Do the story one sentence at a time. Do not try to get everything to work at once. The rules you add should involve the relation between ATRANS and possession, between possession of a weapon and plans for getting money, between plans for getting money and being at a suitable place, between plans for getting money and carrying out a threat, between MTRANS and someone knowing something, and so forth.

4. In this version of McPAM, when you want to give several subgoals to a plan, you have to write separate rules for each subgoal, repeating the plan each time. A better way would be to allow a rule to have more than one pattern-expression pair in the right hand side, and each would specify one subgoal of the plan on the left hand side. Modify McPAM to do this by modifying RHS, RELATE and TRY-RULES to handle more than one pair.
The adaptation of the original theory was not without its differences, even at the level of plans. The plan for communication, DKNOW, in their theory encompasses a variety of communication tasks. In TALE-SPIN, there is a clear distinction between transmitting and acquiring information, and they are described by independent procedures.

I also merged the problem-solving knowledge with information about social relationships and personal characteristics, principally as preconditions to acts of persuasion. The "scales" I used derived from some of the work of Myron Wish (1975), which was extended to describe the more specific requirements of the stories the program was writing.

The second largest part of TALE-SPIN, after the problem-solving component, is the English-language generator. Originally, I used a version of Goldman's BABEL (Goldman, 1975), but the match was not ideal, and in the standard but unfortunate tradition of A.I., I wrote my own from scratch. Gradually, it relied more and more on the memory (chronological record), so that it would say "Joe Bear returned to his cave" instead of "Joe Bear went to the cave," and it knew when to say "Irving Bird wasn't hungry any more" as opposed to the simpler "Irving Bird wasn't hungry."

The inference mechanism began with the simple, "context-free" consequences that are part of the definition of the primitive acts of Conceptual Dependency. These were extended to include three new areas: the social-state knowledge mentioned above; the effects of "noticing" whenever anyone moved from one place to another (unfortunately but unavoidably, a large source of simple inferences); and "reactions", or the inferences from MLOC, nearly equal in number to all the inferences from the rest of the system.

Of the many domains that might have been investigated, the representation of physical space seemed difficult to postpone, and the resulting theory and implementation of "maps" and "blueprints" most closely resemble Minsky's definition of frames. This information was used when new characters and new settings were generated; e.g., if something is a house, then it has a front door, and may have an upstairs bedroom, and so on. Since the characters in TALE-SPIN are talking animals and humans, the world blueprint describes meadows and valleys and the like. They are created on demand (when a journey takes place).

Finally, TALE-SPIN reflects a theory of stories that is easy to state: stories are both coherent and interesting, and they span a number of levels. To explain that statement, let me first describe the overall control for the simulator.

When an event occurs, it is "asserted", which means that it is recorded and all its consequences are computed and likewise asserted. One kind of event is called a goal statement, such as "John wants to visit Mary." When a goal statement is asserted, the corresponding problem-solving procedure is called, i.e., one for the goal of transportation (DPROX), one for the goal of communication (DKNOW), and so on. These procedures invoke subgoals and eventually produce new events, which are then asserted, continuing the cycle. The simulation begins by establishing a "setting" and a cast of characters. One of the characters is given a problem to solve. In the course of solving that problem, the character may interact with other characters. If one character persuades another to do something, then the second character goes about solving problems. The simulator keeps track of all the characters.

This is the model of coherence—the behavior of the characters should be rational—and it provides the test for the theories of problem solving. But beyond that, stories must be interesting. In TALE-SPIN, an interesting story sets up a particular problem and a focal problem domain called the domain of interest. The story is "about" solving that problem. The details of the solution should pertain to the domain of interest, and the problem should not be too easily solved. The level of the domain of interest is what distinguishes a simple adventure story from a work such as Conrad's Heart of Darkness. Left to its own devices, as it were, the simulator will create a low-level story about characters driven by very basic needs such as hunger, and the similarity between these stories and actual folk tales has been noted (de Beaugrande and Coby, 1979). But in order to tell a higher-level story, the simulator requires that some of the setting be established ahead of time, particularly relationships between characters so that their predicted behavior corresponds to the needs of the story. In this way, TALE-SPIN is able to generate stories with a "point", such as the simpler of the Aesop fable.

Examples

Here is an example of a TALE-SPIN story. After an initial dialogue with the user, establishing the characters and the physical setting, the story begins when each of the characters is made thirsty. The story required no information about the social relationships, because all characters are motivated to rescue anyone in danger of death.

Most of the "resultant" inferences have been omitted from the output for brevity. Paragraphs have been added here to indicate separate stories that use the same set of characters and locations.

ONCE UPON A TIME GEORGE ANT LIVED NEAR A PATCH OF GROUND. THERE WAS A NEST IN AN ASH TREE. WILMA BIRD LIVED IN THE NEST. THERE WAS SOME WATER IN A RIVER. WILMA KNEW THAT THE WATER WAS IN THE RIVER. GEORGE KNEW THAT THE WATER WAS IN THE RIVER. ONE DAY WILMA WAS VERY THIRSTY. WILMA WANTED TO GET NEAR SOME WATER. WILMA FLEW FROM HER NEST ACROSS A MEADOW THROUGH A VALLEY TO THE RIVER. WILMA DRANK THE WATER. WILMA WAS NOT THIRSTY ANY MORE.

GEORGE WAS VERY THIRSTY. GEORGE WANTED TO GET NEAR SOME WATER. GEORGE WALKED FROM HIS PATCH OF GROUND ACROSS THE MEADOW THROUGH THE VALLEY TO A
RIVER BANK. GEORGE FELL INTO THE WATER. GEORGE WANTED TO GET NEAR THE VALLEY. GEORGE COULDN'T GET NEAR THE VALLEY. GEORGE WANTED TO GET NEAR THE MEADOW. GEORGE COULDN'T GET NEAR THE MEADOW. WILMA WANTED GEORGE TO GET NEAR THE MEADOW. WILMA WANTED TO GET NEAR GEORGE. WILMA GRABBED GEORGE WITH HER CLAW. WILMA TOOK GEORGE FROM THE RIVER THROUGH THE VALLEY TO THE MEADOW. GEORGE WAS DEVOTED TO WILMA. GEORGE OWED EVERYTHING TO WILMA. WILMA LET GO OF GEORGE. GEORGE FELL TO THE MEADOW. THE END.

The two relationship states at the end are "demon" inferences—George would have reacted that way to anyone who pulled him out of the water.

Here's an example showing "verbose mode" where every inference is printed. The relationships between the characters were established by a dialogue with the user, which has been omitted.

ONCE UPON A TIME JOHN BEAR LIVED IN A CAVE. JOHN KNEW THAT JOHN WAS IN HIS CAVE. THERE WAS A BEEHIVE IN A MAPLE TREE. TOM BEE KNEW THAT THE BEEHIVE WAS IN THE MAPLE TREE. TOM WAS IN HIS BEEHIVE. TOM KNEW THAT TOM WAS IN HIS BEEHIVE. THERE WAS SOME HONEY IN TOM'S BEEHIVE. TOM KNEW THAT THE HONEY WAS IN TOM'S BEEHIVE. TOM HAD THE HONEY. TOM KNEW THAT TOM HAD THE HONEY. THERE WAS A NEST IN A CHERRY TREE. ARTHUR BIRD KNEW THAT THE NEST WAS IN THE CHERRY TREE. ARTHUR WAS IN HIS NEST. ARTHUR KNEW THAT ARTHUR WAS IN HIS NEST. ARTHUR KNEW THAT JOHN WAS IN HIS CAVE. JOHN KNEW THAT ARTHUR WAS IN HIS NEST. JOHN KNEW THAT TOM WAS IN HIS BEEHIVE. ARTHUR KNEW THAT THE BOYSENBERRIES WERE NEAR A BUSH. THERE WAS A LILY FLOWER IN A FLOWERBED. ARTHUR KNEW THAT THE BOYSENBERRIES WERE NEAR THE BUSH. JOHN KNEW THAT THE LILY FLOWER WAS IN THE FLOWERBED.

ONE DAY JOHN WAS VERY HUNGRY. JOHN WANTED TO GET SOME HONEY. JOHN WANTED TO FIND OUT WHERE THERE WAS SOME HONEY. JOHN LIKED ARTHUR. JOHN WANTED ARTHUR TO TELL JOHN WHERE THERE WAS SOME HONEY. JOHN WAS HONEST WITH ARTHUR. JOHN WASN'T COMPETITIVE WITH ARTHUR. JOHN THOUGHT THAT ARTHUR LIKED HIM. JOHN THOUGHT THAT ARTHUR WAS HONEST WITH HIM. JOHN WANTED TO ASK ARTHUR WHETHER ARTHUR WOULD TELL JOHN WHERE THERE WAS SOME HONEY. JOHN WANTED TO GET NEAR ARTHUR. JOHN WALKED FROM A CAVE EXIT THROUGH A PASS THROUGH A VALLEY THROUGH A MEADOW TO THE GROUND BY THE CHERRY TREE. JOHN WAS NEAR THE GROUND BY THE CHERRY TREE. JOHN KNEW THAT JOHN WAS NEAR THE GROUND BY THE CHERRY TREE. JOHN KNEW THAT ARTHUR WAS IN HIS NEST. ARTHUR KNEW THAT JOHN WAS NEAR THE GROUND BY THE CHERRY TREE. JOHN ASKED ARTHUR WHETHER ARTHUR WOULD TELL JOHN WHERE THERE WAS SOME HONEY. ARTHUR DIDN'T HAVE MUCH INFLUENCE OVER JOHN. ARTHUR REMEMBERED THAT ARTHUR LOVED JOHN. ARTHUR WAS HONEST WITH JOHN.

At this point, Arthur has decided to tell John where some honey is. This is just the first third of the story. The rest tells how John Bear goes to Tom Bee's hive to get some honey and he refuses. The point to be made by this is that TALE-SPIN is not putting words together to make a story. It has a very complex model of a small multi-actor world, and the sentences that are finally printed show just a tiny fraction of what the program really knows.

Summary of Program Behavior

The program can currently create eight different kinds of characters that are directly coded, and forty-six kinds of physical locations (e.g., meadows, bridges) that are data-driven (constructed from abstract descriptions). There are explicit problem-solving procedures for transportation (DPROX), acquisition of objects (DCONT), acquisition of information (DKNOW), transfer of information (TELL), persuasion (PERSUADE), bargaining (BARGAIN), and asking favors (ASK). The standard set of CD primitives was extended to include the acts *PLAN* and *WANT*, and thirteen "social" states were used (e.g., honesty, affection). There are forty-one inference procedures.

The generator uses a vocabulary consisting of fifty verbs in addition to the nouns for every object and several adjectives (or adjectival phrases) for every state in the system (e.g., for the "energy" scale, the generator can choose from: wiped out, exhausted, tired, rested, lively, energetic, hyper). It uses past, present, and future tenses, negations, contractions, infinitive phrases ("Joe asked living to tell him..."), interrogatives (who, what, where), conditionals (whether), modals (can, might, would), and "mass" nouns (e.g., "some" honey, not "a" honey).

In addition to the domains already described, TALE-SPIN includes a symbolic arithmetic package, where all numerical concepts are represented as points in a directed graph, so that partial-order comparisons can be made without assigning numerical values. This is used, for example, in representing the time at
which an event occurs; we need to compare the relative times of two events to generate the proper tense in the English output.

One of the enduring points in TALE-SPIN’s favor is that it provides, conceptually, a wealth of “small” topics to be invented or extended. That is, if you wish to improve the stories by improving the representation of physical space, making travel-distance a planning precondition, for instance, it’s not difficult to find out where to add that code. The generator’s vocabulary and range of grammatical constructions are easily extended. “Large” topics, of course, like redesigning the top-level control, are not easy.

The major drawback to TALE-SPIN is its size, not surprisingly. When it ran at Yale under Stanford LISP 1.6 (Quam, 1969), in 1976, it required 75K (PDP-10 words) to load, and an additional 25K would make the response reasonable. Today, TALE-SPIN is entirely compiled and runs at UCI under UCI LISP (Meehan, 1979) requiring about 90K to load. Part of that increase is due to the additional facilities in UCI LISP, but most of it comes simply from the extended capabilities of the system. Both versions of TALE-SPIN were written in MLISP (Smith, 1970), an ALGOL-like language that is converted by a pre-processor into LISP code for execution.

The system is not particularly robust. Some errors are detected by the program itself, which prints a message and waits for a response. Some of these are ignorable. When the generator cannot produce a translation, its “error message” is “Uh... Uh... mumble”, a phrase borrowed from Goldman’s BABEL. (In fact, TALE-SPIN’s generator is called MUMBLE.) But other messages, unfortunately, are meaningless to anyone but me. Errors in the internal syntax of CD expressions, for example, or contradictions in the time-net (A is before B and B is before A), still occur. In long stories, one can encounter the familiar resource limitations. My favorite error message occurred in the middle of generating a sentence: JOE BEAR KNOWS THAT FREE STG EXHAUSTED.

Issues

TALE-SPIN has three main theoretical concerns:

1. **Plans and goals**. The program shows a new side to the Schank-Abelson theory of intentionality, the principal exposition for which has always been in the comprehension, rather than the generation, of text. The theory, for example, doesn’t specify the rules for acquisition or rejection of new plans and goals (sometimes referred to as the goal calculus), nor is much said about the connection between certain social relationships and various problem-solving techniques—issues that I was forced to face in building the system.

2. **The nature of stories**. TALE-SPIN explores the connection between storytelling and problem solving by defining a story’s impetus in terms of characters who are in pursuit of various goals. Even starting with such low-level goals as hunger and thirst, it takes very little to produce a story involving trust, affection, good intentions, and the lack of same. Stories are certainly more than problem-solving narratives, but they may not be less.

3. **Simulation as a form of cognition**. At first, I thought that the difficulties in managing a simulation were primarily issues of programming (how to imitate parallelism, for example). But once I had the characters doing simulations themselves as a technique for planning ahead, I began to believe in simulation as a form of memory, in which case the rules for controlling it become far more significant, and programming tricks must be abandoned.

Much remains to be done in these three areas. The goal calculus has yet to be worked out, and every time the theory of goals changes (usually by expansion), the old calculus requires a major overhaul. For example, how many “background” goals do you have? How are they organized? Without answers to such questions, reasonable-sounding calculus heuristics like “Compare a new goal to the background goals to decide whether to pursue it” are too vague.

Good stories are coherent and interesting on many levels of meaning—the more, the better. This may be the harshest of the three areas to develop, because it presumes that we know how to represent meaning on many levels, and in particular, that the lower levels are simpler. The difficulty I see is that those lower levels require either a great engine to do all the busy-work of computing the obvious consequences of, say, every physical action, or some very smart processor that ignores all those details that we know to be unnecessary. If only we knew how we know they’re unnecessary!

An immediate problem with using simulation as a form of memory is knowing how not to get bogged down with the details that will arise during the actual execution of a plan, and how to make intelligent guesses in the face of incomplete information.

OVERVIEW OF TALE-SPIN

An Example of the Program in Question

We will begin our discussion of the inner workings of TALE-SPIN with a look at what the program does in an interactive session with it.

The upper-case text is from the program. The comments (in lower-case) describe how the program proceeds. We begin with a dialogue to establish the characters.
(START)

********** WELCOME TO TALE-SPIN **********

CHOOSE ANY OF THE FOLLOWING CHARACTERS FOR THE STORY:
1: BEAR 2: BEE 3: BOY 4: GIRL 5: FOX
6: CROW 7: ANT 8: CANARY

*1 2

By answering with 1 and 2, we ask the program to make up a bear and a bee. Their names and sexes (except for 3 and 4) are chosen at random. When characters are created, some local setting is included: their homes, and some objects likely to be found there. Initially, all characters start out at home. The narrative is told in past tense.

ONCE UPON A TIME SAM BEAR LIVED IN A CAVE. SAM KNEW THAT SAM WAS IN HIS CAVE.

All characters are made aware of their initial locations.

THERE WAS A BEEHIVE IN AN APPLE TREE. BETTY BEE KNEW THAT THE BEEHIVE WAS IN THE APPLE TREE. BETTY WAS IN HER BEEHIVE. BETTY KNEW THAT BETTY WAS IN HER BEEHIVE. THERE WAS SOME HONEY IN BETTY'S BEEHIVE. BETTY KNEW THAT THE HONEY WAS IN BETTY'S BEEHIVE. LETTY HAD THE HONEY. BETTY KNEW THAT BETTY HAD THE HONEY.

The model for bees is that they live in beehives and "own" honey. The model for beehives specifies that they are found in trees, so the creator adds all of these items to the simulated world.

The sentences beginning "Betty knew that..." are included to show that the program is storing that information, to be used in solving problems and making inferences.

—DECIDE: DOES BETTY BEE KNOW WHERE SAM BEAR IS? *NO
—DECIDE: DOES SAM BEAR KNOW WHERE BETTY BEE IS? *YES
SAM KNEW THAT BETTY WAS IN HER BEEHIVE.
SAM KNEW THAT BETTY HAD THE HONEY.

We get to decide whether the characters know about each other.

—DECIDE: DO YOU WANT ANY OF THESE IN THE STORY?
(BREADCRUMBS CHEESE BASEBALL) *NO

These are props that can be included and given to the characters, but in this case we decided not to use any of them.

—DECIDE: DO YOU WANT ANY OF THESE IN THE STORY?
(BERRIES FLOWER WATER WORM) *YES
WHICH ONES?
1: BERRIES 2: FLOWER 3: WATER 4: WORM
*2
THERE WAS A ROSE FLOWER IN A FLOWERBED. WHO KNOWS ABOUT THE ROSE FLOWER?
1: BETTY BEE 2: SAM BEAR
*2
SAM KNEW THAT THE ROSE FLOWER WAS IN THE FLOWERBED.

These props can be created, too, but they don’t belong to any of the characters. Instead, we get to decide who knows where they are.

THIS IS A STORY ABOUT...
1: BETTY BEE 2: SAM BEAR
*2
HIS PROBLEM IS THAT HE IS...
1: HUNGRY 2: TIRED 3: THIRSTY 4: HORNY
*1

The last thing we need in the initial setting is a "main character" and a problem for that character to begin working on. The choice of initial problems is the set of S-goals (cyclic goals of satisfaction).

ONE DAY SAM WAS VERY HUNGRY.

From this point on, after we assert that Sam is hungry, the control is out of our hands. The simulator takes over, computing consequences and invoking the planning procedures. The procedure now running is S-HUNGER, and it calls DCONT so that Sam will get some food. The list of foods bears eat is stored, and honey is chosen at random.

SAM WANTED TO GET SOME HONEY.

DCONT calls DKNOW to have Sam find out where there’s some honey. DKNOW first checks to see whether Sam already knows where some honey is. The pattern matcher matches the non-specific honey he’s looking for with the particular honey that Betty owns, since Sam knows about that. Had he not known already, DKNOW would have started its own procedures for finding out information (e.g., ask a friend).

HOW HONEST WAS SAM? *NOT AT ALL.
SAM WAS DISHONEST.
HOW VAIN DID SAM THINK THAT BETTY WAS? *NOT AT ALL
SAM THOUGHT THAT BETTY WAS HUMBLE.
DCONT's first technique is to ask whether the honey is free for the taking. It does that by searching the memory to see whether Sam thinks that anyone owns the honey. Of course, Betty does, so the honey isn't free, and that technique won't work.

The second technique is simply to persuade the owner to hand over the honey, but before we do that, we consider whether a trick might work here. The particular trick works only if the person who wants the honey is dishonest and the owner is vain, and it involves flattery. The memory doesn't say anything about those characteristics of Sam and Betty, so the system asks us directly. The question is written out by the English generator. The answer is matched against a list of canned phrases such as LOTS, NOT MUCH, SOMEWHAT, and so on. In any case, the first condition was met, but not the second, so we give up the idea of that trick and consider simple persuasion.

SAM WANTED BETTY TO GIVE SAM THE HONEY.
HOW DECEPTIVE DID SAM FEEL TOWARDS BETTY? *VERY
SAM WAS INCLINED TO LIE TO BETTY.

The "persuade package" contains four techniques for getting someone to do something for you. You can ask, you can give them a good reason, you can bargain, or you can threaten. If it is your intent to deceive that person, however, you won't ask or bargain. The search for a good reason will be described in detail later; it was not successful here.

HOW MUCH DID SAM LIKE BETTY? *LOTS
SAM LOVED BETTY.
SAM DECIDED THAT BETTY WOULDN'T GIVE SAM THE HONEY.

The fourth planbox (threaten) won't work if you like the person, and that's the last option, so this whole attempt to persuade Betty to give Sam the honey fails. But the "persuade package" was only one of the ways DCONT knows about. The next planbox in DCONT is theft—if you want something, steal it. Before we can use this option, of course, we need to know whether Sam is honest. His dishonesty was established earlier, so we don't need to ask the question again. The plan for theft is very simple: get the owner away from the object, and take it.

SAM WANTED BETTY TO FLY AWAY FROM HER BEEHIVE.

The procedure now active is D-NEG-PROX, where the goal is not to get Betty to some particular place, but rather to get her away from her beehive. There are two techniques in D-NEG-PROX. The first is to persuade Betty to go by herself. The second is to use DPROX to get Betty to some nearby area, using any applicable technique known to DPROX, including simply pulling her out. We start with the persuasion.

The first persuasion technique is a simple request, but it's blocked by the fact that Sam intends to deceive Betty. The second technique is to find a good reason for Betty to leave. The task is to find some goal of Betty's that would be aided by having her leave the beehive; that is, we want to find a causal chain that links leaving the hive to achieving one of her goals. The search is done breadth-first and starts by considering the goals Betty can always be assumed to have. The two goals that bees have in TALE-SPIN are hunger and rest, and bees 'eat' flowers.

The inference mechanism is often used to compute consequences, but it can also compute preconditions, or "backwards" inferences. The precondition for resting (more precisely, for achieving a state of rest) is sleeping, which is described by means of a script. The precondition for that script is being located at your home. The precondition for that is that someone bring you home (possibly you yourself). The preconditions for that are knowing where your home is, and is being located near you, and so on, backing up the causal chains. Every time we get a new precondition, we try to match it against the desired event, which in this case is Betty leaving the beehive. Nothing in the chain leading to resting matches, at least not up to a preset length. However, going backwards from "Betty is not hungry," we compute "Betty eats some food," "Betty has some food," "X (unspecified actor) gives Betty the food," "X is near the food," and "Y (unspecified) moves X to the food." This finally matches "Betty leaves the beehive" (more precisely, Betty moves Betty from the beehive to anywhere except the beehive), since we can extend this (as in unification) to "Betty moves Betty from the beehive to the flower."

But this is not enough. Sam has to trick Betty into doing that. We continue searching back up causal chains until we find a precondition that Sam can fulfill. Before Betty can go to the flower, she must know where a flower is. In order for that to happen, someone must tell her where a flower is. This, finally, is the precondition Sam can fulfill. (If Sam thought that Betty already knew where a flower was, this strategy would be rejected, and other alternatives would be tried.)

SAM DECIDED THAT IF SAM TELLS BETTY WHERE THERE WAS A FLOWER THEN BETTY MIGHT NOT BE NEAR HER BEEHIVE.

This example shows how the inference mechanism that the simulator uses can be used by the characters to do their own planning. In this case, the characters are computing "backward" inferences (preconditions); in the case of considering a request, the characters use the "forward" inferences (consequences) to help them decide whether to comply with the request.

There's no guarantee that the plan will work. Some plans depend on particular social relationships, and even though Bill Fox may think Henry Crow is vain, Henry may not be, in which case Bill's plan could fail.

SAM WANTED TO TELL BETTY WHERE THERE WAS A FLOWER.
In order for this trick to work, Sam has to talk to Betty. The TELL procedure is now active.

SAMI WANTED TO GET NEAR BETTY.

The first planbox in TELL is to go directly to Betty. If that failed, Sam would try using a messenger.

SAMI WALKED FROM A CAVE EXIT THROUGH A PASS THROUGH THE VALLEY TO THE GROUND BY THE APPLE TREE.
SAM WAS NEAR THE GROUND BY THE APPLE TREE.
SAM KNEW THAT SAM WAS NEAR THE GROUND BY THE APPLE TREE.
BETTY KNEW THAT SAM WAS NEAR THE GROUND BY THE APPLE TREE.

The module in charge of physical layouts has created all the intervening landscape, and Sam successfully arrives at Betty’s tree, a fact of which both are made aware.

SAMI TOLD BETTY THAT THE ROSE FLOWER WAS IN THE FLOWERBED.
BETTY KNEW THAT SAM TOLD BETTY THAT THE ROSE FLOWER WAS IN THE FLOWERBED.

This is all he has to do. If the trick works, the rest will happen automatically.

HOW DECEITFUL DID BETTY THINK THAT SAM FELT TOWARD HER?
*NOT AT ALL
BETTY THOUGHT THAT SAM WAS HONEST WITH HER.
BETTY KNEW THAT SAM KNEW THAT THE ROSE FLOWER WAS IN THE FLOWERBED.

If Betty thinks Sam is being honest with her, then she’ll believe that he believes what he just told her.

HOW MUCH DID BETTY TRUST SAM? *LOTS
BETTY TRUSTED SAM COMPLETELY.
BETTY KNEW THAT THE ROSE FLOWER WAS IN THE FLOWERBED.

In order for Betty to accept what Sam said as being true, she must also trust him. While it may seem at first that we’re being overly cautious, this actually allows us to have characters fail in their attempts to deceive one another, which makes for more interesting stories than those in which everyone always believes everyone else.

HOW HUNGRY WAS BETTY? *VERY
BETTY WAS FAMISHED.

When Betty finds out where some food is (remember that bees eat flowers here), a possible consequence (reaction) is that she will go eat it. But we haven’t established yet whether Betty is hungry, so TALE-SPIN asks. Had we said no, Sam’s plan would have failed. Since we reply that she is, S-HUNGER is reactivated, this time with Betty Bee as the hungry character. Again, we call DCONT and then DPROX.

BETTY WANTED TO GET A FLOWER.
BETTY WANTED TO GET NEAR THE ROSE FLOWER.

The unspecific flower has been resolved to the particular rose, whose location she knows (thus bypassing DKNOW).

BETTY FLEW FROM HER BEEHIVE THROUGH A VALLEY THROUGH A VALLEY TO THE ROSE FLOWER.
BETTY WAS NEAR THE ROSE FLOWER.
BETTY KNEW THAT BETTY WAS NEAR THE ROSE FLOWER.

The latter two sentences are consequences of her journey.

BETTY TOOK THE ROSE FLOWER.
BETTY HAD THE ROSE FLOWER.
BETTY KNEW THAT BETTY TOOK THE ROSE FLOWER.
BETTY KNEW THAT BETTY HAD THE ROSE FLOWER.

DCONT, called by S-HUNGER, has now succeeded. S-HUNGER specifies that the next thing to do is to eat the flower.

BETTY ATE THE ROSE FLOWER.
THE ROSE FLOWER WAS GONE.
BETTY WAS NOT HUNGRY.
BETTY THOUGHT THAT THE ROSE FLOWER WAS GONE.

A consequence of eating something is that it is destroyed. At this point, Betty has no more goals to pursue, so this part of the story is done. Still pending, of course, are the actions Sam has in mind.

SAMI TOOK THE HONEY.
SAMI HAD THE HONEY.
SAMI KNEW THAT SAM TOOK THE HONEY.
SAMI KNEW THAT SAM HAD THE HONEY.
SAMI ATE THE HONEY.
THE HONEY WAS GONE.
SAMI WAS NOT HUNGRY.
SAMI THOUGHT THAT THE HONEY WAS GONE.
THE END.
In similar fashion, he takes the honey and eats it. He has no more goals to pursue, and since his was the top-level goal, the story ends.

**MODULES**

Problem Solving Procedures

To produce the above story, it was necessary to give TALE-SPIN detailed knowledge of how to achieve goals by planning. The examples in this section describe the implementation of the planning methods that the characters in the stories use. Some of the preconditions for particular techniques involve the social relationships between characters (e.g., affection, trust) or personality traits (e.g., kindness). These are all represented as scales from \(-10\) to \(+10\). For example, if X likes Y, this is represented as (affection > 0).

**REQUEST \((X, Y, Z)\)**

X has asked Y to do Z.

If Y dislikes X or distrusts X or dominates X, then Y will refuse.
If Y likes X a lot or is indebted to X, then Y will agree.
If there are "positive" consequences, that is, if some consequence includes increased happiness (+joy) or is a precondition to some constant goal, then Y will agree.
If there are "negative" consequences, then Y will refuse.
If Y is kind, then Y will agree. Otherwise, Y will refuse.

**PROMISE \((X, A, Y, B)\)**

Y has offered to do B if X will do A.

If X is inclined to deceive Y, then X will say yes but will actually plan on insulting Y when X learns that Y has done B. (The deferred plan is implemented with a demon.)
If X likes Y, then X will say yes and will plan on doing A. Otherwise, X will refuse.

**BARGAIN \((X, Y, ACT)\)**

X wants to bargain with Y to do ACT

X considers the goals that Y can be assumed to have all the time (the cyclic S-goals). For those goals that require physical objects (e.g., food for S-HUNGER), X asks Y whether Y would do ACT if X brought Y one such object. (If X thinks Y already has that object, or if the acquisition of the object is already on X's goal-stack, then that plan is abandoned.) After asking the question (via TELL), the procedure checks whether X now knows that Y would not agree to the bargain—that piece of information would be generated by the inference mechanism when Y reacts to the question, so the answer could be known when TELL returns. If Y agrees to the deal, then X tries to get the object. If unsuccessful, X abandons the idea. Otherwise, X gives Y the object. Again, as soon as that happens, Y reacts one way or the other (either carrying out his part of the bargain or not), so the bargaining is judged a success if the ACT occurs.

**DCONT \((X, Y)\)**

X wants to acquire Y.

As in all the goal procedures, we check to see whether X already has Y—in which case the D-goal succeeds—or whether X was currently in the process of acquiring Y—in which case the goal fails. If X doesn’t know where Y is (or who owns Y), then DKNOW is called to find out. If that fails, so does DCONT.

The first "planbox" (technique) is to ask whether Y is owned by someone. If not, the X goes directly to Y (calling DPROX); if X can’t get there, DCONT fails. Once X is there, we test again whether X knows who owns Y. This is required in case the idea that no one owned Y was mistaken—there are stories where characters lie to each other about such matters. If all is well, then X attempts to take Y (DO-ATRANS), and DCONT succeeds if the attempt succeeds.

The second technique is for X to persuade the owner to give Y to X. However, if X is dishonest and if X can get to the owner (DPROX), then X will try to trick the owner by figuring out (via "backward" inferences) what act the owner could do that would cause Y to be near X, and then figuring out (via "reason" inferences) what X could do to motivate the owner to do that act. One example of this is flattery in "The Fox and the Crow": the crow is holding a piece of cheese that the fox wants, and the fox thinks that the crow is vain, so he praises the crow’s singing and requests a song, knowing that if the crow begins to sing, he’ll drop the cheese. In another example, one bear challenges another (who’s holding some honey) to a race. The second bear puts down the honey and takes off, so the first bear takes the honey and runs, in the opposite direction. The flattery plan required that the fox think that the crow was vain, and in order for it to work, the crow did, in fact, have to be vain. Those two ideas are independent—the fox could have been wrong, in which case, the plan would have failed. Similarly, the challenge required a competitive attitude between the two characters.

If the trick option is not used, or if it fails, then X tries simple persuasion. If that doesn’t work, then X considers outright theft if he’s dishonest.
X tries to get the owner away from Y (D-NEG-PROX), and then X goes and tries to take Y (DO-ATRANS).

**DKNOW (X, Q)**

X wants to find out the answer to question Q

X tries to persuade a friend, whose location he already knows, to tell him the answer to Q. Barring that, he asks the friend to tell him where he might find his other friends, and then he asks them about Q. Finally, X tries to persuade his friends to find out the answer to Q themselves, and then tell him.

(There is not yet a check on who these friends are, and it is easy to get stuck here. A recent, actual example: "SUE WANTED TO ASK PEGGY WHETHER PEGGY WOULD ASK BETTY WHETHER BETTY WOULD ASK PEGGY WHETHER PEGGY WOULD ASK BETTY WHETHER BETTY WOULD ASK PEGGY WHETHER PEGGY WOULD ASK BETTY WHETHER BETTY WOULD ASK PEGGY WHETHER PEGGY WOULD ASK BETTY WHETHER BETTY WOULD ASK PEGGY WHETHER PEGGY WOULD TELL SUE WHERE THERE WAS SOME CHEESE IF SUE GAVE PEGGY SOME HONEY IF SUE GAVE BETTY SOME HONEY IF SUE GAVE PEGGY SOME HONEY IF SUE GAVE BETTY SOME HONEY IF SUE GAVE PEGGY SOME HONEY IF SUE GAVE BETTY SOME HONEY IF SUE GAVE PEGGY SOME HONEY." The answer, incidentally, was no.)

**TELL (X, A, Y)**

X wants to communicate A to Y

A is a "fact", not a question, so TELL is simpler than DKNOWN. X tries to tell Y directly (calling DPROX). If that fails, X tries to persuade any one of his friends to carry the message.

**DPROX (X, Y, Z)**

X wants Y to be at location Z

The first technique is for X to try to move Y directly to Z. (This requires that Y be movable.) When Joe Bear wants to visit Irving Bird, we call DPROX

DPROX(Joe Bear, Joe Bear, Irving Bird)

which says that Joe Bear wants to change Joe Bear’s location to Irving Bird. Here X=Y (i.e., Joe Bear wants to move himself). If X is not the same as Y, then first we must call DPROX (X, Y, Y) and X picks up Y (DO-GRASP). X has to find out where Z is (via DKNOWN), and there has to be a route from X to Z (DLINK). Then X actually goes to Z (DO-TRANS). Once there, X lets go of Y (unless X=Y), and the plan succeeds. However, X may not really have gotten there. That is, X may have been misinformed (during the DKNOWN) as to Z’s actual location, deliberately or otherwise. In this case, if X can remember who misled him, his relationship with that person deteriorates (trust and affection drop to -6, and suspicion of deceit rises to +6), and the plan fails.

If X couldn’t even get to Y, then he tries to persuade Y to transport himself to Z. Next, he tries to persuade some friend to take Y to Z. Finally, he tries to persuade a friend to bring Z to Y. The obvious preconditions are checked (e.g., trees are immovable, rocks are movable but cannot move themselves, and neither trees nor rocks can be persuaded to do anything).

**PERSUADE (X, Y, Z)**

X wants to persuade Y to do Z

The first technique is for X simply to ask Y, but this is not used if X wants to deceive Y, feels competitive with Y, dislikes Y (affection, or if X thinks that Y dislikes him, or is trying to deceive him. Assuming that all works, X simply asks the question (via TELL). Y will respond one way or the other before we “return” from TELL, so PERSUADE simply checks to see whether Y did Z. If so, then PERSUADE succeeds.

The second method is for X to figure out a good reason why Y should do Z. If Z is a precondition to one of Y’s goals, then X will try to help Y do Z by meeting some of Z’s preconditions. Y may or may not get the hint. The third method is to bargain, which requires only that X not be trying to deceive Y.

The fourth method is for X to threaten Y with bodily harm. This tactic won’t be used if X likes Y, if X is dominated by Y, or if X doesn’t feel at all competitive with Y. Nor will it be used if Y is taller or heavier than X. (Every character is given a height and weight at the time of creation.) X communicates the threat (via TELL). Y will respond one way or the other upon hearing the threat. If Y does not acquiesce and x really dislikes Y, then X carries out the threat and slugs Y.

**S-HUNGER (X)**

X is hungry

The plan here is very simple: A should obtain some food (DCONT) and eat it (DO-INGEST). If X already knows where some food is, he will try that first. Otherwise, we randomly pick any one of the kinds of food that he eats as the desired meal.

**S-THIRST (X)**

X is thirsty

The plan here is for X to get near some water (DPROX) and drink it.
S-SEX (X)
X wants sex
X tries to persuade a friend to “fool around” (the generator’s phrase).

S-REST (X)
X is tired
X simply does a DPROX to his bed (or home) and goes to sleep.

Inferences
Although it may not be obvious at first glance, it is necessary for TALE-SPIN to be able to make inferences. It needs to do so in order to correctly assess the effects of the events that it generates. TALE-SPIN uses four types of inferences for each of the CD primitives. Type-1 inferences specify the previous states or acts, in a strict causal sense. These are the “backwards inferences” that the characters themselves use in planning. The type-1 inferences from “Joe gave the berries to Jack” (ATRANS) are that Joe had the berries (CONT) and that the berries were near Joe (LOC).

Type-2 inferences are the consequences, and they constitute the majority of all the inferences. The simulation is essentially driven by type-2 inferences. The largest subclass of these inferences specifies who becomes aware of the consequences (MLOC), and these are called reaction inferences. When characters are threatened or praised or insulted, for example, their motivation to respond comes from the reaction inferences.

Type-3 inferences apply only to the states, and they specify the acts for which the states are preconditions, according to the planboxes.

Type-4 inferences apply only to the acts, and they specify what other acts might follow.

These inference-procedures depend on the current state of memory. For instance, Joe Bear’s reaction to seeing honey depends on how hungry he is at the time. Apart from that, the range of inferences is the same. However, there is a separate class of demon-inferences that are not constant. While the consequences are immediately true, the demon-inferences are used to represent things that may become true at some point in the future. For example, if Irving promises (in good faith) to tell Joe where some honey is after Joe brings him a worm, then a demon-inference for Irving upon receiving the worm is to keep his promise and tell Joe where some honey is, which is obviously not one of the normal consequences of getting a worm.

Relationships
When a plan has a precondition concerning the relationship between characters (“How much does Joe like Maggie?”) or the personality of a single character (“How kind is Irving?”), the program checks the memory. If no answer is found, the program asks the user. It does this on the assumption that there is an answer, but it just hasn’t been established yet. This treatment is quite different from the other preconditions (“Does Joe know where any honey is?”) where failing to find the answer in memory is equivalent to answering no.

The question is represented in CD and simply passed to the English generator. The program will accept a number of predefined responses that eventually correspond to an integer. For example, when the program asks “How much does Joe think that Maggie likes Henry?” the responses VERY, LOTS, or A LOT will produce an “affection” value of 7. The responses A LITTLE or SOMETHING will produce a value of 3. The responses NOT VERY MUCH, NOT VERY, or NOT MUCH will produce a 0. The responses NONE or NOT AT ALL will produce a −3.

It’s a trivial matter to extend the possible responses, to use different integers for the input, or to change the numbers in the preconditions themselves. The above numbers are simply rough guidelines.

Memory
The term “memory” in TALE-SPIN refers to the chronology of facts, which are grouped according to who believes the facts. The “true” facts are simply those believed by the simulator, and these include such things as details of the physical world. The characters have their own beliefs. There are no “true” beliefs about relationships: in TALE-SPIN, the statement “John likes Henry” means “John believes that he likes Henry.” It would make no sense in TALE-SPIN to say that “it is true that John likes Henry” and that “John thinks that he dislikes Henry.”

When facts are “asserted”, they are added to memory. When that happens, the state of memory is checked for consistency. Contradicted and superseded facts are removed. Memory is stored in TRUEFACTS, contradicted facts are saved in !OLDFACTS, and superseded facts are simply deleted.

Contradictions are mutually exclusive states. If Joe Bear is in his cave, that fact will be in TRUEFACTS. If he goes to visit Irving Bird, his new location is the tree where Irving lives. Thus, “Joe is at the maple tree” contradicts “Joe is at his cave.” The contradicted statement is saved so that, for example, if Joe goes to the cave later on, the generator will be able to say “Joe returned to his cave” instead of “Joe went to his cave.”

Superseded facts are those that are more clearly specified. If Joe is talking to Irving, the CD representation of the statement “Joe is at the maple tree” is stored in TRUEFACTS as one of Irving’s beliefs. If Joe walks away, the statement “Joe is not at the maple tree” is added to Irving’s beliefs, and the former statement is moved to !OLDFACTS since it is now contradicted. But suppose Irving later finds out through a third party that Joe is near the river. The statement “Joe is near the river” supersedes but does not contradict “Joe is not near the maple tree,” so that belief is simply deleted.
The Creator and the Travel Agent

When characters are introduced, a program called the “creator” constructs a physical setting for that character. The particular items in the setting are specified by abstract descriptions of the simulated world, called blueprints. The created versions of these are called maps; they are used to represent all physical locations in TALE-SPIN.

Routes through this world are computed by a program called the “travel agent”. The maps are all part of a hierarchy. The central idea is that an individual map is of fixed size and resolution, and you ascend the hierarchy to find maps covering a larger overall range but less detail, and descend to do the opposite. When the travel agent is asked to find a route from point A to point B, it “zooms out” (ascends the hierarchy) until it finds a map that encompasses, at some depth, both A and B. It figures the route from A’s “terminal” (the point in this high-level map that contains A) to B’s “terminal.” Then it zooms back in on the map “below” A’s terminal and computes the route from A to that terminal, using the same procedure, and does the same for B. The result is a series of low-level trip segments.

If no such route exists, the travel agent calls the creator to construct the necessary pieces, according to the blueprints. When there is more than one possible solution, the creator asks the user questions such as “Do you want a new mountain?”

Symbolic Arithmetic

TALE-SPIN had to be able to handle relationships between numerical quantities. For example, the problem solver had to know if A happened before or after B, as did the generator when choosing tenses. Some plans asked questions like “Is Joe Bear taller than Jack Bear?” We wanted to be able to do this without having to specify exact times for things, or exact heights, or whatever. It was the comparative relationships that were important.

The program, called XARITH, represents these relationships (which in mathematics are considered to be “partially ordered”) with a directed graph. It processes assertions and queries, and it can detect contradictions (A is before B, and B is before A) and redundant information (A is before B, B is before C, and A is before C). When numerical information is known, it uses a total ordering. It can deduce that two points in the graph are actually the same. In TALE-SPIN, it is used primarily to follow transitivity chains, so that it can answer whether A is before B without having seen that explicit assertion. It has not yet been used to represent the duration of events (the difference between the “starting” and “ending” times) nor second-order information (the difference between durations, as in “It took less time to fly from New Orleans to New York than it did to get from JFK airport to Manhattan by bus”).

The English Generator

The generator works on single CD expressions. It first determines whether the expression represents an act, a state, a state-change, or a causal link. It chooses which verb to use depending on the particular CD expression involved (e.g., for MTRANS, it uses ask, offer, deny, refuse, agree, and tell) and on the memory (e.g., for ATRANS, give vs. give back). Embedded sentences can have infinities (“Joe wanted Irving to tell him . . .”) and can omit the subject of infinitives (“Joe wanted to ask Irving . . .”).

The subject and verb are processed by a single function, relying on the fact that in English, the effects of negation, mood, tense, etc., can be local to the subject-verb pair. For example, in the sentence “Joe sees Irving,” the direct object (“Irving”) follows the subject-verb pair, not only in this form but also in these: Joe saw, Joe will see, Did Joe see, Will Joe see, Can Joe see, Was Joe able to see, Joe doesn’t see, Wouldn’t Joe be able to see, Joe might be able to see, etc.

The social states use a variety of phrases depending on the degree of the relationship (e.g., for AFFECTION: hate, dislike, like, love, be devoted to). Finally, there is a post-processor that uses first names of characters, articles (a, an, the, any, some), and punctuation (period or question mark).

Mis-spun Tales

One of the best ways to see why all the above components are necessary to a story generator is to see how we learned that they were necessary. It is not always obvious how a computer program will actually function while it is still in the planning stages. Important parts of a program are often left out because there was no way to know that they would be needed.

TALE-SPIN, in its early stages, frequently told rather strange stories. These “mistakes” caused many re-definitions in the original program. Since this process of “mistakes” followed new theory is characteristic of AI programs in general it is worthwhile to look at these “mistakes” and consider what had to be done to fix them. (The output of the original stories has been simplified for ease of reading.)

****** 1 ******

One day Joe Bear was hungry. He asked his friend Irving Bird where some honey was. Irving told him there was a beehive in the oak tree. Joe threatened to hit Irving if he didn’t tell him where some honey was.

Joe has not understood that Irving really has answered his question, albeit indirectly. Lesson: answers to questions can take more than one form. You’ve got to know about beehives in order to understand that the answer is acceptable.
One day Joe Bear was hungry. He asked his friend Irving Bird where some honey was. Irving told him there was a beehive in the oak tree. Joe walked to the oak tree. He ate the beehive.

Increasing the range of acceptable answers is not enough. You have to know what the answers really mean.

In the early days of TALE-SPIN, all the action focused on a single character. Other characters could respond only in very limited ways, as in answering direct questions, for example. There was no concept of one character “noticing” what another character had done. Hence the following story, which was an attempt to produce “The Ant and the Dove”, one of the Aesop fables:

Henry Ant was thirsty. He walked over to the river bank where his good friend Bill Bird was sitting. Henry slipped and fell in the river. He was unable to call for help. He drowned.

That wasn’t supposed to happen. Falling into the river was deliberately introduced to cause the central “problem” of the story. Had Henry been able to call to Bill for help, Bill would have saved him, but I had just added the rule that being in water prevents speech, which seemed reasonable. Since Bill was not asked a direct question, he didn’t notice his friend drowning in the river. “Noticing” is now an inference from change of location, so Bill sees Henry in the river, deduces that Henry’s in danger, and rescues him.

Here are some rules that were in TALE-SPIN when the next horror occurred. If A moves B to location C, we can infer not only that B is in location C, but that A is also. If you’re in a river, you want to get out, because you’ll drown if you don’t. If you have legs you might be able to swim out. With wings, you might be able to fly away. With friends, you can ask for help. These sound reasonable. However, when I presented “X fell” as “gravity moved X,” I got this story:

Henry Ant was thirsty. He walked over to the river bank where his good friend Bill Bird was sitting. Henry slipped and fell in the river. Gravity drowned.

Poor gravity had neither legs, wings, nor friends. Now “X fell” is represented with PROPEL, not PTRANS, that is, as “the force gravity applied to X,” and the inference from PROPEL are not the same as for PTRANS.

The inclusion of awareness meant that I couldn’t set up the stories that way I used to.

Once upon a time there was a dishonest fox and a vain crow. One day the crow was sitting in his tree, holding a piece of cheese in his mouth. He noticed that he was holding the piece of cheese. He became hungry, and swallowed the cheese. The fox walked over to the crow. The end.

That was supposed to have been “The Fox and the Crow”, of course. The fox was going to trick the crow out of the cheese, but when he got there, there was no cheese. I fixed this by adding the assertion that the crow had eaten recently, so that even when he noticed the cheese, he didn’t become hungry.

Before there was much concern in the program about goals, I got this story:

Joe Bear was hungry. He asked Irving Bird where some honey was. Irving refused to tell him, so Joe offered to bring him a worm if he’d tell him where some honey was. Irving agreed. But Joe didn’t know where any worms were, so he asked Irving, who refused to say. So Joe offered to bring him a worm if he’d tell him where a worm was. Irving agreed. But Joe didn’t know where any worms were, so he asked Irving, who refused to say. So Joe offered to bring him a worm if he’d tell him where a worm was...

Lesson: don’t give a character a goal if he or she already has it. Try something else. If there isn’t anything else, then that goal can’t be achieved.

Here are some more rules. If you’re hungry and you see some food, you’ll want to eat it. If you’re trying to get some food and you fail, you get sick. If you want some object, try bargaining with the object’s owner. Innocuous, right?

One day Henry Crow sat in his tree, holding a piece of cheese in his mouth, when up came Bill Fox. Bill saw the cheese and was hungry. [Bill has just been given the goal of satisfying hunger.] He said, “Henry, I like your singing very much. Won’t you please sing for me?” Henry, flattered by this compliment, began to sing. The cheese fell to the ground. Bill Fox saw the cheese on the ground and was very hungry. [Satisfying hunger is about to be added to Bill’s goals again.] He became ill. [Because satisfying hunger was already a goal of Bill’s, it can’t be added again. Hence, Bill fails to satisfy
his hunger, so he gets sick.] Henry Crow saw the cheese on the ground, and he became hungry, but he knew that he owned the cheese. He felt pretty honest with himself, so he decided not to trick himself into giving up the cheese. He wasn’t trying to deceive himself, either, nor did he feel competitive with himself, but he remembered that he was also in a position of dominance over himself, so he refused to give himself the cheese. He couldn’t think of a good reason why he should give himself the cheese [if he did that, he’d lose the cheese, so he offered to bring himself a worm if he’d give himself the cheese. That sounded okay, but he didn’t know where any worms were. So he said to himself, “Henry, do you know where any worms are?” But of course, he didn’t, so he... [And so on.]

The program eventually ran aground for other reasons. I was surprised it got as far as it did. I fixed it by adding the rule that dropping the cheese results in loss of ownership.

TECHNICAL VIEW OF TALE-SPIN

The Top Level of TALE-SPIN

In this section, I would like to explain in more detail how TALE-SPIN runs—what its basic execution cycle is.

There is an initial dialogue with the user, specifying the characters and some details of setting, but the cycle doesn’t really begin until the user specifies that the main character has a problem. Once that fact is asserted, the top-level cycle takes over automatically.

It is possible, in fact, to start the cycle even during the initial dialogue. If you create a bear and some berries and tell the bear where the berries are, TALE-SPIN will ask how hungry the bear is. That is due to the fact that the inference part of the cycle wants to know whether the bear is now going to use the information about the berries—an automatic trigger.

When an event is asserted, we do one of two things. If it is a CD expression for “John thinks John has X as a goal”, then we save “John has X as a goal” on a list (PLAN-LIST). Otherwise, we simply add the event to the memory. If it is of the form “John thinks Y”, then it is stored with the things John knows. Otherwise it’s stored as a “true” fact (known to the storyteller). Notice that “John thinks that Mary has Z as a goal” does not go onto the PLAN-LIST, but it is stored in memory.

Next, we compute all the consequences of the asserted event, which will also be a list of CD expressions. If there are any, then we repeat the whole process on them, adding them either to the PLAN-LIST or to memory. Finally, when there are no more consequences, we send each of the plans on the list to the problem solver. That is, each item on that list is the statement that some character has some goal, so we now call the appropriate problem-solver for the goal. If the goal is to reduce hunger, we call S-HUNGER. If the goal is acquisition of some object, we call DCONT. If the goal is not a state but a direct act, such as MTRANSing some piece of information, then we call a routine (DO-IT) that simulates the actual events.

DO-IT

For all but a few of the acts, DO-IT simply asserts the event, but for others there are runtime preconditions to take care of. These are not part of the planning process, but rather take care of details, including obstacles that represent the difference between what the characters imagine is going to happen, and what actually does happen.

Part of the bookkeeping detail for DO-MTRANS and DO-PTRANS is calling the creator and the travel agent. If Irving Bird has decided to tell Joe Bear where some honey is and we haven’t already created some honey that he knows about, we do so now. If Irving is being honest, then this is exactly like the “real” creation in the initial dialogue. But if Irving is lying, then we’ve got to be trickier. Some of the material gets created, but other material is only hypothetically created. For instance, Irving might tell Joe that there’s some honey in such-and-such an oak tree. If he’s lying, then there will be an oak tree, but it won’t have a beehive in it, and if Joe goes there, he’ll find that out and realize that he’s been tricked.

DO-PTRANS has some bookkeeping to do, too. When Joe Bear goes to visit Irving Bird who lives in a nest in a maple tree, we want to make sure that Joe doesn’t wind up in the nest, but rather stays on the ground near the maple tree.

Consequences

The most important part of the cycle is the generation of consequences. Some of the consequences are constant, or context-free, but many depend on the state of memory, what specific facts people know, and what attitudes they have about themselves and others.

There are two principal advantages to the data-driven consequence cycle. First, the more information you put into the inferences, the more will happen “automatically” so that you can make the planning procedures less explicit. This makes good methodological sense as well, since all the planning procedures have equal access to the inference mechanism, and you’re more likely to spot conceptual errors. That is, you may have added inference X for situation Y, but the system may surprise you, pleasantly or otherwise, by making inference X in situation Z.

The second advantage is that when the characters themselves are planning, they use the same inference mechanism that the simulator does. That is, when Joe Bear is trying to figure out whether to do something, such as complying with
a request, he considers the consequences, as computed by the inference mechanism from his point of view. They may differ from what would actually occur, since his knowledge, especially of interpersonal relationships, need not be consistent with the truth. For example, he may surmise that Maggie Bee is quite friendly and will gladly offer him some honey, so in his hypothetical simulation he achieves his goal, thus suggesting that that particular plan is a good one. When he actually confronts Maggie Bee, in the real simulation, she may dislike him intensely and refuse to give him anything. In this view, then, hypothetical simulation is a major component of the process of planning.

The disadvantages of the consequence-cycle are the potential for proliferation of facts, especially irrelevant facts, and the tendency for the characters to spend all their time second-guessing each other, endlessly. The first problem is a familiar plague in AI systems. If you need a particular inference once, then you get it all the time, needed or not. One alternative is to “enable” the inference ahead of time, and “disable” it afterwards, which is outright cheating. Another popular approach is somehow to “structure” the information so that the context determines which inferences should be made. Unfortunately, TALE-SPIN’s context is not yet that structured. A third tactic is to have an intelligent garbage collector, also known as a forgetting mechanism, to solve what is essentially a “system” problem, not a theoretical problem, at least not at the level of simulation.

The second-guess syndrome is actually a special case of the inference explosion. The theoretical weakness in planning by simulating is that you need to control the level of detail. When you’re actually doing something, everything actually happens. When you’re merely thinking about doing something, you cannot imagine all the details. Fortunately, TALE-SPIN doesn’t get stuck here since the information that a character has is both incomplete and imperfect, so the inferences do die out. Also, as noted above, the inference cycle does not pursue the consequences of “John thinks that Mary has X as a goal.” If it did, and went on to compute that “John thinks that Mary has Y as a goal” where Y is one of the subgoals of X, we’d quickly get into the game of representing “John thinks that Mary thinks that Bill thinks that John thinks that…” and all the associated problems, theoretical and practical.

Example

In the story given earlier where George Ant falls in the river and Wilma Bird rescues him, the entire rescue scene is part of one call to the inference mechanism. It begins when we assert that George fell into the water. An inference from that is that George is in the water. Two inferences from that are that George knows that he’s in the water and that Wilma does, too. She’s sitting nearby, and a change of location produces such an inference for every nearby character. An inference from knowing that someone is in some water is knowing that the person might die. (Exceptions could be made for fish, etc.) An inference from knowing that some event might cause death is having the goal to avoid or undo that event. (Both George and Wilma infer these things.) The “inference” from having a goal is acting on that goal, i.e., calling the problem solver. In this case, D-NEG-PROX is called: George wants to get out of the river. D-NEG-PROX calls DPROX on nearby areas, and so now “George wants to get near the meadow,” which is the creator’s name for one of the river banks. George knows where the river bank is (a precondition in DPROX), but when DO-PTRANS is called, one of the runtime obstacles prevents him from leaving: very small creatures cannot PTRANS themselves through rivers. George’s DPROX goal thus fails, and so does his D-NEG-PROX. Fortunately, Wilma has the same goal, so D-NEG-PROX is called again. This time, it’s Wilma who wants George out of the river. D-NEG-PROX calls DPROX, getting George to the river bank. Wilma knows where George is, and DO-PTRANS succeeds since she is flying. DPROX continues by having her grab George, bringing him to the river bank, and letting him go. The inference from letting something go (unGRASP) is that they land on whatever is below them (creator/travel agent used again here), so George falls to the meadow. Of course, George was aware of everything that happened to him (more knowledge-state inferences), and when he knew he was in danger, a demon inference was established so that he were to be rescued, his relationship to his rescuer would change (affection + 10, indebtedness + 10); hence, “George was devoted to Wilma. George owed everything to Wilma.”

All of this happened in one call to the assertion mechanism, when we said that George had fallen into the water.

Data Structures and Control Structures

DKNOW, DPROX, etc., are directly coded as procedures. The advantage of this approach is that it makes it easier to handle the flow of control from one planbox to the next and to keep track of the binding of variables. Another typical approach, made popular by MICRO-PLANNER and its like, is to define such procedures as “data” and to program a separate interpreter for them, the advantage being that one could now “read” the plan instead of only being able to execute it. I chose not to use this approach for several reasons. First of all, it suggests that reading a plan is the same as carrying out a plan—that you have access to the same information, which isn’t true. Second, the planning information that the characters do need is simpler and can be encoded in other ways. Third, the interpreter requires a lot of bookkeeping code for binding variables and matching patterns, which didn’t seem worth the effort for the simulator, which is naturally oriented towards execution. Finally, many of the same effects can be achieved by the “hypothetical simulation” discussed earlier. That is, the purpose of reading a plan is to be able to predict in advance what will be needed,

...
what is likely to occur, etc. By having the characters call the planning procedures directly, using "hypothetical" memory (initially the same as current memory), and "observing" what happens, the same effect is possible. While there is still a serious issue about the level of detail in a hypothetical versus actual execution of plans, the simulation idea has some appeal on psychological grounds.

Memory

In the original implementation of TALE-SPIN, the memory, which contains what each character knows and what the storyteller knows, was a list of CD expressions indexed by the person who "knows" the represented events, sub-indexed under the role-fillers, and further sub-indexed under the role-names. This was an attempt to model a theory of memory where all the things John knew specifically about Mary would be grouped together, as opposed to simply grouping together all the things that John knew about anything. In practice, however, TALE-SPIN never used that feature, i.e., nothing in the system depended on the particular grouping of events in memory. In the current implementation, the sub-indices have simply been removed, and while the pattern matcher does a little more work now looking for answers to questions like "Where does John think Mary is?", the substantially reduced overhead for the structure itself more than compensates.

The Inference Mechanism

Each of the CD acts and states has a corresponding set of inference functions (types 1–4). The inference generator uses the top-level act (e.g., PTRANS), state (e.g., MLOC), or connective (e.g., CANCAUSE), and calls the appropriate procedure.

Each procedure constructs a list of CD expressions that are the inferences. For example, the consequences (type-2 inferences) from an ATRANS event are specified as follows:

The TO filler now has the OBJECT filler.
The FROM filler no longer has the OBJECT filler.
The TO filler knows that the ATRANS occurred.
Everyone else at the same location as the ACTOR filler knows that the ATRANS occurred.

Suppose the event were "John gave Mary a book." That is represented as

((ACTOR (JOHN) <=> (ATRANS) OBJECT (BOOK) FROM (JOHN) TO (MARY)))

Then the ATRANS type-2 inference procedure returns the list of inferences: John no longer has the book, Mary has the book, John knows that the ATRANS occurred, Mary knows that the ATRANS occurred. If this event really happened (as opposed to being part of someone's advance planning), then each inference is asserted, the consequences of each inference is computed and asserted, and so on.

The "reaction inferences" are simply invoked by the procedure that computes the consequences for MLOC, in the same fashion as the top level of the inference generator.

The Generator

Like the inference mechanism, the English generator takes a CD expression and calls the procedure associated with the main act, state, or connective. The generator's principal function is to produce the correct order of the words and the correct form of the verb. For each of fifty verbs, it knows the conjugation for singular and plural in the past, present, and future tenses. Those are pre-computed. It constructs at runtime the negations, interrogatives, and so on, which requires only a small amount of information about auxiliaries such as "do" and "will".

The function TN produces the subject-verb pair. TN takes two parameters, the infinitive from of the verb and the subject. The only reason the subject is used is that it may have to be embedded, as in "Did John go...?" The other pieces of information are the tense and the "mode-list". The mode-list is part of the CD expression, i.e., it is a necessary part of the meaning. The tense of the top-level sentence is arbitrary. Narratives are customarily told in the simple past tense. But given the tense of the top-level sentence, the tense to be used for the embedded expression is computed by calling the timekeeper, e.g., "John thinks that Mary was in New York" versus "John thinks that Mary will be in New York." Possible elements of the mode-list are: *NEG* if the concept is a negation; *MAYBE*, which would be added, for instance, when one character can only guess the likely response of another, as in "Joe thought that Irving might tell him where some honey was"; *CAN* to indicate ability; and *??* to indicate a question.

For example, suppose the subject is MARY, the infinitive (as determined by the PTRANS generator procedure) is WALK, and the mode-list is (*NEG* *??* *CAN*), and the tense is future. The procedure first tests for the presence of *CAN* in the mode-list. Given that, and the future tense, the infinitive gets "BE ABLE TO" concatenated to it, and we know that we will need an auxiliary, in this case, DO. The verb we use is either the auxiliary (if we have decided we need one) or the infinitive, so in this case, it's DO, and we look up its future-tense form. For DO, that's stored as WILL. Next, we look up the negative form of that since *NEG* is in the mode-list; that's stored as WON'T. Next, we write out the "verb" and then the subject, since *??* is present, and finally, if we used an auxiliary, we type out the infinitive, thus producing WON'T MARY BE ABLE TO WALK. When the sentence is finished, another function will add the question mark.
Suppose the subject is MARY, the infinitive is WALK, but the mode-list is (*MAYBE*), and the tense is present. If *MAYBE* is in the mode-list, we use the auxiliary MIGHT as the "verb", the present-tense form of which is MIGHT. We write the subject, then the verb, and finally the infinitive, producing MARY MIGHT WALK.

There’s nothing particularly sacred about this procedure. I determined the order of the tests simply by working with conjugations for a while, and this pattern emerged. It is sufficient for all the constructions that TALE-SPIN can produce. It won’t produce examples such as "It was John who went to New York", because there’s nothing in the CD representation to indicate that kind of emphasis.

Nouns are far simpler. The creator puts the English names and first names on the property lists of the system names for each character it creates, e.g., the system name for Joe Bear is *JOEBEAR*. His English name is the list (JOEBEAR), and his first name is stored as JOE. For physical objects, the creator puts only English names on the property lists. The generator uses this information after the order of the words in the output sentence has already been determined. It goes through the list of words, inserting articles (a, an, the, any, some) and punctuation. The generator has already included possessives. That is, if it was producing *NEST*4, it checks to see whether anyone owns that object. If so, it inserts "IRVING'S", for example. When stating the location of some object, it will use "THERE" plus the appropriate form of the verb *to be*.

The generator calls the memory to decide between such things as "go" and "return", "take" and "take back", "isn’t hungry" and "isn’t hungry any more".

Looking Back on the Implementation

The diversity of TALE-SPIN's components is both its pride and its curse. That is, it's nice to see dynamically created mountains, correct English tenses and modals, theft and lying, automatic inferences, competition and indebtedness, pattern matchers, plans and goals, directed graphs, and so on, in the same system, but keeping track of it all isn't easy. Changing it in any major way is a very risky business, because the system is highly interconnected. To ask the user whether John likes Mary, we need the generator. To be able to say "the maple tree", the generator needs information from the creator. The creator needs to know where birds live. DPROX needs to know the difference between Irving Bird, his nest, the tree containing that nest, and the ground around the tree. BARGAIN needs access to the reaction-inferences. As part of a theory of human knowledge, such integration is highly desirable. As a characteristic of a 90K LISP program, such integration complicates the programming tasks. As a feature of a cognitive model, it runs the risk of spreading the knowledge too thin.

10 Micro TALE-SPIN

INTRODUCTION

The micro versions of the storyteller (TALE-SPIN) and the English generator (MUMBLE) have the same basic cycle of assertions and inferences as the full system, but the details have been greatly simplified.

Most of the planning procedures have been retained, but they use simpler techniques, connected by ANDs and ORs ("do this AND this OR else do that AND that"). In particular, the characters cannot plan ahead by hypothetical simulation. Scales, such as hunger and affection, have been replaced by a simple yes or no, represented as MODE (POS) and MODE (NEG).

The top level of the inference mechanism is the same, but only type-1 inferences (consequences) appear here, and simpler versions of them, at that. No social states, for example, are ever inferred in the micro system, so no matter how often Joe slugs Irving, Irving’s feelings toward Joe remain stailwartly the same. The only S-goals used here are S-HUNGER and S-THRIST.

Most of the social states have been preserved, though the restrictions on the social conditions have been loosened, making it somewhat easier for a "successful" story to happen. The creator and the travel agent have been dispensed with altogether. The initial world description is explicitly declared by the function INIT-WORLD, which is called by the top-level function (SPIN).

A generator has been included so the story can be followed. Even in its crude form, it's a much better alternative to tracing certain key functions than decoding CD forms written in LISP. Micro-MUMBLE uses about twenty-five verbs and prefices all physical objects with the article "the". Names for characters and objects come straight from the internal identifier, so Joe Bear always comes out