

SPARCL Tutorial, part 1

(Copyright 2002 by Lindsey Spratt. All Rights Reserved.)

1.Introduction to SPARCL

In this section we review the basic mechanisms of SPARCL through an example program. Although the treatment is informal many important concepts are introduced such as: SPARCL clauses, facts, rules and running queries. [Due to the underlying logic programming semantics of SPARCL, there are many similarities between an introduction to SPARCL and introductions to other logic programming languages. Thus, the contents and organization of this script are adaptations to SPARCL of the first chapter of the second edition of Ivan Bratko's "Prolog Programming for Artificial Intelligence", 2nd edition.]

1.1.Facts - the Parents program. In this subsection we create a program which describes the 'Parent'/2 relationship among seven people, then present several queries of this parent relationship.

In this subsection you are shown how to create a simple "database"-style program, 'Parent'/2, and how to query ("run") it.¹ SPARCL is a visual logic programming language based on sets with partitioning constraints. This subsection introduces some of the simple visual logic programming aspects of SPARCL.

First we define a simple SPARCL program.

1.1.1.Create the 'Parent'/2 program. In this subsection we create a program of six clauses. These clauses describe a 'Parent'/2 relationship among seven people.

A SPARCL program is represented by a window containing one or more clauses. An empty new program is created via the "New Program..." option of the "File " menu. After we create the program window we will populate it with clauses. The first step is described below, and the results of this step are shown in Figure 1.

Step 1.

DO: Create a new program (and window) named "Parent".

BY: Select the "New Program..." option of the "File " menu, enter "Parent" in the program name dialog, and click "OK".

1. Due to the underlying logic programming semantics of SPARCL, there are many similarities between an introduction to SPARCL and introductions to other logic programming languages. We adapted section 1.1 of the second edition of Ivan Bratko's "Prolog Programming for Artificial Intelligence" (a discussion of logic programming and PROLOG) to create this subsection of the tutorial on logic programming and SPARCL.

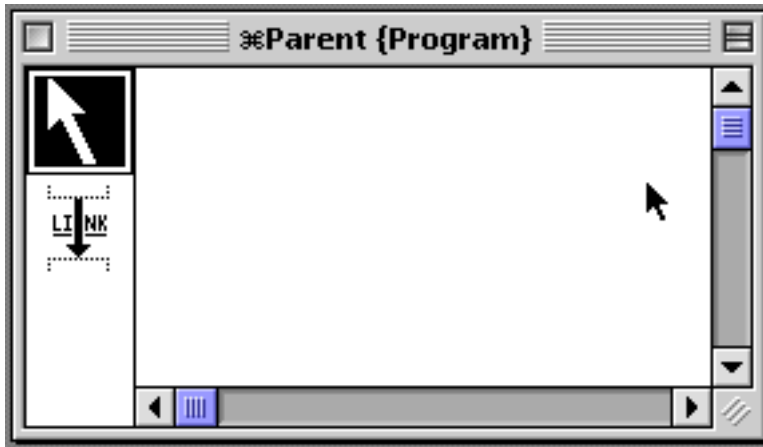


Figure 1: The newly created “Parent” program window.

There are three "panes" in a program window (only two of which are visible in Figure 1): the drawing pane, the tools pane, and the viewer pane. The drawing pane is the large area on the right side of the window. It has scroll bars on its right and bottom sides. The tools pane is on the left side of the

window. There are two tool icons, the top one (an arrow) is the "general" tool and the other tool is the "connect" tool. The viewer pane is optionally visible, it is in the lower left corner of the window. The viewer is used to quickly position the drawing pane. It is left hidden (or "off") to speed up the display operations.

The general tool supports a wide variety of operations, each of which is invoked through a popup menu which is activated by pressing the mouse button while the cursor is over some "object" in the program window (the background of the program window is considered to be the "program" object). Each kind of object has a different popup menu. The connect tool supports connecting two terms together. We discuss this more below.

Now we are ready create the clauses of the ‘Parent’/2 program. Step 2, which creates the first clause, has several sub-steps.

Step 2: Create a clause with arguments Pam and Bob in program Parent.

Step 2.1 creates an “empty” clause in the program window. The popup menu item selection is shown in Figure 2. The result of the interaction described in step 2.1 is shown in Figure 3.

Step 2.1.
 DO: Create a new clause in program "Parent".
 BY: Select the "Create Clause" option in the popup menu for the program (window).

Now that we have the empty clause, we must add the two arguments positions. This is done in Step 2.2. This step uses two sub-steps.

Step 2.2: Add arguments to an existing object.

The next step creates the first argument position. The popup menu item selection is shown in Figure 4. The result of the interaction described in step 2.2.1 is shown in Figure 5.

Step 2.2.1.

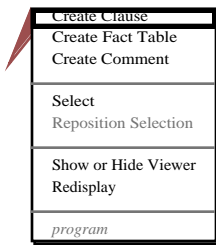


Figure 2: Step 2.1 interaction to create a clause. (The small “flag” at the upper left corner of the clause is half of a “demonstration cursor” used by the tutorial system.)



Figure 3: New “Parent” clause resulting from Step 2.1

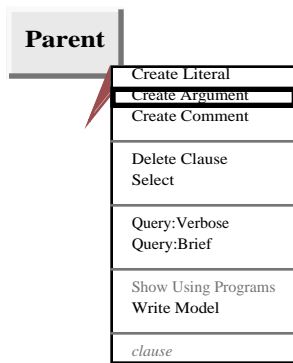


Figure 4: Interaction to create an argument for step 2.2.1.

DO: Create a new argument in the clause.
 BY: Select the "Create Argument" option in the popup menu for clause.

The next step creates the second argument position. The popup menu item selection is shown in Figure 6. The result of the interaction described in step 2.2.2 is shown in Figure 7.

Step2.2.2.(repeat Step 2.2.1).

Having created two argument positions in the ‘Parent’/2 clause, we now place the ur constant “Pam” in the first (upper) argument. This is done by step 2.3, which is done in three sub-steps.

Step 2.3: Create a new ur constant of value Pam in the first argumentof the clause.

The step 2.3.1 creates an ur constant in the first argument position with the default value of “new_ur”, and leaves an edit item open for that

new ur constant. The popup menu item selection is shown in Figure 8. The result of the interaction described in step 2.3.1 is shown in Figure 9.

Step 2.3.1.
 DO: Create a new ur in the first argumentof the clause.
 BY: Select the "Insert:Ur" option in the popup menu for the argument.

Now we enter the text “Pam” into the edit item left open by the previous step. The result of this step is shown in Figure 10.

Step2.3.2.
 DO: Edit the value of the ur constant of the “open” ur constant to "Pam".
 BY: Enter the characters of the new value.

We have entered the text “Pam”, now we close the edit item. The result of this step is shown in Figure 11.

Step 2.3.3.
 DO: Close the current edit "box" for program "Parent".
 BY: Click in the program window anywhere outside of the box.



Figure 5: Result of argument creation by step 2.2.1.

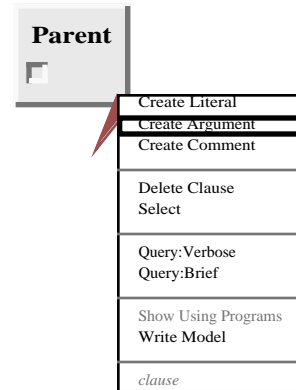


Figure 6: Interaction to create an argument for step 2.2.2.

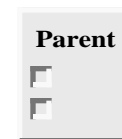


Figure 7: Result of argument creation by step 2.2.2.

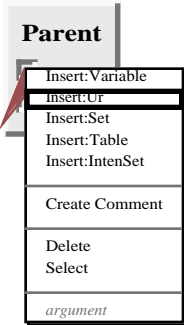


Figure 8: Interaction to create a new ur constant for step 2.3.1.



Figure 9: Result of ur constant creation by step 2.3.1.

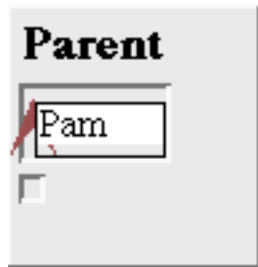


Figure 10: Result of ur constant editing by step 2.3.2.

The next step creates an ur constant “Bob” in the second (lower) argument. It uses the same three sub-steps as did step 2.3. The result of this step is shown in Figure 12.

Step 2.4: Create a new ur constant of value Bob in the second argument of the clause.

Step 2.4.1.

DO: Create a new ur in the second argument of the clause.

BY: Select the "Insert:Ur" option in the popup menu for the argument.

Step 2.4.2.

DO: Edit the value of the “open” ur constant to "Bob".

BY: Enter the characters of the new value.

Step 2.4.3.

DO: Close the current edit "box" for program "Parent".

BY: Click in the program window anywhere outside of the box.

Step 3 creates a second clause. It uses the same sort of sub-steps as used in creating the first clause, with the exception of the argument position creation steps.

When there is just one predicate defined in a program window, as in this case, any “create clause” interaction creates a clause for that predicate with the appropriate number of argument positions. The result of the sub-steps of step 3 are shown in Figure 13.

Step 3: Create a clause with arguments” Tom” and “Bob” in program Parent.

Step 3.1.

DO: Create a new clause in program "Parent". (Call this clause 2.)

BY: Select the "Create Clause" option in the popup menu for the program.

(Use existing arguments in an existing object.)

Step 3.2: Create a new ur constant of value “Tom” in the first argument of clause 2.

Step 3.2.1.

DO: Create a new ur in the first argument of clause 2.

BY: Select the "Insert:Ur" option in the popup menu for the argument.

Step 3.2.2.

DO: Edit the value of “open” ur constant to "Tom".

BY: Enter the characters of the new value.

Step 3.2.3.

DO: Close the current edit "box" for program "Parent".

BY: Click in the program window anywhere outside of the box.

Step 3.3: Create a new ur constant of value “Bob” in the second argument of clause 2. (sub-steps similar to step 3.2)

The remainder of the six clauses are created in the next four steps. These steps achieve their goals by the same process as steps 2 and 3. The resulting set of clauses is shown in .

Step 4: Create a clause with arguments “Tom” and “Liz” in program Parent.



Figure 11: Result of “closing” ur constant edit item by step 2.3.3.



Figure 12: Result of creating new ur constant by step 2.4.



Figure 13: Result of creating new clause by step 3.

- Step 5: Create a clause with arguments "Bob" and "Ann" in program Parent.
- Step 6: Create a clause with arguments "Bob" and "Pat" in program Parent.
- Step 7: Create a clause with arguments "Pat" and "Jim" in program Parent.

Now that we have defined the "Parent"/2 predicate (a predicate "Name/K" is defined by all of the clauses with the name "Name" and K of arguments), we can query it. A query is written in SPARCL as a clause, generally with one or more "literals" in it. A "literal" is a reference to a predicate.

A query is written in SPARCL as a clause, generally with one or more "literals" in it. A "literal" is a reference to a predicate.

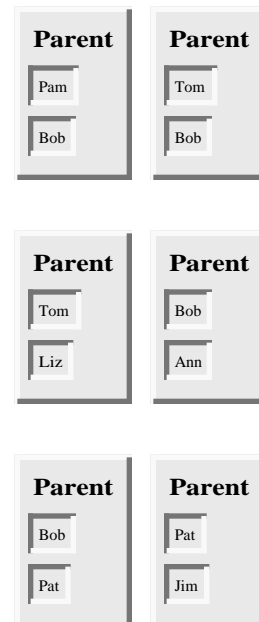


Figure 14: Result of steps to create the six 'Parent'/2 clauses.

1.1.2: Querying the Parent program. In this section we create six clauses for querying the parent program and run these queries.

First we create the program which will contain the six query clauses.

- Step 1.
- DO: Create a new program (and window) named "Parent Query".
- BY: Select the "New Program..." option of the "File " menu, enter "Parent Query" in the program name dialog, and click "OK".

For our first example, we ask "Who is Pam's child?". This is done in two phases: first, create a clause which describes the query; second, use the "Query:Brief" option of the "Clause" popup menu to run the query. Step 2 creates the query using several sub-steps.

- Step 2: Create a "Parent Query" clause for asking the question "Who is Pam's child?" (Call this clause 1)

Step 2.1 creates the "Parent Query" with one argument. The argument is created in the same fashion as shown in step 2.2.1 of section 1.1.1. The result of the creation is shown in Figure 15.

- Step 2.1.
- DO: Create a new clause named "Parent Query" with 1 argument in program "Parent Query". (Call this clause 1)
- BY: Select the "Create Clause" option in the popup menu for the program. Add 1 more argument to the default 0 arguments.

We add a comment to the clause to describe the question it asks in step 2.2. This is done in three sub-steps.

- Step 2.2: Create a new comment "Who is Pam's child?" in clause 1.

The next step creates a comment for the clause with the default value "comment". The interaction is shown in Figure 16, and the result is shown in Figure 17.

- Step 2.2.1.
- DO: Create a new comment in clause 1.
- BY: Select the "Create Comment" option in the popup menu for the clause.

Next the default comment value is replaced with the desired comment "Who is Pam's child?". The result is shown in Figure 18.



Figure 15: Result of step 2.1 to create argument.

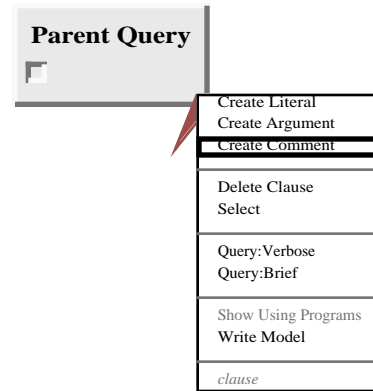


Figure 16: Interaction for step 2.2.1 to create argument.

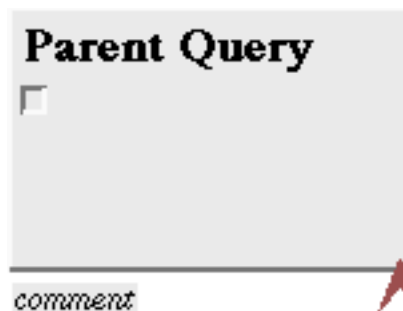


Figure 17: Result of step 2.2.1 to create the comment.

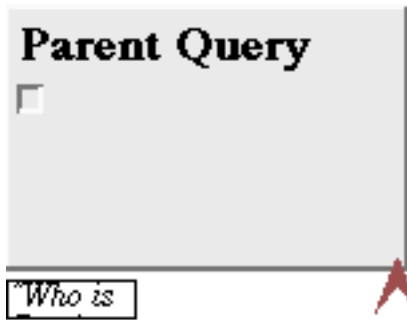


Figure 18: Result of step 2.2.2 to set the value of the comment.

Step 2.2.2.
DO: Edit the value of the "open" comment to "Who is Pam's child?".
BY: Enter the characters of the new value.

The next step finishes the creation of the comment by closing

the edit item for the comment.

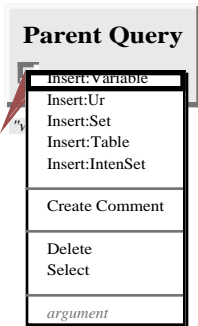


Figure 19: Interaction for step 2.3 to create a variable.

Step 2.2.3.
DO: Close the current edit "box" for program "Parent Query".
BY: Click in the program window anywhere outside of the box.

Next a variable is put in the argument of the clause. The interaction is shown in Figure 19 and the result is shown in Figure 20.

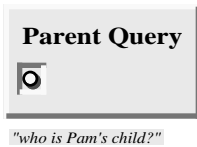


Figure 20: Result for step 2.3 to create a variable.

Step 2.3.
DO: Create a new variable in the argument of clause 1.
BY: Select the "Insert:Variable" option in the popup menu for the argument.

The next step creates a literal. The interaction is shown in Figure 21, and the result is shown in Figure 22.

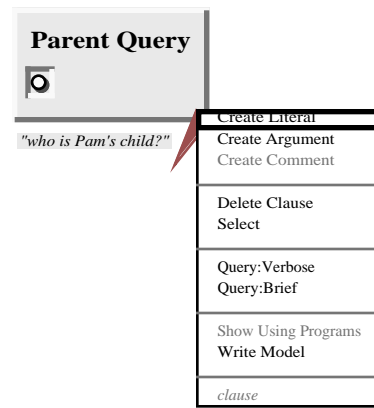


Figure 21: Interaction for step 2.4 to create a literal.

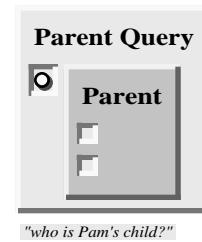


Figure 22: Result for step 2.4 to create a literal.

Step 2.4.
DO: Create a new literal with name "Parent" and 2 arguments in clause 1.
BY: Select the "Create Literal" option in the popup menu for the clause.

Next we put the ur constant valued "Pam" in the first argument of the new literal. This is done by several sub-steps.

Step 2.5: Create a new ur constant of value "Pam" in the argument of the literal of clause 1.

The sub-steps for creating an ur constant value "Pam" start with step 2.5.1. These steps are similar to the ur constant creation steps presented above. The initial interaction is shown in Figure 23 and the final result is in Figure 24.

Step 2.5.1.
DO: Create a new ur in argument 1 of the literal of clause 1.
BY: Select the "Insert:Ur" option in the popup menu for the argument.

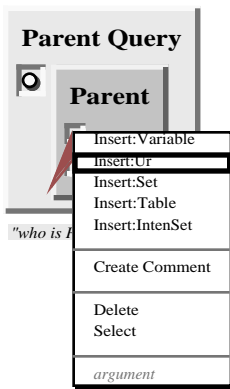


Figure 23: Interaction for step 2.5.1 to create an ur constant.

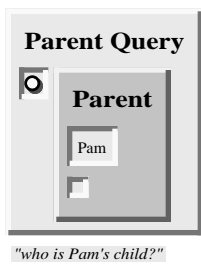


Figure 24: Result for step 2.5.3 to close the edit item for the new ur constant.

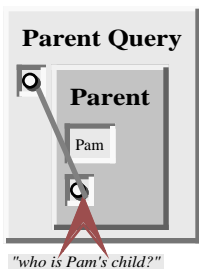


Figure 25: Interaction for step 2.7 to link two variables.

Step 2.5.2.
DO: Edit the value of the “open” ur constant to “Pam”.
BY: Enter the characters of the new value.

Step 2.5.3.
DO: Close the current edit “box” for program “Parent Query”.
BY: Click in the program window anywhere outside of the box.

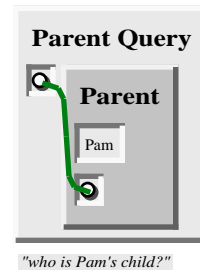


Figure 26: Result for step 2.7 to link two variables.

The next step creates a variable in the second argument of the literal, similarly to step 2.3.

Step 2.6.
DO: Create a new variable in the argument 2 of the literal of clause 1.
BY: Select the “Insert:Variable” option in the popup menu for the argument.

Now we link together the two variables. The interaction is shown in Figure 25. The state of the interaction pictured is after doing a click-and-drag from the variable in the argument of the clause to the variable in the second argument of the literal, but before the mouse button is released. The “demonstration cursor” shows where the user’s cursor would be at this point and the grey line shows where the link will be placed. The result is shown in Figure 26.

Step 2.7.
DO: Create a coreference link including the variable in the argument of clause 1 and the variable in argument 2 of the literal of clause 1.
BY: With “connect tool” as the current tool, depress the mouse button while the cursor is over one of the variables and drag the cursor until it’s over the other variable, then release the mouse button.

This clause has our first uses of SPARCL variables. One is in the argument of the “Parent Query” clause; the other is in the second argument to the ‘Parent’/2 literal. These two variables are linked to show that they must refer to the same thing. In a traditional linear language this would be indicated by having two variable instances with the same name.

Now that we have built a “query” clause, we are ready to evaluate it. The next step does this evaluation. The interaction is shown in Figure 27 and the result is shown in Figure 28.

Step 3.
DO: Execute a query of clause 1, with tracing information suppressed.
BY: Select the “Query:Brief” option in the popup menu for the clause.

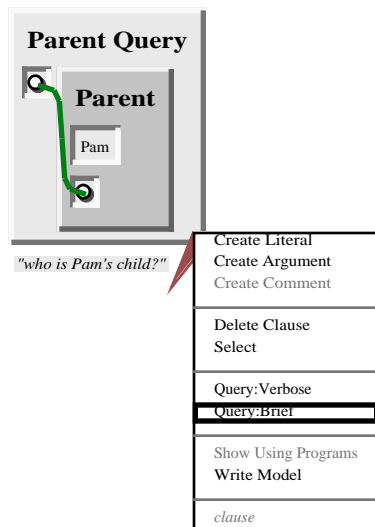


Figure 27: Interaction for step 3 to query SPARCL.

This query creates output in two windows. It writes a "timing" message in the "*Output*" window telling how long the query took to be evaluated (this time does not include time spent rendering the result). The query, when successful, also adds an "N-tuple" to the "Parent Query RESULT" window which is the query clause "head" as it is after the successful query evaluation. In this example, the result in Figure 28 shows that Bob is Pam's child.

For our second example, we ask "Who is Liz's parent?". This is done in the same two phases as were used before. The sub-steps used by step 4 to create the query clause are similar to those of step 2 above. The resulting clause is shown in Figure 29.

Parent Query → Bob

Figure 28: Result for step 3 to query SPARCL.

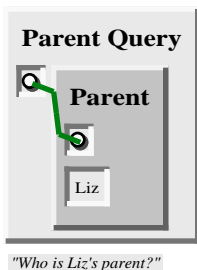


Figure 29: Result for step 4 to create the "Who is Liz's parent?" query clause.

Step 4: Create a "Parent Query" clause for asking the question "Who is Liz's parent?"

Step 5.
 DO: Execute a query of clause with ID Parent Query:13, with tracing information suppressed.
 BY: Select the "Query:Brief" option in the popup menu for clause with ID Parent Query:13.

Parent Query → Tom

Figure 30: Result of step 5 posing the "Who is Liz's parent?" clause as a query.

As a result of the preceding query, SPARCL tells us that Tom is a parent of Liz, as shown in Figure 30.

For our third example, we ask "Who are two people related as parent and child?". The clause for this question and the result of posing this clause as a query are shown in Figure 31.

The next query asks for Jim's grandparent "X". Since our program does not directly know the "Grandparent" relation, this query has to be broken down into two parts:

- 1) Who is a parent of Jim? Assume that this is some Y.
- 2) Who is a parent of Y? Assume that this is some X.

We can ask a query of two parts by putting a literal for each part in the body of the same query clause and connecting the variables of these literals appropriately. The clause for this question and the result of querying SPARCL with it are shown in Figure 32. This result shows us that Bob is Jim's grandparent.

The previous query can be "turned around" and we can ask "Who is a grandchild of Bob?".

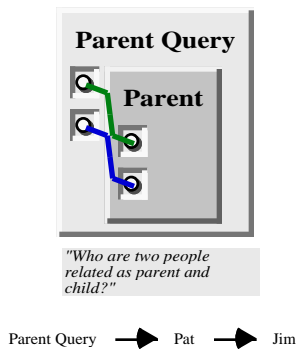


Figure 31: Clause for “Who are two people related as parent and child?” and the result of posing this clause as a query.

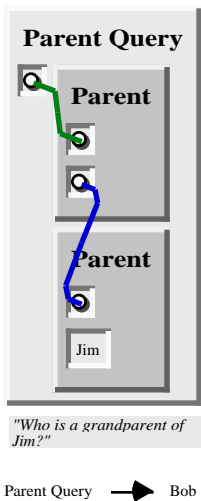


Figure 32: Clause for “Who is the grandparent of Jim?” and the result of posing this clause as a query.

Objects of the first kind are called "atoms" and objects of the second kind are called "variables".

- Questions to the system consist of a clause, which may contain any number of 'literals'. Several literals in the body of a single clause means that clause is true when the conjunction of the literals is true.
- An answer can be either positive or negative: for a positive answer we say that the query was "satisfiable" and it "succeeded"; for a negative answer we say that the query was "unsatisfiable" and it "failed".

The clause for this question and the result of querying SPARCL are shown in Figure 33. This shows us that Jim is a grandchild of Bob.

Another question could be "Who is a common parent of Ann and Pat?". As before, this query has to be broken down into two parts:

- 1) Who is a parent, X, of Ann?
- 2) Is (this same) X a parent of Pat?

The clause for this question and the result of querying SPARCL are shown in Figure 34. The result shows us that Bob is a common parent of Ann and Pat.

This section has presented several points:

- It is easy in SPARCL to define a relation, such as the 'Parent' relation, by stating the N-tuples of objects that satisfy the relation.
- The user can easily query the SPARCL system about relations defined in the program.
- A SPARCL program consists of 'clauses'.
- The arguments of relations can (among other things) be: concrete objects (such as "Tom" and "Ann"), or general objects which are represented by small circles.

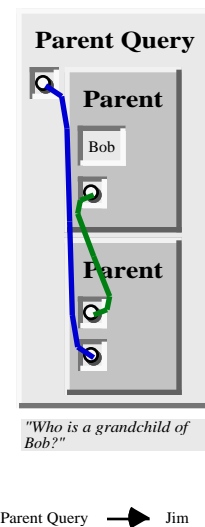


Figure 33: Clause for “Who is a grandchild of Bob?” and the result of posing this clause as a query.

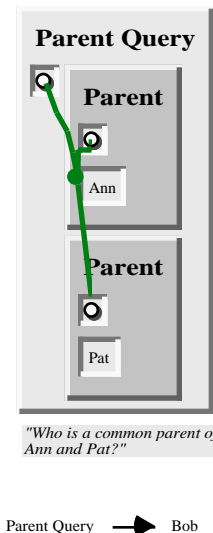


Figure 34: Clause for “Who is a common parent of Ann and Pat?” and the result of posing this clause as a query.

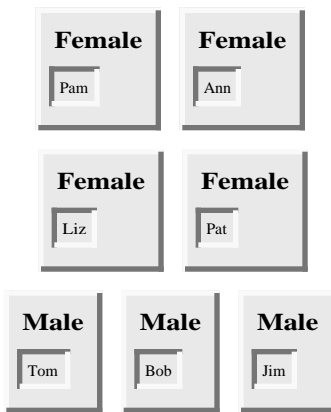


Figure 35: "Sex" program, separate clauses version.

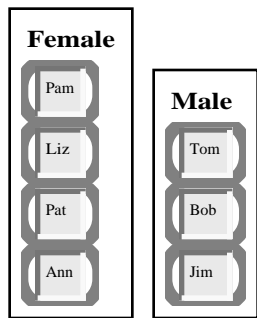


Figure 36: "Sex" program, fact table version.

- If a query has several possible answers then SPARCL will find one of them.

1. Exercise for section 1.1:
1. Formulate in SPARCL the following questions about the 'Parent'/2 relation (your question formulation should be "query" clauses with names of your choosing (e.g. "Query A", "Query B", "Query C")):
 1. Who is Pat's parent?
 2. Does Liz have a child?
 3. Who is Pat's grandparent?

1.2: Rules - extending the Parents program. This script continues the introduction to programming in SPARCL which was begun in the "Facts - the Parents program" script. The "Parents" program can be extended in many interesting ways. Let us first add the information on the sex of the people that occur in the 'Parent'/2 relation. This information has already been entered and saved in a program file, so we can simply open that program. NOTE: The contents and organization of

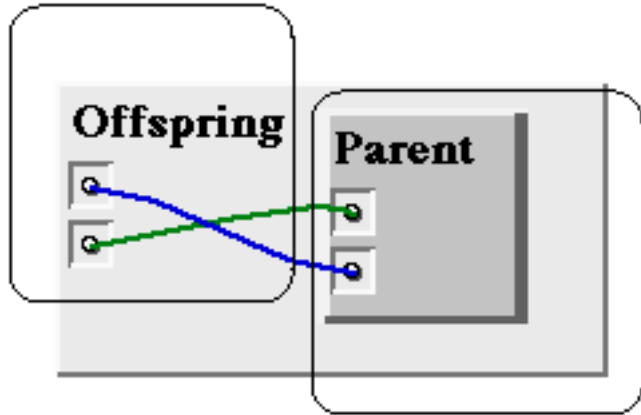
this script are drawn largely from section 1.2 of the second edition of Ivan Bratko's "Prolog Programming for Artificial Intelligence".

The relations introduced in Figure 35 are 'Male'/1 and 'Female'/1. These relations are unary (or one-place) relations. A binary relation like 'Parent'/2 defines a relation between *pairs* of objects; on the other hand, unary relations can be used to declare simple yes/no properties of objects.

These 'Female'/1 and 'Male'/1 relations can be displayed in a different, more compact, way. This alternative representation uses SPARCL's "fact tables". Figure 36 has "fact table" versions of the 'Female'/1 and 'Male'/1 relations. These tables have the same meaning as the collection of shown the in Figure 35. The fact table has the predicate name for all of the facts of the table placed in the upper left-hand corner of the table. Each row of the table is a single "fact" - a clause with an empty "body".

As our next extension to the program let us introduce the 'Offspring'/2 relation as the inverse of the 'Parent'/2 relation. We could define 'Offspring'/2 in a similar way as the 'Parent'/2 relation; that is, by simply providing a list of simple facts about the 'Offspring'/2 relation, each fact

The conclusion part of the rule.



The condition part of the rule.

Figure 37: Offspring clause, with labeled parts.

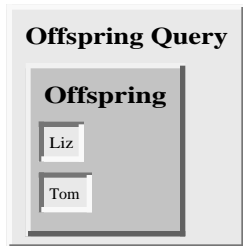


Figure 38: Clause for posing the query "Is Liz an offspring of Tom?"

of clause is called a "rule".

There is an important difference between facts and rules. A fact like those shown in the 'Parent' clauses is something that is always, unconditionally, true. On the other hand, rules specify things that are true if some condition is satisfied. Therefore we say rules have:

- a condition part (the right-hand side of the clause)
- a conclusion part (the predicate name and arguments on the left-hand side of the clause).

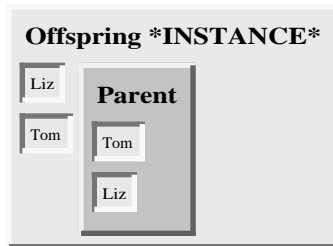


Figure 39: Instantiation of Offspring rule clause. Created in response to the query literal of Figure 38.

The conclusion part is also called the "head" of a clause and the condition part the "body" of the clause.

How rules are actually used by SPARCL is illustrated by the following example. Let us ask our program whether Liz is an offspring of Tom. The "query" clause in Figure 38 represents our question.

There is no fact about offsprings in the program, therefore the only way to consider this question is to apply the rule about offsprings. The rule is general in the sense that it is applicable to

any two objects; therefore it can also be applied to such particular objects as "Liz" and "Tom". We say that the variables become instantiated.

Figure 39 is the special case of the Offspring rule clause after instantiation of the general rule in Figure 37 in satisfying the query literal from Figure 38.

The single literal of the "Offspring *INSTANCE*" body becomes the new goal for SPARCL to solve. It is trivial to solve as it can be found as a fact in the 'Parent' program. This means that the conclusion part of the rule is also true, and SPARCL will succeed in executing the

mentioning one pair of people such that one is an offspring of the other.

However, the offspring relation can be defined much more elegantly making use of the fact that it is the inverse of 'Parent', and that 'Parent' has already been defined. This alternative way can be based on the following logical statement: "For all X and Y, Y is an offspring of X if X is a parent of Y."

The clause in Figure 37 represents the preceding "logical statement". This kind

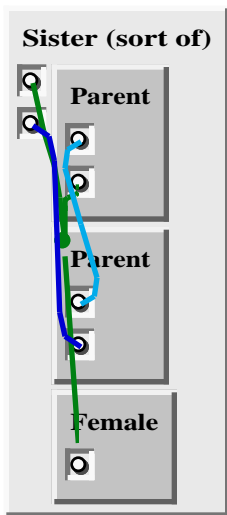


Figure 41: Clause defining the 'Sister (sort of)'/2 predicate.

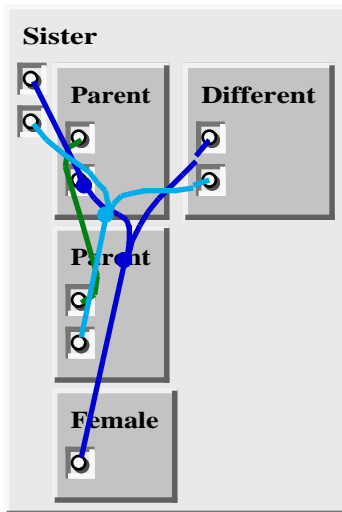


Figure 42: Clause defining the 'Sister' predicate.

original query.

In Figure 40 there are two similar clauses defining two different predicates. The "Mother" clause, which defines the 'Mother'/2 predicate, shows a rule with two literals in its body, 'Parent'/2 and 'Female'/2. The "Grandparent" clause, which defines the 'Grandparent'/2 predicate, shows a rule which uses the same relation, 'Parent'/2, twice in the literals of its body.

The "Sister (sort of)" clause in Figure 41 defines the relationship of someone as the sister of someone if these two people have the same parent. This clause actually is slightly flawed - it allows for someone to be sister to herself. The "Sister" clause in Figure 42 fixes this using the 'Different'/2 relation.

The 'Different'/2 predicate, shown in Figure 43, relies on a partitioned set constraint to specify the terms in its two arguments are different. The '*TERM*'/1

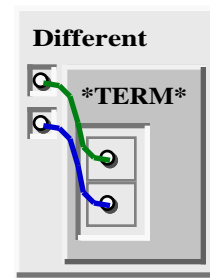


Figure 43: Clause defining the 'Different' predicate.

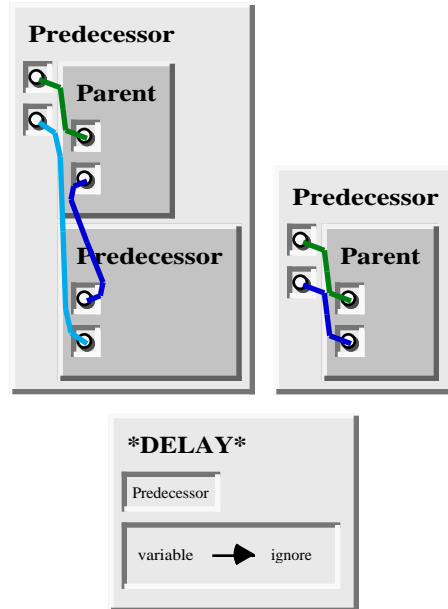


Figure 44: Clauses defining the not-quite-satisfactory version of the 'Predecessor' predicate.

predicate is a builtin predicate of the SPARCL system. It is always true. It is used here to introduce the partitioned set constraint. Since the two "sisters" are in different parts of a partition, they must be different (since parts of a partition are disjoint sets). This is a kind of "not equal" constraint. We will discuss partitioned sets in more detail in a later section.

The 'Predecessor' predicate is defined by the two 'Predecessor' clauses and a '*DELAY*' clause, as shown in Figure 44. This predicate is an example of *recursion*, multi-

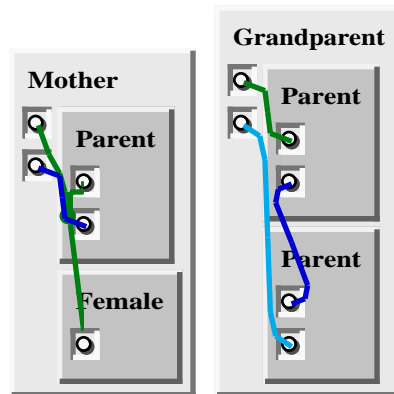


Figure 40: The clauses defining the 'Mother' and 'Grandparent' predicates.

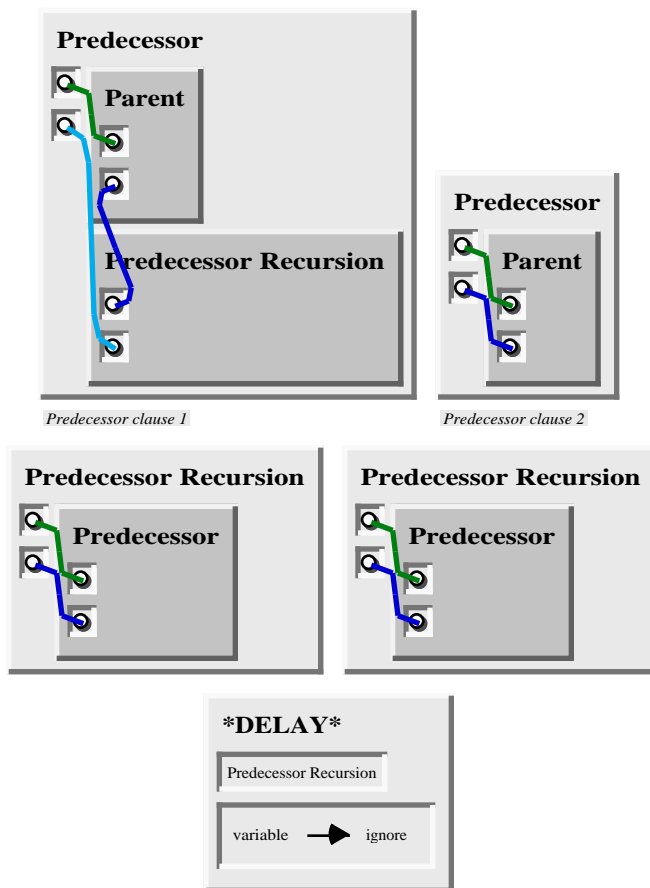


Figure 45: Clauses defining the correct version of the 'Predecessor' predicate.

ple clauses to defining a single predicate, and specifying a *delay* condition. A recursive definition of a predicate uses the predicate being defined in the body of one or more of the defining clauses of that predicate. Some person X is the Predecessor of some other person Z if X is a Parent of Z (this is one of the clauses), OR if X is the parent of some person Y, and Y is a predecessor of Z (this is the other one of the clauses).

The '*DELAY*' predicate is used by the SPARCL interpreter in deciding when to evaluate goal literals. This '*DELAY*' clause instructs the interpreter to "delay" the evaluation of a 'Predecessor' goal if the first argument is a variable (i.e. an unbound variable term). This is necessary in the case of the

'Predecessor' literal before it has solved the 'Parent' literal. This prevents the interpreter from recursing forever. Unfortunately, this is too restrictive. This restricts us to using the 'Predecessor' predicate to ask the question "Who is X the predecessor of?". We would like to also be able to use 'Predecessor' to answer the question "Who is X's predecessor?".

Another version of the 'Predecessor' program is shown in Figure 45 that allows us to ask both of these questions. This version of the 'Predecessor' program allows us to ask both of the questions mentioned before. The difference between this version and the earlier version small: There is a new predicate, "Predecessor Recursion" which is simply a "wrapper" for 'Predecessor'; this predicate is used in the recursive clause of 'Predecessor'; and the '*DELAY*' clause now refers to this new predicate 'Predecessor Recursion' instead of the 'Predecessor' predicate. Now, the interpreter will attempt to solve a 'Predecessor' goal which has an unbound variable first argument (which would have been delayed in the definition in Figure 44), but it will not recurse infinitely in the attempt since the 'Predecessor Recursion' literal will be delayed when its first argument is an unbound variable.

The '*DELAY*' clauses are the primary method the SPARCL programmer has to control

the order in which the SPARCL interpreter attempts to solve goals. This is a particularly important facility when writing recursive predicates to ensure that the recursion terminates.

Important points of this section are:

- SPARCL programs can be extended by simply adding new clauses.
- SPARCL clauses are of three types: facts, rules, and questions.
- Facts declare things that are always, unconditionally true.
- Rules declare things that are true depending on a given condition.
- By means of questions the user can ask the program what things are true.
- SPARCL clauses consist of the "head" and the "body". The head is the name of the predicate and the arguments placed on the left side of the clause. The body is a set of "literals". These literals are understood to be joined by conjunctions.
- Facts are clauses that have a head and the empty body. Rules have the head and the (non-empty) body. A question is a "rule" clause which the programmer chooses to query.
- In the course of computation, a variable can be substituted by another object. We say that a variable becomes "instantiated".
- Variables are assumed to be universally quantified and are read as "for all". Alternative readings are, however, possible for variables that appear only in the body. These can be read as "some" (existential) variables.
- Recursion may be used in defining SPARCL predicates.
- The `"*DELAY"` clause may be used to control the order in which the SPARCL interpreter attempts to solve goals.

1. Exercise for section 1.2

1. Show translations for the following statements:

1. Everybody who has a child is happy (introduce a one-argument relation "Happy").
2. For all X, if X has a child who has a sister then X has two children (introduce a new relation "Has Two Children").
2. Define the relation "Grandchild" using the 'Parent'/2 relation. Hint: It will be similar to the "Grandparent" relation.
3. Define the relation "Aunt" of two arguments in terms of the relations 'Parent'/2 and "Sister".),

1.3-How SPARCL works. This section gives an informal explanation of *how* SPARCL answers questions. [The text of this section is adapted for SPARCL from section 1.4 of Ivan Bratko's "Prolog Programming for Artificial Intelligence", 2nd edition.]

A question to SPARCL is always a set of one or more goal literals. To answer a question, SPARCL tries to satisfy all of the goals. What does it mean to *satisfy* a goal? To satisfy a goal means to demonstrate that the goal is true, assuming that the relations in the program are true. In other words, to satisfy a goal means to demonstrate that the goal *logically follows* from the facts and rules in the program. If the question contains variables, SPARCL also has to find what are the particular objects (in place of variables) for which the goals are satisfied. The particular instantiation of variables to these objects is displayed to the user. If SPARCL cannot demonstrate for some instantiation of variables that the goals logically follow from the program, then SPARCL's answer to the question will be "no".

An appropriate view of the interpretation of a SPARCL program in mathematical terms is then as follows: SPARCL accepts facts and rules as a set of axioms, and the user's question as a *conjectured theorem*; then it tries to prove this theorem—that is, to demonstrate that it can be logically derived from the axioms.

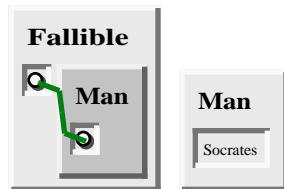


Figure 46: Clauses defining the 'Fallible'/1 and 'Man'/1 predicates.

We will illustrate this view by a classical example. Let the axioms be:

All men are fallible.
Socrates is a man.

A theorem that logically follows from these two axioms is:

Socrates is fallible.

The first axiom above can be rewritten as:

For all X, if X is a man the X is fallible.

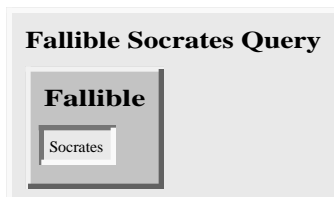


Figure 47: Clause posing the question "Is Socrates fallible?"

The example can be translated into SPARCL as shown in Figure 46.

Now we ask SPARCL the question of Socrates' fallibility by querying the 'Fallible Socrates Query'/0 clause in Figure 47, which succeeds.

To discuss how SPARCL works, we use the concept of a *proof sequence*. Given some program (a set of clauses), a sequence of facts can be constructed starting with any *fact* in the program, then successively adding other facts from the program or any fact derived using a *rule* of the program and any facts already in the sequence. A goal (or 'literal') is *satisfied* if such a sequence of facts can be found which ends with that goal. Let us call such a sequence of facts a *proof sequence*. SPARCL finds an appropriate proof sequence to satisfy a query.

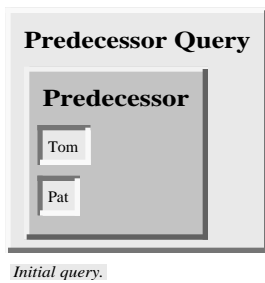


Figure 48: Clause defining initial query for “Is Tom Pat’s predecessor?”

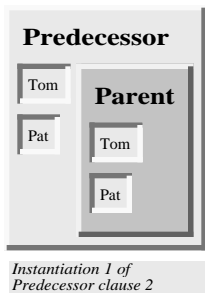
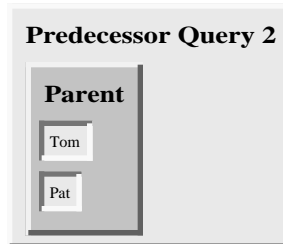


Figure 49: First instantiation of ‘Predecessor’/2 clause 2.

SPARCL searches for a proof sequence satisfying a given goal by starting with that goal and working “backward” to facts of the program. Instead of starting with simple facts given in the program, SPARCL starts with the given goals and, using rules, substitutes the current goals with new goals, until new goals happen to be simple facts (or unify with program facts). Let's look at an example using ‘Predecessor’/2 and ‘Parent’/2 programs.

How does SPARCL “solve” the query asking if Tom is Pat's predecessor? We start with the initial query shown in Figure 48. This initial query clause provides a single goal literal, “Predecessor(Tom, Pat)”. There are two rule clauses which have heads (consequents) which unify with this goal literal. These rules are labeled "Predecessor clause 1" and "Predecessor clause 2". SPARCL may try either of these



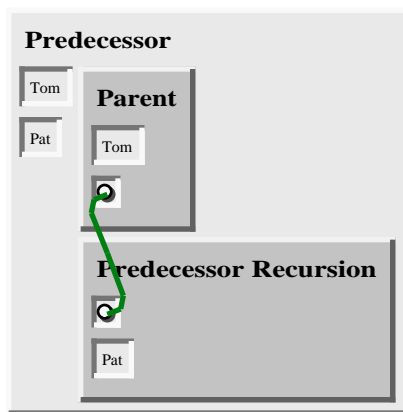
Query 2, substituting body of instantiation 1 of "Predecessor clause 2" into the "initial" query. This query FAILS since there is no corresponding Parent clause.

Figure 50: Revised version of ‘Predecessor Query’/0 (Figure 48) based on substituting the body of instantiation 1 of ‘Predecessor’/2 clause 2 (Figure 49) for the literal in the body of the original ‘Predecessor Query’/0.

clauses first—let's suppose SPARCL tries the rule of "Predecessor clause 2". It unifies the goal literal with the head of the rule, which binds the variables in the head. Since these variables corefer with variables in the body (antecedent) of the rule, these corefering variables are also bound. This instantiates the rule clause as shown in Figure 49.

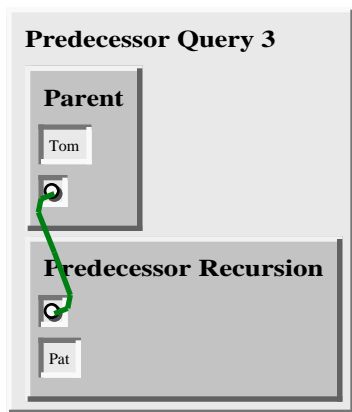
For this instantiation of ‘Predecessor’/2 clause 2 to be applicable in building a proof sequence, the literal in its body must precede the application of this rule in the proof sequence. Thus, SPARCL determines that it must find a proof sequence for this literal. This yields the "query 2" version of the initial query shown in Figure 50. The original goal of "Is Tom Pat's predecessor?" has been replaced by the goal of "Is Tom Pat's parent?".

There is no clause (fact or rule) with a head that matches the ‘Parent’/2 literal in the body of query 2 in Figure 50, so SPARCL fails to solve this goal and it *backtracks* to try an alternative way to derive the top goal ("Is Tom Pat's predecessor?"). There were originally two ways to solve this top goal, and having failed to solve it using the rule “Predecessor clause 2” SPARCL now tries rule "Predecessor clause 1". As was done when using “Predecessor clause2”, SPARCL unifies the goal literal with the head of the rule (“Predecessor clause 1”), which binds the variables in the head. Again, since these variables corefer with variables in the body (antecedent) of



Instantiation 1 of Predecessor clause 1.

Figure 51: Second instantiation of 'Predecessor'/2 clause 2.



Query 3, substituting body of instantiation 1 of "Predecessor clause 1" into the "initial" query.

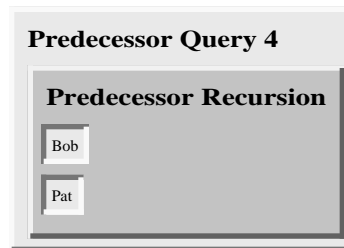
Figure 52: Revised version of 'Predecessor Query'/'0 (Figure 48) based on substituting the body of instantiation 1 of 'Predecessor'/2 clause 1 (Figure 51) for the literal in the body of the original 'Predecessor Query'/'0.

the rule, these corefering variables are also bound. This instantiates the rule clause as shown in Figure 51.

Substituting the body of instantiation 1 of "Predecessor clause 1" for the literal of the original query yields the new version of the query (query 3) shown in . The goal literals, in query 3, share an uninstantiated variable.

SPARCL must now solve the two literals in the body of this revised query. SPARCL is free to satisfy them in any order. Suppose SPARCL tries the 'Predecessor Recursion'/'2 literal first. It checks the *DELAY* clauses and finds that 'Predecessor Recursion'/'2 is to be delayed if the first

argument is an unbound variable. This is the case in query 3, so SPARCL delays solving the 'Predecessor Recursion'/'2 literal and tries the 'Parent'/'2 literal. This one is easily satisfied by matching the fact that "Tom is Bob's parent." This matching binds the shared variable to be "Bob". Removing the "solved" literal "Tom is Bob's parent"



Query 4 is created by removing the solved goal literal for "Parent" from query 3.

Figure 53: Revised version of 'Predecessor Query 3'/'0 (Figure 52) based on removing the solved literal and replacing the remaining variable with its binding ("Bob").

from query 3 leaves us with query 4 (Figure 53).

SPARCL solves the literal of query 4 by instantiating 'Predecessor Recursion'/'2 as shown in Figure 54. It had "delayed" attempting the solution of this literal earlier because

the goal had a variable first argument and there is a *DELAY* clause which specifies this situation as requiring such a goal to be delayed. However, the goal literal no longer has a variable first argument, it is now "Bob". So, SPARCL need not delay solving this literal.

Query 5 in Figure 55 is created by substituting the instantiation of 'Predecessor Recursion'/'2 in Figure 54 into query 4 in Figure 53.

SPARCL solves the literal of query 5 by instantiating "Predecessor clause 2" in Figure 56, similar to the first attempt at solving the initial query in Figure 48.

Query 6 in Figure 57 is created by substituting instantiation 2 of 'Predecessor clause 2'/'0 in

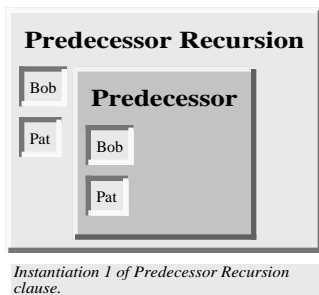


Figure 54: Instantiation 1 of 'Predecessor Recursion'/2.

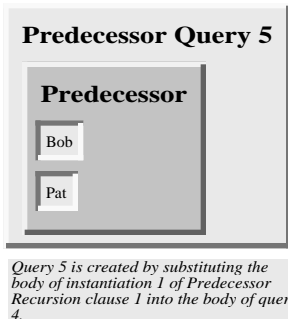


Figure 55: Revised version of 'Predecessor Query 4'/0 (Figure 53) based on substituting the body of instantiation 1 of 'Predecessor Recursion'/2 clause (Figure 54) for the literal in the body of 'Predecessor Query 4'/0.

Figure 56 into query 5. The 'Parent'/2 literal of query 6 matches a 'Parent'/2 fact, and thus is true.

This completes the search for a proof sequence. The sequence being: the literal of query 6 and instance 2 of Predecessor clause 2 derives the literal of query 5; the literal of query 5 and instance 1 of "Predecessor Recursion" derives the literal of query 4; the literal of query 4 and fact "Tom is Bob's parent" derive the literals of query 3; the literals of query 3 and instance 1 of "Predecessor clause 1" derive the the literal of initial query.

The trace of the execution of SPARCL can be pictured as a "tree". The nodes of the tree correspond to goal literals, or lists of goal literals, that are to be satisfied. The arcs between the nodes correspond to the

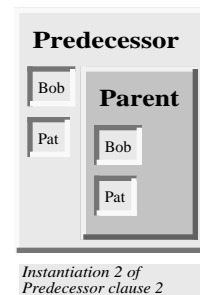


Figure 56: Second instantiation of 'Predecessor'/2 clause 2.

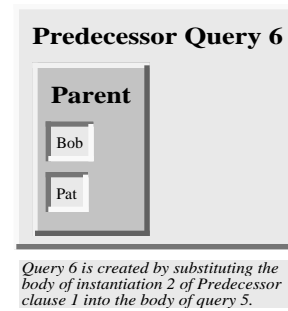


Figure 57: Revised version of 'Predecessor Query 5'/0 (Figure 55) based on substituting the body of instantiation 2 of 'Predecessor'/2 clause (Figure 56) for the literal in the body of 'Predecessor Query 5'/0.

application of (alternative) program clauses that transform the goals at one node into the goals at another node. The top goal is satisfied when a path is found from the root node (top goal) to a leaf node labelled "success". A leaf is labelled "success" if it is a simple fact. The execution of SPARCL programs is the searching for such paths. During the search SPARCL may enter an unsuccessful branch. When SPARCL discovers that a branch fails it automatically backtracks to the previous node and tries to apply an alternative clause at that node.

1.4-What SPARCL programs mean. This section discusses ways to think about the meaning of SPARCL programs.² In the examples so far it has always been possible to understand the results of the program without exactly knowing *how* the system actually found the results. It therefore makes sense to distinguish between two levels of meaning of SPARCL programs;

2. The text of this section is largely drawn from section 1.5 of Ivan Bratko's "Prolog Programming for Artificial Intelligence", 2nd edition.

namely,

- the *declarative meaning* and
- the *procedural meaning*.

The declarative meaning is concerned only with the *relations* defined by the program. The declarative meaning thus determines *what* will be the output of the program. On the other hand, the procedural meaning also determines *how* this output is obtained; that is, how are the relations actually evaluated by the SPARCL system.

The ability of SPARCL to work out many procedural details on its own is considered to be one of its specific advantages. It encourages the programmer to consider the declarative meaning of programs relatively independently of their procedural meaning. Since the results of the program are, in principle, determined by its declarative meaning, this should be (in principle) sufficient for writing programs. This is of practical importance because the declarative aspects of programs are usually easier to understand than the procedural details. To take full advantage of this, the programmer should concentrate mainly on the declarative meaning and, whenever possible, avoid being distracted by the executional details. These should be left to the greatest possible extent to the SPARCL system itself.

This declarative approach indeed often makes programming in SPARCL easier than in typical procedurally oriented programming languages such as Pascal. Unfortunately, however, the declarative approach is not always sufficient. It will later become clear that, especially in large programs, the procedural aspects cannot be completely ignored by the programmer for practical reasons of executional efficiency. Nevertheless, the declarative style of thinking about SPARCL programs should be encouraged and the procedural aspects ignored to the extent that is permitted by practical constraints.

Summary:

- SPARCL programming consists of defining relations and querying about relations.
- A program consists of *clauses*. These are of two types: *facts* and *rules*. A clause can be used to ask a *question*.
- A relation can be specified by *facts*, simply stating the N-tuples of objects that satisfy the relation, or by stating *rules* about the relation.
- A *procedure* (also called a “predicate”) is a set of clauses about the same relation.
- Querying about relations, by means of *questions*, resembles querying a database. SPARCL's answer to a question consists of a set of objects that satisfy the question.
- In SPARCL, to establish whether an object satisfies a query is often a complicated process that involves logical inference, exploring among alternatives and possibly *backtracking*. All this is done automatically by the SPARCL system and is, in principle, hidden from the user.

- Two types of meaning of SPARCL programs are distinguished: declarative and procedural. The declarative view is advantageous from the programming point of view. Nevertheless, the procedural details often have to be considered by the programmer as well.
- The following concepts have been introduced in this section: clause, fact, rule, question; the head of a clause, the body of a clause; recursive rule, recursive definition; procedure; constant, variable; instantiation of a variable; goal (literal); goal is satisfiable, goal succeeds; goal is unsatisfiable, goal fails; backtracking; declarative meaning, procedural meaning.

Next section: 2. SPARCL representation and meaning.