Title	Negation Technique for Context-Free Grammars
Author(s)	YAMASHITA, Yoshiyuki; NAKATA, Ikuo
Citation	数理解析研究所講究録 (1989), 709: 48-67
Issue Date	1989-12
URL	http://hdl.handle.net/2433/101661
Right	
Type	Departmental Bulletin Paper
Textversion	publisher

数理解析研究所講究録
第 709 巻 1989 年 48-67

Negation Technique for Context-Free Grammars

Yoshiyuki YAMASHITA, *Doctoral Program in Engineering,*
(kougaku-kenkyuu-ka)
and
Ikuo NAKATA, *Institute of Information Sciences and Electronics,*
(densi-jyouhou-kougaku-kei)
University of Tsukuba,
(tsukuba-daigaku)
Tsukuba-shi, Ibaraki-ken 305.

1 Introduction

Negation in logic programs has been studied by many reseachers well [Cla] [Ll] [Na] [SaTa] [Sch]. The authors are studying a new programming paradigm called grammatical programming [YaNa 1], which is based on context-free grammars and has the close with logic programming [YaNa 2]. Therefore the study for negation in context-free grammars may contribute to the grammatical programming. This is our motivation to write this paper. Of course, the well known theory of CFGs [AhUl] [Sal] has no concept of negation. Therefore we have to extend the theory of CFGs.

There are four hints about this topics from the view point of the symbolic logic and logic programming [ChLe] [Ll].

Firstly, the naive idea of negation is that it expresses the complementary set of a given set when we discuss on sets of some kind of objects. For example, we suppose that the predicate even(n) is true if n is a member of the set of even numbers. Then ~even(n) is true if n is a member of the complementary set of even numbers, where ~ is the negative symbol. If the universe of discourse is only limited to the set of natural numbers, the complementary set means the set of all the odd numbers. This naive discussion gives us the first idea of an extension of CFGs. The following is the context-free production rules which express even numbers,

Even→0,
Even→s(s(Even)),

where Even is a nonterminal symbol, 0 is a constant, and s is the successor function. We can say that Even means the set of even numbers. Then it is natural to say that ~Even means the

complementary set of even numbers because we regard the negative symbol ~ as a kind of complementary operator. If the universe is the set of natural numbers, it is the set of odd numbers. Therefore the production rule

Odd→~Even,

can be interpreted that Odd means odd numbers. This interpretation is simple. However, we can think of another use of the negative symbol as follows,

What→0,
What→s(~What).

which can not be understood with the above naive interpretation of negation. We need the complete formulation of negation.

Secondly, one of the formal systems which can treat negation is the symbolic logic. In this system so called De Morgan's law holds, namely

$$\sim(p \vee q) = \sim p \wedge \sim q,$$

for any propositions p and q, where $\wedge$ is the conjunctive symbol and $\vee$ is the disjunctive symbol. This law suggests that the formal system which can treat negation and disjunction in the sense of the symbolic logic should also treat conjunction. We can easily introduce a disjunctive symbol "|" into CFGs in the sense of Backus normal form, and interpret $X \rightarrow \alpha \mid \beta$ as the pair of $X \rightarrow \alpha$ and $X \rightarrow \beta$. If we introduce the negative symbol, we should also have a conjunctive symbol "&", and the following rule

$$P \rightarrow \sim(\alpha \mid \beta)$$

should have the same meaning as that of

$$P \rightarrow \sim\alpha \ \& \ \sim\beta.$$

Furthermore, the negative, conjunctive and the disjunctive operations in our extended CFGs should have the same features as those in the symbolic logic.

Thirdly, it has been known that every definite clause program can be automatically transformed into the equivalent **Coupled Context-Free Grammar (CCFG)** and vice versa [YaNa 1] [YaNa 2]. By this transformation rule, a set of definite clause program, for example

even(0),
even(s(s(X))):-even(X),

can be transformed into the equivalent CFG, that is, in this case into

Even→0,

Even→s(s(Even)).

This relation between definite clauses and production rules leads us to the declarative interpretation of production rules. Because the implication p:-q is equivalent to the disjunction p ∨ ~q in the symbolic logic, the production rule P→Q should be interpreted as P | ~Q if we are going to establish our extended CFGs as an analogy of the symbolic logic.

And fourthly, there is the program transformation technique, called the **negation technique** [SaTa] [Sat], which automatically transforms the definite clause program L which defines a predicate symbol p into the negated program Not_L which defines the predicate not_p. Here an atomic formula p(*a*) can not be proven in L if not_p(*a*) can be proven in Not_L. For example, the following program

```
even(0),
even(s(s(X))):-even(X),
```

is transformed into

```
not_even(s(0)),
not_even(s(s(X))):-not_even(X).
```

Because every definite clause program can be automatically transformed into CFGs, we expect that there is the similar technique, which transforms a CFG: G into the negated grammars Not_G (see below).

L	⇒	negation technique	⇒	Not_L
⇑				⇑
⇓				⇓
G	⇒	negation technique?	⇒	Not_G

L is a definite clause program, and Not_L is its negation. G is a CFG equivalent to L, and Not_G is equivalent to Not_L. *Can we define such a negation technique for CFGs ?*

The Relations between Several Transformations

If we can define the negation technique for CFGs, the above three suggestions:

(1) The negative operation in our grammar system is the complementary operation,

(2) Our system has the negative, conjunctive and disjunctive operations as the symbolic logic has.

(3) A production rule in our system is interpreted in the same way as an implication in the symbolic logic,

should play important roles in the negation technique without any contradictions. The negation technique for logic programs gives us more suggestions about the features of negation in CFGs.

In this paper we try to reply to the above suggestions by defining the formal system, called

term logic, which covers the whole theory of CFGs with negations. The term logic is a formal system for terms composed of constants, functors (function symbols), nonterminal symbols, two special symbols $\bot$ and $\top$, the conjunctive symbol &, the disjunctive symbol | and the negative symbol ~. Every term has its interpretations and models similar to Herbrand interpretations and Herbrand models in the first-order logic, respectively. The concepts of equivalence, satisfiability, unsatisfiability and logical consequence are defined in the same way as in the first-order logic. Context-free production rules are interpreted as a special form of terms according to the above suggestion (3), and a CFG is defined as a set of such terms, called a **general Context-Free Grammar (general CFG)**. In the same way as in a general logic program with negations [Ll], the semantics of a general CFG are defined by using its completed definition. For example, the following general logic program

what(0),
what(s(X)):-~what(X).

is completed as

$$\text{what}(X) \leftrightarrow X = 0 \vee \exists Y(X=s(Y) \wedge \sim\text{what}(Y)),$$

and the general CFG

What→0,
What→s(~What).

is completed as

What↔0 | s(~What).

For the completed definition, we define two kinds of semantics. One is the declarative one, the set of logical consequences from the model semantics for the term logic, and the other is the operational one like the derivation procedure in the usual sense.

Based on the term logic, the negation technique for general CFGs is also defined in the same way as for logic programs. By using this technique, for example, we can automatically transform the CFG which defines even numbers into the negated one which defines odd numbers.

The authors are studying CCFGs as general purpose programs [YaNa 3]. We call such a programming methodology grammatical programming or CCFG programming. From this point of view, the introduction of the negation into CFGs becomes the basis of introducing negations into CCFG programming. This will consequently increases the expressive power of CCFGs. However it is difficult to treat negations correctly as we have already learnt in the case of logic programs. For example, the built-in predicate *not* in Edinburgh Prolog [ClMe] can not deliver the positive binding information a goal *not*(...) produces into the other goals. By applying the negation technique, such *not*'s can be eliminated from a logic program. The transformed program can produce positive bindings. The similar effects are also expected in CCFG programming, though we do not discuss about it here any more. We study its theoretical basis here . The introduction of negation into CCFG programming will be our advanced work.

In section two, we show an example of negation techniques for CFGs. There the CFG which defines even numbers is transformed into the CFG which defines odd numbers. We see that each

step of transformation is based on the above suggestions. The discussion in this section leads us to the definition in section three.

In section three, we define the syntax and semantics of the term logic, especially the model semantics similar to that in the first-order logic.

In section four, we define the syntax and semantics of CFGs. As we have already shown above without notice, the CFGs defined here treat *term*s, whereas they treat *string*s in usual. The reason is because strings are more difficult to treat than terms theoretically.

In section four, the negation technique informally introduced in section two is precisely defined. Two kind of transformation rules which preserve the semantics of grammars are given. One is to eliminate negative symbols, and the other is to purge redundant information from the grammars. The negation technique is defined over the rules. Complementarity and duality are also defined in the same way as in logic programs.

2 An example of negation technique

In this section we show a simple example of the negation technique informally as an analogy of the negation technique for logic programs [Sata 1][Sat]. From the negation technique for logic programs, we can automatically transform the following logic program which defines even number

```
even(0).
even(s(s(X))) :- even(X).
```

into the logic program which defines odd numbers as follows (see [Sat] for the details of this transformation).

```
odd(s(0)).
odd(s(s(X))) :- odd(X).
```

In the same way as this technique we can show the transformation technique by which the CFG defining even numbers is transformed into the CFG defining odd numbers.

The following is the CFG which defines even numbers,

```
Even→0,
Even→s(s(Even)).
```

where Even is a nonterminal symbol, 0 a constant, and s(...) a successor function symbol. First of all these two production rules are completed as follows in the same sense as in logic programs

```
Even↔0 | s(s(Even)).
```

Here the symbol "↔" expresses the equality between the left-hand side and the right-hand side, and the symbol "|" means the disjunction as in the BNF notation. The precise meanings of these

symbols are described in the following sections. Therefore the above rule can be read as "Even is equivalent to 0 or s(s(Even))" intuitively. The above rule is negated as follows

$$\sim Even \leftrightarrow \sim(0 \mid s(s(Even))).$$

As the analogy of the logic, it is natural to assume that ~(0 | s(s(Even))) is equivalent to ~0 & ~s(s(Even)) by using the conjunctive symbol &. Thus we transform the above rule into the following one,

$$\sim Even \leftrightarrow \sim 0 \ \& \ \sim s(s(Even)).$$

Since we treat only integers which are expressed as terms by using a constant 0 and a functor s, the term ~0 means one of s(0), s(s(0)), s(s(s(0))), ... Therefore if we define T as the special symbol which denotes an arbitrary term composed of 0 and s, the term ~0 in the above rule can be replaced by s(T) as follows,

$$\sim Even \leftrightarrow s(T) \ \& \ \sim s(s(Even)).$$

In the same way, ~s(s(Even)) in the above rule can be replaced by the disjunction 0 | s(0) | s(s(~Even)) because ~s(s(Even)) is equivalent to 0 | s(~s(Even)) and further s(~s(Even)) is equivalent to s(0) | s(s(~Even)).

$$\sim Even \leftrightarrow s(T) \ \& \ (0 \mid s(0) \mid s(s(\sim Even))).$$

Here we assume that the distributive law holds for & and |. Then

$$\sim Even \leftrightarrow s(T) \ \& \ 0 \mid s(T) \ \& \ s(0) \mid s(T) \ \& \ s(s(\sim Even)).$$

There is no number which satisfies s(T)&0, because s(T) means more than zero. Therefore if we introduce the special symbol ⊥ which does not express any integer, s(T) & 0 can be replaced by ⊥. In the same sense, s(T) & s(0) is equivalent to s(0), and s(T) & s(s(~Even)) is equivalent to s(s(~Even)). Hence the above rule is transformed into the following one,

$$\sim Even \leftrightarrow \bot \mid s(0) \mid s(s(\sim Even)).$$

From the meaning of ⊥, the above rule can be further transformed into the following one,

$$\sim Even \leftrightarrow s(0) \mid s(s(\sim Even)).$$

Now we replace every ~Even by the new nonterminal symbol Not_Even.

$$Not\_Even \leftrightarrow s(0) \mid s(s(Not\_Even)).$$

Transforming the above completed rule into two context-free production rules as follows,

Not_Even→s(0),
Not_Even→s(s(Not_Even)),

we find that these rules define odd numbers. Namely we can derive the CFG which defines odd numbers from the CFG which defines even numbers.

The above transformation is rather informal. Therefore our next work is to formalize this negation technique under a certain formal system.

3 Term logic

In order to formally explain the negation technique, a formal system for CFGs is defined here. Because the formal system treats terms and it should have the same features as those of the first-order logic, we call it as the term logic for convenience. In this section, we define the syntax and semantics of the term logic.

Definition Given a finite set F of **functors** (function symbols) and a finite set N of **nonterminal symbols**, **terms** are defined as follows.

(1) A nonterminal symbol X ($\in N$) is a term.
(2) A 0-ary functor c ($\in F$) is a term, called a **constant**.
(3) If $t_1, ..., t_n$ are terms and f ($\in F$) is an n-ary functor, then $f(t_1, ..., t_n)$ is a term. Every t_i ($i = 1, ..., n$) is a subterm of $f(t_1, ..., t_n)$.
(4) A symbol $\bot$ is a term, called the **bottom**.
(5) A symbol $\top$ is a term, called the **top**.
(6) If s and t are terms, then $s \,\&\, t$ is a term, where & is called the **conjunctive symbol**. The terms s and t are subterms of $s \,\&\, t$.
(7) If s and t are terms, then $s \mid t$ is a term, where | is called the **disjunctive symbol**. The terms s and t are subterms of $s \mid t$.
(8) If t is a term, then $\sim t$ is a term, where ~ is called the **negative symbol**. The term t is a subterm of $\sim t$.

Here we assume that the arity of every functor is fixed as its own. A term which has no nonterminal symbol is called a **ground term**. A nonterminal symbol which is *directly* negated by ~, just like $\sim X$, is said to be a **negative** one, and a non-negative nonterminal symbol is a **positive** one.

Note that we implicitly assume the existence of the sets N and F in the following discussions, and does not take any attention if no confusions occur.

In usual sense, for example in term-rewriting systems, terms are composed of variables and functors whereas they are composed of nonterminal symbols and functors here. In our term logic,

variables are needless.

For convenience, we assume the operator precedence of symbols as follows,

$$\sim \ > \ \& \ > \ |.$$

If necessary the parentheses ( ), { }, ... are used to eliminate the ambiguities of term expressions.

Now we define the interpretation for terms.

Definition The **universe**, denoted by $U(F)$, is the set of all the ground terms which contains neither $\top$, $\bot$, &, | nor ~. We assume that $U(F)$ is not empty. ∎

The set $U(F)$ is the same as the Herbrand universe in the first-order logic [ChLe] [Ll].

Definition An **interpretation** i of each nonterminal symbol X ($\in N$) is an assignment $i(X)$ of a subset of $U(F)$ to X. Under this interpretation, the **value** of a term t, denoted by $[t]_i$ or simply $[t]$, is the subset defined as follows,

(1) $[X]_i = i(X)$ if X ($\in N$) is a nonterminal symbol.
(2) $[c]_i = \{c\}$ if c ($\in F$) is a constant.
(3) $[f(t_1, ..., t_n)]_i = \{f(b_1, ..., b_n) \mid b_j \in [t_j]_i \text{ for } j = 1, ..., n\}$.
(4) $[\top]_i = U(F)$.
(5) $[\bot]_i = \emptyset$.
(6) $[s \ \& \ t]_i = [s]_i \cap [t]_i$.
(7) $[s \mid t]_i = [s]_i \cup [t]_i$.
(8) $[\sim t]_i = U(F) - [t]_i$.

If the values of the terms s and t are the same under any interpretation, it is said that s and t are **equivalent**, denoted by $s = t$. ∎

This interpretation is similar to the Herbrand interpretation in the first-order logic [ChLe] [Ll].

It is clear that the above equivalence relation satisfies the reflexive law, the symmetric law and transitive law. From the above definition we can derive some equivalence relations as follows, that is, for arbitrary terms s, t, u, ...

(1) $s \ \& \ t = t \ \& \ s$
(2) $s \mid t = t \mid s$
(3) $(s \ \& \ t) \ \& \ u = s \ \& \ (t \ \& \ u)$
(4) $(s \mid t) \mid u = s \mid (t \mid u)$
(5) $s \ \& \ (t \mid u) = s \ \& \ t \mid s \ \& \ u$
(6) $(t \mid u) \ \& \ s = t \ \& \ s \mid u \ \& \ s$
(7) $s \mid (t \ \& \ u) = (s \mid t) \ \& \ (s \mid u)$
(8) $(t \ \& \ u) \mid s = (t \mid s) \ \& \ (u \mid s)$

(9) $t \,\&\, \mathsf{T} = t$
(10) $t \,\&\, t = t$
(11) $t \,\&\, \bot = \bot$
(12) $t \mid \mathsf{T} = \mathsf{T}$
(13) $t \mid t = t$
(14) $t \mid \bot = t$
(15) $\sim\sim t = t$
(16) $\sim\mathsf{T} = \bot$
(17) $\sim\bot = \mathsf{T}$
(18) $t \,\&\, \sim t = \bot$
(19) $t \mid \sim t = \mathsf{T}$
(20) $\sim(s \,\&\, t) = \sim s \mid \sim t$
(21) $\sim(s \mid t) = \sim s \,\&\, \sim t$

The equivalences (1) and (2) are the commutative laws, (3) and (4) are the associative laws, (5) ... (8) are the distributive laws, (15) is the discharge of double negation, and (20) and (21) are the similar to DeMorgan's laws in the symbolic logic. And for arbitrary *distinct* functors f and g,

(22) $f(..., s \,\&\, t, ...) = f(..., s, ...) \,\&\, f(..., t, ...)$
(23) $f(..., s \mid t, ...) = f(..., s, ...) \mid f(..., t, ...)$
(24) $f(...) \,\&\, g(...) = \bot$
(25) $f(..., \bot, ...) = \bot$
(26) $\sim f(t_1, ..., t_n) = \{g(\mathsf{T}, ..., \mathsf{T}) \mid g \in F - \{f\}\} \mid f(\sim t_1, \mathsf{T}, ..., \mathsf{T}) \mid ... \mid f(\mathsf{T}, ..., \mathsf{T}, \sim t_n)$

The equivalences (22) and (23) are the distributive laws. The equivalence (26) is important to bring negative symbols to inner positions of a term. And further we easily see the following proposition.

Proposition Let $t[u]$ be the term t which has the subterm u, and $t[v]$ be the term in which the subterm u in $t[u]$ is replaced by the term v. Here if $u = v$, then $t[u] = t[v]$. ∎

Example By using above relations, we can prove that

$$\begin{aligned} \sim s(s(\text{Even})) &= 0 \mid s(\sim s(\text{Even})) && \text{by rule (26)} \\ &= 0 \mid s(\,0 \mid s(\sim \text{Even})\,) && \text{by rule (26)} \\ &= 0 \mid s(0) \mid s(s(\sim \text{Even})) && \text{by rule (23)} \end{aligned}$$

and that

$$\begin{aligned} & s(\mathsf{T}) \,\&\, (\,0 \mid s(0) \mid s(s(\sim \text{Even}))\,) \\ &= s(\mathsf{T}) \,\&\, 0 \mid s(\mathsf{T}) \,\&\, s(0) \mid s(\mathsf{T}) \,\&\, s(s(\sim \text{Even})) && \text{by rule (5)} \\ &= \bot \mid s(\mathsf{T}) \,\&\, s(0) \mid s(\mathsf{T}) \,\&\, s(s(\sim \text{Even})) && \text{by rule (24)} \\ &= \bot \mid s(\,\mathsf{T} \,\&\, 0\,) \mid s(\,\mathsf{T} \,\&\, s(\sim \text{Even})\,) && \text{by rule (22)} \\ &= \bot \mid s(0) \mid s(s(\sim \text{Even})) && \text{by rule (9)} \\ &= s(0) \mid s(s(\sim \text{Even})) && \text{by rule (14)} \end{aligned}$$

We have shown these equivalences informally in the previous section. ∎

Proposition For every ground term which contains negative symbols, there is an equivalent ground term which has no negative symbols. ∎

Proposition For every term which contains negative symbols, there is an equivalent term in which negative symbols appear only in negative nonterminal symbols. ∎

These proposition can be easily proved by using the rule(26).

Using the above equivalence relations, we can transform every term into the equivalent one of the simpler form. Because the above equivalence relations are similar to those in the symbolic logic, we can consider terms of special forms in the same way as in the symbolic logic.

Definition The term t is said to be in a **conjunctive normal form** if and only if t has the form $t_1 \& \ldots \& t_n$ $(n \geq 1)$, where each t_j $(1 \leq j \leq n)$ is a disjunction of one or more terms. The term t is said to be in a **disjunctive normal form** if and only if t has the form $t_1 \mid \ldots \mid t_n$, where each t_j is a conjunction of one or more terms. ∎

Proposition Every term can be transformed into the equivalent term of the conjunctive normal form and of the disjunctive normal form. ∎

The transformation algorithms are the same as the well-known algorithms in the symbolic logic [ChLe]. First we move negative symbols from outer positions to inner positions by the rule (26) and De Morgan's laws (20) and (21), then we arrange the positions of conjunctive and disjunctive symbols by the distribute laws (5) ~ (8).

Now we define the **model semantics** of the term logic.

Definition If the value of a term t is equal to $U(F)$ under the interpretation $\mathbb{M}$, $\mathbb{M}$ is said to be a **model of** t. If $\mathbb{M}$ is a model of every elements in a set P of terms, $\mathbb{M}$ is said to be a model of P. A term t (or a set P of terms) is said to be **satisfiable** if and only if t (or P) has at least one model. Otherwise, it is said to be **unsatisfiable** or to contain **contradiction**. ∎

Example Let X and Y be nonterminal symbols in N. The term X has only one model $\mathbb{M}$ such as $[X]_{\mathbb{M}} = U(F)$ for any F. The term $X \mid \sim Y$ has many models $\mathbb{M}$ such as $[X]_{\mathbb{M}} \supseteq [Y]_{\mathbb{M}}$. ∎

Another semantics, the operational one, is the **proof procedure**. The term logic has the proof procedure, especially the refutation procedure. That is, for every unsatisfiable set P of terms it can be automatically proved that P is unsatisfiable. This is based on the resolution principle of our term logic similar to that of the first-order logic. Briefly speaking, by applying this principle to the pair of terms $s \mid t$ and $\sim s \mid u$ in P, the term $t \mid u$ is derived. This is the similar to Robinson's resolution principle [ChLe]. If the value of a derived term is not equal to $U(F)$ under any

interpretation, we say that the *contradiction* has been found. Namely P is unsatisfiable. We can prove the *completeness* of our resolution principle in the term logic. However we omit the precise discussion because of the space limitation (we will show it in the separate paper). In the next section , a special form of our resolution principle is presented as the derivation procedure of CFGs.

4 General context-free grammars

In this section, CFGs are regarded as a finite set of a special subclass of terms in the term logic. We define the syntax and semantics of CFGs from the view point of the term logic.

From the study of the relationship between logic programs and CCFGs [YaNa 2], we have already known that the definite clause ∀X.p(X):-q(X) can be transformed into the production rule P→Q preserving the meaning. In the symbolic logic, the implication ∀X.p(X):-q(X) is equivalent to ∀X.p(X) ∨ ~q(X). These two facts lead us to the following definition.

Definition For arbitrary terms s and t, the term of the form

$$s \mid\sim t$$

is abbreviated as

$$s \rightarrow t,$$

and called a **production rule**, especially a **context-free production rule** if s is a nonterminal symbol. ∎

We assume that the operator precedence of → is weaker than &, | and ~.

Proposition It holds that $[s]_M \supseteq [t]_M$ if and only if M is a model of $s \rightarrow t$. ∎

If there is the production rule $s \rightarrow t$ in G, it holds that

$$\{ x \mid s \Rightarrow^*_G x \} \supseteq \{ x \mid t \Rightarrow^*_G x \}.$$

This inclusion relation is similar to $[s]_M \supseteq [t]_M$.

Definition A **general Context-Free Grammar (general CFG)**: G is defined by the quadruple (N, F, S, P), where N is a finite set of nonterminal symbols, F a finite set of functors, S a start symbol ($\in N$) and P a finite set of context-free production rules. ∎

If the right-hand side of every production rule in P has neither any negative symbol, conjunctive symbol nor disjunctive symbol, the grammar is in (term-based) CFGs in usual sense. If the right-hand side has no negative symbol but has several conjunctive and disjunctive symbols, the grammar is in a subclass of CCFGs [?], and we can define the *least model* and the *least fixpoint* of

the general CFG in the similar way to those of a definite clause program. Since our main interest in this paper is the property of negation, we do not discuss such topics here any more.

Example The followings are general CFGs.

$\mathbf{G_1}$ = ({Even}, {0, s}, Even, P_{G1}),
P_{G1} = { Even→0,
Even→s(~Even) }.

$\mathbf{G_2}$ = ({Mul6, Mul2, Mul3}, {0, s}, Mul6, P_{G2}),
P_{G2} = { Mul6→Mul2&Mul3,
Mul2→0,
Mul2→s(s(Mul2)),
Mul3→0,
Mul3→s(s(s(Mul3))) }.

The precise meanings of the above grammars are given below. Roughly speaking, $\mathbf{G_1}$ defines even numbers and $\mathbf{G_2}$ defines multiple numbers of six. ∎

Definition For arbitrary terms s and t, the term of the form

$$(s \rightarrow t)\&(t \rightarrow s)$$

is abbreviated as

$$s \leftrightarrow t,$$

and call an **equation**, especially a **context-free equation** if the left-hand side s is a nonterminal symbol. ∎

Note that the equation $s \leftrightarrow t$ is not the same as the equivalence relation $s = t$. The former is a term whereas the latter expresses a relation.

Definition Let P be a set of production rules. For all the production rules in P which have the same left-hand nonterminal symbol X as follows,

$$X \rightarrow \alpha_1, X \rightarrow \alpha_2, \ldots, X \rightarrow \alpha_n, \quad (n \geq 1)$$

the following is said to be the **completed definition** of them,

$$X \leftrightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n.$$

For a general CFG: G = (N, F, S, P), let comp(P) be a collection of completed definitions of P for all nonterminal symbols. If a nonterminal symbol X ($\in N$) does not appear in the left-hand

side of any completed definitions, add X↔⊥ to comp(P). The quadruple (N, F, S, comp(P)) is called the **completion** of G, denoted by comp(G). If an interpretation is a model for comp(P), it is also said to be a model for comp(G). ∎

Now we define the declarative and operational semantics of general CFGs. First we discuss the declarative one, and then the operational one. In the end of this section we discuss the identicalness of these two kinds of semantics.

Definition For the set P of terms and the term t over N and F, t is said to be a **logical consequence** of P if and only if $\mathcal{M}$ is a model for P implies that $\mathcal{M}$ is a model for t. ∎

Lemma If the set P is unsatisfiable, every term t is a logical consequence of P. ∎

This is clear from the definition.

Proposition Let P be the set of terms, t be a term, and a be a ground term in $U(F)$. The production rule $t \to a$ is a logical consequence of P if and only if $P \cup \{\sim t \to a\}$ is unsatisfiable.

Proof If $t \to a$ is a logical consequence of P, it holds that $a \in [t]_{\mathcal{M}}$ for every model $\mathcal{M}$ for P. Hence

$$a \notin U(F) - [t]_{\mathcal{M}} = [\sim t]_{\mathcal{M}},$$

and then

$$a \notin [\sim t]_{\mathcal{M}} \cup (U(F) - \{a\}) = [\sim t \mid \sim a]_{\mathcal{M}} = [\sim t \to a]_{\mathcal{M}} \neq U(F),$$

for every model $\mathcal{M}$ for P. Therefor $P \cup \{\sim t \to a\}$ is unsatisfiable.

If $P \cup \{\sim t \to a\}$ is unsatisfiable, it holds that P is unsatisfiable or that $[\sim t \to a]_{\mathcal{M}} \neq U(F)$ for every model $\mathcal{M}$ for P. In the former case, $\sim t \to a$ is a logical consequence of P from the lemma. In the latter case,it holds that

$$[\sim t]_{\mathcal{M}} \cup (U(F) - \{a\}) \neq U(F).$$

Therefore it holds that

$$a \notin [\sim t]_{\mathcal{M}},$$

and then

$$a \in [t]_{\mathcal{M}}.$$

for every model $\mathcal{M}$ for P. This means that every model $\mathcal{M}$ for P is a model for $t \to a$. ∎

Definition The **declarative semantics** of general CFG: G = (N, F, S, P) is defined as a subset of $U(F)$ such as $S \to a$ is a logical consequence of comp(G) for every a in the subset. ∎

Example(continued) The completion of $\mathbf{G_1}$ is as follows.

$$\text{comp}(\mathbf{G_1}) = (\{\text{Even}\}, \{0, s\}, \text{Even}, \text{comp}(P_{G1})),$$
$$\text{comp}(P_{G1}) = \{\ \text{Even} \leftrightarrow 0 \mid s(\sim\text{Even})\ \}.$$

The completed definition comp($\mathbf{G_1}$) has a unique model M_1 such as [Even]$_{M1}$is the set of all the even numbers. Hence the term Even$\rightarrow s^{2n}(0)$ ($n\geq 0$) is a logical consequence and the set comp(P_{G1})$\cup${~Even$\rightarrow s^{2n}(0)$} is unsatisfiable. We say that $\mathbf{G_1}$ defines even numbers.

The completion of $\mathbf{G_2}$ is as follows,

$$\begin{aligned}\text{comp}(\mathbf{G_2}) &= (\{\text{Mul6, Mul2, Mul3}\}, \{0, \text{s}\}, \text{Mul6}, \text{comp}(P_{G2})),\\ \text{comp}(P_{G2}) &= \{\ \text{Mul6}\leftrightarrow\text{Mul2\&Mul3},\\ &\qquad\qquad \text{Mul2}\leftrightarrow 0 \mid \text{s(s(Mul2))},\\ &\qquad\qquad \text{Mul3}\leftrightarrow 0 \mid \text{s(s(s(Mul3)))}\ \}.\end{aligned}$$

This also has a unique model M_2 such as [Mul2]$_{M2}$ is the set of all the even numbers, [Mul3]$_{M2}$ is the set of all the multiple numbers of three, and [Mul6]$_{M2}$ is the set of all the multiple numbers of six. This grammar define the multiple numbers of six by using the even numbers and the multiple numbers of three. Of course, Mul6$\rightarrow s^{6n}(0)$ is a logical consequence and comp(P_{G2})$\cup${~Mul6$\rightarrow s^{6n}(0)$} is unsatisfiable. ∎

Example The following grammar has only a tautology.

$$\text{comp}(\mathbf{G_3}) = (\{\text{X}\}, F, \text{X}, \{\ \text{X}\leftrightarrow\text{X}\ \}).$$

Because the set { X$\leftrightarrow$X, ~X$\rightarrow a$ } is satisfiable for every ground term a in $U(F)$ under the interpretation such as $a\in$[~X], the semantics of $\mathbf{G_3}$ is defined by the empty set $\emptyset$. ∎

Example The following is an interesting grammar whose rule is unsatisfiable.

$$\text{comp}(\mathbf{G_4}) = (\{\text{X}\}, F, \text{X}, \{\ \text{X}\leftrightarrow\text{~X}\ \}).$$

From the lemma, the term X$\rightarrow a$ is a logical consequence of X$\leftrightarrow$~X for every a in $U(F)$. ∎

The above proposition suggests the existence of the refutation procedure in the term logic just like the resolution principles in the symbolic logic. Now we define a subclass of such a procedure as an operational semantics of general CFG. We see that it is an extension of the derivation procedure of CFGs.

Definition For general CFG: comp(G) = (N, F, S, comp(P)), the **sentential forms** are defined as follows,

(1) The start symbol S is a sentential form.

(2) If α is a sentential form and the derivation relation $\alpha\Rightarrow_{\text{comp(G)}}\alpha'$ holds, then α' is a sentential form.

Here the derivation relation is defined as follows. If the sentential form is a ground term, it is called a **sentence**. The following sequence of sentential forms

$$S \Rightarrow_{comp(G)} \alpha_1 \Rightarrow_{comp(G)} \alpha_2 \Rightarrow_{comp(G)} \cdots \Rightarrow_{comp(G)} \alpha_n \quad (n \geq 1)$$

is called a **derivation sequence.**

Definition For general CFG: comp(G) = (N, F, S, comp(P)), let $\alpha(X)$ be the term which has a nonterminal symbol X as a subterm. If comp(P) has a context-free equation $X \leftrightarrow \beta$, we can substitute β for X in $\alpha(X)$ and obtain $\alpha(\beta)$. We express the relation between $\alpha(X)$ and $\alpha(\beta)$ as

$$\alpha(X) \Leftrightarrow_{comp(G)} \alpha(\beta).$$

Next if the disjunctive normal form of $\alpha(\beta)$ is $\alpha_1 \mid \ldots \mid \alpha_n$ ($n \geq 1$), we say that $\alpha(X)$ derives α_j ($n \geq j \geq 1$) and express this derivation relation as

$$\alpha(X) \Rightarrow_{comp(G)} \alpha_j .$$

The reflective and transitive closure of $\Leftrightarrow_{comp(G)}$ and $\Rightarrow_{comp(G)}$ is expressed as $\Leftrightarrow_{comp(G)}^*$ and $\Rightarrow_{comp(G)}^*$, respectively. If no confusions occur, we write $\Leftrightarrow_{comp(G)}$ and $\Rightarrow_{comp(G)}$ as $\Leftrightarrow$ and $\Rightarrow$.

Definition The union of the values of all the sentences derived by general CFG: comp(G) is called a **General Context-Free Language (GCFL)**, denoted by L(comp(G)). The **operational semantics** of G is defined by L(comp(G)).

Example(continued) We consider the derivation procedure for $\mathbf{G_1}$. First of all, we see that

$$\text{Even} \Leftrightarrow 0 \mid s(\sim\text{Even})$$

for the start symbol Even. Hence

$$\text{Even} \Rightarrow 0,$$

and

$$\text{Even} \Rightarrow s(\sim\text{Even}).$$

The constant 0 is a sentence and s(~Even) is a sentential form of $\mathbf{G_1}$. Next it holds that for s(~Even),

$$s(\sim\text{Even}) \Leftrightarrow s(s(\text{Even}),$$

because

$$\begin{aligned} s(\sim(\,0 \mid s(\sim\text{Even})\,)) &= s(\sim 0 \;\&\; \sim s(\sim\text{Even})) \\ &= s(s(\text{T}) \;\&\; (\,0 \mid s(\text{Even})\,)) \\ &= s(s(\text{Even})). \end{aligned}$$

And further for s(s(Even)),

s(s(Even))⇒s(s(0)),

and that

s(s(Even))⇒s(s(s(~Even)))

because

s(s(Even)⇔s(s(0 | s(~Even))) = s(s(0)) | s(s(s(~Even))).

In this way, the language $L(\mathrm{comp}(\mathbf{G_1}))$ is the set of all the even numbers. ∎

In the end we describe about the identicalness of the declarative and operational semantics of general CFG.

Theorem For every consistent general CFG: G = (N, F, S, P), let a be a ground term in $U(F)$. Here it holds that $S\rightarrow a$ is a logical consequence of comp(P) if L(comp(G)) contains a (*soundness*), and that L(comp(G)) contains a if $S\rightarrow a$ is a logical consequence of comp(P) (*completeness*). ∎

The proof of soundness is clear because $S\rightarrow t$ is a logical consequence if $S\Rightarrow^* t$ for any t. The proof of completeness is omitted because this is difficult without describing the completeness of the refutation procedure in the term logic, which has been discussed in the end of the previous section.

5 Negation technique

Through the discussions in the previous sections the basis has been established to formalize the negation technique described in section two.

First we introduce two kinds of transformations of general CFGs. One is to eliminate negative symbols from a general CFG, and the other is to purge the completed definitions which can be never applied to any sentential forms from a general CFG. The negation technique is the composition of these two transformations with replacing a start symbol S by the negative start symbol ~S. In the following discussions, without the loss of generality we assume that all the negation symbols appear only in *negative nonterminal symbols* in a term.

Transformation rule For a general CFG: comp(G) = (N, F, S, comp(P)), if an equation in comp(P) contains a negative nonterminal symbol ~X and comp(P) contains the equation $X\leftrightarrow\alpha$, add $\sim X\leftrightarrow\alpha'$ to comp(P) and replace every ~X in comp(P) by the new nonterminal symbol *Not_X*, where $\sim\alpha$ is equivalent to α' in which all the negative symbols appear only in negative nonterminal symbols. The obtained set of equations is denoted by comp(P)/~X. The quadruple ($N\cup\{Not\_X\}$, F, S, comp(P)/~X) is the transformed general CFG, denoted by comp(G)/~X. If comp(G) has several negative nonterminal symbols ~X, ~Y, ..., ~Z, obtain comp(G)/~X/~Y/.../~Z until there appears no negative nonterminal symbol in the transformed grammar. ∎

Proposition For the above grammar, it holds that

$$L(\mathrm{comp(G)}) = L(\mathrm{comp(G)}/{\sim}X).$$

Proof There is the following derivation sequence

$$S \Rightarrow_{\mathrm{comp(G)}} \alpha_1 \Rightarrow_{\mathrm{comp(G)}} \alpha_2 \Rightarrow_{\mathrm{comp(G)}} \cdots \Rightarrow_{\mathrm{comp(G)}} \alpha_n, \quad (n \geq 1)$$

in which no $\sim X$ appears if and only if there is the same derivation sequence as follows

$$S \Rightarrow_{\mathrm{comp(G)}/{\sim}X} \alpha_1 \Rightarrow_{\mathrm{comp(G)}/{\sim}X} \alpha_2 \Rightarrow_{\mathrm{comp(G)}/{\sim}X} \cdots \Rightarrow_{\mathrm{comp(G)}/{\sim}X} \alpha_n,$$

in which no *Not_X* appears. The sentential form α_n can derive $\alpha_{n+1}(\sim X, \sim X, ..., \sim X)$ which contains at least one $\sim X$ by comp(G) if and only if α_n can derive $\alpha_{n+1}(Not\_X, Not\_X, ..., Not\_X)$ which contains at least one *Not_X* by comp(G)/$\sim X$ because comp(P) contains the equation $Y \leftrightarrow \beta(\sim X, \sim X, ..., \sim X)$ if and only if comp(P)/$\sim X$ contains $Y \leftrightarrow \beta(Not\_X, Not\_X, ..., Not\_X)$ for a certain nonterminal symbol Y. The sentential form $\alpha_{n+1}(\sim X, \sim X, ..., \sim X)$ can derive $\alpha_{n+2}(\sim X, \gamma(\sim X, ..., \sim X), ..., \sim X)$ by $X \leftrightarrow \gamma$ in comp(G), where $\sim\gamma$ is equivalent to $\gamma(\sim X, ..., \sim X)$, if and only if $\alpha_{n+1}(Not\_X, Not\_X, ..., Not\_X)$ can derive $\alpha_{n+2}(Not\_X, \gamma(Not\_X, ..., Not\_X), ..., Not\_X)$ by $Not\_X \leftrightarrow \gamma(Not\_X, ..., Not\_X)$ in comp(G)/$\sim X$.

In this way, the sentential form $\alpha_k(\sim X, ..., \sim X)$ ($k \geq 1$) can be derived by comp(G) if and only if the sentential form $\alpha_k(Not\_X, ..., Not\_X)$ can be derived by comp(G)/$\sim X$. Here if $\alpha_k(\sim X, ..., \sim X)$ has no $\sim X$, it is a sentence s whose value is unique under any interpretation, and $\alpha_k(Not\_X, Not\_X, ..., Not\_X)$ is also the same sentence s if so. Therefore L(comp(G)) is equal to L(comp(G)/$\sim X$). ∎

By applying this transformation to a general CFG, the equivalent general CFG which contains no negative symbols can be obtained.

Example(continued) For general CFG: comp($\mathbf{G_1}$) has the negative nonterminal symbol ~Even in the equation Even↔0 | s(~Even). Then

$$\mathrm{comp(P_1)}/{\sim}\mathrm{Even} = \{\ \mathrm{Even} \leftrightarrow 0 \mid s(\mathrm{Not\_Even}),$$
$$\mathrm{Not\_Even} \leftrightarrow s(\mathrm{Even}) \quad \},$$

because

$$\sim(0 \mid s(\sim\mathrm{Even})) = \sim 0 \ \&\ \sim s(\sim\mathrm{Even}) = s(\mathsf{T}) \ \&\ (\ 0 \mid s(\mathrm{Even})\ ) = s(\mathrm{Even}).$$ ∎

Transformation rule For a general CFG: comp(G) = (N, F, S, comp(P)), if a nonterminal symbol X can never be derived from the start symbol S, purge the equation $X \leftrightarrow \alpha$ from comp(P), and purge X from N. The purged grammar (N–{X}, F, S, comp(P)–{$X \leftrightarrow \alpha$}) is denoted by comp(G)/X. If comp(G) has such nonterminal symbols X, Y, ..., Z, obtain comp(G)/X/Y/.../Z. ∎

Proposition For the above grammar, it holds that

$$L(\mathrm{comp(G)}) = L(\mathrm{comp(G)}/X).$$

The proof is trivial.

Now we define negation technique.

Transformation rule Suppose that a general CFG: comp(G) = (N, F, S, comp(P)) has the equation $S \leftrightarrow \alpha$. Transform the general CFG: ($N \cup \{Not\_S\}$, F, Not_S, comp(P)$\cup\{\sim S \leftrightarrow \alpha'\}$)/$\sim S$ by applying the above two transformation rules, where $\sim\alpha$ is equivalent to α' in which all the negative symbol appears only in negative nonterminal symbols. The obtained general CFG is denoted by comp(Not_G).

Definition For a general CFG: comp(G) and comp(H), if it holds that

$$U(F)\text{–}L(\mathrm{comp(G)}) \supseteq L(\mathrm{comp(H)}),$$

comp(H) is said to be **dual** to comp(G). If it further holds that

$$U(F)\text{–}L(\mathrm{comp(G)}) = L(\mathrm{comp(H)}),$$

comp(H) is said to be **complementary** to comp(G) [Sat].

Proposition For the above transformation, comp(Not_G) is dual to comp(G).

Proof It is sufficient to prove that

$$U(F)\text{–}L(\mathrm{comp(G)}) \supseteq L((N \cup \{Not\_S\}, F, Not\_S, \mathrm{comp}(P) \cup \{\sim S \leftrightarrow \alpha'\})/\sim S),$$

because

$$L((N \cup \{Not\_S\}, F, Not\_S, \mathrm{comp}(P) \cup \{\sim S \leftrightarrow \alpha'\})/\sim S) = L(\mathrm{comp(Not\_G)}).$$

In the same way as the proof of ..., it holds that $Not\_S \Rightarrow^* t$ by applying the equations in (comp(P)$\cup\{\sim S \leftrightarrow \alpha'\}$)/$\sim S$ if and only if $\sim S \Rightarrow^* t$ by applying the equations in comp(P) for any ground term t. Because $\sim S \Rightarrow^* t$ implies that $\sim S \rightarrow a$ is a logical consequence of comp(P) for any $a \in [t]$, $S \rightarrow a$ is not a logical consequence of comp(P) for any $a \in [t]$ if $Not\_S \Rightarrow^* t$. Namely $a \notin L(\mathrm{comp(G)})$ for any $a \in [t]$ if $Not\_S \Rightarrow^* t$, and then $a \in U(F)\text{–}L(\mathrm{comp(G)})$ for any $a \in [t]$ if $Not\_S \Rightarrow^* t$. Therefore the proposition holds.

In general, comp(Not_G) is not a complementary grammar as we can show the following counter-exmaple.

Example(continued) The following general CFG: comp($\mathbf{G_3}$) has only one equation. This is negated as ~X↔~X. Therefore comp($\mathbf{Not\_G_3}$) is as follows,

$$\mathrm{comp}(\mathbf{Not\_G_3}) = (\{\mathrm{Not\_X}\}, F, \mathrm{Not\_X}, \{\ \mathrm{Not\_X} \leftrightarrow \mathrm{Not\_X}\ \}).$$

It holds that

$$L(\mathrm{comp}(\mathbf{G_3})) = \emptyset,$$

and that

$$L(\mathrm{comp}(\mathbf{Not\_G_3})) = \emptyset.$$

Therefore we see that

$$U(F)\text{–}L(\mathrm{comp}(\mathbf{G_3})) = U(F)\text{–}\emptyset \neq \emptyset = L(\mathrm{comp}(\mathbf{Not\_G_3})),$$

for any non-empty $U(F)$.

6 Discussions

In this paper, we have discussed about the questions what is the concept of negation and how can we establish the negation technique for context-free grammars. We have replied to these questions by establishing the formal system for term, called the term logic,and by giving a new interpretation of context-free production rules as a special form of terms. We see that the semantics of the term logic has the similar mathematical structure to that of the first-order logic, though their syntaxes are quite different to each other.

There are many advanced subjects of the term logic, general CFGs and the negation technique for general CFGs which we do not discuss here. Some of them, which we have already done or we will try soon, are as follows:

(1) to define the resolution principle in the term logic, and to prove its completeness.

(2) to prove the completeness of the derivation procedure for general CFGs.

(3) to improve the negation technique for general CFGs in order to obtain complementary grammars.

(4) to introduce negation into CCFGs, and to reform it from the view point of practical programming methodology.

references

[AhUl] Aho, A. V. and Ullman, J. D. : *The theory of Parsing, Translation, and Compiling, Vol 1: Parsing*, Prentice-Hall (1972).

[ChLe] Chan, C. and Lee, R. C. : *Symbolic logic and Mechanical Theorem Proving*, Academic Press (1973).

[Cla] Clark, K. L. : Negation as Failure, *Logic and Database*, Plenum Press (1978).

[ClMe] Clocksin, W. F. and Mellish, C. S. : *Programming in Prolog*, Springer-Verlag (1981).

[Ll] Lloyd, J. W. : *Foundations of Logic Programming*, Springer-Verlag (1984).

[Na] Naish, L. : Negation and Control in PROLOG, Ph.D. dissertation, University of

Melbourne (1985).

[Sal] Salomaa, A : *Formal Languages*, Academic Press (1973).

[Sat] Sato, T. : *Program Transformation* (Eds. Furukawa, K. and Mizoguchi, H.), Chapter 6, Kyooritsu(1987) (*in Japanese*).

[SaTa] Sato, T. and Tamaki, H. : Transformational Logic Program Synthesis, *Proc. Int. Conf. Fifth Generation Computer Systems* (1984), pp.195-201.

[Sch] Schultz, J. W. : The Use of First-Order Predicate Calculas as a Logic Programming System, M.Sc. dissertation, University of Melbourne (1984).

[YaNa 1] Yamashita, Y. and Nakata, I. : Programming in Coupled Context-Free Grammars, Technical Report ISE-TR-88-70, university of Tsukuba (1988).

[YaNa 2] Yamashita, Y. and Nakata, I. : On the Relation between CCFG Programs and Logic Programs, Technical Report ISE-TR-88-71, university of Tsukuba (1988).

[YaNa 3] Yamashita, Y. and Nakata, I. : Programming in Gramp: a Programming Language based on CCFG, Technical Report ISE-TR-88-73, university of Tsukuba (1988).