ≡

```
int code_size = 0;
```

Macro defined by scraps 47a, 49b, 51c, 53be, 64b, 65b, 67b, 70a, 72e.
Macro referenced in scrap 78c.

When we encounter a label we don't bother checking whether an instruction with this label has already been encountered. This shouldn't happen with properly constructed code anyway.

⟨Compute `n`, the length of code sequence `s`, building label table `labels` in the process 67c⟩ ≡

```
{ code_entry *p = s;
  n = 0;
  while (p != NULL) {
    if (p->inst.op == Lab) { /* Add the entry (p->inst.arg, n) to labels table */
      label_table l; l = new(label_entry);
      l->next = labels; l->label = p->inst.arg; l->code_location = n; labels = l; n--; }
    if (p->inst.op == Pstk) n--;
    n++; p = p->next;
  }
}
```

Macro referenced in scrap 67a.

When packing instructions into the array we must look out for instructions whose arguments are jump destinations. When we encounter one of these we use the `labels` table to replace the label by its actual destination in the array. If we can't find the label we report a fatal error – someone has messed up the code generation... A typical cause of this is to forget to concatenate all the bits of code together before calling the interpreter.

⟨Pack code sequence `s` into `instruction_array` using `labels` table 68a⟩ ≡

```
{ int i = 0;
  while (i<n) {
    code_entry *p = s;
    if (s->inst.op == Lab || s->inst.op == Pstk)
      { s = s->next; dispose(p); p = s; continue; }
    instruction_array[i++] = s->inst;
    if ((s->inst.op == Ujp)   /* Check to see if the argument */
    ||  (s->inst.op == Fjp)   /* is a reference to a label    */
    ||  (s->inst.op == Spawn) /* and replace it if it is.     */
    ||  (s->inst.op == Spawnv)
    ||  (s->inst.op == Call)
    ||  (s->inst.op == In)) {
      int label = s->inst.arg;
      label_table p = labels;
      while ((p != NULL) && (p->label != label)) p = p -> next;
      if (p == NULL)
        error("Interpreter error", "Missing label (%d) in code.\n", label);
      instruction_array[i-1].arg = p->code_location;
    }
    s = s->next; dispose(p);
  }
}
```

Macro referenced in scrap 67a.

⟨Dispose of `labels` table 68b⟩ ≡

```
{ label_table p = labels; label_table o = p;
  while (p != NULL) { o = p; p = p -> next; dispose(o); }
}
```

Macro referenced in scrap 67a.

5.9 The Interpreter

5.9.1 The I/O Processes

The interpreter starts up two processes in addition to those specified by the user. These support input and output via the two channels at the base of the original stack.

The input process is a simple loop of the form

⟨Input process code 68c⟩ ≡

```
il = new_label();
input_process_code =
    build_icode(
      Lab,il,   Input,
                Ldo, -1,
                Ldc,-CHANNEL_BLOCK_SIZE-2,
                Idx,
                Out,
                Ujp,il,   END);
```

Macro referenced in scrap 69c.

The output process is a simple loop of the form

⟨Output process code 69a⟩ ≡

```
{ int l1 = new_label(), l2 = new_label();
  ol = new_label();
  output_process_code =
    build_icode(
      Lab,ol,     Ldo,-1,
                  Ldc,-2*CHANNEL_BLOCK_SIZE-3,
                  Idx,
      Lab,l1,     Ldo,-3,
                  In, l2,
                  Alt, 1,
      Lab,l2,     Pop, 1,
                  Output,
                  Ujp,l1,   END);
}
```

Macro referenced in scrap 69c.

To spawn them is easy. We just execute the code returned by the following function.

⟨Machine prototypes 69b⟩ ≡

```
extern code io_code();
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

⟨Machine utility functions 69c⟩ ≡

```
code io_code()
{ code c, input_process_code, output_process_code;
  int iss, oss, il, ol;
  ⟨Input process code 68c⟩
  ⟨Output process code 69a⟩

  /* The occurrence of 3 in the following statements is due to the display */
  iss = compute_max_stack(input_process_code) + 3;
  oss = compute_max_stack(output_process_code)+ 3;

  return(
    build_icode(
      Ujp, 0,
      input_process_code,
      output_process_code,
Lab,0,Ccb, 1,  /* stdin  */
      Ccb, 1,  /* stdout */
      Ldc, 2,
      Ldc, iss,
      Spawn, il,
      Ldc, 2,
      Ldc, oss,
      Spawn, ol,
      END));
}
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

5.9.2 The Interpreter Loop

To run the interpreter we compute the stack required by the main process, load the code into an array, create a PCB for the main process, set up the initial display, and let it rip! After execution has finished we print out a few statistics if required.

⟨Machine globals 70a⟩ ≡
```
instruction *Code;
```
Macro defined by scraps 47a, 49b, 51c, 53be, 64b, 65b, 67b, 70a, 72e.
Macro referenced in scrap 78c.

⟨Machine prototypes 70b⟩ ≡
```
extern void interpret(code c);
```
Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

⟨The interpreter 70c⟩ ≡
```

void interpret(code program)
{ int stacksize;
  clock_t start_time;
  process p;

  stacksize = compute_max_stack(program) + 2; /* for initial display */
  Code = load_code(program);
  p = create_process(0, stacksize);

  /* This next dubious line is so that the input and output processes
     will not be viewed as children of the main process, just in case the main
     process executes the wait instruction. */
  p->children = -2;

  if (debug_flag) ⟨Setup initial breakpoint 73a⟩

  initialise_display(1, &p->SP, &p->BP);

  ⟨Schedule process p for execution 71c⟩

  start_time = clock();

  interpreter_loop();

  if (pool_flag) {
    printf("CPU time used in interpreter = %d milliseconds.\n",
           (clock()-start_time)*10);
    pool_statistics(); process_statistics(); }
}
```
Macro defined by scraps 70c, 71b.
Macro referenced in scrap 78c.

The structure of the interpreter loop is fairly simple. We just keep executing instructions until all processes have terminated or blocked. When a process communicates it will naturally get rescheduled. However, if it does a lot of computation without any communication then the other processes will get blocked out. An efficient implementation would set a timer and then reschedule the process in an interrupt handler. However, the interpreter is not designed to be particularly efficient and so we adopt a simpler approach. The `counter` variable is decremented each time an instruction is executed and when it reaches 0 the process is rescheduled. To give a bit of randomness to the system the inital value of the counter when a process starts executing is set to a random number in the range 0..`max_slice` When a process is picked for execution we must set up the machine registers appropriately. When the process stops executing because it is waiting on a communication, or its timeslice has expired, we must save the current values of the registers in the processor block. Because instruction execution is nested inside a `for` loop and a `switch` it is convenient to use a `goto` when a process is rescheduled. Hopefully the resulting control flow isn't too obscure.

⟨Machine prototypes 71a⟩ ≡

```
    extern void interpreter_loop();
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

⟨The interpreter 71b⟩ ≡

```
    void interpreter_loop()
    { while (1) {
        ⟨Machine state 32b, ... ⟩
        register int counter;
        register instruction curr_inst;

    load_state:
        PC = active_process->PC; SP = active_process->SP;
        BP = active_process->BP;

        for (counter = rand() % max_slice; counter--; ) {
          if (active_process->stack_overflow_check != OVERFLOW_CHECKSUM)
            error("Fatal error",
                  "Stack overflow for process %d with stack size = %d.\n",
                  active_process, BP-SP+1);

          if (tracing_flag || active_process->trace_process)
            ⟨Display machine state 77d⟩

          ⟨Check for breakpoint 73b⟩

          switch (curr_inst = Code[PC++], curr_inst.op) {
            ⟨Execute instruction curr_inst 32f, ... ⟩ } }

    save_state:
        active_process->PC = PC;       active_process->SP = SP;
        active_process->BP = BP;
        prev_process = active_process; active_process = active_process->next;
      }
    }
```

Macro defined by scraps 70c, 71b.
Macro referenced in scrap 78c.

Finally, a few code fragments for process manipulation. When we schedule a process for execution we put it at the end of the queue so that we don't overtake any waiting processes. We need to treat the case where the queue is empty specially.

⟨Schedule process p for execution 71c⟩ ≡

```
    if (active_process) {
      prev_process->next = p; p->next = active_process; prev_process = p; }
    else { prev_process = active_process = p; p->next = p; }
    p->status = ACTIVE;
```

Macro referenced in scraps 43b, 44c, 53c, 56bc, 70c.

To suspend a process we just remove it from the queue, save its state, and then schedule the next process (if any). We presume its status has already been changed.

⟨Suspend process `p` 72a⟩ ≡

```
⟨Remove p from active process queue 53a⟩
p->SP = SP; p->PC = PC; p->BP = BP;
⟨Schedule next process or return if none 72c⟩
```

Macro referenced in scraps 55a, 56a, 72b.

⟨Suspend if children and schedule next process 72b⟩ ≡

```
if (active_process->children) {
  process p = active_process;
  p->status = WAITING;
  ⟨Suspend process p 72a⟩
}
else break;
```

Macro referenced in scrap 45b.

⟨Schedule next process or `return` if none 72c⟩ ≡

```
if (active_process == NULL) return; else goto load_state;
```

Macro referenced in scraps 44f, 55b, 56a, 72a.

5.10 Breakpoints

When debugging a sequential program it is often useful to be able to set a breakpoint at a particular instruction. The machine will then continue executing instructions until a breakpoint is reached, or execution terminates. A trivial way of supporting such a behaviour in an interpreter is to use one bit in the instruction to indicate the presence or absence of a breakpoint. In a parallel setting such breakpoints are less useful. It would be neater if we could stop execution when a particular process reaches the breakpoint. As space and speed are not really an issue in such an interpreter, we associate a linked list of process IDs (i.e. addresses) with each instruction in the code array. If the active process is contained in this list then we breakpoint. If the list contains an entry with a `NULL` process ID then we stop no matter which process is running.

⟨Machine `typedef`s 72d⟩ ≡

```
typedef struct process_list
{ process p;
  struct process_list *next;
} *process_list;
```

Macro defined by scraps 32ac, 57cd, 72d.
Macro referenced in scrap 77e.

⟨Machine globals 72e⟩ ≡

```
process_list *Bpt;
```

Macro defined by scraps 47a, 49b, 51c, 53be, 64b, 65b, 67b, 70a, 72e.
Macro referenced in scrap 78c.

⟨Create and initialise breakpoint array of size `n` 72f⟩ ≡

```
{ int i = n-1;
  Bpt = malloc(n * sizeof(process_list));
  while (i >= 0) Bpt[i--] = NULL;
}
```

Macro referenced in scrap 67a.

⟨Setup initial breakpoint 73a⟩ ≡

```
{ Bpt[0] = new(struct process_list); Bpt[0]->p = p; Bpt[0]->next = NULL; }
```

Macro referenced in scrap 70c.

We enter the debugger if the current process has the `step_process` flag set to `TRUE` (clearing it in the process), the current instruction's breakpoint list mentions this process, or this list contains the null process.

⟨Check for breakpoint 73b⟩ ≡

```
if (active_process->step_process) {
  active_process->step_process = FALSE;
  debugger(PC,SP,BP); }
else if (Bpt[PC]) { /* Check list for this process, or the NULL process */
  process_list l = Bpt[PC];
  while (l && l->p != NULL && l->p != active_process) l = l->next;
  if (l) debugger(PC,SP,BP);
}
```

Macro referenced in scrap 71b.

When a breakpoint is encountered the user is presented with a number of alternatives. The language accepted by the system at this point is shown below. We could use bison to parse this language, but it's a bit tedious supporting two bison parsers in the same program, and the language is quite simple. We therefore do it by hand. Here are the statements accepted by the debugger (optional items are enclosed in square brackets):

`i [`*m*`] [+ `*n*`]` Display *n* instructions starting at instruction *m*. If *n* is omitted it defaults to the current PC, and *n* defaults to 1.

`s [`*n*`]` Print the top *n* values on the stack. If *n* is omitted then the stack entries between `SP` and `BP` are printed.

`a `*a*` [+ `*n*`]` Print the contents of *n* addresses starting at hex address *a*.

`b [`*i*` [`*a*`]]` Toggle the breakpoint at instruction *i*. If *a* is specified then toggle the breakpoint for the process with this hex address. If no arguments are specified then display a list of breakpoints.

`n` Step to the next instruction. Note that if the instruction involves a communication then the process may be suspended, and other breakpoints might be responded to first.

`g` Continue execution until the next breakpoint is reached (not necessarily in this process), or until execution terminates.

`t [`*a*`]` Toggle tracing for process with hex address *a*, or for all processes if no argument is present.

The main structure of the debugger is fairly obvious. The main arkwardness arises because we must reenable input buffering before interacting with the user. We disable it again when we exit from the debugger.

⟨Machine prototypes 73c⟩ ≡

```
extern void debugger(int PC, stack_slot *SP, stack_slot *BP);
```

Macro defined by scraps 48a, 49ae, 50acd, 51d, 54a, 57e, 58c, 59b, 61b, 62abce, 63b, 66bd, 69b, 70b, 71a, 73c.
Macro referenced in scrap 77e.

⟨Machine utility functions 73d⟩ ≡

```
void debugger(int PC, stack_slot *SP, stack_slot *BP)
{ char buffer[256], command[256];
```

```
    int offset;
    printf("%X @ %3d: ", active_process, PC);
    print_instruction(Code[PC]); printf("\n");
    reset_terminal();

    while (TRUE) {
      printf("? "); fflush(stdout);
      if (gets(buffer) == NULL) exit(EXIT_SUCCESS);

      if (sscanf(buffer, "%s%n", command, &offset) != 1) {
        printf("%X @ %3d: ", active_process, PC);
        print_instruction(Code[PC]); printf("\n"); }

      else if (strlen(command) != 1)
        printf("Illegal command. Type '?' for help.\n");

      else switch (command[0]) {
        case 'i': ⟨Handle debugger instruction 'i' and break 74b⟩
        case 's': ⟨Handle debugger instruction 's' and break 75b⟩
        case 'a': ⟨Handle debugger instruction 'a' and break 75d⟩
        case 't': ⟨Handle debugger instruction 't' and break 75f⟩
        case 'b': ⟨Handle debugger instruction 'b' and break 76b⟩
        case 'n': active_process->step_process = TRUE;
        case 'g': disable_buffering(); return;
        default : printf("Illegal command.\n\n");
        case '?': ⟨Display debugger help 77c⟩
      }
    }
  }
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

Now let's deal with each debugger instruction in turn. To process the optional arguments to the instructions we make multiple calls to `scanf` until one succeeds. This is clearly non-optimal, but this is not important in the current context. In the case of the `'i'` instruction we just need to call `print_instruction` on the appropriate range of instructions. We associate the debugging help information with each entry to ease maintenance.

⟨Debugger help information 74a⟩ ≡

```
  printf("i [m] [+ n]\n");
  printf("   Displays n instructions starting at instruction m.\n");
  printf("   If m is omitted it defaults to the current PC, and n defaults to 1.\n\n");
```

Macro defined by scraps 74a, 75ace, 76a, 77b.
Macro referenced in scrap 77c.

⟨Handle debugger instruction 'i' and break 74b⟩ ≡

```
  { int m, n, i;
    if (sscanf(&(buffer[offset]), "%d +%d", &m, &n) == 2) ;
    else if (sscanf(&(buffer[offset]), " +%d", &n) == 1) m = PC;
    else if (sscanf(&(buffer[offset]), "%d", &m) == 1) n = 1;
    else { m = PC; n = 1; }

    for (i = 1; i <= n && m < code_size; i++) {
      printf("%3d: ", m); print_instruction(Code[m++]); printf("\n"); }
    break;
  }
```

Macro referenced in scrap 73d.

The 's' and 'a' instructions are similar. Note that the world can end if we provide bogus hex addresses. This is particularly important for those instructions that require a valid process address.

⟨Debugger help information 75a⟩ ≡

```
printf("s [n]\n");
printf("   Display the top n entries on the stack. \n");
printf("   If n is omitted then the stack entries between SP and BP are displayed.\n\n");
```

Macro defined by scraps 74a, 75ace, 76a, 77b.
Macro referenced in scrap 77c.

⟨Handle debugger instruction 's' and break 75b⟩ ≡

```
{ stack_slot *m = BP, *p; int i;
  if (sscanf(&(buffer[offset]), "%d", &i) == 1) m = SP+(i-1);
  for (p = SP; p <= m; p++) printf("%X: %X\n", p, p->as_addr);
  break;
}
```

Macro referenced in scrap 73d.

⟨Debugger help information 75c⟩ ≡

```
printf("a m [+ n]\n");
printf("   Print the contents of n addresses starting at hex address m.\n\n");
```

Macro defined by scraps 74a, 75ace, 76a, 77b.
Macro referenced in scrap 77c.

⟨Handle debugger instruction 'a' and break 75d⟩ ≡

```
{ void *p; int i, n = 1;
  if (sscanf(&(buffer[offset]), "%x +%d", &p, &n) == 2) ;
  else if (sscanf(&(buffer[offset]), "%x", &p) == 1) ;
  else p = SP;
  for (i = 0; i < n; ((int *)p)++, i++) printf("%X: %X\n", p, *(int *)p);
  break;
}
```

Macro referenced in scrap 73d.

If an argument is provided to the 't' instruction then we assume that this is the hex address of a process. We toggle the `trace_process` flag of this process. If no argument is provided we toggle the global `tracing_flag`.

⟨Debugger help information 75e⟩ ≡

```
printf("t [a]\n");
printf("   Toggle tracing for process with hex address $a$,\n");
printf("   or for all processes if no argument is present.\n\n");
```

Macro defined by scraps 74a, 75ace, 76a, 77b.
Macro referenced in scrap 77c.

⟨Handle debugger instruction 't' and break 75f⟩ ≡

```
{ process p = NULL;
  sscanf(&(buffer[offset]), "%x", &p);
  if (p) p->trace_process = !p->trace_process;
  else tracing_flag = !tracing_flag;
  break;
}
```

Macro referenced in scrap 73d.

The breakpoint instruction is the trickiest one to process. If no arguments are provided we just traverse and display the contents of the Bpt array. Otherwise we traverse the appropriate entry in this list to see if there is already an appropriate breakpoint. If there is we remove it, and if not we add it.

⟨Debugger help information 76a⟩ ≡

```
printf("b [i [a]]\n");
printf("   Toggle the breakpoint at instruction i.  If a is specified then\n");
printf("   toggle the breakpoint for the process with this hex address.\n");
printf("   If no arguments then display a list of breakpoints.\n\n");
```

Macro defined by scraps 74a, 75ace, 76a, 77b.
Macro referenced in scrap 77c.

⟨Handle debugger instruction 'b' and break 76b⟩ ≡

```
{ int pc = PC; void *p = NULL;
  process_list l;
  if (sscanf(&(buffer[offset]), "%d %x", &pc, &p) == 2) ;
  else if (sscanf(&(buffer[offset]), "%d", &pc) == 1) ;
  else ⟨Display breakpoints and continue with debugger loop 76d⟩

  if (Bpt[pc])
    if (Bpt[pc] && (Bpt[pc]->p == p)) /* Remove it */ Bpt[pc] = Bpt[pc]->next;
    else {
      process_list prev = Bpt[pc];
      l = prev->next;
      while (l && l->p != p) { prev = l; l = l->next; }
      if (l) /* Remove it */ { prev->next = l->next; dispose(l); }
      else ⟨Add breakpoint 76c⟩ }
  else ⟨Add breakpoint 76c⟩
  break;
}
```

Macro referenced in scrap 73d.

⟨Add breakpoint 76c⟩ ≡

```
{ l = new(struct process_list);
  l->p = (process)p; l->next = Bpt[pc]; Bpt[pc] = l;
}
```

Macro referenced in scrap 76b.

⟨Display breakpoints and continue with debugger loop 76d⟩ ≡

```
{ int i;
  printf("Breakpoints:\n");
  for (i = 0; i < code_size; i++) {
    if (Bpt[i]) {
      l = Bpt[i];
      printf("%3d: ", i);
      while (l) {
        if (l->p == NULL) printf("*");
        else printf("%X", l->p);
        if (l = l->next) printf(", "); else printf("\n"); }
      }
    }
  continue;
}
```

Macro referenced in scrap 76b.

To step past an instruction we use the `step_process` flag contained in the PCB of a process. When set, the process will enter the debugger before executing it's next instruction. To step a process we just need to set the flag and clear it when the debugger is next entered.

⟨Other PCB fields 77a⟩ ≡

```
    bool step_process;
```

Macro defined by scraps 48b, 51b, 52a, 77a.
Macro referenced in scrap 50e.

⟨Debugger help information 77b⟩ ≡

```
    printf("n  Step to the next instruction.\n\n");
    printf("g  Continue execution.\n\n");
```

Macro defined by scraps 74a, 75ace, 76a, 77b.
Macro referenced in scrap 77c.

Finally, here is some boring code to print out the help information for the instructions.

⟨Display debugger help 77c⟩ ≡

```
    printf("These are the instructions accepted by the debugger:\n\n");
    ⟨Debugger help information 74a, ... ⟩
```

Macro referenced in scrap 73d.

When tracing the processes we need to display the state of the machine. Here is a simple bit of code to print out the current state of the active process.

⟨Display machine state 77d⟩ ≡

```
    { int stack_slots = BP-SP+1;
      printf("Process %X: PC = %3d, instruction = ", active_process, PC);
      print_instruction(Code[PC]); printf(", SP = %X,\n", SP);
      printf("   stack depth =%3d", stack_slots);
      if (stack_slots > 0) {
        stack_slot *sp;
        printf(", contents = ");
        if (stack_slots > 10) { stack_slots = 10; printf("..., "); }
        sp = SP+(stack_slots-1);
        while (--stack_slots) printf("%X, ", (sp--)->as_int);
        printf("%X.\n", sp->as_int); }
      else printf(".\n");
      fflush(stdout);
    }
```

Macro referenced in scrap 71b.

5.11 The Interpreter Files

"code/interpreter.h" 77e ≡

```
    ⟨Copyright message 2a⟩
    #ifndef _INTERPRETER_H
    #define _INTERPRETER_H
    ⟨Machine opcodes typedef 31⟩
    ⟨Process status typedef 51a⟩
    ⟨Process block typedef 50e⟩
    ⟨Machine typedefs 32a, ... ⟩
    ⟨Machine prototypes 48a, ... ⟩
    #endif
```

⟨Occam object files 78a⟩ ≡

```
      interpreter.o
```

Macro defined by scraps 10d, 23b, 25d, 28d, 78a, 86a.
Macro referenced in scrap 3.

⟨Makefile dependencies 78b⟩ ≡

```

     interpreter.c: occam.h pool.h interpreter.h
```

Macro defined by scraps 10c, 23c, 25c, 28c, 78b, 85f.
Macro referenced in scrap 3.

`"code/interpreter.c"` 78c ≡

⟨Standard μ`OCCAM` includes 2b⟩

```
     #include <ctype.h>
     #include <time.h>
     #include "occam.h"
     #include "pool.h"
     #include "interpreter.h"
```

⟨Machine globals 47a, … ⟩
⟨Machine utility functions 41c, … ⟩
⟨The interpreter 70c, … ⟩

Function name	Return value type	Argument types	Page
build_ccode	code	code, ...	58
build_icode	code	int, ...	58
build_jump_table	jump_table	code	62
code_args	code	int, va_list	58
compute_max_stack	int	code	66
constant_expression	int	code, int *	59
create_process	process	int, int	51
debugger	void	int, stack_slot *, stack_slot *	74
dispose_code	void	code	58
dispose_jump_table	void	jump_table	63
exit_scope_code	code	void	50
initialise_display	void	int, stack_slot **, stack_slot **	40
interpret	void	code	70
interpreter_loop	void	void	71
io_code	code	void	69
load_code	instruction *	code	67
lookup_jt	code_entry *	int, jump_table	63
max_stack	int	code, code_entry *,int, int jump_table	63
print_code	void	code	60
print_instruction	void	instruction	60
process_statistics	void	void	54
stack_effect	int	instruction	61

Table 5.2: Functions defined in this chapter.

Chapter 6

The Occam Program

The overall structure of the program is simple. We first interpret the command line arguments and open the program files. The parser is then called to convert the program into abstract machine code, which is then interpreted if required.

⟨The main program 80a⟩ ≡

```
int main( int argc, char *argv[] )
{ int arg = 1; /* index into argv */

  ⟨Establish signal handlers 85c⟩
  ⟨Interpret command-line arguments 81c, ... ⟩
  ⟨Open program files 83b⟩

  initialise_symbol_table();
  yyparse();              /* code placed in global program_code */
  clear_symbol_table();   /* to reclaim storage */
  line_number = 0;        /* to disable spurious printing of line numbers in run-time errors */

  if (print_flag) print_code(program_code);

  if (run_flag)
    interpret(build_ccode(io_code(), program_code, END));
  return(EXIT_SUCCESS);
}
```

Macro referenced in scrap 86b.

6.0.1 Globals

The parser needs to know how big a channel control block is so that it can calculate the appropriate stack offsets. The size of this block is one stack slot in the interpreter, but will almost certainly be different in a compiler's runtime system. For flexibility we therefore hold the size in the global variable CHANNEL_BLOCK_SIZE.

⟨Global variable declarations 80b⟩ ≡

```
extern int CHANNEL_BLOCK_SIZE;
```

Macro defined by scraps 80b, 81ad, 82a, 83c, 84c.
Macro referenced in scrap 85e.

⟨Global variable definitions 80c⟩ ≡

```
int CHANNEL_BLOCK_SIZE = 1; /* for interpreter */
```

Macro defined by scraps 80c, 81be, 82b, 83d, 84de.
Macro referenced in scrap 86b.

We save the invocation name of the command in a global variable `command_name` for use in error messages.

⟨Global variable declarations 81a⟩ ≡
```
extern char *command_name;
```
Macro defined by scraps 80b, 81ad, 82a, 83c, 84c.
Macro referenced in scrap 85e.

⟨Global variable definitions 81b⟩ ≡
```
char *command_name = NULL;
```
Macro defined by scraps 80c, 81be, 82b, 83d, 84de.
Macro referenced in scrap 86b.

The invocation name is conventionally passed in `argv[0]`.

⟨Interpret command-line arguments 81c⟩ ≡
```
command_name = argv[0];
```
Macro defined by scraps 81c, 82c.
Macro referenced in scrap 80a.

The output from the parser, i.e. the abstract machine code for the program, is held in a global variable called `program_code`.

⟨Global variable declarations 81d⟩ ≡
```
extern struct instruction_sequence *program_code;
```
Macro defined by scraps 80b, 81ad, 82a, 83c, 84c.
Macro referenced in scrap 85e.

⟨Global variable definitions 81e⟩ ≡
```
code program_code = NULL;
```
Macro defined by scraps 80c, 81be, 82b, 83d, 84de.
Macro referenced in scrap 86b.

6.1 Command-Line Arguments

There are a variety of possible command-line arguments:

`-p` Prints out the pool storage statistics after execution.

`-c` Prints out the abstract machine code produced by the compiler.

`-r` Suppresses the evaluation of the abstract machine code.

`-t` Prints out the interpreter state before the execution of each instruction.

`-d` Sets a breakpoint at the first instruction for the initial process.

`-w time` Affects how long the program waits for input, if none available, before scheduling another process. The default is 0.

`-s num` Maximum number of instructions in timeslice (i.e. maximum number of instructions a process can execute before being rescheduled). The actual number of instructions executed is randomly chosen up to this limit.

Global flags are declared for each of the arguments.

⟨Global variable declarations 82a⟩ ≡
```
extern int pool_flag;       /* if TRUE, emit pool statistics at end */
extern int run_flag;        /* if TRUE, evaluate the program */
extern int print_flag;      /* if TRUE, print the code produced by the comp */
extern int tracing_flag;    /* if TRUE, print machine state after each inst */
extern int debug_flag;
extern int waiting_time;
extern int max_slice;
```

Macro defined by scraps 80b, 81ad, 82a, 83c, 84c.
Macro referenced in scrap 85e.

The flags are all initialized for correct default behavior.

⟨Global variable definitions 82b⟩ ≡
```
int pool_flag = FALSE;
int run_flag = TRUE;
int print_flag = FALSE;
int tracing_flag = FALSE;
int debug_flag = FALSE;
int waiting_time = 0;
int max_slice = 10;
```

Macro defined by scraps 80c, 81be, 82b, 83d, 84de.
Macro referenced in scrap 86b.

We need to examine the entries in `argv`, looking for command-line arguments.

⟨Interpret command-line arguments 82c⟩ ≡
```
while (arg < argc) {
  char *s = argv[arg];
  if (*s++ == '-') {
    ⟨Interpret the argument string s 82d⟩
    arg++;
  }
  else break;
}
```
Macro defined by scraps 81c, 82c.
Macro referenced in scrap 80a.

Several flags can be stacked behind a single minus sign; therefore, we've got to loop through the string, handling them all.

⟨Interpret the argument string `s` 82d⟩ ≡
```
{
  char c = *s++;
  while (c) {
    switch (c) {
      case 'p': pool_flag = TRUE; break;
      case 'r': run_flag = FALSE; break;
      case 'c': print_flag = TRUE; break;
      case 't': tracing_flag = TRUE; break;
      case 'd': debug_flag = TRUE; break;
      case 'w': waiting_time = atoi(argv[++arg]); break;
      case 's': max_slice = atoi(argv[++arg]); break;
      default:
        ⟨Report unexpected arguments and exit 83a⟩
    }
    c = *s++;
  }
}
```
Macro referenced in scrap 82c.

⟨Report unexpected arguments and exit 83a⟩ ≡

```
fprintf(stderr, "%s: Unexpected command line arguments.\n"
        "Usage is: %s [-prctd] [-w time] [-s num] sourcefile [inputfile]\n",
        command_name, command_name);
exit(EXIT_FAILURE);
```

Macro referenced in scraps 82d, 83b.

6.2 File Handling

The compiler expects to be passed at least one filename, containing the program to be compiled. If a second filename is passed this is used as the input to the program when it is interpreted. When only one filename is present we assume that the input to the interpreted program will be `stdin`.

⟨Open program files 83b⟩ ≡

```
{ FILE *file;
  int num_files = argc - arg;
  if (num_files == 0 || num_files > 2) {
    ⟨Report unexpected arguments and exit 83a⟩ }

  file = fopen(argv[arg],"r");
  if (file == NULL)
    error("Fatal error", "Cannot open file %s.\n",argv[arg]);
  initialise_lexer(file);

  if (num_files == 2) {
    input_fd = open(argv[arg+1],O_RDONLY);
    if (input_fd == -1)
      error("Fatal error", "Cannot open file %s.\n",argv[arg+1]);
    }
  else if (isatty(input_fd))
    ⟨Disable input buffering 83e⟩
}
```

Macro referenced in scrap 80a.

⟨Global variable declarations 83c⟩ ≡

```
extern int input_fd;
```

Macro defined by scraps 80b, 81ad, 82a, 83c, 84c.
Macro referenced in scrap 85e.

⟨Global variable definitions 83d⟩ ≡

```
int input_fd = 0; /* stdin as default */
```

Macro defined by scraps 80c, 81be, 82b, 83d, 84de.
Macro referenced in scrap 86b.

If the interpreter input is to come from `stdin` then we disable buffering on the stream. This allows us to perform non-blocking reads. Without these, the interpreter would suspend every time the input process checked for input. Unfortunately there doesn't seem to be any portable way of doing this across all Unix systems.

⟨Disable input buffering 83e⟩ ≡

```
disable_buffering();
```

Macro referenced in scrap 83b.

⟨Utility prototypes 84a⟩ ≡
```
extern void disable_buffering();
```
Macro defined by scraps 84af, 85a.
Macro referenced in scrap 85e.

⟨Utility functions 84b⟩ ≡
```

void disable_buffering()
{ struct termios term;
  tcgetattr(0, &term);
  oldterm = term;
  term.c_lflag &= ~ICANON;
  term.c_cc[VMIN] = 0; term.c_cc[VTIME] = waiting_time;
  tcsetattr(0, TCSANOW, &term);
  input_buffered = FALSE;
  atexit(&reset_terminal);
}
```
Macro defined by scraps 84bg, 85bd.
Macro referenced in scrap 86b.

We introduce a flag to record whether the input is being buffered. This is needed so that we can interpret the return values from calls to `read` appropriately.

⟨Global variable declarations 84c⟩ ≡
```
extern int input_buffered;
```
Macro defined by scraps 80b, 81ad, 82a, 83c, 84c.
Macro referenced in scrap 85e.

⟨Global variable definitions 84d⟩ ≡
```
int input_buffered = TRUE; /* buffered by default */
```
Macro defined by scraps 80c, 81be, 82b, 83d, 84de.
Macro referenced in scrap 86b.

We need to reset the terminal when we have finished. We use the global `oldterm` to remember the state of the terminal before we started tampering with it.

⟨Global variable definitions 84e⟩ ≡
```
struct termios oldterm;
```
Macro defined by scraps 80c, 81be, 82b, 83d, 84de.
Macro referenced in scrap 86b.

⟨Utility prototypes 84f⟩ ≡
```
extern void reset_terminal();
```
Macro defined by scraps 84af, 85a.
Macro referenced in scrap 85e.

⟨Utility functions 84g⟩ ≡
```

void reset_terminal() { tcsetattr(0, TCSANOW, &oldterm); }

```
Macro defined by scraps 84bg, 85bd.
Macro referenced in scrap 86b.

6.3 Error Reporting

Here is a simple error reporting function. The first argument should contain the type of the error, e.g. "Lexical error", "Syntax error", "Fatal error" etc. The second argument is treated like an argument to printf, i.e. it is a format specification possibly containing specifiers that are replaced by the values of subsequent arguments. If the global variable `line_number` is non-zero then the line number is also printed. The program terminates with a failure error code after the error message has been displayed.

⟨Utility prototypes 85a⟩ ≡

```
extern void error(char *type, char *format, ...);
```

Macro defined by scraps 84af, 85a.
Macro referenced in scrap 85e.

⟨Utility functions 85b⟩ ≡

```
void error(char *type, char *format, ...)
{ int i; va_list arg; va_start(arg,format);
  fprintf(stderr, "%s: %s", command_name, type);
  if (line_number) fprintf(stderr, " on line %d.\n", line_number);
  else fprintf(stderr, ".\n");
  for (i = strlen(command_name)+2; i>0; i--) fprintf(stderr, " ");
  vfprintf(stderr, format, arg);
  va_end(arg); exit(EXIT_FAILURE);
}
```

Macro defined by scraps 84bg, 85bd.
Macro referenced in scrap 86b.

We catch any errors to allow the test scripts to work smoothly.

⟨Establish signal handlers 85c⟩ ≡

```
(void) signal(SIGFPE, leave_fpe);
(void) signal(SIGSEGV, leave_segv);
```

Macro referenced in scrap 80a.

⟨Utility functions 85d⟩ ≡

```
void leave_fpe(int sig)
{ error("Runtime error", "Floating point exception.\n"); }

void leave_segv(int sig)
{ error("Runtime error", "Segmentation violation.\n"); }
```

Macro defined by scraps 84bg, 85bd.
Macro referenced in scrap 86b.

6.4 The Files

`"code/occam.h"` 85e ≡

```
#ifndef _OCCAM_H
#define _OCCAM_H
⟨Global variable declarations 80b, ... ⟩
⟨Utility prototypes 84a, ... ⟩
#endif
```

⟨Makefile dependencies 85f⟩ ≡

```
occam.c: pool.h symbol_table.h lexer.h parser.h interpreter.h occam.h
```

Macro defined by scraps 10c, 23c, 25c, 28c, 78b, 85f.
Macro referenced in scrap 3.

⟨Occam object files 86a⟩ ≡

```
        occam.o
```

Macro defined by scraps 10d, 23b, 25d, 28d, 78a, 86a.
Macro referenced in scrap 3.

"code/occam.c" 86b ≡

```

        ⟨Standard μOCCAM includes 2b⟩

        #include <termios.h>
        #include <fcntl.h>
        #include <string.h>
        #include <signal.h>
        #include "pool.h"
        #include "symbol_table.h"
        #include "lexer.h"
        #include "parser.h"
        #include "interpreter.h"
        #include "occam.h"

        ⟨Global variable definitions 80c, ... ⟩
        ⟨Utility functions 84b, ... ⟩

        ⟨The main program 80a⟩
```

Function name	Return value type	Argument types	Page No.
main	int	int, char *	80
disable_buffering	void	void	84
reset_terminal	void	void	84
error	void	char *, char *, ...	85
leave_fpe	void	int	85
leave_segv	void	int	85

This table shows all the functions defined in the preceeding chapter, their argument and return parameter types and the page on which the function is defined.

Appendix A

Indices

A.1 Files

A.2 Macros

A.3 Identifiers