Outline

Concurrency: basic principles

Multiprocessor architectures

- SIMD (Single-Instruction, Multiple-Data)
  - each processor executes the same instruction, but on a different part of the data
  - typical instances: *vector processors*
  - today e.g. in graphical cards (CUDA)
- MIMD (Multiple-Instruction, Multiple-Data)
  - processor operate independently, but can synchronize
  - *distributed* – each own data,
    or *shared memory*

Today's computers:

- more processors/cores on one chip
- the speed of one processor grows less dramatically than some years ago

Categories of concurrency

- physical concurrency
  - several program units literally *"execute simultaneously"*
- logical concurrency
  - the execution is *interleaved* on a single processor
  - but to the outside the environment looks as concurrent
- quasi-concurrency
  - coroutines – not true concurrency

Basic concepts

- *task*
  - a program unit, which can be executed independently of other units
  - sometimes called a *process* or a *thread*
- *differences* between tasks and subroutines:
  1. a task can be *executed implicitly*
  2. unit, which executes a task, may not necessarily wait for it to finish
  3. after a task is finished, the execution can continue from a different point than the point of task execution
- *heavyweight tasks*
  - each task has its own address space
- *lightweight tasks*
  - all lightweight tasks share the same address space
  - easier for the programmers
  - can be more efficient (less overhead)

Basic concepts 2

task communication

- through shared variables
- message passing
- using parameters

synchronization

- a mechanism for controlling the task execution order
- *cooperation synchronization*
  - one task `waits` for the other
  - typical problem: producer-consumer
- *competition synchronization*
  - several tasks compete for an access to a limited resource
  - typical problem: mutual exclusion
- *scheduler*
  - run-time system, which governs processor sharing among tasks

Producers-consumers

- classical OS-theory problem
- two types of processes
  - *producers* – create data
  - *consumers* – use data
- data are passed using *buffers* of a limited capacity
- constraints
  - if a consumer finds the *buffer empty*, he must wait for data
  - if a producer finds the *buffer full*, he must wait for an empty slot
- the processes must cooperate

Mutual exclusion

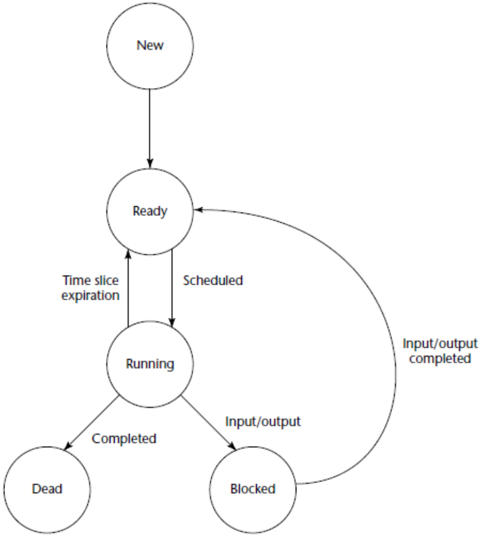
```
task A;                    task B;
  x = x + 1;                   x = x * 2;
end A;                     end B;
```

- in the beginning: x=3
- problem: arithmetic operations on x are not atomic
  1. fetch the value of x
  2. execute the operation
  3. store the value back in x
- after executing A and B: x=4 or x=6 or x=7

methods for providing mutually exclusive access

1. semaphores
2. monitors
3. message passing

Task life cycle



[Sebesta]

Problems

liveness

- typically required
- if “something good” should happen, then it eventually happens
- e.g. a scheduled task is given the processor in a finite amount of time
- the computation proceeds towards a goal

deadlock

- if two or more tasks wait for each other
- example (tasks: A, B; resources: X, Y):
  1. task A requests and obtains the resource X
  2. task B requests and obtains the resource Y
  3. task A requests the resource Y (requires X and Y)
  4. task B requests the resource X (requires X and Y)
  5. A and B mutually wain one for the other

Language designs for oncurrency

	Shared memory	**Message passing**	**Distributed**
Language	Java, C#	Ada	
Extension	OpenMP		RPC
Library	`pthreads` Win32 threads	MPI	Internet libraries

OpenMP example

```
#pragma omp for
for(int n=0; n<10; ++n) {
  printf(" %d", n);
}
printf(".\n");
```

generated code

```
int this_thread = omp_get_thread_num();
int num_threads = omp_get_num_threads();
int my_start = (this_thread  ) * 10 / num_threads;
int my_end   = (this_thread+1) * 10 / num_threads;
for(int n=my_start; n<my_end; ++n)
  printf(" %d", n);
```

possible outcome:

```
0 5 6 7 1 8 2 3 4 9.
```

Semaphores

Semaphores

- E. Dijkstra, 1965
- a simple mechanism
- based on the concept of *guards*
  - condition + block of code
  - allows the code to be executed only if the condition is fulfilled
  - semaphore is an implementation of guard
- semaphore data
  - an integer counter + a queue of waiting tasks
- operations
  - P (*wait*) [*probeer te verlagen* – try to decrease]
  - V (*signal/resume*) [*verhogen* – increase]
- counter values
  - $0, 1$ – binary *(mutex)*
  - $0 \dots n$ – *counting semaphores*

Semaphore implementation

wait(s)

```
if s.counter > 0 then
    s.counter--
else
    (s.queue).insert(P)
    suspend P's execution
end
```

signal(s)

```
if isempty(s.queue) then
    s.counter++
else
    P = (s.queue).remove()
    mark P as ready
end
```

Semaphore example

both competion and cooperation

```
semaphore access, fullCount, emptyCount;
access.counter = 1;
fullCount.counter = 0;
emptyCount.counter = BUFLEN;
```

```
task producer;
loop
  -- produce VALUE --
  wait(emptyCount);
  wait(access);
  DEPOSIT(VALUE);
  signal(access);
  signal(fullCount);

end loop;
end producer;
```

```
task consumer;
loop

  wait(fullCount);
  wait(access);
  FETCH(VALUE);
  signal(access);
  signal(emptyCount);
  -- consume VALUE --
end loop
end consumer;
```

Mutex in C – Linux 2.4.0

```
static int solo1_midi_release(struct inode *inode, struct file *file)
  ...
  lock_kernel();
  if (file->f_mode & FMODE_WRITE) {
    add_wait_queue(&s->midi.owait, &wait);
    for (;;) {
      __set_current_state(TASK_INTERRUPTIBLE);
      spin_lock_irqsave(&s->lock, flags);
      count = s->midi.ocnt;
      spin_unlock_irqrestore(&s->lock, flags);
      if (count <= 0) break;
      if (signal_pending(current)) break;
      if (file->f_flags & O_NONBLOCK) {
        remove_wait_queue(&s->midi.owait, &wait);
        set_current_state(TASK_RUNNING);
        return -EBUSY;
      }
  ...
  unlock_kernel();
  return 0;
}
```

Semaphores: evaluation

pros

- simple and efficient

problems

- prone to errorneous use
  (missing `wait` or `signal`)
- deadlocks

The semaphore is an elegant synchronization tool for an ideal programmer who never makes mistakes.

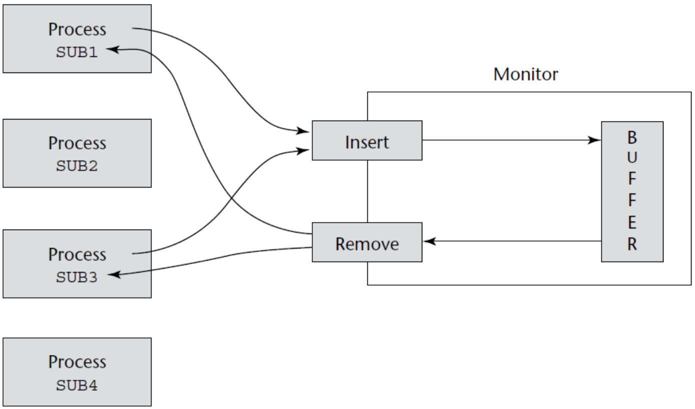
P. B. Hansen, 1973

Monitors

Monitors

- solve some of the problems of semaphores
- formalized 1973 (Hansen), named 1974 (Hoare)
- use *encapsulation*
  - shared data are packaged with operations on these data
  - the representation is hidden – basically a kind of ADT
  - access mechanism is part of the monitor
- if some subroutine of the monitor is running, all other calling tasks are put into the queue
- guarantee mutual exclusion
- cooperation is programmer’s responsibility
- can be implementer using semaphores (and vice versa)
- e.g. `synchronized` in Java

Monitor example



[Sebesta]

Concurrency in Java

Java Threads

- *threads* are lightweight tasks
  (i.e. sharing the same address space)
- communication using shared data
  (and the `join` method)

Concurrent units

1. `run()` method of various objects
   - descendants of the `Thread` class
   - classes implementing `Runnable`
2. the `main` method

```
class MyThread extends Thread {
  public void run() { ... }
}
...
Thread myTh = new MyThread();
myTh.start();
```

The `thread` class

selected methods

start	executes the thread
yield	voluntarily yields the processor
sleep	postpones for a specified time (in ms) (after which the thread joins the ready queue)
join	waits for a (different) thread to finish

```
public void run() {
  ...
  Thread myTh = new Thread();
  myTh.start();
  // do part of the computation of this thread
  myTh.join(); // Wait for myTh to complete
  // do the rest of the computation of this thread
}
```

interrupt	notifies the thread that it should finish

Java scheduler

- threads can have different priorities
- threads with higher priorities are given a preference
- a lower priority thread can run only if there is not any ready thread of a higher priority
- within the same priority the behaviour is implementation-dependent (usually round-robin)

Java – mutual exclusion

1. semaphores
   - `java.util.concurrent.Semaphore`
   - counting semaphores
2. synchronized methods/blocks
   - must be declared `synchronized`
   - each object has its own implicit *lock* and a *ready task queue*
   - a synchronized method must first obtain the lock
   - *monitor* = an object with all methods declared `synchronized`

```
class Buffer {
  private int [100] buf;
  ...
  public synchronized void deposit(int item) { ... }
  public synchronized int fetch() { ... }
  ...
}
```

Java – cooperation sync.

using the `wait, notify, notifyAll` methods
(inherited from `Object`)

`wait`	the task is added to the waiting list
`notify`	a *randomly chosen* task in the waiting list is moved to the ready queue
`notifyAll`	all waiting tasks are moved to the ready queue

```
public synchronized void deposit (int item)
  while (filled == queueSize)
    wait();
  que[nextIn] = item;
  nextIn = (nextIn % queueSize) + 1;
  filled++;
  notifyAll();
}
```

Java atomic variables

- classes in the `java.util.concurrent.atomic` package (e.g. `AtomicInteger`)
- *non-blocking, synchronized access* to
  - types `int, long, boolean`
  - arrays
  - references
- methods
  - getters, setters and arithmetic operations
  - cannot be interrupted (*atomicity*)
- advantages
  - no need for locks
  - efficient
  - faster than using *synchronized*
  - for `int` and `long` often atomic operations support in hardware

Java atomic variables IIa

example

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

Java atomic variables IIb

example

```
class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }

}
```

Java atomic variables IIc

example

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }

}
```

Java – explicit locks

```
Lock lock = new ReentrantLock();
...
Lock.lock();
try {
  // The code that accesses the shared data
  } finally {
  Lock.unlock();
  }
```

- an alternative to `synchronized`
- starting with Java 5.0
- class `ReentrantLock`, interface `Lock`
- methods `lock, tryLock, unlock`
- advantages (compared to `synchronized`):
  - the `tryLock` method waits only for a specified time
  - does not have to follow the nesting structure

Message passing

Message passing

- no relationship to message passing in OOP!
- Hansen (1978), Hoare (1978)
- nondeterminism is needed to achieve *fairness*
  $\Rightarrow$ related to guarded commands
- message passing can be *synchronous* or *asynchronous*

synchronous message passing

- one of the tasks sends a message
- the other task must be ready to receive the message
- only then the communication takes place – *rendezvous*
- the information exchange can then go in both directions

Concurrency in Ada

- implemented by message passing
- basic concept: *tasks*
  - syntactically simillar to packages (specification, body)
  - can be a part of a package, a subroutine or a body
- interface: *entry points*
  - part of the specification
  - locations, where tasks can receive messages
- messages can be parameterized
- method body contains the corresponding `accept` clause
  - here the received message is processed
- two specific types of tasks
  - *actor tasks* – no entry points
  - *server tasks* – no code outside `accept` clauses

Ada – syntax

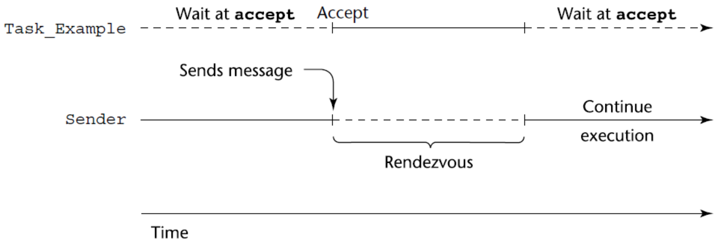
Specification

```
task Task_Example is
  entry Entry_1(Item : in Integer);
end Task_Example;
```

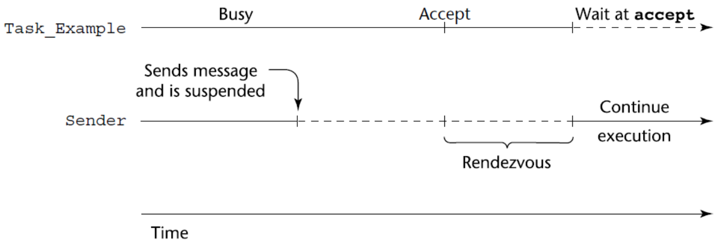
Implementation

```
task body Task_Example is
  begin
  loop
    accept Entry_1(Item : in Integer) do
      ...
    end Entry_1;
  end loop;
end Task_Example;
```

Ada – the course of synchronization



(a) `Task_Example` waits for `Sender`



(b) `Sender` waits for `Task_Example`

Ada – choosing the recipient

- significant *asymmetry*
  - the sending task must know the name of the entry point
  - the receiving task receives from anybody
- multiple entry points per task?
  - considered in the order they appear
  - `select` clauses – in any order

```
select
  accept Foo_Msg(formal parameters) do
  ...
  end Foo_Msg;
  ...
or
  accept Bar_Msg(formal parameters) do
  ...
  end Bar_Msg;
  ...
end select;
```

Ada buffer example

```ada
task body Buf_Task is
  Bufsize : constant Integer := 100;
  Buf: array (1..Bufsize) of Integer;
  Filled : Integer range 0..Bufsize := 0;
  Next_In,Next_Out : Integer range 1..Bufsize := 1;
begin
  loop
    select
      when Filled < Bufsize =>  -- open or closed
        accept Deposit(Item : in Integer) do
          Buf(Next_In) := Item;
        end Deposit;
        Next_In := (Next_In mod Bufsize) + 1;
        Filled := Filled + 1;
    or
      when Filled > 0 =>        -- open or closed
        accept Fetch(Item : out Integer) do
      ...
end Buf_Task;
```

Ada concurrency

there's more . . .

- shared memory in addition to message passing
- tasks can be used to implement monitors,
  but are much more general!

  similar to the relationship between packages and ADTs
- prone to programming errors
- problems efficiently implementating rendezvous
- solution: *protected objects*
  - similar to monitors
  - entry points replaced by protected functions/procedures

Concurrency in functional languages

Multilisp

`pcall`

```
(pcall f a b c d)
```

- parameters `a,b,c,d` of the function `f` can be evaluated concurrently
- safe if there are no side-effects

`future`

```
(future x)
```

- creates a new task evaluating `x`
- the original task continues until it needs the value of `x`

Concurrent ML (CML)

- *threads*, created using `spawn (`*`function`*`)`

```
let val c : string chan = channel ()
  in spawn (fn () => TextIO.print (recv c));
```

- the resulting value is ignored
- effects: output or communication with other threads
- inter-thread communication
  - using *channels*
  - synchronous message passing
  - channels are types (can be passed as a parameter)
  - if multiple messages, the choice among them is random

Formal Semantics for Natural Language

1 Goals of Semantics

Early work on semantics in generative grammar is now felt to be misguided. This work concentrated on specifying translation procedures between syntactic and semantic structures. However, the meaning of these 'semantic' structures was never defined. Several researchers pointed out that this process just pushed the problem one level further down – rather as though I translate an English sentence into Tagalog (or some other language you do not understand) and then tell you that is the meaning of the English sentence. Recent work on semantics in generative grammar has been based on 'logical' truth-conditional semantics. This approach avoids the above criticism by relating linguistic expressions to actual states of affairs in the world by means of the concept of truth. Within generative grammar, this approach is usually called Montague grammar or Montague semantics (after the logician Richard Montague).

1.1 Semantics and Pragmatics

Semantics and Pragmatics are both concerned with 'meaning' and a great deal of ink has been spilt trying to define the boundaries between them. We will adopt the position that Pragmatics = Meaning – Truth Conditions (roughly!). For the most part we will be concerned with the meaning of sentences, rather than the meaning of utterances. That is, we will not be concerned with the *use* of sentences in actual discourse, the speech acts they can be used to perform, and so forth. From this perspective, the three sentences in (1) will all have the same meaning because they all 'involve' the same state of affairs.

(1) a Open the window
    b The window is open
    c Is the window open

The fact that a) is most likely to convey an assertion, b) a command and c) a question is, according to this approach, a pragmatic fact about the type of speech act language users will typically associate with the declarative, imperative and interrogative syntactic constructions. We will say that all the sentences of (1) convey the same *proposition* – the semantic 'value' of a sentence.

1.2 Semantic Intuitions/Evidence

Just as with syntax we used intuitions about 'grammaticality' to judge whether syntactic rules were correct, we will use our semantic intuitions to decide on the correctness of semantic rules. The closest parallel to ungrammaticality is nonsensicality or semantic anomaly. The propositions in (2) are all grammatical but nonsensical.

(2) a Colourless green ideas sleep furiously
  b Kim frightened sincerity
  c Thirteen is very crooked

Other propositions are contradictions, as in (3).

(3) a It is raining and it is not raining
  b A bachelor is a married man
  c Kim killed Mary but she walked away

The assertion of some propositions implies the truth of other propositions; for example (4a) implies b) and c) implies d).

(4) a John walked slowly
  b John walked
  c John sold Mary the book
  d Mary bought the book from John

This relation is called *entailment* and is perhaps the most important of the semantic intuitions to capture in a semantic theory since it is the basis of the inferences we make in language comprehension, and many other semantic notions reduce to entailment. For example, two propositions can be synonymous, as in (5), but the notion of synonymy reduces to the notion of identity of entailments.

(5) a John is a bachelor
  b John is an unmarried man

We also have intuitions about the (semantic) ambiguity of certain sentences; that is they can convey more than one proposition, for example, those in (6).

(6) a Competent women and men go far
  b He fed her dog biscuits
  c Everyone knows one language

We would like our semantic theory to predict and explain these intuitions and thus we will use intuitions of this kind to evaluate semantic theories.

1.3 Semantic Productivity/Creativity

Another important aspect of meaning that we would like our semantic theory to explain is its productivity. We are able to interpret a potentially infinite number of sentences that convey different propositions. Therefore, just as in syntactic theory, we will need to specify a finite set of rules which are able to (recursively) define/interpret an infinite set of propositions.

1.4 Truth-conditional Semantics

There are two aspects to semantics. The first is the inferences that language users make when they hear linguistic expressions. We are all aware that we do this and may feel that this is what understanding and meaning are. But there is also the question of how language relates to the world, because meaning is more than just a mental phenomenon – the inferences that we make and our understanding of language are (often) about the external world around us and not just about our inner states. We would like our semantic theory to explain both the 'internal' and 'external' nature of meaning.

Truth-conditional semantics attempts to do this by taking the external aspect of meaning as basic. According to this approach, a proposition is true or false depending on the state of affairs that obtain in the world and the meaning of a proposition is its truth conditions. For example, *John is clever* conveys a true proposition if and only if John is clever. Of course, we are not interested in verifying the truth or falsity of propositions – we would get into trouble with examples like *God exists* if we tried to equate meaning with verification. Rather knowing the meaning of a proposition is to know what the world would need to be like for the sentence to be true (not knowing what the world actually is like). The idea is that the inferences that we make or equivalently the entailments between propositions can be made to follow from such a theory.

Most formal approaches to the semantics of NL are truth-conditional and model-theoretic; that is, the meaning of a sentence is taken to be a proposition which will be true or false relative to some model of the world. The meanings of referring expressions are taken to be entities / individuals in the model and predicates are functions from entities to truth-values (ie. the meanings of propositions). These functions can also be characterised in an 'external' way in terms of sets in the model – this extended notion of reference is usually called denotation. Ultimately, we will focus on doing semantics in a proof-theoretic way by 'translating' sentences into formulas of predicate / first-order logic (FOL) and then passing these to a theorem prover since our goal is automated text understanding. However, it is useful to start off thinking about model theory, as the validity of rules of inference rests on the model-theoretic intepretation of the logic.

1.5 Sentences and Utterances

An utterance conveys far more than a propositional content. Utterances are social acts by speakers intended to bring about some effect (on hearers).

Locutionary Act: the utterance of sentence (linguistic expression?) with determinate sense and reference (propositional content)
Illocutionary Act (Force): the making of an assertion, request, promise, etc., by virtue of the conventional force associated with it (how associated?)
Perlocutionary Act (Effect): the bringing about of effects on audiences by means of the locutionary act

Natural languages do not 'wear their meaning on their sleeve'. Discourse processing is about recovering/conveying speaker intentions and the context-dependent aspects of propositional content. We argue that there is a logical truth-conditional substrate to the meaning of natural language utterances (semantics). Sentences have propositional content, utterances achieve effects.

Context-dependent aspects of a proposition include reference resolution – which window are we talking about? – especially with indexicals, such as some uses of personal pronouns, *here*, *this*, time of utterance, speaker etc., so we talk about the propositional content conveyed by a sentence to indicate that this may underspecify a proposition in many ways. We'll often use the term logical form to mean (usually) the proposition / propositional content which can be determined from the lexical and compositional semantics of a sentence represented in a given logic.

1.6 Syntax and Semantics

As the ambiguous examples above made clear, syntax affects interpretation because syntactic ambiguity leads to semantic ambiguity. For this reason semantic rules must be sensitive to syntactic structure. Most semantic theories pair syntactic and semantic rules so that the application of a syntactic rule automnatically leads to the application of a semantic rule. So if two or more syntactic rules can be applied at some point, it follows that a sentence will be semantically ambiguous.

Pairing syntactic and semantic rules and guiding the application of semantic rules on the basis of the syntactic analysis of the sentence also leads naturally to an explanation of semantic productivity, because if the syntactic rule system is recursive and finite, so will the semantic rule system be too. This organisation of grammar incorporates the principle that the meaning of a sentence (its propositional content) will be a productive, rule-governed combination of the meaning of its constituents. So to get the meaning of a sentence we combine words, syntactically and semantically to form phrases, phrases to form clauses, and so on. This is known as the Principle of Compositionality. If language is not compositional in this way, then we cannot explain semantic productivity.

1.7 Model-theoretic Semantics

The particular approach to truth-conditional semantics we will study is known as model-theoretic semantics because it represents the world as a mathematical abstraction made up of sets and relates linguistic expressions to this *model*. This is an external theory of meaning *par excellence* because every type of linguistic expression must pick out something in the model. For example, proper nouns refer to objects, so they will pick out entities in the model. (Proof theory is really derivative on model theory in that the ultimate justification of a *syntactic* manipulation of a formula is that it always yields a new formula true in such a model.)

1.8 An Example

Whilst Chomsky's major achievement was to suggest that the syntax of natural languages could be treated analogously to the syntax of formal languages, so Montague's contribution was to propose that not only the syntax but also the semantics of natural language could be treated in this way. In his article entitled 'English as a Formal Language', Montague made this very explicit, writing: 'I reject the contention that an important theoretical difference exists between formal and natural languages' (compare Martin Kay's remark about 'high-level compiling').

As a first introduction to an interpreted language, we will provide a syntax and semantics for an arithmetical language.

```
a) Exp --> Int
b) Exp --> Exp Op Exp
c) Stat --> Exp = Exp
d) Int(eger): 1,2,...9,...17...
e) Op(erator): +, -
```

Notice that this grammar generates a bracket-less language. We can provide a straightforward interpretation for this language by firstly defining the meaning of each symbol of the language and secondly stating how these basic 'meanings' combine in (syntactically permissable) expressions and statements. Lets assume that the interpretation of integers is as the familiar base ten number system, so that *7* is **7**, *19*, **19**, and so on. (Just to make clear the difference between the symbol and its interpretation we will use bold face for the interpretation of a symbol and italics for a symbol of some language.) The interpretation of the operators and equality sign is also the familiar one, but if we are going to characterise the meaning of expressions and statements in terms of these more basic meanings we will need to define them in a manner which makes the way they combine with integers and other expressions clear. We will define them as (mathematical) functions which each take two arguments and give back a value. By function, we mean a relation between two sets, the domain and range, where the domain is the set of possible arguments and the range the set of possible values. For some functions, it is possible to simply list the domain and range and show the mappings between them. We cannot characterise + properly in this fashion because its domain and range will be infinite (as the set of integers is infinite), but we can show a fragment of + as a table of this sort.

Domain	Range	Domain	Range
`<0, 0>`	`0`	`<11, 1>`	`12`
`<0, 1>`	`1`	`<11, 2>`	`13`
`<1 ,0>`	`1`	`<11, 3>`	`14`
`<1, 1>`	`2`	`<11, 4>`	`15`
`<1, 2>`	`3`	`<11, 5>`	`16`
...	...	...	...

The domain of + is a set of ordered pairs, written between angle brackets, the range the set of integers. Ordering the arguments for + is not very important, but it is for −. (You might like to construct a similar table for − to convince yourself of this point.) = is a rather different kind of function whose range is very small, consisting just of the set {F,T} which we will interpret as 'false' and 'true' respectively. The table for = would also be infinite, but we show a fragment:

Domain	Range	Domain	Range
`<0, 0>`	`T`	`<1, 0>`	`F`
`<1, 1>`	`T`	`<0, 1>`	`F`
`<2, 2>`	`T`	`<1, 2>`	`F`
`<3, 3>`	`T`	`<0, 2>`	`F`
`<4, 4>`	`T`	`<2, 3>`	`F`
...	...	...	...

Functions like = which yield truth-values are sometimes called characteristic or Boolean functions (after the logician George Boole). There is a close relationship between the concept of a function and sets because we can always represent a function in terms of sets and mappings between them (although we cannot always exhaustively list the members of these sets).

Now that we have defined the meaning of all the symbols in our language, of its vocabulary, we can define how they combine semantically. We do this by adding a semantic component to each of the syntactic rules in the grammar. The result is shown below:

a) $\text{Exp} \rightarrow \text{Int} : \text{Int}'$
b) $\text{Exp} \rightarrow \text{Exp Op Exp} : \text{Op}'(\text{Exp}'_1, \text{Exp}'_2)$
c) $\text{Stat} \rightarrow \text{Exp} = \text{Exp} : =' (\text{Exp}'_1, \text{Exp}'_2)$

Each rule now has two parts delimited by a colon. The second is the semantic part. The primes are used to indicate 'the semantic value of' some category, so the category Int has values in 1,2,.. whilst Int′ has values in **1, 2,...**. The semantic operation associated with rules a) and b) is function-argument application, which is notated $F(A_1, \ldots A_n)$. The value returned by the function applied to the particular arguments which occur in some expression is the semantic value of that expression. Where the same category labels occur twice on the right hand side of some rule, we use subscripted numbers to pick them out uniquely, by linear order, for the semantic part of the rule.

Applying this interpretation of our language to some actual expressions and statements should make the mechanics of the system clearer. Below we show one of the syntactic structures assigned to $4 + 5 - 3 = 6$ and give the corresponding semantic interpretation where each symbol has been replaced by its interpretation and each node of the tree by the interpretations derived from applying the semantic rule associated with each syntactic rule to the semantic values associated with the daughter categories.

```
            Stat                               =(6 6) T
           /  | \                               /  |  \
        Exp   \  \                       -(9 3) 6  |   |
       /  | \  \  \                     /  |\      |   |
    Exp   |  \  \  \            +(4 5) 9   | \     |   |
   / | \  \   \  \  \           /  |  \    |  \    |   |
Exp  | Exp |  Exp \  Exp       4   |   5   |   \   |   6
 |   |  |  |   |   |  |        |   |   |   |    |  |   |
Int  | Int |  Int  | Int       4   |   5   |    3  |   6
 |   |  |  |   |   |  |        |   |   |   |    |  |   |
 4   +  5  -   3   =  6        4   +   5   -    3  =   6
```

Rule a) just states that an integer can be an expression and that the semantic value of that expression is the semantic value of the integer. Accordingly, we have substituted the semantic values of the integers which occur in our example for the corresponding categories in the syntactic tree diagram. Rule b) is used in to form an expression from 4, + and 5. The associated semantic operation is function-argument application, so we apply the semantic value of the operator + to the semantic value of the arguments, 4 and 5. The same syntactic rule is used again, so we perform another function-argument application using the result of the previous application as one of the arguments. Finally, c) is used, so we apply = to 6 and 6, yielding 'T' or 'true'. You might like to draw the other tree that can be assigned to this example according to the grammar and work through its semantic interpretation. Does it yield a true or false statement?

It may seem that we have introduced a large amount of machinery and associated notation to solve a very simple problem. Nevertheless, this apparently simple and familiar arithmetical language, for which we have now given a syntax and semantics, shares some similarities with natural language and serves well to illustrate the approach that we will take. Firstly, there are an infinite number of expressions and statements in this language, yet for each one our semantic rules provide an interpretation which can be built up unit-by-unit from the interpretation of each symbol, and each expression in turn. This interpretation proceeds hand-in-hand with the application of syntactic rules, because each syntactic rule is paired with a corresponding semantic operation. Therefore, it is guaranteed that every syntactically permissable expression and statement will receive an interpretation. Furthermore, our grammar consists of only three rules; yet this, together with a lexicon describing the interpretation of the basic symbols, is enough to describe completely this infinite language. This expressive power derives from recursion. Notice that the semantic rules 'inherit' this recursive property from their syntactic counterparts simply by virtue of being paired with them. Secondly, this language is highly ambiguous – consider the number of different interpretations for $4 + 5 - 2 + 3 - 1 = 6 - 4 + 9 - 6$ – but the grammar captures this ambiguity because for each distinct syntactic tree diagram which can be generated, the rules of semantic interpretation will yield a distinct analysis often with different final values.

1.9 Exercises

1) Can you think of any arithmetical expressions and statements which cannot be made given the grammar which do not require further symbols? How would you modify the grammar syntactically and semantically to accommodate them?

2) What is the relationship between brackets and tree structure? Can you describe informally a semantic interpretation scheme for the bracketed language generated by (1) which does not require reference to tree diagrams or syntactic rules?

3) The interpretation we have provided for the bracket-less arithmetical language corresponds to one which we are all familiar with but is not the only possible one. Find an interpretation which makes the statements in a) true and those in b) false. Define a new grammar and lexicon which incorporates this interpretation.

```
a)                  b)
4 + 1 = 4             3 - 2 = 1
4 - 1 = 4             4 - 6 = 8
5 + 3 = 1             7 + 4 = 10
9 + 2 = 6             5 + 2 = 3
6 - 2 = 5             3 - 4 = 2
```

5) Provide a grammar for a language compatible with the examples illustrated in below. Interpret the integers in the usual way, but choose an interpretation for the new symbols chosen from $+$, $-$, and $*$ (multiply). Give the interpretation that your grammar assigns to each example and demonstrate how it is obtained.

```
a) @ 2 3
b) #  2 @ 3 4
```

```
c) ^ 4 @ 2 #   6 8
d) #   @ 4 ^ 3 2 5
e) @ #   ^ 2 3 7 9
```

2 The Meaning of Sentence Connectives

In this section, we will begin developing a truth-conditional theory of sentence semantics in earnest. We will proceed by trying to capture the meaning of increasingly large fragments of English. To start with, we will tackle words like conjunctions used to coordinate very simple sentences consisting of proper names and intransitive verbs. The system we will develop is a slight extension to Propositional Logic (PL)

2.1 Denotation and Truth

Truth-conditional semantics attempts to capture the notion of meaning by specifying the way linguistic exressions are 'linked' to the world. For example, we argued that the semantic value of a sentence is (ultimately) a proposition which is true or false (of some state of affairs in some world). What then are the semantic values of other linguistic expressions, such as NPs, VPs, and so forth? If we are going to account for semantic productivity we must show how the semantic values of words are combined to produce phrases, which are in turn combined to produce propositions. It is not enough to just specify the semantic value of sentences.

One obvious place to start is with proper names, like *Max* or *Belinda* because the meaning of a proper name seems to be intimately connected to the entity it picks out in the world (ie. the entity it refers to, eg. max1, a particular unique entity named *Max*). So now we have the semantic values of proper names and propositions but we still need to know the semantic values of verbs before we can construct the meaning of even the simplest propositions. So what is the 'link' between verbs and the world? Intransitive verbs combine with proper names to form propositions – intransitive verbs pick out properties of entities. But how can we describe a 'property' in terms of a semantic theory which attempts to reduce all meaning to the external, referential aspect of meaning? One answer is to say that the semantic value of an intransitive verb is the set of entities which have that property in a particular world. For example, the semantic value of *snore* might be {max1, fido1}. Actually, we will say that a set like this is the semantic value of a predicate like snore1, a particular sense of *snore*. Now we are in a position to say specify the meaning of (7) in a compositional fashion.

(7) Max snores

First find the referent of *Max* and then check to see whether that entity, say max1, is in the set of entities denoted by snore1. Now we have specified the truth-conditions of the proposition conveyed by (7) (ignoring any possible ambiguities concerning the referent of *Max* and the sense of *snore*).

Developing a truth-conditional semantics is a question of working out the appropriate 'links' between all the different types of linguistic expression and the world in such a way that they combine together to build propositions. To distinguish this extended notion of reference from its more general use, we call this relation **denotation**. Thus the denotation of an intransitive verb will be a set of entities and of a proper name, an entity.

At this point we should consider more carefully what sentences denote. So far we have assumed that the semantic value of a sentence is a proposition and that propositions are true or false. But what is the link with the world? How is this to be described in external, referential terms? One answer is to say that sentences denote their truth-value (ie. true or false) in a particular world, since this is the semantic value of a proposition. So we add the 'entities' true and false to the world and let sentences denote these 'entities'. However, there is an immediate problem with this idea – all true sentences will mean the same thing, because truth-conditional semantics claims in effect that denotation exhausts the non-pragmatic aspects of meaning. This appears to be a problem because *Mr. Blair is prime minister* and *Mr. Clinton is president* are both true but don't mean the same thing.

2.2 Sense and Reference

The problem of the denotation of sentences brings us back to the internal and external aspects of meaning again. What we want to say is that there is more to the meaning of a sentence than the truth-value it denotes in order to distinguish between different true (or false) sentences. There are other problems to; consider, for example, the sentence in (8)

(8) The morning star is the evening star.

It was a great astronomical discovery when someone worked that a star seen at a certain position in the sky in the morning and one seen at another position in the evening were both Venus. Yet according to our theory of semantics this ought to be a tautologous or logically true statement analogous to (9) because the meaning of a definite description or a proper name is just the entity (individual or object) it denotes.

(9) Venus is Venus.

Traditionally, linguistic expressions are said to have both a sense and a reference, so the meaning of *the morning star* is both its referent (Venus) and the concept it conveys (star seen in morning).

At this point you might feel that it is time to give up truth-conditional semantics, because we started out by saying that the whole idea was to explain the internal aspect of meaning in terms of the external, referential part. In fact things are not so bad because it is possible to deal with those aspects of meaning that cannot be reduced to reference in model-theoretic, truth-conditional semantics based on an intensional 'possible worlds' logic. The bad news is though that such logics use higher-order constructs in ways which are difficult to reduce to first-order terms for the purposes of automated theorem proving. More on this later. For the moment we will continue to develop a purely 'extensional' semantics to see how far we can get.

2.3 Propositional Logic

Consider a sentence such as (10) made up of two simple sentences conjoined by *and*.

(10) Max snores and Belinda smiles

Lets assume that *Max snores* and *Belinda smiles* convey true propositions – then what is the truth-value of the proposition conveyed by the complex sentence in (10)?

Propositional logic (PL) addresses the meaning of sentence connectives by ignoring the internal structure of sentences / propositions entirely. In PL we would represent (10) and many other English sentences as $p \wedge q$ where p and q stand in for arbitrary propositions. Now we can characterise the meaning of *and* in terms of a truth-table for $\wedge$ which exhausts the set of logical possibilities for the truth-values of the conjoined propositions (p, q) and specifies the truth of the complex proposition as a function of the truth-values for the simple propositions, as below:

p	q	p ∧ q
t	t	t
t	f	f
f	t	f
f	f	f

Can you write down the truth-tables for $\vee$, $\neg$ and $\Rightarrow$? (I use $\Rightarrow$ to denote logical implication.)

The semantics of PL must specify the truth-conditions for the truth or falsity of well-formed formulas in terms of the truth or falsity of the simple unanalysed propositions represented by sentential variables and the truth-conditions for each connective. We have already seen what the truth-conditions for *and* are; we gloss this as below, where α and β are metavariables standing for any sentential variable:

$\alpha \wedge \beta$ is true iff both α and β are true.

Notice that is a statement in the meta language we are using to describe the semantics of PL. We could state the semantics of each sentence connective in this way. This semantic model-theoretic interpretation licences entailments or rules of valid inference like $p, q \models\ p \wedge\ q$ – can you see why? The truth-table characterisations of the logical constants guarantee that such entailments must hold, given any valid model of a PL language.

Each connective combines with two propositions to form a new complex proposition. Therefore we can represent the general form of the semantics of a connective as a function which takes two arguments and yields a result. In the case of sentential connectives this function will be what is called a truth function because both arguments and result will be truth-values. The precise function will vary depending on which connective we are discussing. Similarly, the negation operator can be described as a truth function which takes one argument. Truth-tables define these functions. However, we still need some way of incorporating these semantic rules into a grammar. This will ensure that we can interpret any well-formed formula that the grammar generates. The following CF rules generate a bracketless variant of PL, in which Var represents any propositional variable (p,q,r) and Scon any 2-place connective ($\wedge, \vee, \Rightarrow$). and the semantic rules have the interpretation given in the previous section; that is, primes denote the 'semantic value of' and $[F, A_1, \dots A_n]$ function-argument application:

1) S $\rightarrow$ Var : Var$'$
2) S $\rightarrow$ S Scon S : [Con$'$,S$'$,S$'$]
3) S $\rightarrow$ Neg S : [Neg$'$,S$'$]

We can read the semantic interpretation of a formula of bracketless PL off its structural description. To see how this works it is probably best to imagine that you are applying the semantic rules to the syntax tree working from the 'deepest' point in the tree up to its root node. The interpretation of the simple propositions depends on the truth-value that we (arbitrarily) assign them to get the process started. This process is analogous to choosing a model of the world in which to interpret the formula (ie. assuming some state of affairs). The interpretation of the connectives, however, remains the same in any model we care to imagine; for this reason the connectives are often called **logical constants**.

2.4 English Fragment 1

Our first semantically interpreted fragment of English (F1) is going to consist of the natural language 'equivalents' of the sentential connectives of PL and basic sentences constructed from proper names and transitive or intransitive verbs; for example (11) is one sentence in F1.

(11) Max smiles and Belinda likes Fido

We will use the semantics of the connectives developed for PL and then consider how good this semantics is for the English words *and*, *or*, and so forth. We will develop a semantics of simple sentences along the lines sketched above.

2.4.1 Lexicon for F1

The lexicon stores syntactic and semantic information about each word (in that order). The semantic interpretation of a proper name is the particular entity it refers to. Semantic interpretations are written with numbers to indicate which sense of the predicate or referent in the model is intended – since we are not dealing with reference or word meaning at the moment this will always be 1.

```
Max : Name : max1
Belinda : Name : belinda1
Fido : Name : fido1
Felix : Name : felix1

and : Conj : and
or : Conj : or
it-is-not-the-case-that : Neg : not

snores : Vintrans : snore1
smiles : Vintrans : smile1
likes : Vtrans : like1
loves : Vtrans : love1
```

2.4.2 Grammar for F1

The syntactic rules are only marginally more complex than those for PL. The interpretation of the semantic part of the rules is identical to that described for PL:

1) S $\rightarrow$ NP VP : [VP$'$,NP$'$]
2) S $\rightarrow$ S Conj S : [Conj$'$,S$'$,S$'$]
3) S $\rightarrow$ Neg S : [Neg$'$,S$'$]
4) VP $\rightarrow$ Vtrans NP : [V$'$,NP$'$]
5) VP $\rightarrow$ Vintrans : V$'$
6) NP $\rightarrow$ Name : Name$'$

2.4.3 Some Examples

Unless we construct a model in which to interpret F1 we cannot illustrate the workings of the semantic rules directly in terms of truth-values. But we can also think of the rules as a specification of how to build up a 'logical form' for a sentence. The logical form of the sentences in F1 look rather like formulas in PL except that we are using prefix notation and using 'and', 'or' and 'not' instead of the less computer friendly $\wedge$, $\vee$ and $\neg$. The example below shows how the logical form is built up in tandem with the construction of the syntax tree:

```
a)
     S                        S
    / \                     /   \
  NP   VP                 NP      VP
  |    |                  |       |
  Name Vintrans     Conj  Name    Vintrans
  |    |            |     |       |
  Max  snores       and   Belinda smiles

  [snore1,max1]           [smile1,belinda1]

b)
                    S
          /         |   \
     S              |         S
    / \             |       /   \
  NP   VP           |     NP      VP
  |    |            |     |       |
  Name Vintrans     Conj  Name    Vintrans
  |    |             |    |       |
  Max  snores       and   Belinda smiles

  [and,[snore1,max1],[smile1,belinda1]]
```

One difference between the sentences of PL, the logical forms of F1 and F1 itself is the absence of brackets in F1. This means that many of the (infinite) grammatical sentences of F1 are ambiguous; for example, the sentence in (12a) has the two logical forms shown in b) and c).

(12) a It-is-not-the-case-that Max snores and Belinda smiles
not,[and,[snore1,max1 ,[smile1,belinda1]]]
and,[not,[snore1,max1],[smile1,belinda1]]

The interpretation in b) means that neither Max snores nor Belinda smiles, whilst that in c) means that Max doesn't snore but Belinda does smile. The different interpretations are a direct consequence of the syntactic ambiguity and the corresponding different order in which the semantic rules are applied. (Incidentally *it-is-not-the-case that* is not a word of English, but *not* is rather different from the sentential connective of PL – so I'm cheating for now)

2.5 A Model for F1

Model-theoretic, truth-conditional semantics interprets linguistic expressions with respect to some model. Thus the truth or falsity of a proposition is calculated with respect to a particular model and a truth-conditional semantics consists of a set of general procedures for calculating the truth or falsity of any proposition in any model (ie. the truth-conditions for the proposition).

A model is represented in terms of sets. It is defined as a domain of entities (abbreviated E) and a function (F) which supplies the denotations of the vocabulary of the language being interpreted in that model. Here is one model in which we can interpret sentences of F1:

```
E {max1, belinda1, fido1, felix1}

F (snore1) {max1, fido1}

F (smile1) {belinda1, felix1}

F (like1) {<max1, belinda1> <belinda1, felix1>}

F (love1) {<fido1, belinda1> <fido1, max1>}
```

In this model, Max and Fido snore, Belinda and Felix smile, Max likes Belinda and Belinda likes Felix, and Fido loves Belinda and Max. Now we are in a position to interpret a sentence relative to this model:

```
   [and,[snore1,max1],[smile1,belinda1]]
   t    t             t
```

I illustrate the interpretation by indicating the truth-values which result from each function-argument application under the logical form for the sentence. We can now see more clearly what it means to say that snore1 is a function which we apply to the argument max1 (analogous to the description of *and* ($\wedge$) as a truth function above). snore1 is a function from entities to truth-values – a characteristic function. We can define it by exhaustively listing its range of inputs and corresponding outputs in this model:

```
max1       -->   t
fido1      -->   t
belinda1   -->   f
felix1     -->   f
```

The set-theoretic operation used to compute this function is just to check whether the argument is a member of the denotation of the relevant intransitive verb and return T if it is and F if it isn't.

3 Entailment and Possible Models

We can define entailment, contradiction, synonymy, and so forth in terms of the notion of possible models for a language. Lets start with contradictory propositions. The proposition conveyed by (13) is a contradiction in F1, because it is impossible to construct a model for F1 which would make it true.

(13) a Max snores and it-is-not-the-case-that Max snores
and,[snore1,max1 ,[not,[snore1,max1]]]

The reason for this is that whatever model you construct for F1, *Max snores* must either come out true or false. Now from the truth-tables for *it-is-not-the-case-that* and *and* the truth-value for (13) will always be false, because a proposition containing *and* is only ever true if both the coordinated simple propositions are true. However, this can never be the case when we coordinate a proposition and its negation because of the truth-table for negation. In F1, the semantic 'translation' of the connectives, unlike say the denotation of *snores*, remains the same in every possible model. The opposite of a contradictory proposition is one which is logically true; that is, true in every possible model. An example in F1 is *Max snores or it-is-not-the-case-that Max snores*. See if you can explain why using the same kind of reasoning that I used to explain the contradictoriness of (13).

One proposition A entails another proposition B if and only if in every possible model in which A is true, B is true as well. For example, in F1 (14a) entails (14b) because it is impossible to make up a model for F1 in which you can make a) come out true and b) come out false (unless you change the semantic rules). Try it!

(14) a Max snores and Belinda smiles
b Max snores

Can you think of any other entailment relations in F1?

Finally, two propositions are synonymous, or more accurately logically equivalent, if they are both true in exactly the same set of models. For example (15a) and b) are logically equivalent in F1 because it is impossible to make up a model in which one is true and the other false (unless you change the semantic rules). Try it!

(15) a a) Max snores and Belinda smiles
b b) Belinda smiles and Max snores

Synonymy goes beyond logical equivalence (in PL and F1) because sometimes it involves equivalence of meaning between words or words and phrases. However, to account for this we need to say something more about word meaning – so we will tackle this next term.

3.1 Exercises

1) Using the grammar for PL above and choosing an assignment of t-values to propositional variables generate some formulas and calculate their t-values

2) Construct the syntax trees and associated logical forms for the following sentences of F1:

a) It-is-not-the-case that Fido likes Felix
b) Felix loves Max or Felix loves Belinda
c) Felix loves Max or Max loves Belinda and Felix loves Fido
d) Max snores and Belinda loves Felix or it-is-not-the-case-that Felix smiles

3) Construct a different model in which to interpret the sentences of 2) illustrate how the interpretation proceeds by writing truth-values under the logical forms you have associated with each sentence.

4) Can you think of any problems with using the PL semantics of sentential connectives to describe the semantics of *and* and *or* in English? Think about the following examples:

a) The lone ranger rode off into the sunset and mounted his horse
b) Either we'll go to the pub or we'll go to the disco or we'll do both

5) What sort of a function will we need to associate with a transitive verb? Write down the function for love1 (as I did for snore1). What is the general set-theoretic operation which underlies this function? (See next section if in doubt.)

4 First Order Logic (FOL)

In this section, we will describe FOL, also often called (First-order) Predicate Logic / Calculus (FOPC) in more detail.

FOL extends PL by incorporating an analysis of propositions and of the existential and universal quantifiers. We have already informally introduced the analysis of one-place predicates like *snores* in the fragment F1. The semantics of FOL is just the semantics of F1 augmented with an account of the meaning of quantifiers, and free and bound variables. The interpretation of the connectives and of predicates and their arguments will remain the same, though we will now consider predicates with more than one argument.

4.1 FOL syntax

The FOL lexicon contains symbols of the following types:

Individual Constants such as a,b, max1, cat10, etc, indicated by lower case early letters of the alphabet, and possibly a number

Individual Variables such as w,x,y,z, indicated by lowercase late usually single letters of the alphabet (or uppercase letters in Prolog-like implementations)

Predicates such as snore1, love1, *P,Q,R,S* indicated by usually non-singular letters possiby ending with a number, or a single italicised capital

Connectives $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, (**and, or, if, iff** in implementations)

Negation $\neg$, **not** in implementations

Quantifiers $\exists$, $\forall$ (**exists, forall** in implementations)

Functions such as F_1, F_2, mother1-of, etc, indicated by pred-of or F_n

Brackets ()

We will describe the syntax of FOL using a CFG to generate well-formed formulas:

1) S $\rightarrow$ Pred(Term)
2) S $\rightarrow$ Pred(Term,Term)
3) S $\rightarrow$ Pred(Term,Term,Term)
4) S $\rightarrow$ (S Conn S)
5) S $\rightarrow$ Quan Var S
6) S $\rightarrow$ Neg S
7) Term $\rightarrow$ Const
8) Term $\rightarrow$ Var
9) Term $\rightarrow$ Fun(Term)

(Note that we only allow 1/2/3-place predicates, although this could be generalised by replacing rules 1,2,and 3 with a rule schema using Kleene plus: S $\rightarrow$ Pred(Term$^+$).) Can you write a variant grammar replacing infix connectives with prefix connectives and using square brackets around quantifiers, the hat operator etc to generate Prolog-style list syntax (see egs. in (16e,g))? – the point is there can be more than one syntax for FOL.

4.2 FOL semantics

A model for FOL consists of entities, denoted by entity constants, and properties or relations, denoted by predicate or function symbols. One-place predicates denote properties and others denote relations. Functions denote relations which yield an entity rather than a truth-value. Terms denote entities. Sentences or well-formed formulas denote truth-values. Some examples of well-formed formulas and terms are given in (16). See if you can say which is which and also name the atomic components of each one. It might help to draw the derivations.

(16) a mother1(belinda1)
 b (mother1(belinda1) $\wedge$ snore1(max1))
 c mother1-of(max1)
 d $\exists\, x(P(x) \Rightarrow\ Q(x))$
 e `[exists,x^[and,[snore1,x],[smile1,x]]]`
 f $\forall$ x $\exists$ y (mother1(x) $\Rightarrow$ equal(mother1-of(y),x))
 g `[exists,y^[forall,x^[if[snore1,x],[love1,x,y]]]]`

To model relations set-theoretically we need to introduce a notation for ordered pairs, triples and ultimately n-ary relations. `<belinda1 max1>` is an ordered pair which might be in the set denoted by love1. Notice that the two-place predicate love1 is not a symmetric relation so it does not follow (unfortunately) that `<max1 belinda1>` will also be a member of this set. So a formula $P(t_1, t_2, t_n)$ will be true in a model M iff the valuation function, F(P) yields a set containing the ordered entities denoted by terms, t_1, t_2, t_n. Otherwise a model for FOL is identical to the model we developed for F1.

FOL allows us to make general statements about entities using quantifiers. The interpretation of the existential quantifier is that there must be at least one entity (in the model) which can be substituted for the bound variable 'x' in e.g. (16d) to produce a true proposition. The interpretation of the universal quantifier ($\forall$) is that every entity (in the model) can be substituted for the variable bound by the quantifier (eg. 'x' in (16f)) to yield a true proposition. A variable is free in a formula if it is not bound by a quantifier. Vacuous quantifiers can also be generated whose variables do not occur inside the formula within their scope. Can you derive formulas with free variables and vacuous quantifiers using the grammar above?

To interpret a formula containing a variable bound by a quantifier (in a model) we need a **value assignment function** which assigns values to variables in formulas. For FOL, we want a function which assigns an entity to variables over entities. Now we can talk about the truth of a proposition in a model given some value assignment to its variables. For example, if we assume the model below (where F (A) {m b} just specifies the members of the domain of A which yield a value of t(rue) so implicitly F(A(g)) $\rightarrow$ f):

```
E {m b g}    F (A) {m b}    F (B) {<b g>}
```

then the value assignment function will assign one of the entities from the model to a variable. Thus the set of possible assignments to ‘x’ in (17a) are shown in b,c,d).

(17) a B(x g)
b B(m g)
c B(b g)
d B(g g)

Of these, only c) will yield a true proposition in this model. When variables are bound by an existential or universal quantifier, this imposes extra conditions on the interpretation of the formula, which are given below:

Existential Quantifier: The formula $\exists\ \alpha$ is true iff for some value assignment to the variable bound by $\exists$ in α, the resulting variable free formula is true.

Universal Quantifier: The formula $\forall\ \alpha$ is true iff for every value assignment to the variable bound by $\forall$ in α, the resulting variable free formula is true.

Thus the process of computing the truth-value of a formula with a variable bound by an existential quantifier is a question of mechanically substituting each entity (in the model) for the variable until you find one which makes the formula true, whilst in the case of one bound by a universal quantifier, every possible substitution must come out true. So the process of interpreting the following formula in the model given above, can be illustrated as below:

$\forall\ x\ B(x\ g)$

```
B(m g) --> f
B(b g) --> t   --> f
B(f g) --> f
```

A well-formed formula may contain more than one quantifier in which case the interpretation of the formula will vary depending on the relative scope of the quantifiers (just as we saw that the interpretation of sentences of F1 varied depending on the relative scope of *and*. For example, if we are interpreting the formula in (18a), then we must first choose a value assignment for the variable ‘x’ bound by the existential quantifier and calculate the truth of the universally quantified sub-formula with respect to this value assignment, because the universal quantifier binding ‘y’ is ‘inside’ (ie. in the scope of) the existential.

(18) a $\exists\ x\ \forall\ y\ B(x\ y)$
b $\forall\ y\ \exists\ x\ B(x\ y)$

On the other hand, in b), we must first choose an assignment for 'y' and then calculate the truth of the existentially quantified sub-formula with respect to each possible value assignment to 'y'. The interpretation of a) in the model above proceeds as illustrated below:

```
x/b                x/g                x/m

B(b b) --> f        B(g b) --> f        B(m b) --> f
B(b g) --> t        B(g g) --> f        B(m g) --> f
B(b m) --> f        B(g m) --> f        B(m m) --> f
```

We try to find one assignment to 'x' such that that one entity stands in the B relation to every entity in the model. None of the three possible assignments to 'x' yield a complete set of true propositions when we try all possible value assignments to 'y'; therefore, a) is false in the model above. However, when we compute b) we fix the assignment to the universally quantified variable first and then vary the assignment to the existentially quantified one:

```
y/b

B(b b) --> f
B(g b) --> f
B(m b) --> f
```

For a universally quantified formula to be true in a model it must be true for every possible value assignment. For an existentially quantified formula to be true in a model it must be true for one value assignment. We have assigned 'b' to 'y', but this fails to yield a true formula under any assignment of a value to 'x' so the formula must be false in this model. On the other hand if one of these formulas had been true, then we would need to continue and compute all the other possible assignments to 'y' until we found another value which did not yield a true formula under any assignment to 'x'. Thus the relative scope of the quantifiers affects which assignment is made first and therefore which is fixed with respect to the other assignment.

We can think of a quantifier as a function which is applied to the result of applying the variable it binds to the sub-formula in its scope. Applying the variable to the sub-formula means performing all the possible substitutions of entities for the variable licensed by the model and computing the truth-value of the resulting propositions. This will yield a set of truth-values. Quantifiers are then, functions from sets of truth-values to a truth-value. For example, applying the variable function to a term such as B(x f) might yield truth-values for the propositions B(f f), B(m f) and B(b f), say {t t f}. Now applying the quantifier function to {t t f} will yield a truth-value for the whole formula – 't' if the quantifier is existential, 'f' if it is universal.

(This is a fairly informal account of the model-theoretic semantics of FOL. Cann *Formal Semantics* and the other textbooks go into more detail.)

4.3 Proof Theory and Automated Theorem Proving

So far, we have been developing a model-theoretic version of truth-conditional semantics in which we interpret linguistic expressions 'with respect to' or 'relative to' or just 'in' an abstract set-theoretically defined model of the world. We have seen that it is possible to characterise judgements of synonymy, contradictoriness, relations of entailment, and so forth, in terms of the possible models for a language. However, this only works if each model is complete, in the sense that it represents every state of affairs, and we have access to every possible and complete model. (If you can't see why, read the definition of entailment again in section 1.)

We would like our semantic theory to not only characterise the semantics of a language correctly (competence) but also to shed light on the process of language comprehension (performance). However, if language users do inference in a model-theoretic fashion, they would need to carry around the 'whole actual world' (and all the other possible variations on it) 'inside their heads'. This sounds unlikely, because most of us are aware that there are big gaps in our knowledge. One answer to this is to say the competence theory characterises the ideal (omnipotent) language user and that we all operate with partial information and therefore make wrong inferences occasionally. Clearly, doing semantics by machine we cannot hope to model the whole world so we will in practice make inferences (i.e. generate useful entailments) by applying proof-theoretic rules in a goal-directed fashion (i.e. by doing something analogous to automated theorem proving).

Logics, such as PL and FOL, were invented by philosophers to study the **form** of valid argumentation, independently of its content. So philosophers have looked for rules which define valid ways of reasoning in terms of the syntax of logical expressions (regardless of their semantic content). For example, two such rules for PL are shown in (19).

(19) a And-elimination: $p \wedge \ q \vdash \ p$
b Modus Ponens: $p \Rightarrow \ q, p \vdash \ q$

Each of these rules has some premises and a conclusion (written after the ($\vdash$) entails metasymbol). These rules are valid because if the premises are true, the conclusion is guaranteed to be true as well, regardless of the semantic content of the propositional variables p and q. The rules work because of the semantics of the connectives, but given this it is possible to perform inferences using proof-theory 'mechanically'. Proof theory may well be a better way to approach the psychology of inference (and is often a better way to perform inferences mechanically by computer). Later we will look at automated theorem proving techniques and consider the issues of partiality, completeness and soundness of inference in more detail. For now, it is important to recognise how such rules are justified as rules of *valid* entailment in terms of reasoning about possible models for the logics and English fragments we are looking at.

Here is an outline of an axiomatic proof theory for FOL. For any well-formed formulas, ψ, ϕ, φ, the following rules of inference hold:

Modus Ponens: $\psi, \psi \Rightarrow \phi \vdash \phi$
And-introduction: $\psi, \phi \vdash \psi \wedge \phi$
And-elimination: $\psi \wedge \phi \vdash \psi$
Or-introduction: $\psi \vdash \psi \vee \phi$
Or-elimination: $\psi \vee \phi, \neg\phi \vdash \psi$
Universal-introduction: $\psi(x) \vdash \forall x\ \psi(x)$
(any free variable is implicitly universally quantified)
Universal-elimination: $\forall x\ \psi(x) \vdash \psi(t/x)$
(where 't' is any term substituted for all occurrences of 'x' in ψ)

and the following logical equivalences:

De Morgan: $\neg(\psi \wedge \phi) \Leftrightarrow \neg\psi \vee \neg\phi$
De Morgan: $\neg(\psi \vee \phi) \Leftrightarrow \neg\psi \wedge \neg\phi$
De Morgan: $\forall x\ \neg\psi \Leftrightarrow \neg\exists x\ \psi$
De Morgan: $\neg\forall x\ \psi \Leftrightarrow \exists x\ \neg\psi$
Distributivity: $\psi \wedge (\phi \vee \varphi) \Leftrightarrow (\psi \wedge \phi) \vee (\psi \wedge \varphi)$
Distributivity: $\psi \vee (\phi \wedge \varphi) \Leftrightarrow (\psi \vee \phi) \wedge (\psi \vee \varphi)$
Contraposition: $\psi \Rightarrow \phi \Leftrightarrow \neg\psi \Rightarrow \neg\phi$
Contraposition: $\psi \Leftrightarrow \phi \Leftrightarrow \psi \Rightarrow \phi \wedge \phi \Rightarrow \psi$

Rules of inference and logical equivalences allow purely syntactic manipulation of formulas to derive valid conclusions (proofs).

Can you reformulate the syllogism in (20) in FOL and show that it is valid?

(20) a All men are mortal
b Socrates is a man
c Socrates is mortal

Soundness: if $\Gamma \vdash \psi$ then $\Gamma \models \psi$

Completeness: if $\Gamma \models \psi$ then $\Gamma \vdash \psi$

Decidability: no for FOL
That is, FOL proof theory is sound because every proposition which is entailed by a FOL language is also in any model of that language, and it is complete because every fact in any model of a FOL language is also an entailment (see e.g. Ramsay, A. *Formal Methods in Artificial Intelligence*, CUP, 1988 for proofs

and further discussion)

From proof theory to theorem proving involves control principles to avoid non-termination, ‘irrelevant’ inferences etc. How can one of the rules of inference given above lead to non-termination?

5 An extended FOL-like English fragment, F2

Our second fragment (F2) extends and builds on F1 to include sentences such as those in (21).

(21) a Every man smiles
 b Every man likes some woman
 c No man smiles
 d Every man$_i$ loves himself$_i$
 e He loves Belinda
 f A man$_i$ likes Fido and he$_i$ likes Belinda (too)
 g Belinda gives Max a dog

5.1 Verb Complementation

We have already seen how to represent intransitive verbs / one-place predicates in F1. Now we can add transitive, ditransitive etc. (22) and capture (some of) the semantics of an extended fragment of English, F2, in terms of FOL.

(22) a Kim loves Sandy
 b love1(kim1, sandy1)
 c Kim gave Sandy Fido
 d give1(kim1, sandy1, fido1)

(Note that we are ignoring tense/aspect.) Can you construct models which will make (22b,d) true? Can you ‘translate’ the examples in (23) into FOL formulas?

(23) a Fido is on Sandy
 b Sandy needs a computer
 c Sandy thinks Kim owns a computer

Can you construct models again? What problems do these examples raise? The exstensional semantics of complementation commits us to the existence of ‘a computer’ in order to assign a logical form to (23b) – can you see why? Verbs like *need* are often called intensional verbs because they require an account of sense/intension to capture their truth-conditions and entailments properly. (23c) raises similar but even more difficult problems concerning so-called propositional attitude verbs (covered briefly in the handout *Theories of Syntax, Semantics and Discourse for NL* on module L100 and in more detail in Cann *Formal Semantics.*

5.2 Quantifiers and pronouns

We will also try to extend our truth-conditional semantics of English to cover English quantifers, such as *every* and *some*, and pronouns, such as *he/him* and *she/her*.

We will treat the English quantifiers *every* and *some* as analogous to the universal and existential quantifiers of FOL, respectively. The meaning of other English 'quantifiers', such as *no*, *a*, *the*, and so forth, will hopefully be reducible to the meaning of these two 'basic' quantifiers. Pronouns will be treated analogously to bound variables in FOL. F2 also includes nouns, such as *man*, *woman*, and so forth. Unlike proper names, nouns do not denote entities but rather properties of entities. Therefore, their meaning is the same as that of an intransitive verb, such as *snore*.

If we consider the meaning of the two sentences in (24a) and c), it should be clear that we can't capture their meaning by translating them into the FOL expressions in b) and d) respectively.

(24) a Every man snores
 b $\forall\ x\ snore1(x)$
 c Some woman smiles
 d $\exists\ x\ smile1(x)$

The problem is that the FOL expressions will be true of any entity in the model who snores or smiles. They aren't restricted in the way the English sentences are to apply to just men or just women, respectively. Obviously, we have failed to include the meaning of the nouns in our translation. The question is how to combine the noun denotations and verb denotations correctly to arrive at the correct truth-conditions for sentences of this type? a) has the logical form of an if-then conditional statement. It says that for any entity if that entity is a man then that entity snores. On the other hand, c) says that there exists at least one entity who is both a woman and smiles, so it has the logical form of a conjunction of propositions. Therefore, the correct translations of these two sentences are given in (25a,b) and (25c,d), respectively.

(25) a $\forall$ x man1(x) $\Rightarrow$ snore1(x)
 b `[forall,X^[if,[man1,X],[snore1,X]]]`
 c $\exists$ x woman1(x) $\wedge$ smile1(x)
 d `[exists,X^[and,[woman1,X],[smile1,X]]]`

We want our grammar of F2 to associate these logical forms with these sentences. To achieve this in a compositional fashion we will need to associate a 'template' logical form for the entire sentence with each of the different quantifiers – otherwise we won't be able to capture the different ways in which the NP and VP are combined semantically, depending on which quantifier is chosen. In (26) I give a 'translation' of *every*.

(26) a $\forall$ x P(x) $\Rightarrow$ Q(x)
  b `[forall,X^[if,[P,X],[Q,X]]]`

'P' and 'Q' are to be interpreted as (one-place) predicate variables. The process of combining the meaning of the quantifier with that of the noun and then that of VP is now one of substituting the meaning of the noun for 'P' and the meaning of the VP for 'Q'. The templates for the other quantifiers are shown in the lexicon for F2 below. However, note that FOL does not include predicate variables, so I am cheating in order to try to construct a compositional FOL-like treatment of F2. (We'll return to this problem in section 6.)

Many pronouns in English sentences pick out (at least) one entity in the universe of discourse, so one way to treat them semantically is to translate them into existentially-quantified variables ranging over the entities in the model with appropriate gender constraints, etc. For example, (27a) might translate as b).

(27) a He loves belinda
  b $\exists$ x male1(x) $\wedge$ love1(x belinda1)

But this analysis ignores the fact that the referent of a pronoun is normally determined anaphorically or indexically; that is, from the linguistic or extralinguistic context, respectively. However, it provides a reasonable first approximation of the semantic part of the meaning of a pronoun (ie. the part which is independent of the context of utterance). In other examples, such as (28a), it seems more appropriate to have the variable translating the pronoun bound by the universal quantifier, as in b).

(28) a Every man$_i$ thinks that he$_i$ snores
  b $\forall$ x man1(x) $\Rightarrow$ think1(x snore1(x)))
  c `[forall,X^[if,[man1,X],[think1,X,[snore1,X]]]]`

This is not the only possible interpretation of examples like (28a), but when the pronoun is interpreted as being anaphorically linked with the subject NP, it does not pick out one entity but rather the set of entities who are men. This is exactly what the translation as a universally-quantified bound variable predicts. We'll leave examples like (28a) out of F2 because of the problems with propositional attitude verbs, but examples like (29a) are in F2, and here translating the pronoun as a bound variable in the scope of the existential quantifier seems to capture their truth-conditions correctly.

(29) a A man_i likes Belinda and he_i likes felix (too)
b $\exists$ x man1(x) $\Rightarrow$ like1(x belinda1) $\wedge$ like1(x felix1)
c `[exists,X^[and,[and,[man1,X],[like1,X,belinda1]],[like1,X,felix1]]]`

However, we still have a problem because the scope of the quantifier can only be determined when we have decided whether the pronoun is anaphoric and coreferential with a quantified NP antecedent, so the 'translation' of a pronoun as an 'externally' bound variable only works for F2 with subscripts indicating coreference, as in (29a). Note also the nested prefix 'and's in (29c) – can you see how these get produced given the lexicon and grammar for F2 given in the next sections?

5.2.1 Lexicon for F2

The lexicon for F2 is the same as F1 with the addition of some determiners, nouns and pronouns, and a ditransitive verb (so that we have one three-place predicate). The complete lexicon is given below:

```
Max : Name : max1
Fred : Name : fred1
Belinda : Name : belinda1
Fido : Name : fido1
Felix : Name : felix1

he_x : PN : X
he : PN : [exists,X^[and,[male1,X],[P,X]]]
her_x : PN : X
her : PN : [exists,X^[and,[female1,X],[P,X]]]
himself_x : PN : X
herself_x : PN : X

and : Conj : and
or : Conj : or
it-is-not-the-case-that : Neg : not

snores : Vintrans : snore1
smiles : Vintrans : smile1
likes : Vtrans : like1
loves : Vtrans : love1
gives : Vditrans : give1

a : Det : [exists,X^[and,[P,X],[Q,X]]]
no : Det : [not,[exists,X^[and,[P,X],[Q,X]]]]
some : Det : [exists,X^[and,[P,X],[Q,X]]]
every : Det : [forall,X^[if,[P,X],[Q,X]]]
```

```
man : N : man1
woman : N : woman1
dog : N : dog1
cat : N : cat1
```

5.2.2 Grammar for F2

The syntactic rules of F2 are the same as those for F1 with the addition of three further rules, shown below:

7) VP → Vditrans NP NP : [V′,NP′,NP′]
8) NP → Det N : [Det′,N′]
9) NP → PN : PN′

Rule 7) allows us to cover sentences containing ditransitive verbs, and says that semantically the denotation of the verb represents a function which takes the denotations of both NP objects as arguments. Rule 8) introduces NPs containing a determiner and a noun and rule 9) pronouns. The semantics of 9) is straightforward and identical to that for the rule which introduces proper names. Determiners are treated semantically as functions which take nouns as arguments. The complete new grammar is shown below:

1) S → NP VP : [NP′,VP′]
2) S → S Conj S : [Conj′,S′,S′]
3) S → Neg S : [Neg′,S′]
4) VP → Vtrans NP : [V′,NP′]
5) VP → Vintrns : V′
6) NP → Name : Name′
7) VP → Vditrans NP NP : [V′,NP′,NP′]
8) NP → Det N : [Det′,N′]
9) NP → PN : PN′

There is one other change to this grammar involving rule 1). The subject NP is now treated as a function which takes the denotation of the VP as its argument. This is because we are treating the semantics of quantifiers as the semantic template into which the meaning of the rest of the sentence slots. (In fact, this means that there is a problem with treating non-subject NPs as arguments of verbs. However, we will ignore this until later when we consider the process building up a logical form through a series of function-argument applications in more detail.)

5.3 Logical Form

There is a difference between the way we have treated the semantics of the two artificial languages (logics) PL and FOL and the way we have treated the semantics of our two English fragments F1 and F2. For the former, we have given semantic rules which work out the truth of formulas relative to some model directly because the semantic rules are interpreted as instructions to check the model in various ways. For example, applying the predicate snore1 to the entity max1 is defined in FOL as a semantic operation which reduces to seeing whether max1 is a member of the set denoted by snore1. On the other hand, in specifying the semantics of F1 and F2 we have translated sentences of F1 and F2 into formulas which we have called logical forms or function-argument structures which themselves are equivalent to (ie. just notational variants of) formulas in FOL.

One way to think about this is to imagine that we are giving the semantics of English indirectly by first translating English into logic forms and then interpreting the resulting logical expressions in some model. The advantage of translating into FOL (or some similar logic) is firstly, that this representation is unambiguous (because of the brackets and the substitution of predicates and variables or constants for words) and secondly, that the semantics of, say, FOL is relatively clear-cut (compared to the semantics of English).

5.4 Scope Ambiguities

We saw that formulas of FOL containing more than one quantifier have different truth-conditional interpretations depending on the order or relative scope of these quantifiers. However, our grammar for F2 only produces one of these orderings for analogous sentences of F2. For example, (30a) only receives the interpretation (30b,c).

(30) a Every man loves some woman

b $\forall$ x man1(x) $\Rightarrow$ $\exists$ y woman1(y) $\wedge$ love1(x y))

c `[forall,X^[if,[man1,X],[exists,Y^[and,[woman1,Y],[love1,X,Y]]]]]`

can you verify this is the interpretation yielded by the rules above? This is the interpretation which is true provided for each man there is at least one woman (who may be different in each case) who he loves. However, to get the 'every man loves Marilyn Monroe' sort of reading we would need to reverse the order of the quantifiers so that the existential has (so-called) wide scope. Incidentally, if you find this reading a little forced for (30a), there are other examples where it seems more natural, as (31) illustrates.

(31) a Every man loves a woman
b Every man loves one woman
c One language is spoken by everyone (here)
d A student guide took every prospective candidate round the lab

In (31a) and b) it is easier to get the wide scope existential reading where there is just one woman, but most people still feel that this is not the preferred interpretation. In c), where the existential *one* occurs before *everyone* in the surface form of the sentence, the existential wide scope reading seems more readily available, whilst in d) it is definitely preferred. We adopt the existential wide scope reading of d) very readily because of our general knowledge about the likely routine for showing prospective new students aound – that one existing student takes them on a tour.

Further evidence that lexical semantic and general knowledge affects the quantifier scoping that we choose comes from the examples in (32).

(32) a There was a name tag near every door
b A flag was hanging from every window

In these examples, the existential *a* precedes the universal *every* in the surface realisation of these examples, yet the universal is given wide scope, and we naturally assume there is more than one flag or name tag. Presumably, we reach these conclusions on the basis of our knowledge about the relative size of flags and windows, name tags and doors, their function, and so forth.

Scope ambiguities of this type are not restricted to quantifiers. There are scope ambiguities in F1 concerning the relative scope of the negation, conjunction and disjunction operators. These also interact with quantifiers to create further ambiguities. For example, (33a) is ambiguous between b) and c) depending on the relative scope of *not* and *every.*

(33) a Everyone doesn't snore
b $\forall$ x $\neg$ snore1(x)
c $\neg\forall$ x snore1(x)

c) is compatible with some people snoring, whilst b) is not. In addition, there are scope ambiguities in connection with the interpretation of examples containing pronouns. In (34a) the preferred reading is the one in which *he* is treated as co-referential with *man*, but in (34b) it seems more natural to assume *he* does not have an antecedent within the sentence.

(34) a A man likes Belinda and he likes Felix (too)
b Every man likes Belinda and he likes Felix (too)

We can describe this difference in terms of whether the variable which translates *he* is in the scope of the quantifier translating *a* or *every.* Once again many factors seem to influence the calculation of pronominal reference. For example, stressing *he* in a) prevents the otherwise preferred interpretation, whilst in (35a) lexical semantics and general world knowledge play a role in calculating the antecedent of *their*.

(35) a The men allowed the women to join their club
  b The men allowed the women to found their club

Unlike the scope ambiguities between connectives and negation in F1, scope ambiguities involving quantifiers are not the result of syntactic ambiguity. There is no syntactic reason to believe that these examples are syntactically ambiguous, but nevertheless they are semantically ambiguous. This is a problem for the theory we have been developing because it predicts that sentences should only be semantically ambiguous if they are syntactically or lexically ambiguous. Thus we can produce two logical forms for (36a) and b) because we can associate *bank* with two concepts (financial institution and river side) and because PPs can function adverbially or adjectivally (attaching to NP or VP).

(36) a Max sat by the bank
  b Max hit the woman with the umbrella

We have tried to treat pronouns as lexically ambiguous, at the cost of subscripting F2, but it it is even harder to see how to treat quantifier scope as a lexical ambiguity. A lot of research has gone into this problem and there are several proposals as to how to produce sets of logical forms from a lexically and syntactically unambiguous sentence. The technical details of these proposals don't matter. What is important is that they weaken the claim that syntax determines interpretation and undermine our initial proposal that syntactic and semantic rules are paired one-to-one in the grammar.

5.5 Exercises

1) Interpret the following expressions of FOL in the model given underneath, show the process of interpretation by displaying the possible substitutions of entities for variables and then the contribution of each quantifier.

(37) a $\forall\ x\ A(x)$
  b $\exists\ x\ (A(x) \Rightarrow\ B(x\ b))$
  c $\forall\ x\ (A(x) \Rightarrow\ C(a\ x\ e))$
  d $\forall\ x\ \exists\ y\ (A(x) \Rightarrow\ B(a\ x\ y))$

```
Model:

I {a, b, c, d, e}
```

```
F (A) {a, b, c}

F (B) {<a b> <a c>}

F (C) {<a d e> <b d e> <c d e>}
```

2) Think of English sentences which the formulas in question 1) could be used to represent or 'translate'.

3) Say whether the following sentences are part of F2. If they are draw their syntax trees and give their logical forms. If not change them so that they retain the same meaning but are in F2 and then do the same. If any examples are ambiguous give all the possibilities:
a) Every woman loves Max and he loves Felix
b) It-is-not-the-case-that a dog snores and every cat smiles
c) Fred gives some woman a dog
d) Fred loves Belinda and likes Felix
e) Every man likes Belinda and Belinda likes her dog
f) No woman loves every man and every dog

4) Construct a model for F2 which makes all of the sentences of 3) true under at least one interpretation. Indicate which interpretation you are assuming by marking the appropriate logical form with an asterisk in your answer to question 3).

5) Can you find sentences which the grammar for F2 assigns incorrect logical forms or not enough logical forms as a result of scope ambiguities? Write them down and write down one other appropriate logical form not produced by the grammar.

6 Syntax-Directed Compositional Translation

In the previous sections, I cheated a bit over the processes involved in building up the logical form of sentences of F2 in tandem with performing a syntactic analysis. For F1, it was possible to characterise this process as one of function-argument application for all of the rules of the grammar. However, this doesn't work for F2, mainly because we included more complex NPs involving quantifiers. In this section, we won't extend the English fragment much, but will clarify the semantic rules which lie behind F2 in more detail.

6.1 The Typed Lambda Calculus

The Lambda Calculus (LC) is another artificial language like PL and FOL. (For historical reasons its more often called a calculus rather than a logic – the word 'calculus' emphasises the proof theoretic aspect of the language, but like FOL it (now) has a complete model-theoretic semantics). The variety we will look at is typed because variables range over particular syntactic categories or types of the language; for example one-place predicates or entity constants.

LC is an extension of FOL to include the lambda (λ) operator. This operator operates syntactically rather like a quantifier in FOL; for example, (38a) is a well-formed formula of LC in which the lambda operator binds the variable 'x'.

(38) a $\lambda\ x\ A(x) \wedge\ B(x\ a)$
b $\lambda\ x\ [A(x) \wedge\ B(x\ a)](b)$
c $A(b) \wedge\ B(b\ a)$

However, the interpretation of lambda expressions is rather different to that of expressions containing quantifiers. The lambda operator is a function forming device which can be used to compose existing functions to build one new composite, complex function. So (38a) should be read as 'the property of being simultaneously A and being in the B relation to a'. We can find constants to substitute for 'x' and thereby obtain a well-formed formula of FOL again. The process of doing this is called lambda reduction (or, more correctly, beta reduction). In b) we have written the same formula but indicating the scope of the lambda operator with square brackets. The constant in brackets outside the lambda expression is interpreted as the argument to the function expressed by the lambda formula and the process of applying the function to its argument we will call beta reduction. This results in the formula in c). The expressions in b) and c) are logically equivalent (truth-conditionally synonymous) because of the semantics of LC. Another way of saying this in terms of proof theory is to say that c) can be validly inferred from b).

To understand the semantics of beta reduction in model-theoretic terms we need to revise slightly the way we represent predicates. We have said that a one-place predicate such as snore1 denotes a set of entities and is a characteristic function from entities (domain) to truth-values (range). One way of writing this function down is as a lambda expression, as in (39a).

(39) a λ x [snore1(x)]
b λ x [snore1(x)] (max1)
c snore1(max1)

Now both b) and c) represent the application of that function to the entity max1. However, in order to be able to extract over n-place predicates to form new 'n minus 1'-place predicates we must interpret two and three -place predicates as functions from entities to functions from entities to truth-values and as functions from entities to functions from entities to functions from individuals to truth-values. (Got that?!) For example, we have thought of two-place predicates like love1 as a function which takes two individuals as arguments and yields a truth-value. However, we can also break it down into two functions, the first of which takes one entity as an argument and yields a new function which is exactly like the function associated with a one-place predicate. Thus, the domain of this function will be the set of entities and the range will be a set of (characteristic) functions. Assuming the following model, we can represent love1 as shown underneath:

```
I {max1, belinda1, fido1, felix1}

F (love1) {<max1 belinda1> <felix1 belinda1> <fido1 felix1>
           <felix1 fido1>}

love1:       1                2

max1        -->    max1      --> f
                   belinda1  --> f
                   fido1     --> f
                   felix1    --> f

belinda1    -->    max1      --> t
                   belinda1  --> f
                   fido1     --> f
                   felix1    --> f

felix1      -->    max1      --> f
                   belinda1  --> t
                   fido1     --> t
                   felix1    --> f

fido1       -->    max1      --> f
                   belinda1  --> f
                   fido1     --> f
                   felix1    --> t
```

The first function takes us from one entity to a new function from entities to truth-values. When we supply the second entity, the second function yields a truth-value. Using this technique, we can now see how the lambda operator is able to create new one-place predicates out of two-place predicates where one argument is fixed. So, for example, the lambda expression in (40a) is one way of expressing or referring to the function which appears to the right of the arrow after max1 above (assuming that it is interpreted in this model). Note that we assume that we always combine with the object NP before the subject NP.

(40) a λ x [love1(x max1)]
b λ x [love1(x max1)] (belinda1)
c love1(belinda1 max1)

So far we have used the lambda operator to create new functions which abstract over entities (ie. the variable bound by the lambda operator has ranged over the entities in the model). However, lambda abstraction can be applied to any category or type in the logical language; applying it to entities gives Second Order Logic, applying it to one-place predicates, Third Order Logic, two-place predicates Fourth Order Logic and so forth. We will want to apply lambda abstraction to some predicates in the new version of the grammar for F2, so the logical forms we will be constructing will be formulas of (at least) the third order typed lambda calculus.

Quite rapidly lambda expressions get hard to manipulate syntactically; for example, the formula in (41a) is equivalent to b) by beta reduction.

(41) a $\lambda\ x[A(x\ a) \vee\ B(x) \Rightarrow\ \lambda\ y\ [C(y\ b) \wedge\ \lambda\ z\ [D(x\ z) \wedge E(y\ c)](d)](e)](f)]$
b $A(fa) \vee\ B(f) \Rightarrow\ C(eb) \wedge\ D(fd) \wedge\ E(ec)$

In Prolog-style notation lambda bound variables are represented as `X^`$[\psi]$ and pulled to the front of formulas, as in (42).

(42) a
```
[X^[Y^[Z^[if,[or,[A,X,a],[B,X]],[and,[C,Y,b],[and,[D,X,Z],[E,Y,c]]]],d],e],f]
```
b
```
[if,[or,[A,f,a],[B,f]],[and,[C,e,b],[and[D,f,d],[E,e,c]]]]
```

6.2 Beta Reduction

Whenever possible, our strategy will be to use LC as a way of specifying the meaning of sub-expressions but reduce these via beta reduction to formulas of FOL for specifying the semantics of propositions. This is because we want to make use of these formulas with automated theorem provers and these are (mostly) restricted to first-order logic.

Beta reduction applied to a 'beta-redex' (a formula containing an argument to a lambda term) such as $\lambda\ x[P(x) \wedge\ Q(x)](a)$ is defined as the substitution of 'a' for all occurrences of 'x' in $P(x) \wedge\ Q(x)$. It is necessary to be careful about variable names, since it is possible to accidentally bind or substitute for variables with identical names in different formulas when performing reduction. For instance, reducing $\lambda P\ [\exists x\ P(x)]$ with $\lambda\ x\ R(x)$ should produce $\exists x\ \lambda x1\ [R(x1)](x)$ and not $\exists x\ \lambda x\ [R(x)](x)$ in which further lambda reduction would fail because all the variables would be bound by the existential quantifier. This problem can be avoided by assigning distinct names to variables in terms before reduction.

6.3 Types

We use a typed version of LC to avoid logical paradoxes and to keep the notion of a function within the bounds of standard set theory. For instance, in untyped LC it is possible to construct terms which denote things like 'the set of all sets which are not members of themselves' – $\lambda x\ \neg Member(x\ x)$ – and to ask questions like 'is this set a member of itself?' – $(\lambda x\ \neg Member(x\ x)\lambda x\ \neg Member(x\ x))$. If it is, then it is a set which is a member of itself and if it isn't, then it is a set which is a not a member of itself, and so should already be part of its own denotation. Typing prevents such paradoxical results.

An extensional type system can be built up from the primitives 'e' for entity and 't' for truth-value. So an entity constant or variable is of type 'e' and a proposition of type 't'. The type of everything else follows from this. For example, a one-place predicate is of type `<e t>` or `e --> t` because it is a function from an entity to a truth-value; a two-place predicate is a function from an entity to a function from an entity to a truth value; ie. `<e <e t>>` or `(e --> (e --> t))`. So if we are being careful, we should explicitly type all the lambda bound variables in a LC formula rather than relying on loose conventions like 'x','y' or 'X', 'Y' are entity variables and 'P' and 'Q' are x-place predicate variables. However, it all gets a bit verbose, as (43) indicates, so a lot of the time we'll suppress the types BUT it is important to check for typing, at least implicitly, when constructing typed LC formulas.

(43) $\lambda P_{<et>} \lambda x_e P(x)$

6.4 Rule-to-rule Translation

Armed with typed LC, we can now specify more precisely the manner in which logical forms are built up compositionally in tandem with the application of syntactic rules in F2. For example, consider the analysis of *Max snores* again. The syntactic analysis of this sentence requires the following three rules of F2:

1) S $\rightarrow$ NP VP : [NP$'$,VP$'$]

2) VP $\rightarrow$ V : `X^`[V$'$,X]

3) NP $\rightarrow$ Name : `P^`[P,Name$'$]

These rules are similar to those used in the grammars of F1/2 except that we have specified the semantic part of the rules in more detail using the notation of LC. The denotation of *snores* is a set of entities (written snore1). When we apply rule 2 to *snores* to obtain the partial syntactic analysis shown in a) below, we also instantiate the variable in the LC formula paired with this rule to form the semantic interpretation of *snores*, as in b):

```
a)      VP                     b) X^[snore1,X]
        |
        V

      snores
```

This lambda expression is an instruction to form a function over the denotation of snore1 from entities to truth-values. The denotation of *Max* is max1, so the analysis of the subject NP proceeds in a similar way to produce:

```
a)     NP                      b) P^[P,max1]
       |
       Name
       |
       Max
```

However, the semantic rule associated with NPs is an instruction to form a function which abstracts over the set of properties predicated of max1 (ie. the set of functions representing one-place predicates). This more complex account of the denotation of an NP is needed so that NPs containing either proper names or determiners (esp. quantifiers) and nouns will end up with the same denotation and can therefore be combined with predicates using the same semantic rules. Thus in F1, where all NPs are proper names, the denotation of an NP is always an entity constant. Therefore, the semantic rule combining NPs and VPs is function-argument application, where the function associated with the VP is a characteristic function which checks for membership of the argument in the denotation of the verb and returns a truth-value. However, in F2 this won't work because NPs like *every man* cannot denote entities. Instead, we make NPs denote functions from characteristic functions to truth-values. The only difference between proper name NPs and quantified NPs will be whether the lambda abstracted set of properties is predicated of an entity constant or (bound) variable.

Since this treatment of NPs is more complex, we'll try to make things clearer with a concrete example. Assuming the model given below, the denotation of the NP *Max* will be the function shown underneath:

```
I {max1 belinda1}

F (snore1) {max1}

F (love1) {<max1, belinda1>}
```

```
F (like1) {<belinda1 max1>}

NP(max1)

{
snore1:
                           --> t

love1:
               max1        --> f
               belinda1    --> t

like1:
               max1        --> f
               belinda1    --> f
                       }
```

In other words, the denotation of the NP *Max* contains the set of possible properties which can be predicated of max1 in this model, such as snoring, liking himself, loving Belinda etc., and the lambda operator creates a function from these VP (one-place predicate) denotations to truth-values.

Combining the denotation of the NP and VP above using rule 1) produces the syntactic and semantic analysis shown in a) and b) below:

```
a)          S              b)  [P^[P,max1],X^[snore1,X]]
        /       \
    NP            VP           [X^[snore1,X],max1]
    |             |
    Name          V            [snore1,max1]
    |             |
    Max           snores
```

The formulas in b) are logically equivalent, the third is the result of applying beta reduction (twice) to the first. Here beta reduction is like 'merging' the two lambda expressions by matching the (typed) variables 'P' and 'X' to expressions of the appropriate type. In fact, what we have done here is compose two functions – one abstracting over predicates, the other over entities – to produce a formula equivalent to the one we got in F1 for *Max snores* by treating max1 as an argument of the one-place predicate snore1.

This may seem like a lot of extra complexity for nothing, but this analysis is essential when we move on to sentences like *Every man snores*. To analyse this sentence, we need the other NP rule given below:

4) NP → Det N : [Det′,N′]

This rule applies the denotation of the determiner to that of the noun. In section 5.2.1, we anticipated the use of LC by describing the meaning of *every* as the 'template' for a logical form in (44a).

(44) a `[forall,X^[if,[P,X],[Q,X]]]`
     b `P^[Q^[forall,X^[if,[P,X],[Q,X]]]]`

The correct way to write this in LC is as a lambda abstraction over two one-place predicates, as in (44b). So the analysis of *Every man* will come out like this:

```
a)     NP      b) [P^[Q^[forall,X^[if,[P,X],[Q,X]]]],X^[man1,X]]
    /   \

  Det      N      Q^[forall,X^[if,[X1^[man1,X1],X],[Q,X]]]
  |        |
 Every   man      Q^[forall,X^[if,[man1,X],[Q,X]]]
```

By two applications of beta reduction, we obtain the third formula. Now when we analyse *snores* we get:

```
a)       S          b) [Q^[forall,X^[if,[man1,X],[Q,X]]],Y^[snore1,Y]]
      /    \

  NP         VP         [forall,X^[if,[man1,X],[snore1,X]]]
 /  \        |
Det   N      V
|     |      |
Every man    snores
```

Using the apparatus of LC, we are able to get the different logical forms for quantified and unquantified NPs, but at the same time achieve this without needing different semantic rules for combining different types of NPs with VPs. This is essential if we are going to have an account of semantic productivity. In effect, using LC and raising the type of NPs allows us to maintain compositionality but provides more flexibility in the way we semantically combine the meanings of words and phrases.

6.5 Types revisited

We can now give a more precise account of the relationship between syntactic and semantic rules and categories in the grammar. We can associate with each syntactic category of the grammar, a semantic type which specifies the type of object denoted by expressions of that category and also specifies how it combines with other types. For example, we have said that proper names denote entities (i.e. any discrete object – car1, table1, max1 etc.). We can write this `<e>`. Sentences denote propositions (ie truth-values) written `<t>`, intransitive verbs denote functions from entities to truth-values `<e t>`, NPs denote functions from functions from entities to truth-values to truth-values `<<e t> t>`. Given this new notation we could represent the semantic analysis of *Every man snores* given above slightly more abstractly as below:

```
a)      S                b)                    t
      /    \                                 /       \
    NP      VP                  <<e t> t>           <e t>
   /  \     |                      |      \           |
Det    N    V      <<e t> <<e t> t>>   <e t>       <e t>
|      |    |                  |           |           |
Every man  snores            all         man1       snore1
```

The type notation illustrates how the generic functions combine or cancel with each other to yield new functions and ultimately a truth-value. Once we know the semantic type of every syntactic category, we know how to combine them to form the semantic types of larger categories. Of course, when interpreting an actual sentence, we replace the semantic types with the functions derived from the denotations of the words in the sentence in the model in which we are performing the interpretation (ie. we replace types with tokens).

6.6 English Fragment 2 Redone

We will now rewrite the grammar for F2, making use of LC to specify more precisely the form of the semantic rules and giving the semantic type of each syntactic category in the grammar.

Lexicon

The only changes to the lexicon concerns the semantics of the quantifiers, which are now written as double lambda abstractions over two one-place predicates – the noun and VP denotations, respectively, and of the pronouns, where we now write the 'externally' bound version as a lambda abstracted entity variable.

```
Max : Name : max1
Fred : Name : fred1
Belinda : Name : belinda1
Fido : Name : fido1
Felix : Name : felix1

he_x : PN : P^[X^[and,[male1,X],[P,X]]]
```

```
he : PN : P^[exists,X^[and,[male1,X],[P,X]]]
her_x : PN : P^[X^[and,[female1,X],[P,X]]]
her : PN : P^[exists,X^[and,[female1,X],[P,X]]]
himself_x : PN : P^[X^[and,[male1,X],[P,X]]]
herself_x : PN : P^[X^[and,[female1,X],[P,X]]]

and : Conj : and
or : Conj : or
it-is-not-the-case-that : Neg : not

snores : Vintrans : snore1
smiles : Vintrans : smile1
likes : Vtrans : like1
loves : Vtrans : love1
gives : Vditrans : give1

a : Det : P^[Q^[exists,X^[and,[P,X],[Q,X]]]]
no : Det : P^[Q^[not,[exists,X^[and,[P,X],[Q,X]]]]]
some : Det : P^[Q^[exists,X^[and,[P,X],[Q,X]]]]
every : Det : P^[Q^[forall,X^[if,[P,X],[Q,X]]]]

man : N : man1
woman : N : woman1
dog : N : dog1
cat : N : cat1
```

Grammar for F2

1) S → NP VP : [NP′,VP′]
2) S → S Conj S : [Conj′,S′,S′]
3) S → Neg S : [Neg′,S′]
4) VP → Vtrans NP : [NP′,`Y^[X^`[V′,X,Y]]]
5) VP → Vintrans : `X^`[V′,X]
6) NP → Name : `P^`[Name′]
7) VP → Vditrans NP NP : [NP'_2,[NP'_1,`Z^[Y^[X^`[V′,X,Y,Z]]]]]
8) NP → Det N : [Det′,N′]
9) NP → PN : `P^`[PN′]

The changes from the old version just involve the semantics of the VP rules. These are now written as lambda expressions. For example, the semantic rule associated with 5) says that the meaning of an intransitive verb is a lambda abstraction over whatever the denotation of that intransitive verb is to form a function from entities to truth-values. The semantic rule for 4) says that the denotation of the transitive verb should be the argument to a function from entities to a second function from entities to truth-values. Since the semantic type of the NP will be a lambda expression denoting a function from VP type functions to further functions, the resulting formula is quite complex. However, after beta reductions it reduces to the appropriate FOL logical form.

6.7 Pronouns and Quantifier Scoping Revisited

The LC treatment of quantifiers in F2 does not resolve the problem of how to get wide-scope existential readings of examples like *Every man loves one woman.* Can you show this? Nor does treating pronouns as ambiguous between existientially-bound for diectic uses and lambda bound for coreferential uses get us that far. Concentrating again on the coreferential cases like *A man$_i$ likes Belinda and he$_i$ likes Felix (too)*, the rules of F2 will not actually reduce the lambda bound variable translating *he* so that it is bound by the existential associated with *A*. Can you prove this to yourself? The problem is that the semantics of the first conjunct gets ‘closed off’ before the second is interpreted. For this reason (and others), several semantic theories (DRT, Dynamic Semantics: see eg. Bach) have explored variant logics which allow the scope of quantifiers to be kept open as the interpretation of a sentence is built up incrementally. (More on this next term.)

Since we’ve spent some time looking at the problem scope ambiguities raise for syntax-directed semantic interpretation, you might wonder why we are explaining in more detail exactly how syntactic and semantic rules are paired together. However, if we gave up organising the grammar in this fashion we would be in danger of losing our account of semantic productivity. We want to show how the meaning of words and phrases are combined using general rules. The apparatus of LC is likely to be essential to this enterprise because it provides a very flexible way of specifying the denotations and modes of combination of linguistic expressions. Therefore, we have a chance of coming up with a general rule which states how **all** NPs and VPs combine, say; and not one rule for NPs containing proper names, another for quantified NPs, another for NPs with pronouns, another for NPs with relative clauses, and so forth. It may be that ultimately, we want to make these rules sensitive to more than just the syntactic analysis of these linguistic expressions, but we will still want a compositional account of the meaning of words, phrase, clauses, and sentences; even if the semantic rules are sensitive to, say, the presence of specific words or intonation, and aspects of the wider (non-)linguistic context too.

6.8 Exercises

1) Write down the logically equivalent first order formulas corresponding to the following lambda expressions by performing as many beta reductions as are necessary:
a) $\lambda x\ [M(x) \wedge\ L(x\ b)](m)$
b) $\lambda x\ [\lambda y\ [L(x\ y) \vee\ H(y\ b)](m)](f)$

c) $\lambda x\ [\forall y\ M(y) \Rightarrow\ L(x\ y)](f)$

2) Convert the following formulas to equivalent lambda expressions by abstracting over the entity or predicate listed after them (ie. perform the opposite of beta reduction on them):
a) $M(m) \wedge\ L(mb)$ (abstract over 'm')
b) $\forall x\ M(x) \Rightarrow\ L(x\ b)$ (abstract over 'M')
c) $\exists x\ M(x) \wedge\ L(x\ b)$ (abstract over 'M' and 'b')

3) If we have a model M containing a set of entities E and the model contains the following set { x : x snore1} (ie. the set of entities which snore), then we can represent the same set as a function from entities to truth-values. Write this function as a lambda expression.

4) I defined love1 (a two-place predicate) as a function from entities to a function from entities to truth-values. This can be expressed as a lambda expression – λ x [λ y [love1(x y)]] – and, given a model, as two functions mapping from the entities in the model to more entities and finally a truth-value. Construct a model which includes denotations for like1 and give1. Then define these functions by exhaustively listing the possibilities in your model.

5) It would be nice to get rid of *it-is-not-the-case-that* from our English fragment (since it isn't English!) and replace it with *not* or *n't*. To do this we need to extend the fragment in several ways to cover examples like:

Max doesn't snore
Max doesn't love Belinda

We need an entry for *doesn't* and new syntactic rules for analysing VPs containing this word. Lets assume the following entry and rule:

doesn't : Aux : `P^[X^`[not,[P,X]]]

and the following PS rule:

VP $\rightarrow$ Aux VP : Aux$'$(VP')

see if you can work out how this works and check that you understand by adding the entry and rule to the grammar for F2 above and parsing these sentences.

7 Conclusions

The best way to do semantics is to specify the 'translations' of sentences into a logic with a model theory and proof theory. We have considered the semantics of a small fragment of English (which we have only dealt with partially, ignoring e.g. tense/aspect). We can construct these translations compositionally so that the meaning of a sentence is a function of the meaning of its constituents. We can implement such a compositional semantics in CFG.