Chapter 18

KNOWLEDGE REPRESENTATION FOR ELECTRONIC CIRCUITS IN LOGIC PROGRAMMING

Takushi Tanaka
Department of Computer Science, Fukuoka Institute of Technology
3-30-1 Wajiro-Higashi, Higashi-ku, Fukuoka Japan, 811-0295,
tanaka@fit.ac.jp

This chapter presents a brief introduction to logic programming and its application to knowledge representation for electronic circuits. We first show how circuits can be represented in terms of predicate logic. Basic concepts in logic programming such as facts, rules, and goals are introduced with circuits as examples. Next, we develop a grammatical method for circuit representation. Circuits are viewed as sentences, and their elements as words. Circuit structures are defined by a logic grammar called DCSG for word-order free languages. A set of grammar rules, when converted into Prolog clauses, forms a logic program which performs top-down parsing. Furthermore, we extend DCSG by introducing semantic terms into the grammar rules. These semantic terms define relationships between syntactic structures and their meanings. By assuming circuit functions to be the meanings of the circuit structures, knowledge of circuit structures and functions is coded into grammar rules. When a given circuit is parsed, not only its syntactic structure is determined, but also its functions are derived as meanings of the sentence.

18.1. Introduction

Prolog is a programming language based on predicate logic. It is well suited for symbolic computations that handle problems between objects and their relations. Programs written in ordinary programming language define procedures to solve problems, while Prolog programs define problems themselves and relationships between objects. Prolog programs are executed by a mechanism of theorem proving. Instead of defining the procedure explicitly, solving problems with the mechanism of theorem proving is called logic programming.

Using electronic circuits as examples, we introduce the basic concepts of logic programming. We first show how circuits are represented in terms of predicate logic. A given circuit is defined by a set of Prolog facts. Circuit structures are defined by Prolog rules. Finding structures in the given circuit is realized by Prolog goals. The examples presented here also show the problems and limitations of this method.

Next, we develop a new method for circuit representation to overcome these problems. In this new method, circuits are viewed as sentences in a word-order free language, and their elements as words. Circuit structures are defined by grammar rules. In order to implement this method, we introduce a logic grammar called DCSG (Definite Clause Set Grammar) for word-order free languages.[4] A set of grammar rules, when converted into Prolog clauses, forms a logic program which performs top-down parsing.

Electronic circuits, both analogue and digital, are designed as a hierarchical structure of functional blocks. Each functional block consists of sub-functional blocks, and each sub-functional block is also decomposed further into sub-sub-functional blocks until the individual parts are reached. Since these hierarchical structures are analogous to syntactic structures of language, DCSG is useful for defining syntactic structures of electronic circuits. However, DCSG is not able to represent knowledge about circuit functions.

Since we need a method to represent knowledge of the relationship between circuit structures and their functions, we develop a new circuit grammar[8] by extending DCSG. The new circuit grammar has semantic fields to define relationships between syntactic structures and their meanings. By assuming circuit functions to be the meanings of the circuit structures, knowledge concerning circuit structures and functions is coded into the grammar rules. When a given circuit is parsed, not only its syntactic structure but also its functions are derived as meanings of the sentence.

18.2. Circuit Representation in Prolog

18.2.1. *Facts*

Using electronic circuits as examples, we will introduce basic concepts of logic programming in Prolog. All objects in circuits are represented by logical nouns called terms, and all relationships between those objects are represented by logical predicates. The terms and the predicates are combined to form atomic formulas, which are logical sentences. The following six atomic formulas represent the circuit *ca*39 shown in Figure 18.1.

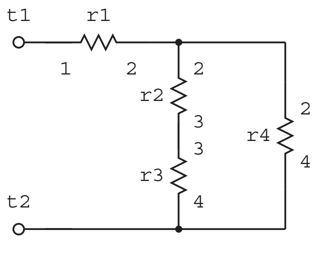

Fig. 18.1. Circuit ca39

$resistor(r1, 1, 2).$
$resistor(r2, 2, 3).$
$resistor(r3, 3, 4).$
$resistor(r4, 2, 4).$
$terminal(t1, 1).$
$terminal(t2, 4).$

The atomic formula "$resistor(r1, 1, 2)$" consists of a predicate symbol "$resistor(...)$" and three constant terms "$r1$", "1" and "2". The atomic formula states that the resistor named $r1$ is connected to node 1 and node 2. The atomic formula "$terminal(t1, 1)$" states that the external terminal named $t1$ is connected to node 1. These six logical sentences describe whole circuit topology of $ca39$. When we consider the circuit in Figure 18.1, these sentences are called "facts", and placed in the Prolog database, which holds true sentences.

18.2.2. *Goal*

Since we are considering the circuit $ca39$ in Figure 18.1, all the formulas that represent the circuit are already placed in the Prolog database. The following atomic formula prefixed by "$?-$" is called a goal clause, which asks the Prolog system whether the resistor $r1$ is connected to the nodes 1 and 2. Here, "$?-$" is a prompt for goal clause generated by the Prolog system. When the goal clause is given, the Prolog system looks for the goal in the Prolog database. Since $resistor(r1, 1, 2)$ is in the database, the goal becomes *true*, and the Prolog system answers *yes*.

$?-\ resistor(r1, 1, 2).$
yes

We can find the nodes to which the resistor "$r2$" is connected by the following goal clause with variables A and B. These variables are also terms which construct atomic formulas. Here, the variables are represented by a string beginning with an upper case letter.

$?-\ resistor(r2, A, B).$

The variables A and B in the goal clause are assumed to be bound by existential quantifiers in predicate logic. Namely, the Prolog system is asked to prove the sentence

$$(\exists A)(\exists B)resistor(r2, A, B)$$

which states that "there exist nodes A and B connected to the resistor $r2$". The Prolog system looks in the database and finds the fact $resistor(r2, 2, 3)$ as an instance for which $resistor(r2, A, B)$ becomes $true$, and the system outputs the variable-value bindings as:

$A = 2$
$B = 3$
yes

These variable-value bindings are made by a mechanism called unification. Unification discovers a variable substitution for the two formulas $resistor(r2, 2, 3)$ and $resistor(r2, A, B)$ that makes them equal.

The following goal clause finds a resistor "X" connected to the nodes 3 and 4 as:

$?-\ resistor(X, 3, 4).$
$X = r3$
yes

Note, however, that if the node order is reversed, the goal fails to find the resistor connected to nodes 4 and 3:

$?-\ resistor(X, 4, 3).$
no

Since resistors are non-polar elements, we want to refer to resistors regardless of their node order. This can be done by defining a new predicate using rules.

18.2.3. *Rules*

A rule in Prolog is a conditional sentence called a definite clause, which is also placed in the Prolog database. A rule consists of a left-hand side called a head, a special symbol " $:-$", and a right-hand side called a body. The head consists of an atomic formula which is the result of the conditional sentence. The symbol "$:-$" is the logical connective of implication "$\leftarrow$", although the order of the condition and result are reversed. The body, which is the conditional part, consists of atomic formulas. Facts in Section 18.2.1 can be viewed as a special case of rules which do not have conditions.

The following rule defines the new predicate "$res(R, A, B)$" which can refer either $resistor(R, A, B)$ or $resistor(R, B, A)$. Here, the symbol ";" works as a logical connective "*or*". This rule can be read if $resistor(R, A, B)$ or $resistor(R, B, A)$ is $true$, $res(R, A, B)$ becomes $true$. Procedurally, in order to show that $res(R, A, B)$ is $true$, the Prolog system tries to show that either $resistor(R, A, B)$ or $resistor(R, B, A)$ is $true$. The variables R, A, and B in facts and rules are assumed to be bound by universal quantifiers in predicate logic.

$$\begin{aligned} res(R, A, B)\ :-\ & resistor(R, A, B);\\ & resistor(R, B, A). \end{aligned}$$

When the following goal is given, the goal is unified with the head of this rule, and the body of the rule becomes a new goal.

$$?-\ res(X, 4, 3).$$

The body generates the following disjunction as the new goal.

$$?-\ resistor(X, 4, 3);\ resistor(X, 3, 4).$$

The first sub-goal of disjunction fails because there is no fact unified with $resistor(X, 4, 3)$ in the database, but the second sub-goal $resistor(X, 3, 4)$ succeeds, and outputs as:

$$X = r3$$
$$yes$$

Similar rules can be defined for other non-polar elements such as capacitors and inductors:

$$\begin{aligned} cap(C, A, B)\ &:-\ capacitor(C, A, B);\\ &\qquad capacitor(C, B, A). \end{aligned}$$

$$\begin{aligned} ind(L, A, B)\ &:-\ inductor(L, A, B);\\ &\qquad inductor(L, B, A). \end{aligned}$$

The following rule defines the predicate "$anyElm(X, A)$" which refers to any element X connected to node A. Here, we assume the external terminal "$terminal(X, A)$" to be a kind of any element. The underscores "_" in atomic formulas are anonymous variables which are not of concern in the rule.

$$\begin{aligned} anyElm(X, A)\ &:-\ terminal(X, A);\\ &\qquad res(X, A, \_\ );\\ &\qquad cap(X, A, \_\ );\\ &\qquad ind(X, A, \_\ ). \end{aligned}$$

18.2.4. *Predicates for circuit structures*

In order to find resistors connected in series (Figure 18.2) in the circuit $ca39$, we attempt to satisfy the following conjunctive goal. Here, "," between two atomic formulas works as the logical connective "*and*".

$$?-\ res(X, A, B),\ res(Y, B, C).$$

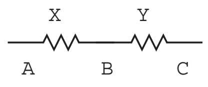

Fig. 18.2. Resistors connected in series

This conjunctive goal successfully finds resistors connected in series. The first sub-goal $res(X, A, B)$ finds the resistor $r2$ and its connecting nodes 2 and 3, then the second sub-goal $res(Y, B, C)$ finds the resistor $r3$ connecting node 3 as:

$A = 2$
$B = 3$
$C = 4$
$X = r2$
$Y = r3$;

The ";" after $Y = r3$ is typed by the user and directs the Prolog system to find another answer. The Prolog system discard the first answer, and tries to find another answer using the Prolog backtracking mechanism. The conjunctive goal also outputs unexpected answers such as:

$A = 1$
$B = 2$
$C = 1$
$X = r1$
$Y = r1$

In the series circuit shown in Figure 18.2, neither elements X and Y nor nodes A and C may be the same. Since the condition "*not* $A = C$" topologically includes the condition "*not* $X = Y$", we next try proving the following goal:

$$?-\ res(X, A, B),\ res(Y, B, C),\ not\ A = C.$$

The Prolog system ceases to output undesired answers of the above type, but it still outputs another type of unexpected answers as follows.

$A = 1$
$B = 2$
$C = 3$
$X = r1$
$Y = r2$

No element other than X and Y may be connected to the central node B of the series circuit. This can be expressed by first defining an element Z other than X and Y connected to B as:

$$\begin{aligned} otherElm(Z, B, X, Y) :-\ & anyElm(Z, B),\\ & not\ Z = X,\\ & not\ Z = Y. \end{aligned}$$

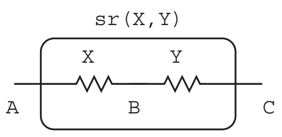

Fig. 18.3. Series connection of resistors

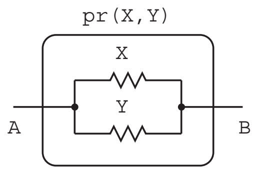

Fig. 18.4. Parallel connection of resistors

Now, we can define the predicate $rSeries(sr(X,Y),A,C)$ for series circuit of resistors by adding the conditions "$not\ A = C$" and "$not\ otherElm(\_\,,B,X,Y)$" to reject undesired answers.

$$\begin{aligned}rSeries(sr(X,Y),A,C) :-\ & res(X,A,B),\\ & res(Y,B,C),\\ & not\ A = C,\\ & not\ otherElm(\_\,,B,X,Y).\end{aligned}$$

The first argument "$sr(X,Y)$" of the predicate $rSeries(\ldots)$ is the name given to the series circuit of resistors X and Y. The name is a form of compound term made of the function symbol $sr(\ldots)$ and variables X and Y. Namely, the name is given by depending on the value of X and Y. This is a technique to give an unique name to a new object, and the function is called a Skolem function.

The predicate for parallel connections is also defined as the same manner:

$$\begin{aligned}rParallel(pr(X,Y),A,C) :-\ & res(X,A,B),\\ & res(Y,A,B),\\ & not\ X = Y.\end{aligned}$$

18.2.5. *Difficulties in circuit representation using predicates*

First we defined terms and predicates for circuit elements, and then we defined predicates for abstract elements and circuit structures. If we could define predicates for all concepts in circuits in the same manner, we would have an axiomatic system of circuits which can automatically derive true sentences on circuits.

However, our circuit representation faces difficulties when we try to define predicates for relationships between two circuits such as equivalent circuit. In order to refer to relationships between circuits, we need a mechanism to identify a set of facts for a specific circuit.

One method is to introduce a new argument for circuit identification into each circuit predicate as:

$resistor(ca39, r1, 1, 2).$
$resistor(ca39, r2, 2, 3).$
$resistor(ca39, r3, 3, 4).$
$resistor(ca39, r4, 2, 4).$
$terminal(ca39, t1, 1).$
$terminal(ca39, t2, 4).$

The first argument $ca39$ of each predicate indicates that each element belongs to the circuit $ca39$.

Although we can refer to relationships between two circuits using identification, we have another problem, namely rewriting circuits. In circuit analysis, we often rewrite circuits into equivalent circuits. Rewriting facts in the Prolog database can be done using the built-in predicates "*assert*" and "*retract*". However using *retract* is problematical from a logical point of view, because using *retract* means erasing facts which were given as true sentence, and the erased facts can never be used again. In the following sections, we resolve these problems by assuming circuits to be a kind of formal language.

18.2.6. *Changing circuit representation*

We have already introduced two kinds of terms, constants and variables, to represent objects in circuits. These terms are viewed as nouns to make logical sentences. Predicate logic has another kind of term, called a compound term, which consists of a function symbol and other terms. In the previous sections, the atomic formula $resistor(r1, 1, 2)$ is a logical sentence which states that "$r1$" is a resistor connected to node 1 and node 2. Here, the $resistor(\ldots)$ worked as a predicate symbol. As predicates and functions are the same in style, we will change $resistor(\ldots)$ from a predicate symbol to a function symbol. As a result, $resistor(r1, 1, 2)$ becomes a compound term instead of an atomic formula. Unlike mathematics, in logic, functions are not computed. The compound term works as a noun phrase, while the atomic formula works as a simple sentence. An atomic formula states a relationship between objects while a compound term represents a new object using other objects. Therefore, the compound term $resistor(r1, 1, 2)$ can be read "resistor $r1$ connected to node 1 and 2" as though it were a noun phrase.

Prolog supports a special data structure called "list". A list is a sequence of any number of terms surrounded by "[" and "]". We can use a list as a new circuit representation for ca39 as follows:

$[resistor(r1, 1, 2),\ resistor(r2, 2, 3),\ resistor(r3, 3, 4),$
$\ resistor(r4, 2, 4),\ terminal(t1, 1),\ terminal(t2, 4)].$

Here, the list is used to represent the set of all the elements that make up the circuit $ca39$. In the next section, we develop a mechanism called DCSG to treat these lists as a word-order free sentence.

18.2.7. *Lists*

Since we will use lists to represent circuits, we discuss lists further here. A list can be viewed as a compound term made by iteratively applying the special function ".". For example, the list $[a,\ b,\ c]$ consists of the function "$.(a,\ .(b,\ .(c, [\ ])))$". Here, a list without elements is called the empty list and is simply written as "$[\ ]$". Using the empty list, the special function "$.(c, [\ ])$" makes the list "$[c]$". The function "$.(b,\ [c])$" makes the list "$[b,\ c]$". And the function "$.(a,\ [b,\ c])$ makes the list "$[a,\ b,\ c]$".

The first element of a list is called the head of the list. The remaining part is another list and is called the tail. That is, a list consists of "$.(Head,\ Tail)$". The special symbol "|" also separates a list into a head and a tail and composes a list as shown in the following goal clauses:

$?-\ [a] = [Head \mid Tail].$
$Head = a$
$Tail = [\ ].$

$?-\ [a,\ b,\ c] = [Head \mid Tail].$
$Head = a$
$Tail = [b,\ c]$

$?-\ X = [a,\ b \mid [c,\ d]].$
$X = [a,\ b,\ c,\ d]$

$?-\ X = [a \mid [b \mid [c \mid [\ ]]]].$
$X = [a,\ b,\ c]$

18.3. Logic Grammar DCSG

18.3.1. *Word-order free language*

Most Prolog systems provide a mechanism for parsing context-free languages called DCG(Definite Clause Grammar).[3] A set of the grammar rules, when converted into Prolog clauses, forms a logic program which executes top-down parsing. Here, we develop a logic grammar DCSG(Definite Clause Set Grammar)[4] for word-order free language similar to the method of DCG.

First, we will introduce a concept of word-order free language. A word-order free language **L(G')** is defined by modifying the definition of a formal grammar. We define a context-free word-order free grammar **G'** to be a quadruple $< V_N, V_T, P, S >$ where: V_N is a finite set of non-terminal symbols, V_T is a finite set of terminal symbols, P is a finite set of grammar rules of the form:

$$\begin{array}{lll} A \longrightarrow B_1, B_2, ..., B_n. & (n \geq 1) \\ A \in V_N, \quad B_i \in V_N \cup V_T & (i = 1, ..., n) \end{array}$$

and $S(\in V_N)$ is the starting symbol. The above grammar rule means that the symbol A is rewritten not with the string of symbols "$B_1, B_2, \ldots, B_n$", but with the set of symbols $\{B_1, B_2, \ldots, B_n\}$. A sentence in the language **L(G')** is a set of terminal symbols which is derived from S by successive application of grammar rules. Here the sentence is a multi-set which admits multiple occurrences of elements taken from V_T. Each non-terminal symbol used to derive a sentence can be viewed as a name given to a subset of the multi-set.

18.3.2. *DCSG conversion*

When a set of grammar rules is given to a Prolog system, the DCG mechanism is used to convert the grammar rules into Prolog clauses. We now develop a conversion for word-order free languages that is analogous to DCG conversion. The general form of the conversion procedure from a grammar rule

$$A \longrightarrow B_1, B_2, ..., B_n. \tag{1}$$

to a Prolog clause is:

$$\begin{array}{ll} \mathit{subset}(A, S_0, S_n) \ :- & \mathit{subset}(B_1, S_0, S_1), \\ & \mathit{subset}(B_2, S_1, S_2), \\ & \ldots \\ & \mathit{subset}(B_n, S_{n-1}, S_n). \end{array} \tag{1'}$$

Here, all symbols in the grammar rule are assumed to be non-terminal symbols. If "$[B_i]$"$(1 \leq i \leq n)$ is found in the right hand side of grammar rules, where "B_i" is assumed to be a terminal symbol, then "$\mathit{member}(B_i, S_{i-1}, S_i)$" is used instead of "$\mathit{subset}(B_i, S_{i-1}, S_i)$" in the conversion.

The arguments $S_0, S_1, \ldots, S_n$ in $(1)'$ are multisets of V_T, represented as lists of elements. The predicate "*subset*" is used to refer to a subset of an object set which is given as the second argument, while the first argument is the name of its subset. The third argument is a complementary set which is the remainder of the second argument less the first; e.g. "$\mathit{subset}(A, S_0, S_n)$" states that "$A$" is a subset of S_0 and that S_n is the remainder.

The predicate "*member*" is defined by the Prolog clauses (2) and (3) below. It has three arguments. The first is an element of a set. The second is the whole set. The third is the complementary set of the first argument.

$$\mathit{member}(M, [M|X], X). \tag{2}$$
$$\mathit{member}(M, [A|X], [A|Y]) \ :- \ \mathit{member}(M, X, Y). \tag{3}$$

When the clause (1)' is used in parsing, an object sentence (multiset of terminal symbols) is given to the argument S_0. In order to find the subset A in S_0, the first sub-goal finds the subset B_1 in S_0 then put the remainder into S_1, the next sub-goal finds B_2 in S_1 then put the remainder into $S_2, \ldots$, and the last sub-goal finds B_n in S_{n-1} then put the remainder into S_n. That is, when a grammar rule is used in parsing, each non-terminal symbol in the grammar rule makes a new set from the given set by removing itself as its subset. While, each terminal symbol used in the grammar rule also makes a new set from the given set by removing itself as its member.

DCSG uses the predicates *subset* and *member* to convert grammar rules into Prolog clauses, but the differences between DCG and DCSG are minimal. If we replace the predicate *subset* with *substring* and remove the clause (3) from the definition of *member*, the conversion will be equivalent to DCG conversion, although ordinary DCG does not use the predicate of *substring* for simplification.

18.3.3. *Backward chaining and top down parsing*

Usually, a Prolog program consists of two kinds of definite clauses, called facts $\{F_1, F_2, \ldots, F_n\}$ and rules $\{R_1, R_2, \ldots, R_m\}$ both viewed as axioms. The execution of a Prolog program can be viewed as a process of deriving a theorem by backward chaining from the axioms. The top-down parsing of a word-order free sentence somewhat resembles the process of backward chaining. The object sentence is given as a set of terminal symbols $\{W_1, W_2, \ldots, W_n\}$. The starting symbol "$S$" is decomposed into terminal symbols using grammar rules $\{G_1, G_2, \ldots, G_m\}$ until they coincide with the given sentence. That is, the set of facts corresponds to the sentence, and the set of backward chaining rules corresponds to the set of grammar rules. Deriving theorems in backward chaining corresponds to identifying non-terminal symbols in the top-down method.

There is an important difference between backward chaining and top-down parsing. Backward chaining allows multiple use of the same fact to derive a theorem, while in a context-free language, each terminal symbol in a sentence contributes only once to the reduction of non-terminal symbols. This characteristic is very useful for avoiding a common looping problem in backward chaining, a problem which is caused by multiple use of the same fact.

18.3.4. *The looping problem*

When problems to be solved are formalized and expressed in Prolog, we often encounter a certain kind of looping problem. We will clarify a cause of this looping problem using voltage derivation as an example.

Assume that the voltages in a circuit are those given in Figure 18.5. We might consider representing this voltage data by the following facts:

$voltage(1, 2, 20).$
$voltage(3, 2, 15).$
$voltage(3, 4, 8).$

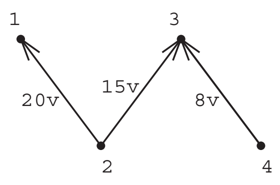

Fig. 18.5. Voltages on a circuit

"$voltage(1, 2, 20)$" states that the voltage between node 1 and node 2 is 20 volts. In order to derive the voltage data independently of the node order, we could consider defining "$volt$" by the following rules:

$$volt(A, B, V) \ :- \ voltage(A, B, V).$$
$$volt(A, B, -V) \ :- \ voltage(B, A, V).$$

Furthermore, we will define the predicate "v" that derives voltages between two arbitrary nodes A and C as

$$v(A, C, V) \ :- \ volt(A, C, V).$$
$$v(A, C, V + W) \ :- \ volt(A, B, V), \ v(B, C, W).$$

It turns out, however, that these definitions will not work as intended. In order to derive the voltage between 1 and 4, we will attempt to execute the following goal clause:

$$?- \ v(1, 4, X).$$

As the voltage between 1 and 4 is not given, the goal is decomposed into sub-goals by the second definition of "$v(\ldots)$". The first sub-goal succeeds by "$volt(1, 2, 20)$" binding node B with 2. The second sub-goal "$v(2, 4, W)$" is also decomposed into sub-goals by the second definition of "$v(\ldots)$" again. The first sub-goal succeeds as "$volt(2, 1, -20)$" using the same voltage data, and the second sub-goal becomes the same as the initial goal. Thus, the system loops.

One method to avoid this problem is to erase voltage data as they are used, so that the same datum is not used twice. This can be done by replacing the definition of "$volt(\ldots)$" with

$$\begin{aligned} volt(A, B, V) \ :- \ & voltage(A, B, V). \\ & retract(voltage(A, B, V)). \\ volt(A, B, -V) \ :- \ & voltage(B, A, V). \\ & retract(voltage(B, A, V)). \end{aligned}$$

But the erased data cannot be recovered in backtracking.

Another common method keeps track of the data used, so that the same datum is not used twice. In this method, the definition of "$v(\ldots)$" is replaced with the following clauses, and we acquire the voltage between nodes 1 and 4 by the goal clause "$?- \ v(1, 4, X, [\,])$". The fourth argument of the goal is a list of nodes which have already been used to calculate the voltages and must not be used twice. This method has the disadvantage of requiring the overhead of explicitly keeping track of the data used:

$$\begin{aligned}
&v(A,C,V,\_) \;:-\; volt(A,C,V).\\
&v(A,C,V+W,T) \;:-\; volt(A,B,V),\\
&\qquad\qquad\qquad\qquad not\ member(B,T,\_),\\
&\qquad\qquad\qquad\qquad v(B,C,W,[A|T]).
\end{aligned}$$

In the next section, we show how to avoid this problem by viewing problem solving as a generalized parsing problem.

18.3.5. *Solution of the looping problem*

To solve the above looping problem, we introduce a change of representation which involves viewing the voltage derivation problem not as a backward search problem, but as a parsing problem. The node-voltage data are not representated by a set of facts. Each expression "$voltage(A,B,V)$" forms a compound term. The voltages in Figure 18.5 are represented by a set of those terms using a list:

$$vData([voltage(1,2,20),\ voltage(3,2,15),\ voltage(3,4,8)]).$$

Accordingly, we represent the voltage derivation not as clauses for backward chaining but as grammar rules for parsing. The following grammar rules correspond to the clauses of "$volt(\ldots)$":

$$\begin{aligned}
&volt(A,B,V) \;\longrightarrow\; [voltage(A,B,V)].\\
&volt(A,B,-V) \;\longrightarrow\; [voltage(B,A,V)].
\end{aligned}$$

"$voltage(A,B,V)$" surrounded by "[" and "]" is a terminal symbol, while "$volt(A,B,V)$" is a nonterminal symbol. Here, we have introduced universally quantified variables (A, B, and V) into the grammar rules. These variables are instantiated when they are applied to object sentences. According to the DCSG conversion procedure, the grammar rules are converted into the following clauses:

$$\begin{aligned}
&subset(volt(A,B,V),S0,S1) \;:-\; member(voltage(A,B,V),S0,S1)\\
&subset(volt(A,B,-V),S0,S1) \;:-\; member(voltage(B,A,V),S0,S1)
\end{aligned}$$

In order to derive the voltage between two arbitrary nodes, we define the following grammar rules corresponding to the clauses of "$v(\ldots)$":

$$\begin{aligned}
&v(A,C,V) \;\longrightarrow\; volt(A,C,V).\\
&v(A,C,V+W) \;\longrightarrow\; volt(A,B,V), v(B,C,W).
\end{aligned}$$

These grammar rules are converted into the following clauses:

$$\begin{aligned}
&subset(v(A,C,V),S0,S1) \;:-\; subset(volt(A,C,V),S0,S1).\\
&subset(v(A,C,V+W),S0,S2) \;:-\; subset(volt(A,B,V),S0,S1),\\
&\qquad\qquad\qquad\qquad\qquad\qquad subset(v(B,C,W),S1,S2).
\end{aligned}$$

Deriving the voltage X between nodes 1 and 4 is accomplished by identifying the nonterminal symbol "$v(1, 4, X)$" in the word-order free sentence of voltage data as follows:

$$?-\ vData(VD),\ subset(v(1,4,X), VD, \_\ ).$$

$$X\ =\ 20\ +\ (-15\ +\ 8)$$

Terminal symbols associated with the nonterminal symbol "$v(1, 4, X)$" are removed sequentially from the object sentence. Therefore, the looping problem due to using the same data repeatedly does not occur. This method is similar to removing data using the predicate "*retract*(...)" described in the previous section. But the removed data can be recovered by backtracking.

We have overcome a common looping problem in backward chaining by simply rewriting backward chaining rules as grammar rules and by viewing a set of facts as a word-order free sentence. The looping problem caused by multiple use of the same fact is avoided by viewing DCSG as a generalized parsing problem.

18.4. Finding Structures in Circuits

18.4.1. *Circuits represented as sentences*

Since we have a mechanism for parsing word-order free languages, we will treat a circuit represented as a list as a word-order free sentence. The circuit *ca*49 in Figure 18.6 is represented by the fact (4) shown below. The predicate *ca*49([...]) states that the word-order free sentence "[...]" represents circuit *ca*49. The compound term *battery*(*b*1, 1, 2) represents the battery *b*1 connected its positive terminal to the node 1 and its negative terminal to the node 2.

$$\begin{aligned}&ca49([resistor(r1,1,3),\ resistor(r2,2,3),\ resistor(r3,1,4),\\ &\quad resistor(r4,2,4),\ resistor(r5,3,4),\ battery(b1,1,2)]).\end{aligned} \tag{4}$$

18.4.2. *Grammar rules without recursion*

Since a resistor is a non-polar element, we need to refer a resistor regardless of its node order. The non-terminal symbol $res(R, A, B)$ can refer either $resistor(R, A, B)$

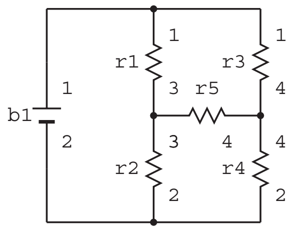

Fig. 18.6. Circuit ca49

or $resistor(R, B, A)$. Here, DCSG allows the symbol ";" as abbreviation of two grammar rules with the same left hand side. The non-terminal symbol $batt(...)$ enables us to refer the same battery with different names. If a battery is referred to with reverse node-order, the name of battery is prefixed by a minus symbol. The non-terminal symbol $anyElm(X, A)$ can refer to any element X connected to the node A.

$$\begin{aligned} res(R, A, B) \longrightarrow\ & [resistor(R, A, B)];\\ & [resistor(R, B, A)]. \end{aligned}$$

$$\begin{aligned} batt(E, A, B) &\longrightarrow [battery(E, A, B)].\\ batt(-E, A, B) &\longrightarrow [battery(E, B, A)]. \end{aligned}$$

$$\begin{aligned} anyElm(X, A) \longrightarrow\ & res(X, A, \_\,);\\ & batt(X, A, \_\,). \end{aligned}$$

18.4.3. *All elements connected to a node*

In circuit analysis, Kirchhoff's current law (KCL) requires that the sum of all branch currents into a node be zero. In order to apply KCL at a specific node, we must find all elements connected to the node. The non-terminal symbol $allElm(X, A)$ successfully finds these elements connected to the node A. Here, X is replaced by a list of the all elements. If the same rules were written in backward chaining by replacing the symbol "$\longrightarrow$" with "`:-`", a looping problem due to using the same data repeatedly would occur.

$$\begin{aligned} allElm([\,], A) \longrightarrow\ & not\ anyElm(\_\,, A).\\ allElm([X|Y], A) \longrightarrow\ & anyElm(X, A),\\ & allElm(Y, A). \end{aligned}$$

The following goal finds all elements connected to node 1 in the circuit $ca49$ as:

$$?-\ ca49(CT), subset(allElm(X, 1), CT, \_\,).$$

$$X = [r1, r3, b1]$$

18.4.4. *Paths and loops*

According to Kirchhoff's voltage law (KVL), the sum of branch voltages along to a loop must be zero. In order to find loops in a circuit, we first define the non-terminal symbol $path(X, A, B)$ which finds routes between two nodes A and B.

$$\begin{aligned} path([X], A, B) &\longrightarrow anyElm(X, A, B).\\ path([X, B|Y], A, C) &\longrightarrow anyElm(X, A, B),\\ & \qquad path(Y, B, C). \end{aligned}$$

The variable X in $path(X, A, B)$ is substituted for by elements and nodes from the starting node A to the ending node B. The following goal attempts to find all routes from the starting node 1 to the ending node 2 in the circuit $ca49$.

$? -\ ca49(CT), subset(path(X, 1, 2), CT, \_).$

$X = [b1];$
$X = [r1, 3, r2];$
$X = [r1, 3, r5, 4, r4];$
$X = [r1, 3, r5, 4, r3, 1, b1]$

The first answer $[b1]$ shows the path from the node 1 via element $b1$ to the node 2. The second answer shows the path from the node 1 via $r1$, node 3, and $r2$ to the node 2. The first three answers show paths from the node 1 to the node 2, but the fourth answer is not desired. It goes back to the starting node 1 and then goes to the ending node 2 via $b1$. Since parsing sentence does not use the same word twice, no element in the circuit appeared twice in the answer. The problem is that our definition does not inhibit the use of the same node twice.

Since we want to get correct answers which do not include loops in the paths, we modify the grammar rules as follows:

$$\begin{aligned}
&path([X], A, B, \_) \longrightarrow\ anyElm(X, A, B).\\
&path([X, B|Y], A, C, T) \longrightarrow\ anyElm(X, A, B),\\
&\qquad\qquad\qquad\qquad\qquad\quad quote\ \ not\ member(B, T, \_),\\
&\qquad\qquad\qquad\qquad\qquad\quad path(Y, B, C, [A|T]).
\end{aligned}$$

The "*quote*" in the grammar rule is a command to the DCSG converter. It directs the converter to insert the following strings as is. Namely, "$not\ member(B, T, \_)$," is inserted as a Prolog clause in the DCSG conversion. The last argument of *path* is substituted for by a list of nodes which are already in the path and not to be used twice as a relay node in the process of finding path.

The following goal successfully finds all paths from the node 1 to the node 2 in the circuit $ca49$. The last argument of *path* is substituted for by "[2]" which shows the ending node 2 must not be used as a relay node.

$? -\ ca49(CT), subset(path(X, 1, 2, [2]), CT, \_).$

$X = [b1];$
$X = [r1, 3, r2];$
$X = [r1, 3, r5, 4, r4];$
$X = [r3, 4, r4];$
$X = [r3, 4, r5, 3, r2];$
No

If the ending node is equal to the starting node, the goal enumerates all loops through the node as follows:

$$?-\ ca49(CT), subset(path(X, 1, 1, [\,]), CT, \_\,).$$

$X = [r1, 3, r5, 4, r3];$
$X = [r1, 3, r5, 4, r4, 2, -b1];$
$X = [r1, 3, r2, 2, -b1];$
$X = [r1, 3, r2, 2, r4, 4, r3];$
$X = [r3, 4, r4, 2, -b1];$
...

18.5. Circuit Grammar for Knowledge Representation

This section extends DCSG to create a new formalism called circuit grammar. This circuit grammar has fields for semantic terms, and defines not only syntactic structures but also the relationships between those structures and their meaning. Unlike DCSG, circuit grammar does not distinguish terminal symbols from non-terminal symbols in the grammar rules. This makes it possible to handle the meanings of words in the same way we handle the meanings of structures.

18.5.1. *Semantic field in left-hand side*

The semantic terms are placed in curly brackets in a grammar rule as follows.

$$A, \{F_1, F_2, ..., F_m\} \ \longrightarrow \ B_1, B_2, ..., B_n. \tag{5}$$

This grammar rule can be read as stating that the symbol A with meaning $\{F_1, F_2, \ldots, F_m\}$ consists of the syntactic structure $B_1, B_2, \ldots, B_n$. This rule is converted into a Prolog clause as follows:

$$\begin{aligned} ss(A, S_0, S_n, E_0, [F_1, F_2, ..., F_m|E_n]) :-\ & ss(B_1, S_0, S_1, E_0, E_1),\\ & ss(B_2, S_1, S_2, E_1, E_2),\\ & \cdots\ ,\\ & ss(B_n, S_{n-1}, S_n, E_{n-1}, E_n). \qquad (5)' \end{aligned}$$

Since this conversion differs from that used in DCSG, we use the predicate "ss" instead of "$subset$". When a rule is used in parsing, the goal $ss(A, S_0, S_n, E_0, E)$ is executed, where the variable S_0 is replaced by an object set (object circuit) and the variable E_0 is replaced by an empty set. The subsets "$B_1, B_2, \ldots, B_n$" are successively identified in the object set S_0. After all of these subsets are identified, the remainder of these subsets (the complementary set) is put into S_n. While, the semantic information from B_1 is added with E_0 and put into E_1, the semantic information of B_2 is added with E_1 and put into $E_2, \ldots$, and the semantic information of B_n is added with E_{n-1} and put into E_n. Finally, the semantic information

$\{F_1, F_2, \ldots, F_m\}$, which is the meaning associated with symbol A, is added and all of the semantic information is put into E.

18.5.2. *Semantic field in right-hand side*

Semantic terms on the right-hand side define the semantic conditions for the grammar rule. For example, the following rule (6) is converted into the Prolog clause (6)' as follows.

$$A \longrightarrow B_1, \{C_1, C_2\}, B_2. \tag{6}$$

$$\begin{aligned} ss(A, S_0, S_n, E_0, E_n) \ :- \ & ss(B_1, S_0, S_1, E_0, E_1), \\ & member(C_1, E_1, \_\,), \\ & member(C_2, E_1, \_\,), \\ & ss(B_2, S_1, S_2, E_1, E_2). \end{aligned} \tag{6'}$$

When the clause (6)' is used in parsing, the conditions C_1 and C_2 are tested to see if the semantic information E_1 meets these conditions after identifying the symbol B_1. If it succeeds, the parsing process goes on to identify the symbol B_2.

18.5.3. *Terminal symbols with semantic fields*

In the new circuit grammar, terminal symbols are defined as grammar rules without a right-hand side to rewrite. This means that terminal symbols are not distinguished from non-terminal symbols. Both terminal and non-terminal symbols are converted with the predicate "*ss*". The terminal symbol A with meaning $\{F_1, F_2, ..., F_m\}$ is written as (7).

$$A, \{F_1, F_2, ..., F_m\}. \tag{7}$$

This rule is converted into the following clause (7)' in circuit grammar.

$$ss(A, S_0, S_1, E_0, [F_1, F_2, ..., F_m | E_0]) \ :- \ member(A, S_0, S_1). \tag{7'}$$

That is, when the rule (7) is used in parsing, the terminal symbol A is searched for in the object set S_0. If it is found, the complementary set is returned from S_1, and the semantic term $\{F_1, F_2, ..., F_m\}$ associated with A is added with the current semantic information E_0 to make the fifth argument of "*ss*". Thus, we can handle the meaning of words in the same way we handle the meanings of structures.

18.5.4. *English interface for semantic term*

Circuit functions are electrical behaviors that are useful to users of the circuit. We first define semantic terms for voltages and currents, the basic components of electrical behaviors of circuits. The compound term "$voltage(A, B)$" represents the voltage at node A to node B. The compound term "$current(A, X)$" represents the current flowing from node A into element X.

The most important knowledge for understanding circuit behavior consists of understanding the causal relationships between these voltages and currents. Some causal relationships are used for their function and others for their side effects. These causal relationships are often represented by numerical formulae and characteristic curves. Since we do not intend to develop a precise circuit simulator but rather aim to create a model for understanding circuits, we formalize only the dependencies between these voltages and currents using the predicate "*cause*". We also use the predicates "*control*" and "*enable*" to represent dependencies between circuits and functions.

Although only a small number of terminal symbols are required to represent circuit topologies, unlike the circuit representation problem, handling semantics, that is, circuit functions, requires that many more functional blocks be defined and that many more terms and predicates for functions be provided. Since the number of these terms and predicates increases as we integrate more knowledge, we define English interfaces whenever we introduce new terms and predicates. The special predicate "$t2e(U, V)$" (Term to English) shows how to read the semantic term U as the noun phrase V in English, and the special predicate "$p2e(U, V)$" (Predicate to English) shows how to read the semantic predicate U as the sentence V in English.

$$
\begin{aligned}
&t2e(voltage(A, B), ['voltage\ at', A, to, B]).\\
&t2e(current(A, X), ['current\ from', A, to, X]).\\
&p2e(cause(A, B), [A, causes, B]).\\
&p2e(control(A, B), [A, controls, B]).\\
&p2e(enable(A, B), [A, enables, B]).
\end{aligned}
$$

18.6. Grammar Rules

18.6.1. *Circuits as Functional Blocks*

Electronic circuits are designed as a hierarchical structure of functional blocks. We now develop grammar rules using the functional blocks appearing in the circuit $cd42$, which is a simple operational amplifier (Figure 18.7). The circuit $cd42$ is represented as the following word-order free sentence. Here, the compound term $npnTr(q1, 3, 5, 6)$ represents the NPN transistor named $q1$ with the base connected to node 3, the emitter to node 5, and the collector to node 6 respectively.

$$
\begin{aligned}
\text{cd42}([\ &resistor(r1, 2, 10), resistor(r2, 9, 1), npnTr(q1, 3, 5, 6),\\
&npnTr(q2, 4, 5, 7), npnTr(q3, 10, 1, 5), npnTr(q4, 10, 1, 10),\\
&npnTr(q5, 10, 1, 8), npnTr(q6, 8, 9, 2), pnpTr(q7, 7, 2, 6),\\
&pnpTr(q8, 7, 2, 7), pnpTr(q9, 6, 2, 8), terminal(t1, 3),\\
&terminal(t2, 4), terminal(t3, 2), terminal(t4, 9),\\
&terminal(t5, 1)]).
\end{aligned}
$$

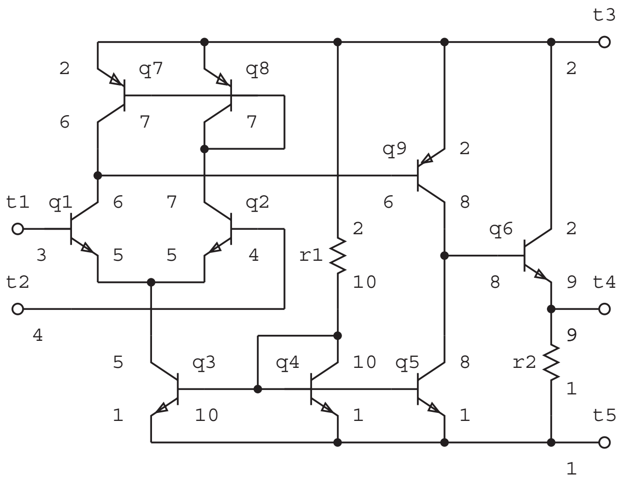

Fig. 18.7. Circuit cd42

18.6.2. *Terminal Symbols*

Unlike the DCSG formalism described in Section 18.3, this new circuit grammar does not distinguish terminal symbols from non-terminal symbols in grammar rules. Terminal symbols are defined by rules that do not have a right-hand side for rewriting. For example, the following grammar rule without a right-hand side defines a resistor as a terminal symbol.

$$resistor(R, A, B).$$

When a $resistor(R, A, B)$ is identified in a circuit, we can assume the existence of a voltage across the resistor and a current through the resistor. The voltage and the current obey the constraints of Ohm's law as represented by the equation $V = IR$. Since we intend to construct a model for understanding circuits, we only focus on the causality of the voltage and the current. Although it is difficult to decide which is the cause and which is the effect between the voltage and the current, we often say: *"Since a voltage is applied to a resistor, a current flows through the resistor"*, or *"Since a current flows through a resistor, a voltage appears across the resistor"*. These views form a causal chain of voltages and currents, which becomes important to understanding the electrical behavior of circuits. Fault diagnosis for an electronic circuit often consists of finding a defect in this causal chain.

These causalities can be added to the definition of *resistor* as semantic terms as follows.

$$\begin{aligned}&resistor(R, A, B),\\&\{\ cause(voltage(A, B), current(A, R)),\\&\quad cause(current(A, R), voltage(A, B))\}.\end{aligned}$$

Using the English interface described in Section 18.5, the compound term "$cause(voltage(A, B), current(A, R))$" can be read: *The voltage at A to B causes the current from A to R.* While "$cause(current(A, R), voltage(A, B))$" can be read: *The current from A to R causes the voltage at A to B.* Actually, these causalities are added to the rules for non-polar element defining $res(R, A, B)$ rather than the terminal symbol so that resistors can be referred to in reverse node-order.

When an NPN transistor is found in a circuit, we can assume the existence of its voltages and currents. Usually, relationships between these voltages and currents are represented by characteristic curves of the transistor. Here, we formalize only the qualitative aspects of these voltages and currents in the following grammar rules.

$$
\begin{array}{l}
npnTr(Q, E, B, C),\\
\{\ state(Q, active),\\
\quad gt(voltage(C, E), vst),\\
\quad equ(voltage(B, E), vbe),\\
\quad gt(current(B, Q), 0),\\
\quad gt(current(C, Q), 0),\\
\quad cause(voltage(B, E), current(B, Q)),\\
\quad cause(current(B, Q), current(C, Q)),\\
\quad cause(signal(voltage(B, E)), signal(current(B, Q))),\\
\quad cause(signal(current(B, Q)), signal(current(C, Q)))\}.
\end{array}
$$

This grammar rule defines an NPN transistor in the active state. The compound terms "$gt(voltage(C, E), vst)$" represents that $voltage(C, E)$ is greater than the collector saturation voltage vst. The compound term "$equ(voltage(B, E), vbe)$" represents that $voltage(B, E)$ is equal to the base-emitter forward voltage vbe. The compound term "$cause(current(B, Q), current(C, Q))$" represents that $current(B, Q)$ causes $current(C, Q)$. Here, $current(B, Q)$ is intended to represent DC-current, while $signal(current(B, Q))$ represents a small signal current.

Since we have introduced new terms and predicates in the grammar rules, we also define the followings English interfaces.

$$
\begin{array}{l}
t2e(vst, ['collector\ saturation\ voltage']).\\
t2e(vbe, ['base\ forward\ voltage']).\\
t2e(signal(X), ['signal\ of', X]).\\
p2e(state(Q, S), [Q, '\ is\ in', S, state]).\\
p2e(gt(A, B), [A, '\ is\ greater\ than', B]).\\
p2e(equ(A, B), [A, '\ is\ equal\ to', B]).
\end{array}
$$

Grammar rules for NPN-transistors in the saturated state and the cutoff state are also defined in the same manner. Similar rules are also defined for PNP-transistors.

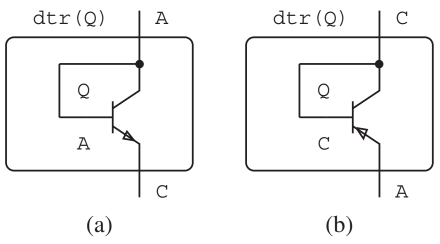

Fig. 18.8. Diode-connected transistor

If a terminal symbol $npnTr(Q, E, B, C)$ is identified in a given sentence, one of its semantic terms (active state, saturation state, and cutoff state) is added as a meaning of the terminal symbol non-deterministically. Usually, if the functional block containing a transistor requires a specific electrical state, the required state will be selected.

18.6.3. *Non-Terminal Symbols*

A transistor in which the base and collector are connected together works as a diode (Figure 18.8). The following grammar rule defines a diode-connected transistor "$dtr(dtr(Q), A, C)$" in the *conductive* state. The syntactic part of the right-hand side defines either an NPN-transistor Q or a PNP-transistor Q whose base and collector are connected to the same node. Here, Q is a name of the original transistor, and $dtr(Q)$ is a name given to the diode (Skolem function). The semantic terms in the left-hand side show that the current flows from A to $dtr(Q)$ in the conductive state, and that the current causes the $voltage(A, C)$. The semantic term $state(Q, active)$ in the right-hand side is an electrical condition which requires that the transistor Q must be in the *active* state. The grammar rule for a diode-connected transistor in the cutoff state is also defined in the same manner.

$$\begin{aligned}
&dtr(dtr(Q), A, C),\\
&\{state(dtr(Q), conductive),\\
&\quad gt(current(A, dtr(Q)), 0),\\
&\quad cause(current(A, dtr(Q)), voltage(A, C))\} \longrightarrow\\
&\qquad\qquad\qquad\qquad (\, npnTr(Q, A, C, A);\ pnpTr(Q, C, A, C)\,),\\
&\qquad\qquad\qquad\qquad \{state(Q, active)\}.
\end{aligned}$$

The following grammar rule defines a simple voltage regulator with a Vbe (0.6 volt) output. The syntactic part of the right-hand side defines the structure shown in Figure 18.9. The semantic condition $state(D, conductive)$ in the right-hand side requires that the diode D must be in conductive state. The semantic term $control(vreg(D, R), voltage(Out, Com))$ in the left-hand side defines a function of

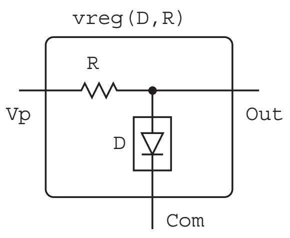

Fig. 18.9. Vbe-voltage regulator

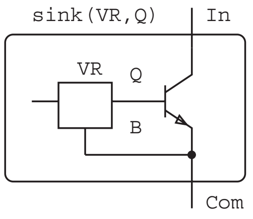

Fig. 18.10. Current source (sink type)

voltage regulator $vreg(D, R)$ that controls the voltage between Out and Com.

$$\begin{aligned}
&vbeReg(vreg(D,R), Vp, Com, Out),\\
&\{control(vreg(D,R), voltage(Out, Com))\} \longrightarrow\\
&\qquad\qquad\qquad dtr(D, Out, Com),\\
&\qquad\qquad\qquad \{state(D, conductive)\},\\
&\qquad\qquad\qquad res(R, Vp, Out).
\end{aligned}$$

The following grammar rule defines the current source (sink type) shown in Figure 18.10. The right-hand side of the grammar rule has a disjunction of syntactic and semantic conditions, which requires either an existence of a voltage regulator VR or the semantic information that the voltage between B and Com is controlled by a voltage regulator VR. This type of grammar rule becomes useful to parse context-dependent circuits (Section 18.7).

$$\begin{aligned}
&cSink(sink(VR,Q), In, Com),\\
&\{control(sink(VR,Q), current(In, Q))\} \longrightarrow\\
&\qquad\qquad\qquad (\ vbeReg(VR, \_\ , Com, B);\\
&\qquad\qquad\qquad \ \ \{control(VR, voltage(B, Com))\}),\\
&\qquad\qquad\qquad npnTr(Q, B, Com, In),\\
&\qquad\qquad\qquad \{state(Q, active)\}.
\end{aligned}$$

The following grammar rule defines the active load (current mirror) shown in Figure 18.11. The semantic terms in the right-hand side are the electrical conditions that operate the circuit. The semantic term in the left-hand side shows that the

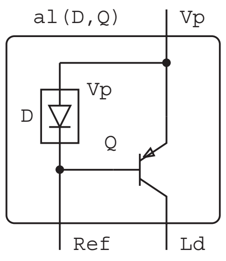

Fig. 18.11. Active load

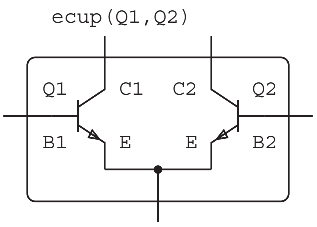

Fig. 18.12. Emitter coupled pair

current from Q to Ld is controlled by the active load $al(D,Q)$, and the current is caused by the current through Ref, and that these two currents become equal.

$$\begin{aligned}
&activeLoad(al(D,Q), Ref, Vp, Ld),\\
&\{\ control(al(D,Q), current(Q,Ld)),\\
&\quad cause(current(al(D,Q), Ref), current(Q,Ld)),\\
&\quad equ(current(Q,Ld), current(al(D,Q), Ref))\} \longrightarrow\\
&\qquad\qquad\qquad dtr(D, Vp, Ref),\\
&\qquad\qquad\qquad \{state(D, conductive)\},\\
&\qquad\qquad\qquad pnpTr(Q, Ref, Vp, Ld),\\
&\qquad\qquad\qquad \{state(Q, active)\}.
\end{aligned}$$

An emitter-coupled pair is defined as follows (Figure 18.12). Although it has no semantic terms, it is useful to define the syntactic structure.

$$\begin{aligned}
&eCoupledPair(ecup(Q1,Q2), B1, B2, E, C1, C2) \longrightarrow\\
&\qquad\qquad\qquad npnTr(Q1, B1, E, C1),\\
&\qquad\qquad\qquad npnTr(Q2, B2, E, C2).
\end{aligned}$$

A single-ended differential amplifier consists of an active load, an emitter-coupled pair, and a current source as shown in Figure 18.13.

$$\begin{aligned}
&sDiffAmp(sdAmp(EC, AL, CS), B1, B2, C1, Vp, Vm),\\
&\{input(sdAmp(EC, AL, CS),\\
&\qquad\qquad\qquad difference(voltage(B1, Vm), voltage(B2, Vm))),\\
&\quad output(sdAmp(EC, AL, CS), current(C1, sdAmp(EC, AL, CS))),
\end{aligned}$$

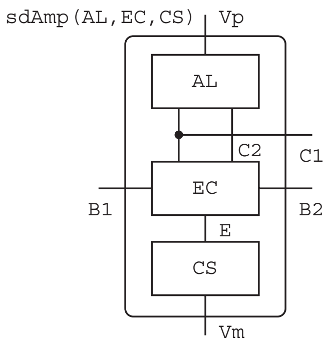

Fig. 18.13. Single-ended differential amplifier

$$\begin{aligned}
&cause(voltage(B1, B2), current(C1, sdAmp(EC, AL, CS))),\\
&suppress(CS, common\_mode\_gain(sdAmp(EC, AL, CS))),\\
&double(AL, current\_gain(EC))\} \longrightarrow\\
&\qquad\qquad\qquad eCoupledPair(EC, B1, B2, E, C1, C2),\\
&\qquad\qquad\qquad activeLoad(AL, C2, Vp, C1),\\
&\qquad\qquad\qquad cSink(CS, E, Vm).
\end{aligned}$$

In order to parse the circuit $cd42$, the common-emitter, the emitter-follower, and the operational amplifier are defined similarly.

Since we have introduced new semantic terms and predicates for circuit functions, we now define their English interfaces as follows:

$$\begin{aligned}
&t2e(common\_mode\_gain(X), ['common\_mode\ gain\ of', X]).\\
&t2e(current\_gain(X), ['current\_gain\ of', X]).\\
&t2e(voltage\_gain(X), ['voltage\_gain\ of', X]).\\
&t2e(input\_impedance(X), ['input\_impedance\ of', X]).\\
&t2e(output\_impedance(X), ['output\_impedance\ of', X]).\\
&t2e(load\_impedance(X), ['load\_impedance\ of', X]).\\
&t2e(difference(X, Y), [difference, X, from, Y]).\\
&p2e(high(X), [X, is, high]).\\
&p2e(low(X), [X, is, low]).\\
&p2e(amplify(X, Y), [X, amplifies, Y]).\\
&p2e(control(X, Y), [X, controls, Y]).\\
&p2e(input(X, Y), [X, is, input, to, Y]).\\
&p2e(output(X, Y), [X, outputs, Y]).\\
&p2e(suppress(X, Y), [X, suppresses, Y]).\\
&p2e(double(X, Y), [X, doubles, Y]).\\
&p2e(cause(C, E), [C, causes, E]).\\
&p2e(enable(X, Y), [X, enables, Y]).
\end{aligned}$$

18.7. Parsing Circuits

All of the grammar rules defined in the previous section are converted into Prolog clauses according to the circuit grammar conversion method described in Section 18.5. The clauses form a logic program that performs top-down parsing. The following goal parses the circuit *cd*42 and derives the circuit structure and functions. The first sub-goal substitutes the circuit *cd*42 into the variable *Circuit*. The first three arguments of the predicate *ss* are the same arguments as the predicate *subset* in DCSG. That is, the first argument is the name of a subset, the second argument is the whole set, and the third argument is its remainder. Since the third argument of *ss* is empty, the second sub-goal tries to identify the whole *Circuit* as a functional block X. Since the forth argument of *ss* is empty, no semantic information is given to parse the circuit. When the goal successfully parses the circuit, the circuit functions are substituted into Y as semantic information.

$$\begin{aligned} ?- \quad & cd42(Circuit),\\ & ss(X, Circuit, [\,], [\,], Y). \end{aligned}$$

$$\begin{aligned} X = \; & operationalAmp(\,opAmp(sdAmp(ecup(q1,q2),\\ &\qquad\qquad al(pdtr(q8),q7),\\ &\qquad\qquad sink(vreg(dtr(q4),r1),q3)),\\ &\qquad pnpCE(q9, sink(vreg(dtr(q4),r1),q5)),\\ &\qquad npnEF(q6,r2)),\\ &\quad 3,4,9,2,1) \end{aligned}$$

The value of X shows that the circuit *ca*42 is identified as an operational amplifier "$operationalAmp(opAmp(\ldots),3,4,9,2,1)$". The first argument $opAmp(...)$ is a name given to the identified circuit, and the rests are the connecting nodes in the circuit. The given name keeps track of identified functional blocks, and is viewed as a parse tree which shows the syntactic structure of the circuit (Figure 18.14). Each node represents a functional block identified in the circuit *cd*42.

If a circuit is generated with only context-free grammar rules, its parse tree becomes a simple tree. Note, however, that the parse tree for *cd*42 has a structure with shared nodes. This means that two current sinks share a single voltage regulator. That is, the circuit designer used a context-dependent grammar rule which collapsed two voltage regulators into one voltage regulator. The semantic condition of the current sink (Figure 18.10) enables us to parse this context dependent circuit.[5]

Each time a functional block is identified in parsing, semantic information about the functional block is added. After parsing, the value of Y has a large amount of semantic information about the circuit as follows.

$$\begin{aligned} Y = [& input(opAmp(...), voltage(3,4)),\\ & output(opAmp(...), voltage(9,1)),\\ & cause(voltage(3,4), voltage(9,1)), \end{aligned}$$

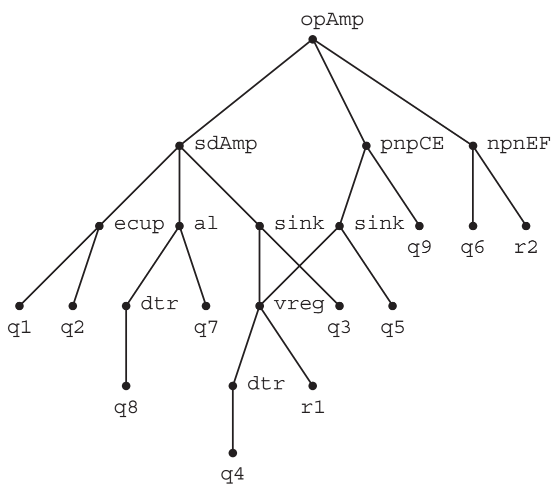

Fig. 18.14. Parse Tree for *cd*42

$enable(sdAmp(...), amplify(opAmp(...), differential\_inputs)),$
$enable(pnpCE(...), high(voltage\_gain(opAmp(...)))),$
$enable(npnEF(...), low(output\_impedance(opAmp(...)))),$
$input(npnEF(...), voltage(8, 1)),$
$output(npnEF(...), voltage(9, 1)),$
$cause(voltage(8, 1), voltage(9, 1)),$
$high(input\_impedance(npnEF(...))),$
$low(output\_impedance(npnEF(...))),$
$equ(voltage\_gain(npnEF(...)), 1),$
$cause(voltage(9, 1), current(9, r2)),$
$cause(current(9, r2), voltage(9, 1)),$
$state(q6, active),$
$gt(voltage(2, 9), vst),$
... 120 lines omitted ...)]

18.8. Functional Explanations in English

After parsing a circuit, we will have acquired a large amount of semantic information about the circuit *cd*42. This information consists of causal relationships between voltages and currents, electrical states of devices, and electrical behaviors and functions of the circuit. Using the English interface and the data defined by "*p*2*e*" and "*t*2*e*", the following sentences are generated from the semantic information.

$? - cd42function(X), write\_function(X).$

voltage at 3 *to* 4 *is input to opAmp*(...).
opAmp(...) *outputs voltage at* 9 *to* 1.

```
voltage at 3 to 4 causes voltage at 9 to 1.
sdAmp(...) enables that opAmp(...) amplifies differential_inputs.
pnpCE(...) enables that voltage_gain of opAmp(...) is high.
npnEF(...) enables that output_impedance of opAmp(...) is low.
voltage at 8 to 1 is input to npnEF(...).
npnEF(...) outputs voltage at 9 to 1.
voltage at 8 to 1 causes voltage at 9 to 1.
input_impedance of npnEF(...) is high.
output_impedance of npnEF(...) is low.
voltage_gain of npnEF(...) is equal to 1.
voltage at 9 to 1 causes current from 9 to r2.
current from 9 to r2 causes voltage at 9 to 1.
q6 is in active state.
voltage at 2 to 8 is greater than vst.
... 120 lines omitted ...
```

18.9. Conclusions

We have developed a grammatical method for circuit representation. Circuits were viewed as sentences, and their elements as words. Circuit structures were defined by a logic grammar called DCSG for word-order free languages. A set of grammar rules, when converted into Prolog clauses, forms a logic program which performs top-down parsing. Although DCSG can recognize the syntactic structures of circuits, it could not handle the relationships between circuit functions.

Since electronic circuits are designed to achieve specific functions, we need a method to represent knowledge of circuit structures and functions. The newly developed circuit grammar described here defines not only syntactic structures, but also the relationships between structures and meanings. That is, we take a circuit's function to be the meaning of that circuit. Knowledge of circuit structures and functions was then coded into grammar rules as semantic information. Using these grammar rules, sample circuits were parsed and their structures and functions were derived successfully.

Our English interface generated more than 130 sentences for the circuit functions of even the simple operational amplifier *cd*42. Although these sentences are not ordered in contents, each sentence is useful in understanding circuit behavior. We can also derive causal chains of voltages and currents from the parse tree and semantic information, and these chains will be useful in troubleshooting.

Since grammar rules are additive, knowledge of new circuits can be added as new grammar rules. As more grammar rules are defined, more circuits can be parsed. The knowledge and mechanisms developed here achieve automatic circuit understanding which can help engineers design and troubleshoot circuits.

References

1. S.A. Greibach, *A New Normal Form Theorem for Context-Free Phrase Structure Grammars*, JACM, vol. 12, pp. 42–52, 1965.
2. J. deKleer, *Causal and Teleological Reasoning in Circuit Recognition*, MIT, AI-TR-529, 1979.
3. F.C.N. Pereira and D.H.D. Warren, *Definite Clause Grammars for Language Analysis*, Artificial Intell., Vol. 13, pp. 231–278, 1980.
4. Takushi Tanaka, *Definite Clause Set Grammars: A Formalism for Problem Solving*, J. Logic Programming, Vol. 10, pp. 1–17, 1991.
5. Takushi Tanaka, *Parsing Circuit Topology in A Logic Grammar*, IEEE-Trans. Knowledge and Data Eng., Vol. 5, No. 2, pp. 225–239, 1993.
6. T. Tanaka and O. Bartenstein, *DCSG-Converters in Yacc/Lex and Prolog*, Proc. 12th International Conference on Applications of Prolog, pp. 44–49, 1999.
7. Takushi Tanaka, *A Logic Grammar for Circuit Analysis — Problems of Recursive Definition*, Lecture Notes in Artificial Intelligence Vol. 4693, Springer, pp. 852–860, 2007.
8. Takushi Tanaka, *Circuit grammar: knowledge representation for structure and function of electronic circuits*, Int. J. Reasoning-based Intelligent Systems, Vol. 1, Nos. 1/2, pp. 56–67, 2009.
9. Takushi Tanaka, *Deriving Electrical Dependencies from Circuit Topologies Using Logic Grammar*, Lecture Notes in Artificial Intelligence Vol. 5712, Springer, pp. 325–332, 2009.
10. Takushi Tanaka, *A Mechanism for Converting Circuit Grammars to Definite Clauses*, Lecture Notes in Computer Science, Vol. 6278, Springer, pp. 190–199, 2010.
11. Takushi Tanaka, *Deriving electrical dependencies from circuit topologies using logic grammar*, Int. Journal of Reasoning-based Intelligent Systems, Vol. 3, No. 1, pp. 28–33, 2011.
12. Takushi Tanaka, *Analyzing Circuit Structures as Language*, to appear in Circuit Analysis (Virginia E. Wright ed.), Nova Science Publishers Inc. NY.
13. *101 Analog IC Designs*, Sunnyvale, CA: Interdesign Inc., 1976.