Conditional Derivation Grammar and Its Parsing Algorithm

ANONYMOUS AUTHOR(S)

Context-free grammars(CFGs) have been used to define the syntaxes of programming languages for decades. However, CFGs are not expressive enough to define the complete syntaxes of practical programming languages. Language developers are forced to define the syntaxes in two separate sets of lexical and hierarchical grammars. The parser implementations are also often designed to have two-layer structures. The separation of lexical and hierarchical grammars causes difficulties in defining complicated syntax features, such as nested template expression of C++ or string interpolation. This paper proposes Conditional Derivation Grammar(CDG), which has a similar format with CFG but offers more features. CDG provides enough expressive power to define the complete syntax of a practical programming language in a single definition. CDG offers new symbols such as intersection, exclusion, followed-by, not-followed-by, and longest match. This paper also proposes a parsing algorithm for CDGs. The algorithm correctly recognizes the language of the given grammar and always terminates. The time and space complexity of the algorithm varies by the characteristics of the given grammar, but further research is needed. Nonetheless, the proposed algorithm is believed to have practical time complexity for the syntaxes of many programming languages.

CCS Concepts: •**Software and its engineering** → **General programming languages;** •**Social and professional topics** → *History of programming languages;*

Additional Key Words and Phrases: keyword1, keyword2, keyword3

ACM Reference format:
Anonymous Author(s). 2017. Conditional Derivation Grammar and Its Parsing Algorithm. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 25 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Context-Free Grammar(CFG) is a useful tool to describe the syntaxes of programming languages. It is intuitive to read and easy to write. Therefore, most programming languages have defined their syntaxes in CFGs for decades.

Unfortunately, the expressive power of CFG is too limited to define the complete syntax of a practical programming language. For example, CFG cannot imitates the behavior of typical lexical analyzers. Consider the following CFG:

$$S \longrightarrow S\ Token \mid \epsilon$$
$$Token \longrightarrow Name \mid Keyword$$
$$Name \longrightarrow Name\ Char \mid Char$$
$$Char \longrightarrow \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \ \ldots\ \mid \texttt{z}$$
$$Keyword \longrightarrow \texttt{i}\ \texttt{f}$$

This grammar looks like a simple lexical grammar. However, the grammar does not work as the lexical analyzers would do. Lexical analyzers often assume longest match policy and priorities among tokens, but CFG does not have such functionalities. For example, the string "`if`" is ambiguous, since it can be *Name* and *Keyword* at the same time. Also, the string can be two tokens of "`i`" and

A note.
2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

"f", both *Name*s. Due to the lack of such functionalities, it is impossible to define the complete syntax of a practical programming language in a single CFG.

It has led the parsers of most programming languages to be constructed with separated two layers of lexical and hierarchical grammar analyzers. Lexical analyzer translates a string into a sequence of tokens, and hierarchical grammar analyzer transforms a stream of tokens to parse tree or forest. Lexical grammars are commonly defined in Regular Expressions(REs) and hierarchical grammars are usually defined in CFGs. Also, most lexical analyzers assume longest match and support priorities among tokens.

The separation of lexical and hierarchical grammars not only increases complexity of parser, but also causes difficulties in modeling complicated language structures of modern programming languages. One example is the nested template type expressions of C++. For example, consider a C++ template type expression `vector<vector<int>>`. The code looks perfectly fine, but old C++ parsers consider the code is invalid. In C++03, two adjacent closing brackets >> is always a right shift or stream extraction operator, even though it is at where such operators cannot be placed. To recognize >> is whether an operator or two closing brackets, the lexical analyzers must understand the context. the parsers of C++11(ISO/IEC 14882: 2011(E) 2011) have fixed the problem, but it requires complicated techniques.

Another example is string interpolation feature, which many modern programming languages have. In Ruby(ISO/IEC 30170: 2012(E) 2012), for example, a string `"1+2=#{1+2}"` is evaluated to `"1+2=3"`. The expression following # in the string is replaced with the result of the evaluated expression. Unfortunately, it is extremely difficult for a scanner to just find out where a string ends. For example, if `"#{a+","+b}"` is given, typical lexical analyzers have no ability to find whether the second double quote(") is closing the string or opening another string inside the embedded expression in the string.

Context-Sensitive Grammar(CSG) can be an alternative to define the syntaxes of programming languages. However, they are not very familiar to programming language developers, and no suitable parsing algorithm for the use cases of programming languages is known.

We propose Conditional Derivation Grammar(CDG). CDG resembles the look of CFG, but it offers new features such as intersection, exclusion, followed-by, not-followed-by, and longest match. It is expected that the syntaxes of most practical programming languages can be defined in a single CDG. Section 2 describes conditional derivation grammars in detail.

This paper also proposes parsing algorithm for conditional derivation grammars. The proposed parser can handle any CDG unless it is illegal. We have proved that the algorithm correctly parses and always terminates for any non-illegal CDG. The time and space complexity of the algorithm varies by the characteristics of the given grammar, and further research is needed. It is believed that the parser has practical time complexity on syntaxes of many programming languages. Section 3 discusses the parsing algorithm.

2 CONDITIONAL DERIVATION GRAMMAR

This section describes Conditional Derivation Grammar(CDG). Section 2.1 describes CDG informally, and Section 2.2 defines CDG formally. Section 2.3 describes illegal CDGs, and Section 2.4 explores several properties of CDG and compares CDG to other formal grammars.

2.1 Informal Descriptions

A context-free grammar is composed of 4 elements: a set of nonterminals, a set of terminals, a set of production rules from a nonterminal to a sequence of terminals and nonterminals, and a start symbol. To get a string of the grammar, we start from the start symbol and repeatedly replace a

nonterminal with one of the productions of the nonterminal. When we get a string of terminals by repeating the nonterminal substitution, it is a string of the language of the CFG.

A CDG also has terminal symbols, nonterminal symbols, production rules from a nonterminal to a sequence of symbols, and a start nonterminal. Besides, CDG has *conditional nonterminals*, and the right-hand-sides of production rules can have the conditional nonterminals as well as nonterminals and terminals. A conditional nonterminal produces a sequence of symbols as a normal nonterminal does, but it adds new conditions to determine whether the derived string is in the language or not. That is, CDG may reject a derived string according to the conditions added by conditional nonterminals during the derivation, while CFG always accepts all strings derived from the start symbol.

Figure 1 shows an example arithmetic expression grammar defined in CDG. The grammar defines mathematical expressions with numbers, variables, and two operators + and *. The priority between operators is implemented.

```
expression = term | expression '+' term
term = factor | term '*' factor
factor = number | variable | '(' expression ')'
number = '0' | {1-9} {0-9}*
variable = {A-Za-z}+
```

Fig. 1. An expression grammar in CDG

CDG has a set of production rules from a nonterminal to a sequence of symbols. As an equal sign(=) being a delimiter, a left-hand-side nonterminal comes before, and a right-hand-side symbol sequence comes after. A right-hand-side is written as a consecutive sequence of symbols with whitespace characters in between. If there are multiple rules that have the same left-hand-side, the right-hand-sides of the rules are written composed with vertical bars(|). The first rule defines the production rules of the start symbol.

A string of English alphabets and numbers represents a nonterminal. If a nonterminal has special characters in its name, you can write the name in a pair of backticks(`). Nonterminals can be on both sides of the rules, while others only can be on the right-hand-sides.

A terminal character is surrounded by a pair of single quotes('). Curly brackets indicate a set of characters, that includes single characters and ranges. A string literal is enclosed by a pair of double quotes("). In definition of terminal characters or string literals, special characters such as newline character are escaped by C-like escape sequences. A dot(.) indicates any single character.

CDG also has optional and repeated symbols. An asterisk(*) is postfixed to repeat a symbol zero or more times, a plus sign(+) indicates the preceding symbol repeating one or more times, and ? makes a symbol optional.

A pair of square brackets([]) indicates a nested sequence, which is a sequence inside another sequence. It is useful when part of right-hand-side is repeated or when a conditional nonterminal has a sequence. A pair of round brackets(()) containing one or more vertical bars indicates an unnamed nonterminal. As its name suggests, an unnamed nonterminal works like a nonterminal with production rules in it.

Syntactic sugars are set of characters symbols, any character symbol, repeated and optional symbols, nested sequences, and unnamed nonterminals. We substitute them by the rules as follows:

- A set of characters $\{t\}$ is considered as a nonterminal that produces the terminals of t.
- An any character symbol . is considered as a nonterminal that produces all of the terminals of the grammar.

- A string is considered as a nested sequence of terminals.
- $A*$ is considered as a nonterminal that produces ϵ or $A*$ A.
- $A+$ is considered as a nonterminal that produces A or $A+$ A.
- $A?$ is considered as a nonterminal that produces ϵ or A.
- $[A_1\ A_2\ \ldots\ A_n]$ is considered as a nonterminal that produces $A_1\ A_2\ \ldots\ A_n$.
- $(A_1|A_2|\ldots|A_n)$ is considered as a nonterminal that produces A_1, A_2, ..., or A_n.

The symbols described so far are similar to the ones of CFGs. A CDG can have conditional nonterminals on the right-hand-sides of its rules. There are 5 types of conditional nonterminals: intersection, exclusion, followed-by, not-followed-by, and longest.

Figure 2 shows an example grammar that imitates the common behaviors of lexical analyzers. This grammar has longest, intersection, and exclusion symbols.

```
S = token*
token = keyword | operator | identifier
keyword = ("if" | "else") & name
operator = <op>
op = '+' | "++"
identifier = name - keyword
name = <[{A-Za-z} {0-9A-Za-z}*]>
```

Fig. 2. A lexical grammar in CDG

A symbol surrounded by a pair of angle brackets(<>) only matches to the longest match. For a symbol A, <A> produces A with a condition that there is no longer match of A. In Figure 2, `name` and `operator` are defined using longest symbol. `name` is defined as the longest match of a nested sequence of an English alphabet followed by numbers or alphabets. `operator` is defined as the longest one among `+` and `++`. A string `++` will be considered as one operator, not two `+` operators, since `operator` is defined as the longest one among the operators.

An intersection symbol is defined with a binary operator of symbols `&`. A string is matched to $A\&B$ if and only if the string can be derived from A and B. That is, $A\&B$ produces A with a condition that the string derived from A must also be derivable from B.

In Figure 2, `keyword` is defined as the intersection of `name` and one of the keywords. A `keyword` must be the longest English alphabet string and one of the keywords at the same time. A string `ifx`, for example, is not a `keyword` since it is not one of the keywords even though it is a `name`. It could be two tokens of a keyword `if` followed by a name `x` without the longest match and intersection.

An exclusion symbol is defined with a binary operator of symbols `-`. For an arbitrary symbol A and B, A-B matches to a string if and only if the string can be derived from A, but not from B. In other words, A-B produces A with a condition that the string derived from A must not be derivable from B. In Figure 2, `identifier` is defined using exclusion. A string `if`, for example, is not an `identifier` since it is a `keyword` even though it is a `name`.

`$`A is followed-by symbol and `!`A is not-followed-by symbol. These symbols produce empty sequence ϵ but add conditions to the following string. `$`A produces ϵ with a condition that there must exist a prefix of the following string that can be derived from A. `!`A produces ϵ with a condition that there must not exist a prefix of the following string that can be derived from A.

For example, consider the following grammar:

```
S = 's' A {0-9}+
A = 'a' $C 'b'*
C = 'b'* '0'
```

This grammar is purely artificial, but it demonstrates how the followed-by symbol works. Consider a string `sab0`. The first character `s` belongs to the nonterminal `S`, the second character `a` belongs to the nonterminal `A`. The followed-by symbol `$C` adds a condition that there must exist a string that can be derived from `C` in the prefixes of `b0`, that is, $\{\epsilon, \mathtt{b}, \mathtt{b0}\}$. It turns out that `C` can derive `b0`, and `sab0` is in the language. However, `sab1` is not accepted since nonterminal `C` cannot derive any string of $\{\epsilon, \mathtt{b}, \mathtt{b0}\}$.

Note that `$C` examines the string out of the derivation of `A`, where `$C` is belonged to. That is, `A` derives only `ab`, but `$C` examines the string including `0` after that.

We write a conditional nonterminal has *body*. A conditional nonterminal produces its body with a new condition. The body of $A\&B$ is A, and the body of $A-B$ is A. Followed-by symbols and not-followed-by symbols have ϵ body. The body of <A> is A.

Figure 3 shows CDG describing CDG itself. The grammar defines the complete specification of the CDG. It does not require lexical analyzer or custom code writing, but it describes the full specification of the language.

2.2 Formal Descriptions

Definition 2.1. A conditional derivation grammar G is a 5-tuple (V_N, V_T, V_C, R, S) such that

(1) V_N, V_T, V_C are finite sets of symbols,
(2) $V_N \cap V_T = \emptyset$,
(3) $V_N \cap V_C = \emptyset$,
(4) $V_T \cap V_C = \emptyset$,
(5) $V_C = V_\& \cup V_- \cup V_\$ \cup V_! \cup V_L$ such that
    $V_\&, V_-, V_\$, V_!$, and V_L are all disjoint with each other,
    $V_\&$ is a finite set of intersection symbols $A\&B$ such that $A, B \in (V_N \cup V_T \cup V_C)$,
    V_- is a finite set of exclusion symbols $A-B$ such that $A, B \in (V_N \cup V_T \cup V_C)$,
    $V_\$$ is a finite set of followed-by symbols $\$A$ such that $A \in (V_N \cup V_T \cup V_C)$,
    $V_!$ is a finite set of not-followed-by symbols $!A$ such that $A \in (V_N \cup V_T \cup V_C)$, and
    V_L is a finite set of longest match symbols <A> such that $A \in (V_N \cup V_T \cup V_C)$.
(6) R is a set of pairs (P, Q) such that
    $P \in V_N$ and
    $Q \in (V_N \cup V_T \cup V_C)^*$,
(7) $S \in V_N$.

V_N is a set of nonterminals and V_T is a set of terminals. R is a set of produce rules from a nonterminal to a sequence of symbols. The left-hand-side of a production rule is always a single nonterminal as in CFGs. We write $R(A)$ is the right-hand-sides of nonterminal A, that is, $R(A) = \{Q | (P, Q) \in R, P = A\}$. V_C is a set of conditional nonterminals. Each symbol of V_C is neither a nonterminal nor a terminal. A conditional nonterminal refers to one or two symbols, and the referred symbols can be V_N, V_T, or V_C.

As a convention, we will use uppercase alphabets $A, B, \ldots$ to represent all kinds of symbols. Unless explicitly specified, an uppercase alphabet can be a terminal, a nonterminal, or a conditional nonterminal.

Definition 2.2 (terminal strings). For a grammar $G = (V_N, V_T, V_C, R, S)$,

- a terminal string $X = X_1 X_2 \ldots X_n$ is a sequence of terminals, i.e. $X \in V_T^*$,
- $|X|$ is the length of X, i.e. $|X| = n$ where $X = X_1 X_2 \ldots X_n$,
- a substring of X from index i to j is $X_{i..j} = X_{i+1} X_{i+2} \ldots X_j$ for $0 \le i \le j \le |X|$, e.g. $\mathtt{abcd}_{0..2} = \mathtt{ab}$, $\mathtt{abcd}_{1..1} = \epsilon$, $\mathtt{abcd}_{1..4} = \mathtt{bcd}$, and

```
Grammar = ws* Rules ws*
Rules = Rules <(ws-'\n')*> '\n' ws* Rule | Rule
Rule = Nonterminal ws* '=' ws* RHSs
RHSs = RHSs ws* '|' ws* Sequence | Sequence
EmptySequence = 'ϵ'
Sequence = EmptySequence | Symbol | SymbolSeq
SymbolSeq = SymbolSeq ws+ Symbol | Symbol ws+ Symbol
Symbol = Exclusion-Symbol4 | Symbol4
Exclusion = Symbol4 | Exclusion ws* '-' ws* Symbol4
Symbol4 = Intersection-Symbol3 | Symbol3
Intersection = Symbol3 | Intersection ws* '&' ws* Symbol3
Symbol3 = Repeat0 | Repeat1 | Optional | Symbol2
Repeat0 = Symbol3 ws* '*'
Repeat1 = Symbol3 ws* '+'
Optional = Symbol3 ws* '?'
Symbol2 = FollowedBy | NotFollowedBy | Symbol1
FollowedBy = '$' ws* Symbol2
NotFollowedBy = '!' ws* Symbol2
Symbol1 = Terminal | String | Nonterminal | Proxy | Longest
        | '(' ws* Symbol ws* ')' | '(' ws* Either ws* ')'
Terminal = anychar | '\'' char '\'' | TerminalCharSet
TerminalCharSet = '{' TerminalCharRange+ '}'
TerminalCharRange = charSetChar | charSetChar '-' charSetChar
String = '"' stringChar* '"'
Nonterminal = PlainNonterminalName | QuoteNonterminalName
PlainNonterminalName = {0-9A-Z_a-z}+
QuoteNonterminalName = '`' nontermNameChar* '`'
Proxy = '[' ws* Sequence ws* ']'
Either = Symbol ws* '|' ws* Symbol | Either ws* '|' ws* Symbol
Longest = '<' ws* Symbol ws* '>'
anychar = .
char = .-'\\' | '\\' {"'\\bnrt} | unicodeChar
charSetChar = .-{\-\\}} | '\\' {"'\-\\bnrt\}} | unicodeChar
stringChar = .-{"\\} | '\\' {"'\\bnrt} | unicodeChar
nontermNameChar = .-{\\`} | '\\' {\\`bnrt} | unicodeChar
unicodeChar = '\\' 'u' {0-9A-Fa-f} {0-9A-Fa-f} {0-9A-Fa-f} {0-9A-Fa-f}
ws = {\t\n\r␣}
```

Fig. 3. CDG describing its own syntax

- $prefixes(X) = \{X_{0..i} | 0 \leq i \leq |X|\}$.

To specify the semantics of the conditional nonterminals, we need a tool to mark specific locations in the derived sentence. Therefore, we introduce *location marker strings*.

Definition 2.3 (location marker string). For a grammar $G = (V_N, V_T, V_C, R, S)$, a location marker string $\alpha \in (V_N \cup V_T \cup V_C \cup M)^*$ where M is a set of location markers such that

- $M \cap V_N = \emptyset$,
- $M \cap V_T = \emptyset$,
- $M \cap V_C = \emptyset$, and
- $\forall m \in M$ appears at most once in α.

We write a location marker as sharps(#) with subscripts, e.g. $\#_1$, $\#_2$, $\#_i$, or $\#_j$. Location markers with number subscripts are concrete ones, and alphabet subscripts represents location marker variables. A location marker only represents a specific location in the string, and it does not affect the derivation. As a convention, we will write a location marker string with Greek letters e.g. α, β, or ω. Greek letter with number subscript represents a character in the location marker string.

For a location marker string α,

- $\lfloor\alpha\rfloor$ is the sequence of symbols which all markers are removed from α.
- $\alpha_{\#_i..\#_j}$ is the slice of α from $\#_i$ to $\#_j$.
- $\alpha_{\#_i..}$ is the slice of α from $\#_i$ to the end of the string.
- $\alpha_{..\#_i}$ is the slice of α from the beginning of the string to $\#_i$.
- α is a terminal location marker string if it does not have any nonterminals or conditional nonterminals, i.e. $\alpha \in (V_T \cup M)^*$.

For example, if $\alpha = \mathtt{A}\ \#_1\ \mathtt{b}\ \mathtt{C}\ \#_2\ \mathtt{d}$, $\lfloor\alpha\rfloor = \mathtt{A\ b\ C\ d}$, $\alpha_{\#_1..\#_2} = \mathtt{b\ C}$, $\alpha_{\#_1..} = \mathtt{b\ C}\ \#_2\ \mathtt{d}$, and $\alpha_{..\#_2} = \mathtt{A}\ \#_1\ \mathtt{b\ C}$. Also, α is not a terminal location marker string since it contains nonterminals.

The conditions on the derivations are represented by five types of *accept conditions*: `unless`, `onlyif`, `followed`, `notfollowed`, and `longest`. As the names imply, each type of accept condition is associated with a type of conditional nonterminal.

Definition 2.4 (accept conditions). For a grammar $G = (V_N, V_T, V_C, R, S)$ and a set of location markers M, an accept condition is one of the followings:

- `unless` $\#_i\ \#_j\ A$ such that $\#_i, \#_j \in M$, and $A \in (V_N \cup V_T \cup V_C)$.
- `onlyif` $\#_i\ \#_j\ A$ such that $\#_i, \#_j \in M$, and $A \in (V_N \cup V_T \cup V_C)$.
- `followed` $\#_i\ A$ such that $\#_i \in M$ and $A \in (V_N \cup V_T \cup V_C)$.
- `notfollowed` $\#_i\ A$ such that $\#_i \in M$ and $A \in (V_N \cup V_T \cup V_C)$.
- `longest` $\#_i\ \#_j\ A$ such that $\#_i, \#_j \in M$, and $A \in (V_N \cup V_T \cup V_C)$.

Using the location marker string and accept conditions, we can define the derive relation $\Rightarrow_G$ for a grammar G. This relation is from and to a pair of a location marker string and a set of accept conditions. The relation is defined as follows:

- For a nonterminal symbol A, $(\alpha A\beta, c) \Rightarrow_G (\alpha Q\beta, c)$ iff $\exists Q \in R(A)$.
- $(\alpha A\&B\beta, c) \Rightarrow_G (\alpha \#_i A \#_j \beta, c \cup \{\texttt{onlyif}\ \#_i\ \#_j\ B\})$.
- $(\alpha A\text{-}B\beta, c) \Rightarrow_G (\alpha \#_i A \#_j \beta, c \cup \{\texttt{unless}\ \#_i\ \#_j\ B\})$.
- $(\alpha \$A\beta, c) \Rightarrow_G (\alpha \#_i \beta, c \cup \{\texttt{followed}\ \#_i\ A\})$.
- $(\alpha !A\beta, c) \Rightarrow_G (\alpha \#_i \beta, c \cup \{\texttt{notfollowed}\ \#_i\ A\})$.
- $(\alpha \texttt{<}A\texttt{>}\beta, c) \Rightarrow_G (\alpha \#_i A \#_j \beta, c \cup \{\texttt{longest}\ \#_i\ \#_j\ A\})$.

The mechanism of the derivation is similar to CFG where a nonterminal is substituted to one of the production rules of it. A conditional nonterminal is substituted to its body and markers adding new accept condition. An accept condition is created when a conditional nonterminal is replaced, and it is only accumulated but never mutated nor removed. Also, accept conditions never affect the derivation process.

An alternative and more formal definition of the derivation relation is as follows:

Definition 2.5 ($\Rightarrow_G$ relation). For a grammar $G = (V_N, V_T, V_C, R, S)$, letting $V = V_N \cup V_T \cup V_C$,

- $(\psi, c) \Rightarrow_G (\omega, c')$ iff $\exists P \in V_N, Q \in V^*, \alpha, \beta, \psi, \omega \in (V \cup M)^* : (\psi = \alpha P \beta) \wedge (P, Q \in R) \wedge (\omega = \alpha Q \beta) \wedge (c = c')$.
- $(\psi, c) \Rightarrow_G (\omega, c')$ iff $\exists A\&B \in V_C, \alpha, \beta, \psi, \omega \in (V \cup M)^*, \#_i, \#_j \in M : (\psi = \alpha A\&B\beta) \wedge (\#_i \notin \psi) \wedge (\#_j \notin \psi) \wedge (\omega = \alpha \#_i A \#_j \beta) \wedge (c' = c \cup \{\texttt{onlyif}\ \#_i\ \#_j\ B\})$.

- $(\psi, c) \Rightarrow_G (\omega, c')$ iff $\exists A\texttt{-}B \in V_C, \alpha, \beta, \psi, \omega \in (V \cup M)^*, \#_i, \#_j \in M : (\psi = \alpha A\texttt{-}B\beta) \wedge (\#_i \notin \psi) \wedge (\#_j \notin \psi) \wedge (\omega = \alpha \#_i A \#_j \beta) \wedge (c' = c \cup \{\texttt{unless}\ \#_i\ \#_j\ B\})$.
- $(\psi, c) \Rightarrow_G (\omega, c')$ iff $\exists \$A \in V_C, \alpha, \beta, \psi, \omega \in (V \cup M)^*, \#_i \in M : (\psi = \alpha \$ A\beta) \wedge (\#_i \notin \psi) \wedge (\omega = \alpha \#_i \beta) \wedge (c' = c \cup \{\texttt{followed}\ \#_i\ B\})$.
- $(\psi, c) \Rightarrow_G (\omega, c')$ iff $\exists !A \in V_C, \alpha, \beta, \psi, \omega \in (V \cup M)^*, \#_i \in M : (\psi = \alpha !A\beta) \wedge (\#_i \notin \psi)(\omega = \alpha \#_i \beta) \wedge (c' = c \cup \{\texttt{notfollowed}\ \#_i\ B\})$.
- $(\psi, c) \Rightarrow_G (\omega, c')$ iff $\exists \texttt{<}A\texttt{>} \in V_C, \alpha, \beta, \psi, \omega \in (V \cup M)^*, \#_i, \#_j \in M : (\psi = \alpha \texttt{<}A\texttt{>} \beta) \wedge (\#_i \notin \psi) \wedge (\#_j \notin \psi) \wedge (\omega = \alpha \#_i A \#_j \beta) \wedge (c' = c \cup \{\texttt{longest}\ \#_i\ \#_j\ B\})$.

We write $(\alpha, c) \Rightarrow_G^* (\beta, c')$ iff $\exists \omega_1, \ldots, \omega_{n-1}$ such that $(\alpha, c) = (\omega_0, c_0) \Rightarrow_G (\omega_1, c_1) \Rightarrow_G \cdots \Rightarrow_G (\omega_n, c_n) = (\beta, c')$. A sequence of location marker strings $\omega_0, \omega_1, \ldots, \omega_n$ is a derivation from α to β. Note that $\Rightarrow_G^*$ is reflexive.

Even if $(S, \emptyset) \Rightarrow_G^* (\alpha, c)$ such that $\alpha \in (V_T \cup M)^*$, $\lfloor \alpha \rfloor$ may not be acceptable by the accept conditions. Only if every accept condition of c is satisfied, $\lfloor \alpha \rfloor$ is accepted as a string of the language. We write a derivation to a terminal string is acceptable if all the accept conditions are satisfied.

We define *acceptable* function that takes two arguments of a location marker string α and an accept condition to determine whether the accept condition is satisfied in α. We write, for a symbol A and a terminal string X, $A \mapsto X$ iff $(A, \emptyset) \Rightarrow_G^* (\alpha, c)$ such that $\lfloor \alpha \rfloor = X$ and $\forall x \in c, \mathit{acceptable}(\alpha, x)$.

Definition 2.6. For a terminal location marker string $\alpha \in (V_T \cup M)^*$ and an accept condition,

- $\mathit{acceptable}(\alpha, \texttt{onlyif}\ \#_i\ \#_j\ A)$ iff $A \mapsto \lfloor \alpha_{\#_i .. \#_j} \rfloor$,
- $\mathit{acceptable}(\alpha, \texttt{unless}\ \#_i\ \#_j\ A)$ iff $A \not\mapsto \lfloor \alpha_{\#_i .. \#_j} \rfloor$,
- $\mathit{acceptable}(\alpha, \texttt{followed}\ \#_i\ A)$ iff $\exists X \in \mathit{prefixes}(\lfloor \alpha_{\#_i ..} \rfloor)$ such that $A \mapsto X$,
- $\mathit{acceptable}(\alpha, \texttt{notfollowed}\ \#_i\ A)$ iff $\nexists X \in \mathit{prefixes}(\lfloor \alpha_{\#_i ..} \rfloor)$ such that $A \mapsto X$, and
- $\mathit{acceptable}(\alpha, \texttt{longest}\ \#_i\ \#_j\ A)$ iff $\nexists X \in (\mathit{prefixes}(\lfloor \alpha_{\#_j} \rfloor) - \epsilon)$ such that $A \mapsto (\lfloor \alpha_{\#_i .. \#_j} \rfloor X)$.

Note that *acceptable* is defined only on terminal location marker strings $(V_T \cup M)^*$. It means that we can determine whether an accept condition is satisfied only when we have a derivation to a terminal location marker string.

Definition 2.7. The *language* $L(G)$ of a CDG $G = (V_N, V_T, V_C, R, S)$ is the set of strings $X \in V_T^*$ such that $S \mapsto X$, i.e. $L(G) = \{X \in V_T^* | S \mapsto X\}$.

Definition 2.8. A language L over an alphabet V_T is a *Conditional Derivation Language*, or CDL iff there exists a conditional derivation grammar that defines the language.

The derivation relation in Definition 2.5 does not specify the order of nonterminals or conditional nonterminals to be replaced. We define *left-most derivation*, that always replaces the left-most nonterminal or conditional nonterminal. We will write left-most derivation relation as $\Rightarrow_G^L$.

We call a symbol A is *nullable* iff $(A, \emptyset) \Rightarrow_G^* (\alpha, c)$ such that $\lfloor \alpha \rfloor = \epsilon$ no matter whether the derivation is acceptable or not. We call a grammar G is nullable iff it has a nullable symbol. A grammar G is called *ambiguous* iff $\exists X \in V_T^*$ such that there exists more than one left-most derivations from S to X. If a grammar G has a production rule $(P, Q) \in R$ such that $Q_1 = P$, G is left-recursive.

Figure 4 shows an example derivation of the grammar of Figure 1. It derives a string `1+2` from the start symbol `expression`. The accept conditions in the derivation are all empty since the grammar has no conditional nonterminals.

Figure 5 and 6 show two derivations of the grammar of Figure 2. Those derivations derive a string `++` from start symbol `S`. The derivation of Figure 5 finished as $(\#_1\ \texttt{'+'}\ \texttt{'+'}\ \#_2, \{\texttt{longest}\ \#_1\ \#_2\ \texttt{op}\})$,

$$(\texttt{expression}, \emptyset) \Rightarrow (\texttt{expression '+' term}, \emptyset) \Rightarrow (\texttt{term '+' term}, \emptyset) \Rightarrow$$
$$(\texttt{factor '+' term}, \emptyset) \Rightarrow (\texttt{number '+' term}, \emptyset) \Rightarrow (\texttt{[\{1-9\} \{0-9\}*] '+' term}, \emptyset) \Rightarrow$$
$$(\texttt{\{1-9\} \{0-9\}* '+' term}, \emptyset) \Rightarrow (\texttt{'1' \{0-9\}* '+' term}, \emptyset) \Rightarrow (\texttt{'1' '+' term}, \emptyset) \Rightarrow$$
$$\cdots \Rightarrow (\texttt{'1' '+' '2'}, \emptyset)$$

Fig. 4. A derivation of expression grammar of Figure 1 to a terminal string 1+2

and the `longest` accept condition is met since $\#_2$ is at the end of the string. On the other hand, the derivation of Figure 6 finished as ($\#_1$ '+' $\#_2$ $\#_3$ '+' $\#_4$, {`longest` $\#_1$ $\#_2$ op, `longest` $\#_3$ $\#_4$ op}), and the accept condition `longest` $\#_1$ $\#_2$ op is not satisfied since ++ is a longer match. These two derivations derive the same string, but one derivation is acceptable while the other is not. The string ++ is in the language anyway since there exists an acceptable derivation.

$$(\texttt{S}, \emptyset) \Rightarrow (\texttt{token*}, \emptyset) \Rightarrow (\texttt{token* token}, \emptyset) \Rightarrow (\texttt{token}, \emptyset) \Rightarrow (\texttt{operator}, \emptyset) \Rightarrow (\texttt{<op>}, \emptyset) \Rightarrow$$
$$(\#_1 \text{ op } \#_2, \{\texttt{longest } \#_1 \ \#_2 \text{ op}\}) \Rightarrow (\#_1 \texttt{ '+' '+' } \#_2, \{\texttt{longest } \#_1 \ \#_2 \text{ op}\})$$

Fig. 5. A derivation of lexical grammar of Figure 2 to a string ++

$$(\texttt{S}, \emptyset) \Rightarrow (\texttt{token*}, \emptyset) \Rightarrow (\texttt{token* token}, \emptyset) \Rightarrow (\texttt{token* token token}, \emptyset) \Rightarrow$$
$$(\texttt{token token}, \emptyset) \Rightarrow \cdots \Rightarrow (\texttt{<op> <op>}, \emptyset) \Rightarrow (\#_1 \text{ op } \#_2 \texttt{ <op>}, \{\texttt{longest } \#_1 \ \#_2 \text{ op}\}) \Rightarrow$$
$$(\#_1 \text{ op } \#_2 \ \#_3 \text{ op } \#_4, \{\texttt{longest } \#_1 \ \#_2 \text{ op}, \texttt{longest } \#_3 \ \#_4 \text{ op}\}) \Rightarrow \cdots \Rightarrow$$
$$(\#_1 \texttt{ '+' } \#_2 \ \#_3 \texttt{ '+' } \#_4, \{\texttt{longest } \#_1 \ \#_2 \text{ op}, \texttt{longest } \#_3 \ \#_4 \text{ op}\})$$

Fig. 6. Another derivation of lexical grammar of Figure 2 to a string ++

2.3 Illegal Grammars

Consider a grammar with one nonterminal:
`S = !S {ab} | 'a'.`

A possible derivation of the grammar is:

$$(\texttt{S}, \emptyset) \Rightarrow (\texttt{!S \{ab\}}, \emptyset) \Rightarrow (\texttt{!S b}, \emptyset) \Rightarrow (\#_1\texttt{b}, \{\texttt{notfollowed } \#_1 \texttt{ S}\}) \tag{1}$$

After we get a derivation, we need to evaluate *acceptable*($\#_1$b, `notfollowed` $\#_1$ S). In order to evaluate this, we need to prove at least one of the followings is true:

(1) $(\texttt{S}, \emptyset) \Rightarrow^* (\alpha, c)$ such that $\lfloor \alpha \rfloor = \epsilon$ and *acceptable*(α, c).
(2) $(\texttt{S}, \emptyset) \Rightarrow^* (\alpha, c)$ such that $\lfloor \alpha \rfloor =$ b and *acceptable*(α, c).

The first case cannot be true since S is not nullable. However, the second case is tricky. The derivation (1) is the only possible conditional derivation from S to b. It means that the result of *acceptable*($\#_1$b, `notfollowed` $\#_1$ S) depends on the result of itself. A circular definition occurs, and we cannot property define the *acceptable* function in such cases. Therefore, we write a CDG is *illegal* if such circular definition problem can occur.

Definition 2.9. A grammar G is *illegal* iff

- $\exists A \in (V_N \cup V_C), (A, \emptyset) \Rightarrow^* (\alpha \$ A \beta, c)$ such that $\lfloor \alpha \rfloor = \epsilon$,
- $\exists A \in (V_N \cup V_C), (A, \emptyset) \Rightarrow^* (\alpha \, ! A \beta, c)$ such that $\lfloor \alpha \rfloor = \epsilon$,
- $\exists B \in (V_N \cup V_C), (B, \emptyset) \Rightarrow^* (\alpha A \& B \beta, c)$ such that $\lfloor \alpha \rfloor = \lfloor \beta \rfloor = \epsilon$, or
- $\exists B \in (V_N \cup V_C), (B, \emptyset) \Rightarrow^* (\alpha A \text{-} B \beta, c)$ such that $\lfloor \alpha \rfloor = \lfloor \beta \rfloor = \epsilon$.

Longest match symbols never cause such problems since `longest` condition always includes the following string in its evaluation.

In this paper, we assume that all CDGs are non-illegal.

2.4 Properties of CDG

This section discusses several properties of CDG including comparisons to other formal grammars.

Theorem 2.10. *The class of conditional derivation languages is closed under union, intersection, and complement.*

Proof. Suppose we have two CDGs $G_1 = (V_N^1, V_T, V_C^1, R_1, S_1)$ and $G_2 = (V_N^2, V_T, V_C^2, R_2, S_2)$. Without loss of generality, we can assume that $V_N^1 \cap V_N^2 = \emptyset$ by renaming nonterminals if required.

- If we construct a new grammar $G' = (V_N^1 \cup V_N^2 \cup \{S'\}, V_T, V_C^1 \cup V_C^2, R_1 \cup R_2 \cup \{(S', S_1), (S', S_2)\}, S')$ where $S' \notin (V_N^1 \cup V_N^2)$, then $L(G') = L(G_1) \cup L(G_2)$.
- If we construct a new grammar $G' = (V_N^1 \cup V_N^2 \cup \{S'\}, V_T, V_C^1 \cup V_C^2, R_1 \cup R_2 \cup \{(S', S_1 \& S_2)\}, S')$ where $S' \notin (V_N^1 \cup V_N^2)$, then $L(G') = L(G_1) \cap L(G_2)$.
- If we construct a new grammar $G' = (V_N^1 \cup \{S'\}, V_T, V_C^1, R_1 \cup \{(S', (.*) \text{-} S_1)\}, S')$ where $S' \notin V_N^1$, then $L(G') = V_T^* - L(G_1)$.

□

A CFG $G_0 = (V_N, V_T, R, S)$ can be directly transformed into a CDG $G = (V_N, V_T, \emptyset, R, S)$. The language of G_0 and G are identical since the nonterminal derivation rule of CDG is identical to CFG.

A context-sensitive grammar may not be directly transformed to a CDG. However, any context-sensitive language can be defined as an intersection of two context-free grammars(Grune and Jacobs 2007), and CDG has intersection symbol. Therefore, any context-sensitive language can be defined in CDG.

For example, language $\{a^n b^n c^n | n \geq 1\}$ is not a context-free language. Using CDG, it can be defined using intersection symbol as shown in Figure 7 or using followed-by symbol as shown in Figure 8.

```
S = P&Q
P = A 'c'*
A = 'a' A 'b' | 'a' 'b'
Q = 'a'* B
B = 'b' B 'c' | 'b' 'c'
```

Fig. 7. CDG describing $\{a^n b^n c^n | n \geq 1\}$ using intersection symbol

Parsing Expression Grammar(Ford 2004), or PEG has similar objective to provide a more expressive and user-friendly grammar to define a language, which is similar to CDG. A parsing expression of a PEG can be locally transformed to CDG symbols as follows:

```
S = $P 'a'* B
P = A 'c'
A = 'a' A 'b' | 'a' 'b'
B = 'b' B 'c' | 'b' 'c'
```

Fig. 8. CDG describing $\{a^n b^n c^n | n \geq 1\}$ using followed-by symbol

(1) an empty string ϵ to `[`ϵ`]`
(2) a terminal a to a
(3) a nonterminal A to A
(4) a sequence $e_1 e_2$ to `[<`e_1`> <`e_2`>]`.
(5) a prioritized choice e_1 / e_2 to `(<`e_1`> | [!`e_1 `<`e_2`>])`
(6) a zero-or-more repetitions e^* to `<`e`*>`
(7) a not-predicate `!`e to `!`e

3 PARSING ALGORITHM

This section describes an algorithm to recognize the language by given conditional derivation grammar. The proposed parser can parse all non-illegal conditional derivation grammars, including ambiguous, left recursive, or nullable grammars, with no modification. The proposed parsing algorithm simulates left-most derivations. The algorithm takes a CDG G and an input terminal string X and *accepts* X if $X \in L(G)$ or *rejects* X otherwise.

The proposed parsing algorithm is left-to-right directional parser. It *consumes* terminals of the input string from left to right, one by one. The algorithm constructs a *parsing context* every time it consumes a terminal. A parsing context contains the information about the progresses of the parsing by the consumed terminals and possible progresses to come.

We write the parser is in *generation* 0 when the parser has consumed no terminals, and the generation increases by 1 as it consumes a terminal. Therefore, parser in generation n represents the state after it has consumed the first n terminals of X, i.e. $X_{0..n}$. We write that a parser *proceeds* when it consumes a terminal and the generation number increases.

Section 3.1 introduces parsing context, and Section 3.2 describes how a parsing context is constructed. Section 3.3 shows a working example. Section 3.4 sketches the correctness proof and Section 3.5 discusses the time and space complexity. Section 3.6 introduces an implementation of the proposed algorithm.

3.1 Parsing Context

Given grammar G and terminal string X, the proposed algorithm constructs a *parsing context* for each generation from 0 to $|X|$. A parsing context of generation i is a graph, that represents the progresses of the parsing by $X_{0..i}$ and possible future progresses. The parsing context of generation 0 is specifically called the *initial parsing context*.

Figure 9 shows the initial parsing context of the arithmetic expression grammar shown in Figure 1. $((\bullet$`expression`$, 0..0), \emptyset)$ node means that as of generation 0, nonterminal `expression` may be followed, but parser has been recognized nothing. The node $v = ((\bullet$`'(' expression ')'`$, 0..0), \emptyset)$ means that as of generation 0, sequence `'(' expression ')'` may be followed, and nothing has been recognized. The edge from v to $((\bullet$`'('`$, 0..0), \emptyset)$ means that the former node can be progressed if the terminal `'('` is found from the following string.

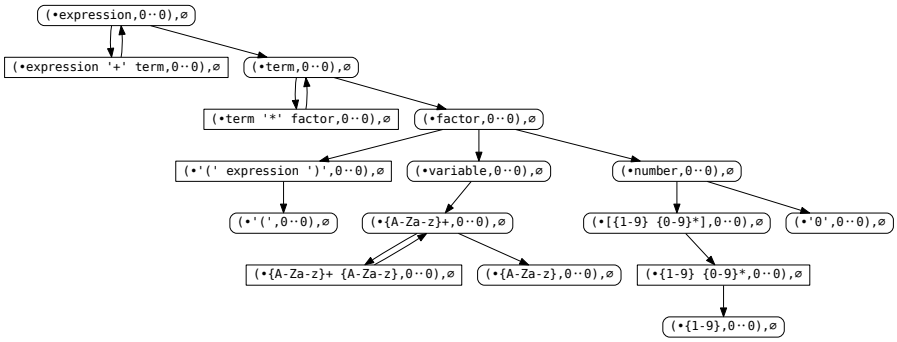

Fig. 9. Initial parsing context of expression grammar of Figure 1

Note that a parsing context graph may have cycles. The graph of Figure 9 has 3 cycles starting from $((\bullet\texttt{expression}, 0..0), \emptyset)$, $((\bullet\texttt{term}, 0..0), \emptyset)$, and $((\bullet\{\texttt{A-Za-z}\}\texttt{+}, 0..0), \emptyset)$. A left recursion causes a cycle in the graph.

Definition 3.1. $C_i = (V_i, E_i)$ is a parsing context at generation i such that

- V_i is a set of nodes v such that
  a node v is a pair (k, d) of kernel k and a set of indexed accept conditions d, and
- E_i is a set of pair of nodes (v_1, v_2) such that $v_1 \in V_g$ and $v_2 \in V_g$.

A node of a parsing context is a pair of a *kernel* and a set of *indexed accept conditions*. A kernel has 4 components: kernel body Z, pointer p, beginning generation number g_0, and ending generation number g_1. A kernel body is either one of the symbols of $(V_N \cup V_T \cup V_C)$ or one of the right-hand-side of the production rules R, i.e. $\{Q \in (V_N \cup V_T \cup V_C)^* | (P, Q) \in R\}$. A kernel is called a *symbol kernel* if the kernel body is a symbol, and it is called a *sequence kernel* if the kernel body is a right-hand-side of R unless its length is 1. A kernel body from the right-hand-sides of R of which length is 1 is a symbol kernel.

We write $Z = Z_1$ if Z is a symbol, and $Z = Z_1 Z_2 \ldots Z_{|Z|}$ if Z is a sequence. For a kernel body Z, $\overline{Z}$ is the length of Z, and it is defined as follows:

$$\overline{Z} = \begin{cases} 1 & \text{if } Z \in (V_N \cup V_T \cup V_C), \\ |Z| & \text{if } Z \in (V_N \cup V_T \cup V_C)^* \end{cases}$$

Definition 3.2. A kernel k of a node $(k, d) \in V_i$ is a quadruple (Z, p, g_0, g_1) such that

- $Z \in (V_N \cup V_T \cup V_C)$ or $Z \in \{Q \in (V_N \cup V_T \cup V_C)^* | (P, Q) \in R\}$,
- $0 \leq p \leq \overline{Z}$, and
- $0 \leq g_0 \leq g_1 \leq i$.

A kernel may be written in more readable form as shown in Figure 9. The pointer will be written as a dot in the middle of kernel body, and the beginning and ending generation numbers are written as the form of substring index. That is, $(Z, p, g_0, g_0) = (Z_1 Z_2 \ldots Z_p \bullet Z_{p+1} Z_{p+2} \ldots Z_{\overline{Z}}, g_0..g_1)$. A kernel (Z, p, g_0, g_q) is called *initial* if $p = 0$ and *final* if $p = \overline{Z}$. A node $v = (k, d)$ is called initial if k is initial and final if k is final. We write $((Z, 0, g_0, g_0), \emptyset)$ is the *initial node* of node $((Z, p, g_0, g_1), d)$.

A node also has a set of indexed accept conditions. We have defined accept conditions in Definition 2.4, and indexed accept conditions are modified version of accept conditions. The main difference is that integer numbers replaces the markers.

When we have a terminal location marker string, markers in the string represent the indices of the string. For example, given grammar G, consider a derived terminal location marker string $\alpha = \mathsf{a}\ \#_1\ \#_3\ \mathsf{b}\ \#_4\ \mathsf{c}\ \#_2\ \mathsf{d}$ and an accept condition $\texttt{onlyif}\ \#_1\ \#_2\ \mathsf{A}$. To know whether $\lfloor\alpha\rfloor = \mathsf{abcd} \in L(G)$, we need to evaluate $\mathit{acceptable}(\alpha, \texttt{onlyif}\ \#_1\ \#_2\ \mathsf{A})$, that is, whether $\mathsf{A} \mapsto \lfloor\alpha_{\#_1..\#_2}\rfloor = \lfloor\#_3\ \mathsf{b}\ \#_4\ \mathsf{c}\rfloor = \mathsf{bc} = \mathsf{abcd}_{1..3}$. Since all the markers are removed after we get a slice from the terminal location marker string, we can think a marker represents an index. That is, when we have derived a terminal location marker string α, a marker represents an index in a terminal string $\lfloor\alpha\rfloor$. In the example above, $\#_1$ and $\#_3$ represent 1, $\#_4$ represents 2, and $\#_2$ represents 3.

In general, we call a marker $\#_x$ is *fixed* in α if the preceding part of the marker only has terminals and markers, i.e. $\alpha_{..\#_x} \in (V_T \cup M)^*$. A fixed marker can be transformed into the index $|\lfloor\alpha_{..\#_x}\rfloor|$. If all markers of an accept condition are fixed, the accept condition can be transformed into an indexed accept condition by replacing the markers into the indices.

Definition 3.3. *indexed* is a function that takes a location marker string α and an accept condition and returns an indexed accept condition. We assume that all markers in the accept condition are fixed in α.

- $\mathit{indexed}(\alpha, \texttt{onlyif}\ \#_i\ \#_j\ A) = \texttt{onlyif}\ (|\lfloor\alpha_{..\#_i}\rfloor|)\ (|\lfloor\alpha_{..\#_j}\rfloor|)\ A$,
- $\mathit{indexed}(\alpha, \texttt{unless}\ \#_i\ \#_j\ A) = \texttt{unless}\ (|\lfloor\alpha_{..\#_i}\rfloor|)\ (|\lfloor\alpha_{..\#_j}\rfloor|)\ A$,
- $\mathit{indexed}(\alpha, \texttt{followed}\ \#_i\ A) = \texttt{followed}\ (|\lfloor\alpha_{..\#_i}\rfloor|)\ A$,
- $\mathit{indexed}(\alpha, \texttt{notfollowed}\ \#_i\ A) = \texttt{notfollowed}\ (|\lfloor\alpha_{..\#_i}\rfloor|)\ A$, and
- $\mathit{indexed}(\alpha, \texttt{longest}\ \#_i\ \#_j\ A) = \texttt{longest}\ (|\lfloor\alpha_{..\#_i}\rfloor|)\ (|\lfloor\alpha_{..\#_j}\rfloor|)\ A$.

Definition 3.4. *acceptable* is a function that takes a terminal string $X \in V_T^*$ and an indexed accept condition and returns whether the condition is acceptable in X.

- $\mathit{acceptable}(X, \texttt{onlyif}\ i\ j\ A)$ iff $A \mapsto X_{i..j}$,
- $\mathit{acceptable}(X, \texttt{unless}\ i\ j\ A)$ iff $A \not\mapsto X_{i..j}$,
- $\mathit{acceptable}(X, \texttt{followed}\ i\ A)$ iff $\exists k \geq i$ such that $A \mapsto X_{i..k}$,
- $\mathit{acceptable}(X, \texttt{notfollowed}\ i\ A)$ iff $\nexists k \geq i$ such that $A \mapsto X_{i..k}$, and
- $\mathit{acceptable}(X, \texttt{longest}\ i\ j\ A)$ iff $\nexists k > j$ such that $A \mapsto X_{i..k}$.

A node $((Z, p, g_0, g_1), d)$ in a parsing context means that $(X_{0..g_0}Z_1Z_2\ldots Z_p, \emptyset) \Rightarrow_G^* (X_{0..g_0}\alpha, c)$ such that $\lfloor\alpha\rfloor = X_{g_0..g_1}$ and $d = \{\mathit{indexed}(X_{0..g_0}\alpha, x) | x \in c\}$. The node also means that $Z_{p+1}Z_{p+2}..Z_{\overline{Z}}$ may be found in the following string $X_{g_1..|X|}$. If the node v is initial, i.e. $p = 0$, $g_0 = g_1$ and $d = \emptyset$.

An edge (v, v') is a directed edge from v to v'. We write the edge is an outgoing edge of v and an incoming edge to v'. We also write v is an incoming node of v' and v' is an outgoing node of v.

If an edge $(v = ((Z, p, g_0, g_1), d), v' = ((Z', p', g_0', g_1'), d'))$ is in a parsing context, v is not final and g_0' is equal to g_1. The edge means that if Z' is matched to $X_{g_1'..g_2}$, then Z_{p+1} is also matched to $X_{g_1'..g_2}$.

When a node $v = ((Z, p, g_0, g_1), d)$ *progresses*, it adds a new node $v' = ((Z', p', g_0', g_1'), d')$ derived from v such that $Z = Z'$, $p + 1 = p'$, $g_0 = g_0'$, $g_1' \geq g_1$, and $d \subseteq d'$. The new node has the same kernel body with v, the pointer will be increased by 1, the beginning generation number will be same, the ending generation number may be increased, and a new accept condition may be added.

In the proposed algorithm, nodes and edges are immutable. Once a node or edge is created, it is never mutated.

3.2 Recognition Algorithm

Recognize procedure shown in Figure 10 takes two arguments of a conditional derivation grammar $G = (V_N, V_T, V_C, R, S)$ and an input string $X = X_1X_2 \ldots X_n$ and accepts or rejects X.

```
function Recognize(G = (V_N, V_T, V_C, R, S), X = X_1 X_2 ... X_n)
    C_0 ← ({((S, 0, 0, 0), ∅)}, ∅)                          ▷ Construction of C_0
    T ← [derive ((S, 0, 0, 0), ∅)]
    U ← ∅
    while T is not empty do
        Process(0, C_0, T, U)
    for i ← 1 to |X| do
        let v = ((X_i, 0, i − 1, i − 1), ∅)                 ▷ Proceed from C_{i−1} to C_i
        if v ∉ V_{i−1} then
            Reject X
        C_i ← C_{i−1}
        T ← [progress v ∅]
        U ← ∅
        while T is not empty do
            Process(i, C_i, T, U)
    let B = {((S, 1, 0, |X|), d) ∈ V_{|X|} | ∀x ∈ d, Acceptable(C_{|X|}, x)}
    if B = ∅ then
        Reject X
    else
        Accept X
```

Fig. 10. Recognize procedure

Recognize has the following internal mutable states: working parsing contexts C_i where $0 \leq i \leq |X|$, a collection of *parsing operations* T, and a set of pairs of nodes U.

When it constructs the initial parsing context, the working parsing context C_0 is initialized with only one node $((S, 0, 0, 0), \emptyset)$ and no edges. The node represents that a start symbol S may be followed. T initially has only one `derive` parsing operation for the start symbol node. A parsing operation is a unit of work that modifies the working parsing context. The set of pairs of nodes U prevents the execution order of operations from affecting the resulting parsing context. U is initially an empty set.

After initialization is done, parser starts to call Process procedure. Process pops a parsing operation from T, adds new nodes and/or edges to C_0, pushes new tasks to T, and puts new pairs to U. When T becomes empty, it means C_0 is established.

Proceeding from generation $i - 1$ to i starts by finding a node of which kernel body is a terminal of the next character X_i. If there is no such node, parser rejects X and terminates. If it finds such node v, it copies C_{i-1} to C_i, sets T to `progress` operation of v, and empties U. Then, it repeatedly calls Process until T becomes empty. When T becomes empty, C_i is established.

Recognize tries to find nodes with kernel $(S, 1, 0, |X|)$ in $C_{|X|}$ after it consumed all terminals of X. If there are no such nodes, parser rejects X and terminates. If there exist such nodes, parser finds a node of which indexed accept conditions are all acceptable. Acceptable function is used to evaluate the indexed accept conditions. Parser finally accepts X if there exists at least one such node in $C_{|X|}$, or rejects otherwise.

A parsing operation is one of the following:

- `derive` v such that $v \in V_i$
- `finish` v such that $v \in V_i$
- `progress` v d such that $v \in V_i$ and d is a set of indexed accept conditions.

`derive` v is created when a non-final node v is added to C_i. `derive` operation is executed exactly once for each and every non-final node. `finish` v is created when a final node v is added to C_i. `finish` operation is executed exactly once for each and every final node. `progress` v d is created when the pointer of v can be progressed with new indexed accept conditions d.

Process procedure processes a parsing operation. It takes 4 arguments: the target generation number i, a working parsing context C_i, a collection of parsing operations T, and a set of pairs of nodes U. The pseudo-code of Process is shown in Figure 11.

Process first pops an operation from T. The order of popped parsing operations is not specified since the result will be unaffected by the execution order of the operations.

If the popped operation is `finish` $v = (k, d)$, it first searches the incoming edges to the initial node of v. Then it creates new `progress` operations for the found nodes with the indexed accept conditions d, and pushes the operations to T.

If a node $v = ((Z, p, g_0, g_1), d)$ is final, it means that $(X_{0..g_0}Z, \emptyset) \Rightarrow_G^* (X_{0..g_0}\alpha, c)$ such that $\lfloor \alpha \rfloor = g_0..g_1$ and $d = \{indexed(X_{0..g_0}\alpha, x) | x \in c\}$. An edge $(v = ((Z, p, g_0, g_1), d), v' = ((Z', 0, g_0', g_0'), \emptyset))$ means that if Z' is matched in $X_{i..j}$, Z_{p+1} also can be matched in $X_{i..j}$. Therefore, a `finish` operation creates `progress` operations to the incoming nodes to the initial node of v.

If the popped operation is `progress` v d', v is a non-final node. It first creates a new node v' derived from v. v' has the same kernel body with v, and the pointer is increased by 1. The beginning generation number will be same with v, but the ending generation number will be i. The accept conditions of v' is based on the union of accept conditions of v and d' from the operation. If the kernel body of v is a conditional nonterminal, new accept condition will be added. If v' is already in V_i, it does nothing further. If v' is not in V_i, it first adds v' to V_i and pushes corresponding finish or derive operation to T. It then adds a pair of the initial node of v and v' to U.

Lastly, if the popped operation is `derive` $v = ((Z, p, g_0, g_1), d)$, the operation adds new nodes that can be derived from Z_{p+1} and edges from v to the new nodes, and pushes `derive` or `finish` operations corresponding to the new nodes. If Z is a sequence, it adds the node for the symbol following the dot, i.e. Z_{p+1}. If Z is a nonterminal, it adds the nodes for the right-hand-sides of the nonterminal Z. If Z is a conditional nonterminal, it adds the node for the body of Z and the node to be used in the evaluation of accept conditions. The derivation relation $\Rightarrow_G$ adds an accept condition when a conditional nonterminal is replaced, but `derive` operation does not create or add new accept condition since the required indices are not fixed yet. The accept condition is handled by `progress` operations.

U is used to correct the problem occurred from the execution order of `derive` and `progress` operations. U keeps track of the result of `progress` operations. Every time $v = ((Z, p, g_0, g_1), d)$ is progressed to create v', the pair of the initial node of v and v', i.e. $(((Z, 0, g_0, g_0), \emptyset), v')$ is added to U. Therefore, for every pair $(v, v') \in U$, v and v' have the same kernel body and v is always an initial node.

Suppose that `derive` v operation tries to add a new node v' but v' is already in the parsing context. It means that there exists another path to derive v', and progress operations for v' may be executed before the derive operation. It is also possible that v' may be progressed to a final node. In such cases, it needs to create progress operations for v to correct the effects of already running finish operations. Since U keeps the results of progress operations of v', we can find the required information.

function PROCESS(i, $C_i = (V_i, E_i)$, T, U)
  $t \leftarrow$ pop an operation from T
  function ADDNODE($v' = ((Z', p', g_0', g_1'), d')$) ▷ $g_1' = i$
    if $v' \notin V_i$ **then**
      $V_i \leftarrow V_i \cup \{v'\}$
      let $t' = \begin{cases} \texttt{finish}\ v' & \text{if } v' \text{ is final, i.e. } p' = \overline{Z'}, \\ \texttt{derive}\ v' & \textit{otherwise} \end{cases}$
      Push t' to T
  if $t = \texttt{derive}\ (v = ((Z, p, g_0, g_1), d))$ **then** ▷ $g_1 = i$
    function DERIVE(Z')
      let $v' = ((Z', 0, g_1, i), \emptyset)$
      $E_i \leftarrow E_i \cup \{(v, v')\}$
      if $v' \notin V_i$ **then**
        ADDNODE(v')
      else if v' is final **then**
        Push $\texttt{progress}\ v\ \emptyset$ to T
      else
        let $V' = \{v_2 \in V_i | (v_1, v_2) \in U, v_1 = v', v_2 \text{ is final}\}$
        Push all operations of $\{\texttt{progress}\ v\ d' | \{(k', d') \in V'\}\}$ to T
    if Z is a sequence **then**
      DERIVE(Z_{p+1})
    else if $Z \in V_N$ **then**
      for $Q \in R(Z)$ **do**
        DERIVE(Q)
    else if Z is $A\&B$ or $A\text{-}B$ **then** ▷ $Z \in V_C$
      DERIVE(A); ADDNODE($((B, 0, i, i), \emptyset)$)
    else if Z is $\$A$ or $!A$ **then**
      DERIVE(ϵ); ADDNODE($((A, 0, i, i), \emptyset)$)
    else if Z is <A> **then**
      DERIVE(A)
  else if $t = \texttt{progress}\ (v = ((Z, p, g_0, g_1), d))\ d'$ **then**
    let $d'' = \begin{cases} \{\texttt{unless}\ g_0\ i\ B\} & \text{if } Z = A\text{-}B, \\ \{\texttt{onlyif}\ g_0\ i\ B\} & \text{if } Z = A\&B, \\ \{\texttt{followed}\ i\ A\} & \text{if } Z = \$A, \\ \{\texttt{notfollowed}\ i\ A\} & \text{if } Z = !A, \\ \{\texttt{longest}\ g_0\ i\ A\} & \text{if } Z = \texttt{<A>}, \\ \emptyset & \textit{otherwise} \end{cases}$
    let $v' = ((Z, p + 1, g_0, i), d \cup d' \cup d'')$
    if $v' \notin V_i$ **then** ▷ Do nothing if $v' \in C_i$
      ADDNODE(v')
      $U \leftarrow U \cup \{(((Z, 0, g_0, g_0), \emptyset), v')\}$ ▷ Pair of initial node of v and v'
  else if $t = \texttt{finish}\ (v = ((Z, p, g_0, g_1), d))$ **then** ▷ $g_1 = i$
    let $v_0 = ((Z, 0, g_0, g_0), \emptyset)$ ▷ Initial node of v
    Push all operations of $\{\texttt{progress}\ v_1\ d | (v_1, v_2) \in E_i, v_2 = v_0\}$ to T

Fig. 11. PROCESS procedure

There are several properties about the algorithm and the resulting parsing context as follows:

- All `finish` operations have final nodes and all `derive` and `progress` operations have non-final nodes.
- For each and every node in the parsing context, exactly one `derive` or `finish` operation is executed.
- The ending generation numbers of nodes of `derive` or `finish` operations are the target generation number i.
- Every initial node has an empty set of accept conditions.
- For every edge (v, v') in a parsing context, the ending generation of v is equal to the beginning generation of v'.
- If there exists a non-initial node $v = ((Z, p, g_0, g_1), d)$, there exists another node $v' = ((Z', p', g_0', g_1'), d')$ such that $Z = Z'$, $p' < p$, $g_0 = g_0'$, $g_1' \leq g_1$, and $d' \subseteq d$.

Acceptable function takes two parameters of a parsing context C and an indexed accept condition x and returns true if x is acceptable or false otherwise.

function Acceptable($C = (V, E)$, x)
  if x = `onlyif i j A` **then**
    return $\exists (v = (k, d)) \in V$ such that $k = (A, 1, i, j)$ and $\forall y \in d$, Acceptable(C, y)
  else if x = `unless i j A` **then**
    return $\nexists (v = (k, d)) \in V$ such that $k = (A, 1, i, j)$ and $\forall y \in d$, Acceptable(C, y)
  else if x = `followed i A` **then**
    return $\exists (v = (k, d)) \in V, l \geq i$ such that $k = (A, 1, i, l)$ and $\forall y \in d$, Acceptable(C, y)
  else if x = `notfollowed i A` **then**
    return $\nexists (v = (k, d)) \in V, l \geq i$ such that $k = (A, 1, i, l)$ and $\forall y \in d$, Acceptable(C, y)
  else if x = `longest i j A` **then**
    return $\nexists (v = (k, d)) \in V, l > j$ such that $k = (A, 1, i, l)$ and $\forall y \in d$, Acceptable(C, y)

If a non-illegal grammar is given, Acceptable always terminates, never falling into infinite loops.

3.3 An Example

This section shows the parsing of the input string `++` on the lexical grammar shown in Figure 2. Figure 12 shows the initial parsing context. The figure does not show the outgoing nodes and edges from $((\bullet\{\texttt{A-Za-z}\}, 0..0), \emptyset)$ for readability, where the node in fact has outgoing nodes such as $((\bullet\texttt{'A'}, 0..0), \emptyset)$, $((\bullet\texttt{'B'}, 0..0), \emptyset)$, ..., $((\bullet\texttt{'z'}, 0..0), \emptyset)$.

In order to construct the graph in Figure 12, parser first adds a node $v_0 = ((\bullet\texttt{S}, 0..0), \emptyset)$ to C_0 and initializes T with `derive` v_0 operation. When Process is first called, `derive` v_0 operation is popped. Since the kernel body of v_0 is nonterminal `S`, it finds the right-hand-sides of `S` in the production rules and creates a node $v_1 = ((\bullet\texttt{token*}, 0..0), \emptyset)$. Since `token*` is a single symbol, v_1 is not a sequence node but a symbol node. An edge (v_0, v_1) is also added to C_0 and `derive` v_1 is pushed into T.

Since T is not empty, Process procedure is called again. The next operation to execute is `derive` $v_1 = ((\bullet\texttt{token*}, 0..0), \emptyset)$. `token*` is nonterminal with two production rules: `token* token` and ϵ. Therefore, parser adds two nodes $v_2 = ((\bullet\texttt{token* token}, 0..0), \emptyset)$ and $v_3 = ((\bullet, 0..0), \emptyset)$ to C_0. Two edges (v_1, v_2) and (v_1, v_3) are also added to C_0, and `derive` v_2 and `finish` v_3 are pushed into T. A `derive` operation is created for v_2 since it is non-final, and a `finish` operation is created for v_3 since it is a final node.

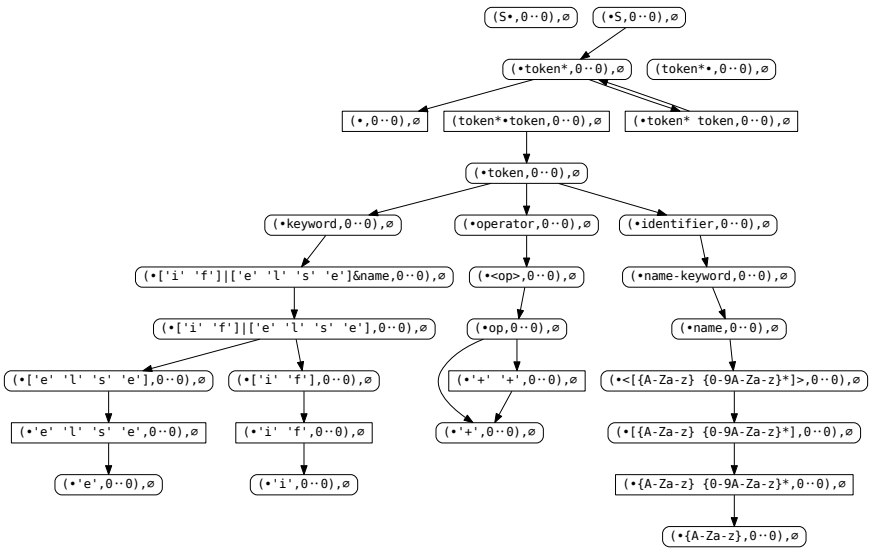

Fig. 12. Initial parsing context of lexical grammar of Figure 2

Suppose the `derive` operation for $v_2 = ((\bullet\texttt{token*}\ \texttt{token}, 0..0), \emptyset)$ is executed first. The symbol following the dot in v_2 is `token*`. Therefore, it tries to create a node $((\bullet\texttt{token*}, 0..0), \emptyset)$, but it is an already existing node v_1 in C_0. No new nodes or operations are created, but an edge (v_2, v_1) is added to C_0. A cycle is formed by a left-recursive production rule.

It is time to run `finish` $v_3 = ((\bullet, 0..0), \emptyset)$. It creates a new operation `progress` $v_1\ \emptyset$ by the incoming node and the accept conditions set of v_3.

`progress` $v_1 = ((\bullet\texttt{token*}, 0..0), \emptyset)\ \emptyset$ creates a new node $v_1' = ((\texttt{token*}\bullet, 0..0), \emptyset)$. The new node v_1' is a final node, so `finish` v_1' is pushed to T.

`finish` v_1' creates two progress operations by the incoming nodes to v_1, the initial node of v_1': `progress` $v_0\ \emptyset$ and `progress` $v_2\ \emptyset$. As a result, the finish operation of v_3 is chained through v_1 to progress v_0 to be final, and $((\texttt{S}\bullet, 0..0), \emptyset)$ is added. `progress` $v_2\ \emptyset$ creates a new node $v_2' = ((\texttt{token*} \bullet \texttt{token}, 0..0), \emptyset)$ and its `derive` operation. The new node $((\bullet\texttt{token}, 0..0), \emptyset)$ is added to C_0 as the result of `derive` v_2', and its derive operation recursively creates the rest of C_0.

Figure 13 shows the parsing context proceeded from generation 0 to generation 1 consuming a terminal $X_1 =$ `'+'`. Some parts of the graph in Figure 13 are abbreviated as a node labeled with dots. The omitted parts include the subgraph starting from $((\bullet\texttt{token}, 0..0), \emptyset)$, $((\bullet\texttt{identifier}, 1..1), \emptyset)$, and $((\bullet\texttt{keyword}, 1..1), \emptyset)$. They are exactly same as of generation 0 or almost same except the generation numbers.

To proceed to generation 1, parser first finds if there is a node $v_t = ((\bullet\texttt{'+'}, 0..0), \emptyset)$, which is created from X_1. Since C_0 has v_t, parser initializes C_1 as a copy of C_0, T to have only one operation `progress` $v_t\ \emptyset$, and U is empty.

The first operation `progress` $v_t\ \emptyset$ creates a new node $v_t' = ((\texttt{'+'}\bullet, 0..1), \emptyset)$ and a `finish` operation for v_t'.

`finish` $v_t' = ((\texttt{'+'}\bullet, 0..1), \emptyset)$ creates two operations by the incoming nodes of v_t, which is the initial node of v_t': `progress` $v_4 = ((\bullet\texttt{'+' '+'}, 0..0), \emptyset)\ \emptyset$ and `progress` $v_5 = ((\bullet\texttt{op}, 0..0), \emptyset)\ \emptyset$

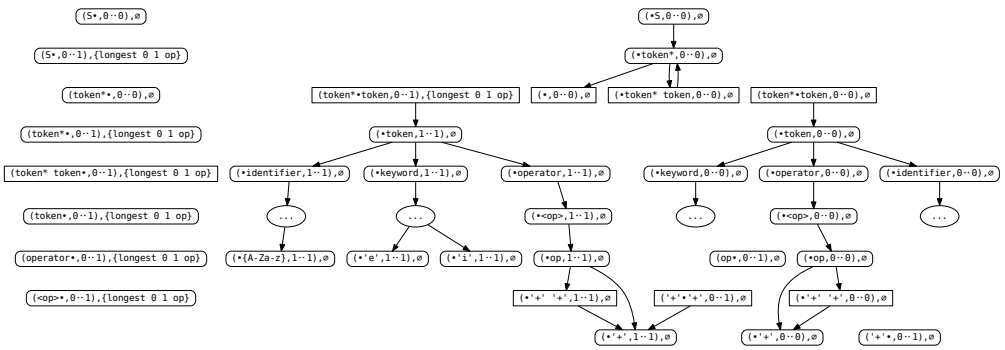

Fig. 13. Parsing context at generation 1

progress $v_4 = ((\bullet\texttt{'+' '+'}, 0..0), \emptyset)\ \emptyset$ adds a new node $v_4' = ((\texttt{'+' } \bullet \texttt{ '+'}, 0..1), \emptyset)$. Since v_4' is a new non-final node, derive v_4' is pushed to T. derive $v_4' = ((\texttt{'+' } \bullet \texttt{ '+'}, 0..1), \emptyset)$ adds a new node $((\bullet \texttt{ '+'}, 1..1), \emptyset)$ and an edge from v_4' to the node. The derive operation is also created for the new node, but it does nothing since the node is terminal node. The progress operation for v_4 considers the case where the first character + is the part of token ++, where another + should be followed.

progress $v_5 = ((\bullet\texttt{op}, 0..0), \emptyset)\ \emptyset$, created by the finish operation of v_t', considers the case where the consumed terminal + is a complete token. This operation creates a new node $((\texttt{op} \bullet, 0..1), \emptyset)$, and it is chained to create a progress operation for the node $v_6 = ((\bullet\texttt{<op>}, 0..0), \emptyset)$.

progress $v_6 = ((\bullet\texttt{<op>}, 0..0), \emptyset)\ \emptyset$ has a node of which kernel body is a conditional nonterminal. Therefore, this node creates a new node with a new indexed accept condition. A longest condition is added since the kernel body is a longest match symbol, and the created node is $v_6' = ((\texttt{<op>}\bullet, 0..1), \{\texttt{longest 0 1 op}\})$. The new node v_6' is a final node, and it creates a finish operation, that creates progress $((\bullet\texttt{operator}, 0..0), \emptyset)\ \{\texttt{longest 0 1 op}\}$. The longest condition will be propagated to all the chains of progress and finish operations from v_6'. As a result, the start symbol node is progressed to $((\texttt{S}\bullet, 0..1), \{\texttt{longest 0 1 op}\})$. The accept condition longest 0 1 op is acceptable in C_1, since there is no node such as $((\texttt{op}\bullet, 0..2), c)$ or $((\texttt{op}\bullet, 0..3), c)$, that means no longer match is found so far.

Figure 14 shows the parsing context at generation 2 consumed $X_2 = \texttt{'+'}$ from the parsing context at generation 1 shown in Figure 13. As in Figure 13, some parts that can be easily inferred are abbreviated as the nodes labeled with dots.

The overall process of proceeding from generation 1 to 2 is similar to proceeding from generation 0 to 1. It starts from a progress $((\bullet\texttt{'+'}, 1..1), \emptyset)\ \emptyset$, and repeats running Process procedure until the operation collection becomes empty and the parsing context established.

After parser finishes the proceeding to generation 2, parser realizes that there is a longer match of op in $X_{0..2}$ by the existence of node $((\texttt{op}\bullet, 0..2), \emptyset)$. It means that all nodes of which accept conditions set has longest 0 1 op are not acceptable anymore. For example, the nodes $((\texttt{token*} \bullet \texttt{ token}, 0..1), \{\texttt{longest 0 1 op}\})$ and $((\texttt{S}\bullet, 0..2), \{\texttt{longest 0 1 op, longest 1 2 op}\})$ are not acceptable.

After parser consumed all terminals of the input string, it finds nodes of which kernel is $(\texttt{S}\bullet, 0..2)$. There are two nodes with the kernel: $v_7 = ((\texttt{S}\bullet, 0..2), \{\texttt{longest 0 1 op, longest 1 2 op}\})$ and $v_8 = ((\texttt{S}\bullet, 0..2), \{\texttt{longest 0 2 op}\})$. Among these two, parser rules out v_7 since it is an unacceptable node, that has an unacceptable condition longest 0 1 op. On the other hand, the only accept

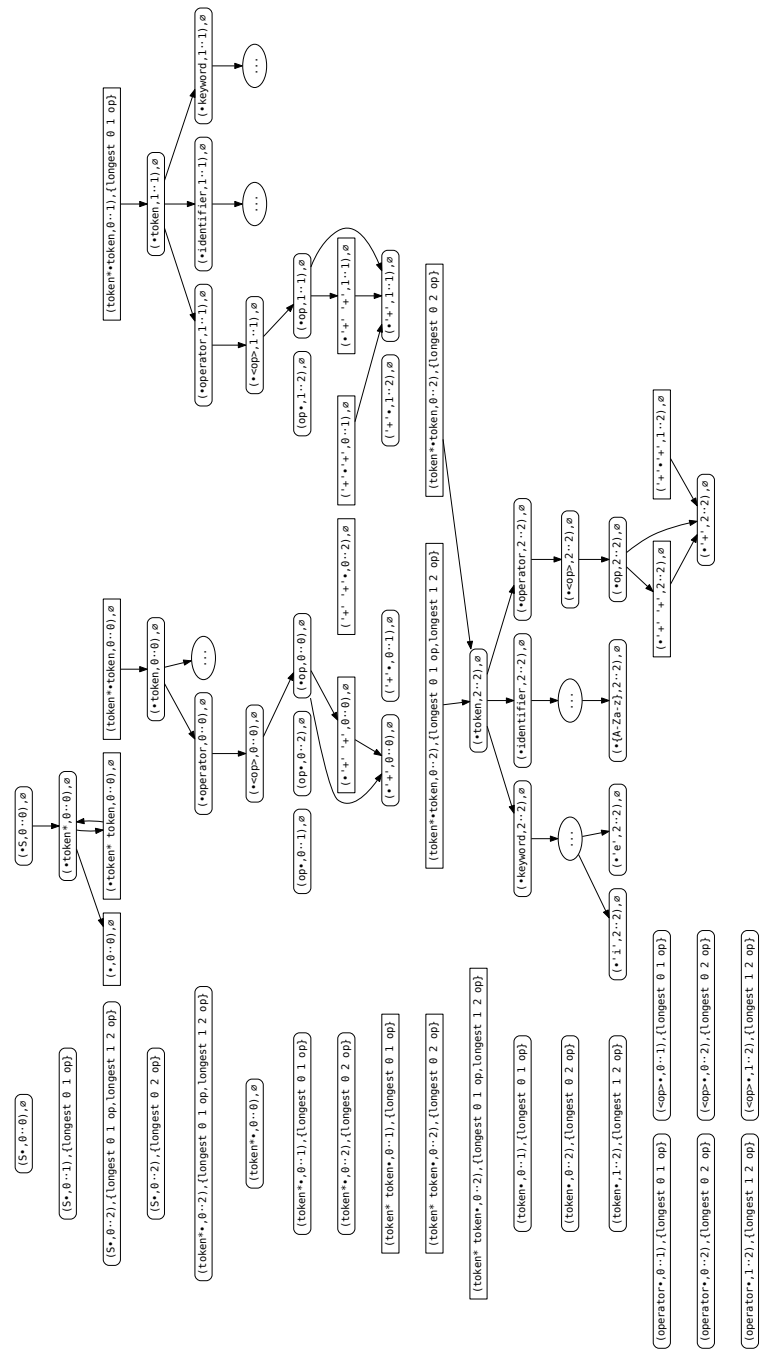

Fig. 14. Parsing context at generation 2

condition of v_8, `longest 0 2 op` is acceptable since there is no longer match of `op` found. Therefore, parser concludes to accept the input string X.

3.4 Correctness

This section sketches the proof of the correctness of the proposed algorithm. We write the algorithm is *correct* if the parser properly accepts or rejects X according to G for a grammar $G = (V_N, V_T, V_C, R, S)$ and a terminal string X.

First, the following lemmas are required to the discussions:

Lemma 3.5.
- *If* $(\alpha, c) \Rightarrow_G^* (\beta, c')$, *then* $c \subseteq c'$.
- *If* $(\alpha, c) \Rightarrow_G^* (\beta, c')$, *then* $(\alpha, c \cup c'') \Rightarrow_G^* (\beta, c' \cup c'')$.
- *For* $X \in V_T^*$, $A, B, C \in (V_N \cup V_T \cup V_C)^*$, $0 \leq p < n$, $\alpha \in (V_T \cup M)^*$, $\beta \in (V_N \cup V_T \cup V_C \cup M)^*$, *and markers of* α *and markers of* β *are disjoint,*
  if $(X\ A\ B\ C, \emptyset) \Rightarrow_G^{L*} (X\ \alpha\ B\ C, c)$ *and* $(X\ \lfloor\alpha\rfloor\ B, \emptyset) \Rightarrow_G^{L*} (X\ \lfloor\alpha\rfloor\ \beta, c')$,
  then we can combine two derivations as $(X\ A\ B\ C, \emptyset) \Rightarrow_G^{L*} (X\ \alpha\ \beta\ C, c \cup c')$.

Lemma 3.6. *If an edge* $(v = ((Z, p, g_0, g_1), d), v' = ((Z', 0, g_1, g_1), \emptyset)) \in E_{g_1}$,
- *If* $Z \in V_N$, *then* $p = 0$, $g_0 = g_1$, $d = \emptyset$, *and* $Z' \in R(Z)$.
- *If* $Z = A\&B$, *then* $p = 0$, $g_0 = g_1$, $d = \emptyset$, *and* $Z' = A$.
- *If* $Z = A\text{-}B$, *then* $p = 0$, $g_0 = g_1$, $d = \emptyset$, *and* $Z' = A$.
- *If* $Z = \$A$, *then* $p = 0$, $g_0 = g_1$, $d = \emptyset$, *and* $Z' = \epsilon$.
- *If* $Z = !A$, *then* $p = 0$, $g_0 = g_1$, $d = \emptyset$, *and* $Z' = \epsilon$.
- *If* $Z =$ <A>, *then* $p = 0$, $g_0 = g_1$, $d = \emptyset$, *and* $Z' = A$.
- *If* Z *is a sequence, then* $|Z| \geq 2$, $p < |Z|$ *and* $Z' = Z_{p+1}$.

Lemma 3.7. *If a node* $v = ((Z, p, g_0, g_1), d) \in V_{g_1}$, *let* $O = \{(v_1, v_2) \in E_{g_1}, v_1 = v\}$,
- *If* $Z \in V_N$ *and* $p = 0$, *then* $O = \{(v, ((Z', 0, g_1, g_1), \emptyset)) | Z' \in R(Z)\}$.
- *If* $Z \in V_C$ *and* $p = 0$, *then* $O = \{(v, ((Z', 0, g_1, g_1), \emptyset))\}$ *where* $Z' = body\ of\ Z$.
- *If* $Z \in (V_N \cup V_C)$, $p = 0$, *and* $(Z, \emptyset) \Rightarrow_G (Z', c)$, *then* $((v, ((Z', 0, g_1, g_1), \emptyset))) \in O$.
- *If* Z *is a sequence, then* $O = \{(v, ((Z_{p+1}, 0, g_1, g_1), \emptyset))\}$.
- $O = \emptyset$ *otherwise.*

Lemma 3.5, 3.6, and 3.7 can be proved by simple induction.

After Recognize consumes all terminals, it finds whether node $((S, 1, 0, |X|), d)$ is in $V_{|X|}$ and finds a node of which accept conditions are all acceptable among them. If $((S, 1, 0, |X|), d) \in V_{|X|}$, it means that parser finds a valid derivation $(S, \emptyset) \Rightarrow_G^* (\alpha, c)$ such that $\lfloor\alpha\rfloor = X$ and $\{indexed((, x) | x \in \alpha\}, c) = d$. Theorem 3.8 proves it.

Theorem 3.8. *If* $((Z, p, g_0, g_1), d) \in V_{g_1}$,
then $(X_{0..g_0} Z_1 \ldots Z_p Z_{p+1} \ldots Z_{\overline{Z}}, \emptyset) \Rightarrow_G^{L*} (X_{0..g_0} \alpha Z_{p+1} \ldots Z_{\overline{Z}}, c)$
such that $\lfloor\alpha\rfloor = X_{g_0..g_1}$ *and* $\{indexed(X_{0..g_0}\alpha, x) | x \in c\} = d$.

Proof. It can be proved by the induction on the nodes of the parsing context.

Base case: If an initial node $v = ((Z, 0, g_0, g_0), \emptyset) \in V_{g_0}$, $(X_{0..g_0} Z_1 \ldots Z_{\overline{Z}}, \emptyset) \Rightarrow_G^{L*} (X_{0..g_0} Z_1 \ldots Z_{\overline{Z}}, \emptyset)$ since $\Rightarrow_G^{L*}$ is a reflexive relation. All nodes created by Recognize procedure and `derive` operations are all initial nodes, therefore the nodes hold the theorem.

Recognize procedure creates `progress` $v = ((Z, p, g_0, g_1), d)$ d' operation when proceeding to generation i if $Z = X_i$, $p = 0$, $g_0 = g_1$, $g_0 + 1 = g_1 + 1 = i$, and $d = d' = \emptyset$. The `progress` creates $((Z, 1, g_0, i), \emptyset)$, and the theorem holds. Only Recognize creates the `progress` operations for terminal symbol nodes.

All non-initial nodes are created by progress operations. Only Recognize can creates `progress` operations for terminal symbol nodes, and this case is covered in the base case. Other `progress` operations are created by final nodes:

Inductive step: `progress` operations for the kernel bodies that are not terminal symbols are created as follows: When proceeding to generation i, for each final node $(v_f = ((Z_f, \overline{Z_f}, g_1, i), d_f)) \in V_i$, the algorithm searches the incoming nodes to $v'_f = ((Z_f, 0, g_1, g_1), \emptyset)$, the initial node of v_f. And for each incoming node $v = ((Z, p, g_0, g_1), d)$, parser creates `progress` v d_f operation. Therefore, if a `progress` v d_f is executed in the proceeding to generation i, then there exists an edge from v to $((Z_f, \overline{Z_f}, g_1, i), d_f)$.

By the induction hypothesis, the final node v_f means that

$$(X_{0..g_1} Z_f, \emptyset) \Rightarrow_G^{L*} (X_{0..g_1}\alpha, c) \text{ such that } \lfloor\alpha\rfloor = X_{g_1..i} \text{ and } \{indexed(X_{0..g_1}\alpha, x) | x \in c\} = d_f \quad (2)$$

Consider each case by the type of the kernel body Z of incoming node v.

If Z is a nonterminal, $p = 0$, $g_0 = g_1$, $d = \emptyset$, and $Z_f \in R(Z)$ by Lemma 3.6. Therefore, $(X_{0..g_0} Z, \emptyset) \Rightarrow_G^L (X_{0..g_0} Z_f, \emptyset) \Rightarrow_G^{L*} (X_{0..g_0}\alpha, c)$ such that $\lfloor\alpha\rfloor = X_{g_0..i}$ and $\{indexed(X_{0..g_0}\alpha, x) | x \in c\} = d_f$ by induction hypothesis (2). The `progress` operation adds a new node $((Z, 1, g_0, i), d_f)$, so the theorem holds.

If Z is an intersection symbol $A\&B$, $p = 0$, $g_0 = g_1$, $d = \emptyset$, and $Z_f = A$ by Lemma 3.6. $(X_{0..g_0} Z, \emptyset) = (X_{0..g_0} A\&B, \emptyset) \Rightarrow_G^L (X_{0..g_0} \#_1 A \#_2, \{\texttt{onlyif } \#_1 \ \#_2 \ B\}) = (X_{0..g_0} \#_1 Z_f \#_2, \{\texttt{onlyif } \#_1 \ \#_2 \ B\}) \Rightarrow_G^{L*} (X_{0..g_0} \#_1 \alpha \#_2, c)$ such that $\lfloor\alpha\rfloor = X_{g_0..i}$ and $\{indexed(X_{0..g_0} \#_1 \alpha \#_2, x) | x \in (c - \{\texttt{onlyif } \#_1 \ \#_2 \ B\})\} = d_f$ by induction hypothesis (2). Also, $indexed(X_{0..g_0} \#_1 \alpha \#_2, \texttt{onlyif } \#_1 \ \#_2 \ B) = \texttt{onlyif } g_0 \ i \ B$. The `progress` operation adds a new node $((Z, 1, g_0, i), d_f \cup \{\texttt{onlyif } g_0 \ i \ B\})$, so the theorem holds.

If Z is exclusion, followed-by, not-followed-by, or longest symbol, the proof is similar to the intersection symbol.

If Z is a sequence symbol $Z_1 Z_2 \dots Z_{|Z|}$, then $|Z| \geq 2$, $Z_f = Z_{p+1}$ by Lemma 3.6. By induction hypothesis on v, $(X_{0..g_0} Z_1 \dots Z_p Z_{p+1} \dots Z_{\overline{Z}}, \emptyset) \Rightarrow_G^{L*} (X_{0..g_0} \alpha Z_{p+1} \dots Z_{\overline{Z}}, c)$ such that $\lfloor\alpha\rfloor = X_{g_0..g_1}$ and $\{indexed(X_{0..g_0}\alpha, x) | x \in c\} = d$. Also, by induction hypothesis (2), $(X_{0..g_1} Z_{p+1}, \emptyset) = (X_{0..g_1} Z_f, \emptyset) \Rightarrow_G^{L*} (X_{0..g_1}\beta, c')$ such that $\lfloor\beta\rfloor = X_{g_1..i}$ and $\{indexed(X_{0..g_1}\alpha, x) | x \in c'\} = d_f$. By combining these two,

$(X_{0..g_0} Z_1 \dots Z_p Z_{p+1} Z_{p+2} \dots Z_{\overline{Z}}, \emptyset) \Rightarrow_G^{L*} (X_{0..g_0} \alpha Z_{p+1} Z_{p+2} \dots Z_{\overline{Z}}, c) =$
$(X_{0..g_0} \alpha Z_f Z_{p+2} \dots Z_{\overline{Z}}, c) \Rightarrow_G^{L*} (X_{0..g_0} \alpha\beta Z_{p+2} \dots Z_{\overline{Z}}, c')$

such that $\lfloor\alpha\rfloor = X_{g_0..g_1}$, $\lfloor\beta\rfloor = X_{g_1..i}$, $\{indexed(X_{0..g_0}\alpha, x) | x \in c\} = d$, $\{indexed(X_{0..g_0}\alpha\beta, x) | x \in c'\} = d \cup d_f$. The `progress` operation adds a new node $((Z, p+1, g_0, i), d \cup d_f)$, so the theorem holds. □

Theorem 3.9. *If a non-final node $v = ((Z, p, g_0, g_1), d) \in V_{g_1}$ and $(X_{0..g_1} Z_{p+1} Z_{p+2} \dots Z_{\overline{Z}}, \emptyset) \Rightarrow_G^{L*} (X_{0..g_1} \alpha Z_{p+2} \dots Z_{\overline{Z}}, c)$ such that $\lfloor\alpha\rfloor = X_{g_1..g_2}$ for some $0 \leq p < \overline{Z}$ and $g_1 \leq g_2 \leq |X|$, then $v' = ((Z, p+1, g_0, g_2), d \cup d') \in V_{g_2}$ where $d' = \{indexed(X_{0..g_1}\alpha, x) | x \in c\}$.*

Proof. It can be proved by induction on the nodes of the parsing context. Consider each case by the type of Z.

Base case: If $Z \in V_T$, Recognize creates `progress` operation for v if $Z = X_i$ when proceeding to generation i. The `progress` operation creates a node $((Z, 1, g_0, i), \emptyset)$, and the theorem holds.

Inductive step:

If $Z \in V_N$, let $A = Z$ and $v = ((A, 0, g_0, g_0), \emptyset) \in V_{g_0}$.

By Lemma 3.7, if $(A, \emptyset) \Rightarrow_G (Q, \emptyset)$, then there exists an edge from v to $v' = ((Q, 0, g_0, g_0), \emptyset) \in E_{g_0}$.

By induction hypothesis on v', if $(X_{0..g_0}Q, \emptyset) \Rightarrow_G^{L*} (X_{0..g_0}\alpha, c)$ such that $\lfloor\alpha\rfloor = X_{g_0..g_1}$ for some $g_1 \geq g_0$, then $v'' = ((Q, \overline{Q}, g_0, g_1), \{indexed(X_{0..g_1}\alpha, x) | x \in c\}) \in V_{g_2}$.

If $(X_{0..g_1}A, \emptyset) \Rightarrow_G^{L} (X_{0..g_1}Q, \emptyset) \Rightarrow_G^{L*} (X_{0..g_1}\alpha, c)$ such that $\lfloor\alpha\rfloor = X_{g_1..g_2}$, the `finish` operation for v'' creates a `progress` operation for v, that adds $((A, 1, g_0, g_1), d) \in V_{g_1}$ where $d = \{indexed(X_{0..g_1}\alpha, x) | x \in c\}$. Therefore, the theorem holds.

If $Z \in V_C$, it can be proved in the similar way to the case where $Z \in V_N$.

If Z is a sequence, let $v = ((Z_1 \dots Z_{\overline{Z}}, p, g_0, g_1), d)$.

By Lemma 3.7, there exists an edge from v to $v' = ((Z_{p+1}, 0, g_1, g_1), \emptyset) \in E_{g_1}$.

By induction hypothesis on v', if $(X_{0..g_1}Z_{p+1}, \emptyset) \Rightarrow_G^{*} (X_{0..g_1}\alpha, c)$ such that $\lfloor\alpha\rfloor = X_{g_1..g_2}$, then $v'' = ((Z_{p+1}, \overline{Z_{p+1}}, g_1, g_2), d')$ where $d' = \{indexed(X_{0..g_1}\alpha, x) | x \in c\}$ for some $g_2 \geq g_1$.

If $(X_{0..g_1}Z_{p+1} \dots Z_{\overline{Z}}, \emptyset) \Rightarrow_G^{L*} (X_{0..g_1}\alpha Z_{p+2} \dots Z_{\overline{Z}}, c)$ such that $\lfloor\alpha\rfloor = X_{g_1..g_2}$, then the `finish` operation for v'' creates a `progress` operation for v, that adds $((Z, p+1, g_0, g_2), d \cup d')$. Therefore, the theorem holds.

□

3.5 Time and Space Complexity

This section discusses the time and space complexity of the proposed algorithm. We will only consider a rough upper bound of complexity. For given grammar G and input string X of length n, we will focus on the correlation of the time and space complexity with the length of the input string n.

We first consider the grammars with no conditional nonterminals, i.e. $V_C = \emptyset$. When parsing such grammars, all nodes of all parsing contexts have empty sets of accept conditions, and we should only consider the possible combinations of the kernels and edges.

The space complexity is proportional to the size of the parsing context, that is the sum of the number of nodes and the number of edges.

In the parsing context at generation i, a node will have a form of $((Z, p, g_0, g_1), \emptyset)$ such that $Z \in (V_N \cup V_T \cup V_C) \cup \{Q | (P, Q) \in R\}$, $0 \leq p \leq \overline{Z}$, and $0 \leq g_0 \leq g_1 \leq i$. Let $z \leq |V_N \cup V_T \cup V_C| + |R|$ and $\sigma \leq 2|V_N \cup V_T \cup V_C| + \sum_{(P,Q)\in R}(|Q| + 1)$. z is the number of possible kernel body and σ is the number of all possible combinations of kernel body and pointer. If we count the number of all possible combinations of g_0 and g_1, it is $(i+1) + (i) + \dots + 2 + 1 = ((i+2)(i+1))/2$. Therefore, the parsing context at generation i has at most $\sigma((i+2)(i+1))/2 \propto i^2$ nodes.

A node at generation i can have at most $\sigma(i+1)$ incoming edges and $\sigma(i+1)$ outgoing edges. For a node $v = (Z, p, g_0, g_1)$, an incoming node to v will have the form of (Z', p', g', g_0) such that $0 \leq g' \leq g_0 \leq i$. Therefore, the rough upper bound of the number of incoming edges to a node is $\sigma(i+1) \propto i$. In the same way, the upper bound of the number of outgoing edges of a node is $\sigma(i+1) \propto i$.

In the parsing context at generation i, there could be $\sigma((i+2)(i+1))/2$ nodes and each node can have at most $2\sigma(i+1)$ edges. Therefore, the upper bound of total number of edges is $(\sigma((i+2)(i+1))/2)(2\sigma(i+1)) = \sigma^2(i+1)^2(i+2)$, and the upper bound of the size of the parsing context at generation i is $(\sigma((i+2)(i+1))/2) + (\sigma^2(i+1)^2(i+2))$, that is $\propto i^3$. As a result, the space complexity of the proposed algorithm is $O(n^3)$ if a grammar with no conditional nonterminal is given.

To calculate the time complexity of the proposed algorithm, we assume that the following tasks can be done in constant times: adding a new node or a new edge to C_i, adding a new pair to U, retrieving updated nodes from U, pushing a new operation to T, checking the existence of a node in C_i, and retrieving the incoming nodes of a node from C_i.

From these assumptions, the execution of a parsing operation is done in constant time even though it could be affected by the complexity or size of the grammar. It is easy to see that `progress` operation is done in constant time, since it simply adds a node to the parsing context and a pair to U. `finish` operation finds the incoming nodes of the given node and creates a progress operation for each incoming node. If we consider that the time to push a progress operation is belonged to the operation being pushed, `finish` is done in constant time. `derive` operation also can be done in constant time. It could be affected by the size of production rules, but we only count the execution time in the matter of the length of the input string.

We now need to calculate the upper bound of the number of operations that can be executed while parsing an input string of length n.

Since a `derive` or `finish` operation is created for each and every node in the parsing context, the number of `derive` and `finish` operations executed to construct the parsing context at generation i is equal to the number of new nodes added in generation i. We already know that the upper bound of the number of nodes in the parsing context of generation i is $\sigma((i+2)(i+1))/2$. Therefore, $((\sigma((i+2)(i+1))/2) - (\sigma((i+1)i)/2)) = \sigma(i+1)$ `derive` or `finish` operations may be required to construct C_i.

A `progress` operation is created when parser consumes a terminal or when a final node is created. The total number of `progress` operations executed to proceed to generation i is the total number of all incoming nodes to the initial nodes of all final nodes that are created in C_i. In addition, one `progress` operation is created when to consume a terminal. The final nodes that are created in C_i have the form of $(Z, \overline{Z}, g_0, i)$ such that $0 \leq g_0 \leq i$. If we count the possible g_0, the number of final nodes in C_i cannot exceed $z(i+1)$. And a node can have at most $\sigma(i+1)$ incoming edges. Therefore, at most $1 + z\sigma(i+1)^2$ `progress` operations may be executed to construct C_i.

To construct the parsing context at generation i, $\sigma(i+1)$ `derive` or `finish` operations and $1 + z\sigma(i+1)^2$ `progress` operations are required at most. By summing up from 0 to n, $\sum_{i=0}^{n}(\sigma(i+1) + z\sigma(i+1)^2) \propto n^3$. Therefore, the proposed algorithm has the time complexity of $O(n^3)$ if a grammar with no conditional nonterminal is given.

The proposed algorithm shares several properties with Earley Parser(Earley 1968). The basic mechanism of the proposed algorithm is almost identical to the Earley parser. Therefore, it is believed that the proposed algorithm will have similar time complexity with Earley parser. Earley parser has time complexity of $O(n^3)$ for an arbitrary CFG, $O(n^2)$ for an unambiguous CFG, $O(n)$ for LR(k) grammars.

Now consider a grammar G with $V_C \neq \emptyset$. We need to calculate the size of the graph again. The possible number of kernel is same with the case where $V_C = \emptyset$, but we need to consider how many possible accept conditions set that a node can have. Given grammar G and input terminal string X, let κ be the number of all possible accept conditions set that a node can have. Then we can get the size of the graph by multiplying κ to the result of the case where $V_C = \emptyset$. Therefore, the upper bound of time complexity will be $\sum_{i=0}^{n}(\sigma\kappa(i+1) + 1 + z\sigma\kappa^2((i+1)^2))$. $\kappa = 1$ if $V_C = \emptyset$.

The upper bound of the number of new indexed accept conditions created while proceeding to generation i is $|V_C|(i+1)$, since an indexed accept condition created when proceeding to generation i has the form of `onlyif` g_0 i A or `followed` i A such that $g_0 \leq i$. Then the upper bound of total number of indexed accept conditions that can exist in generation n is $\sum_{i=0}^{n}|V_C|(i+1) = (|V_C|(n+1)(n+2))/2$. Suppose a node can have any combination of the indexed accept conditions, then κ becomes $2^{(|V_C|(n+1)(n+2))/2}$. Then the time complexity becomes exponential to the length of the input string. However, it is an extremely rare or an impossible case. It only shows that the algorithm terminates for any grammar. It requires further researches on the time and space complexity of the proposed algorithm.

We can add an optimization to the algorithm. After each proceed is done, an indexed accept condition may become impossible to be acceptable no matter what string follows. For example, an accept condition `onlyif 2 5` A cannot be acceptable if there is no node of which kernel is $(A, 1, 2, 5)$ in V_5. We can remove the nodes with such accept conditions to reduce the size of the parsing context. Also, an indexed accept condition may become impossible to be unacceptable. Such accept conditions can be removed from the accept conditions sets of nodes. This optimization can improve the time complexity of the algorithm.

3.6 Implementation

(jpa 2017) is the implementation of the proposed parsing algorithm written in Scala Programming Language. It includes the parsing algorithm, parse tree reconstruction algorithm, and graphical user interfaces. The parse tree reconstruction algorithm was designed based on the meaning of the nodes as discussed in Section 3.4. GUI provides the menus to try CDGs and visualize the parsing contexts and parse trees.

An optimization technique is also implemented; The implementation trims unnecessary nodes from the parsing context after it proceeds. Unnecessary nodes include the ones that are not reachable to the new terminal nodes and ones that have an accept condition that are proved to be unacceptable no matter what string follows. It helps to reduce the size of the parsing context and improve the efficiency of the algorithm.

Preprocessing the grammar will also help to improve the efficiency of the algorithm. For example, in Figure 9, if $((\bullet\texttt{'0'}, 0..0), \emptyset)$ is progressed and finished, it eventually propagates to the progress operation of $((\bullet\texttt{expression}, 0..0), \emptyset)$. Such trivial chains of operations by preprocessing the grammar can be removed algorithmically.

4 CONCLUSION

In this paper, we proposed Conditional Derivation Grammar(CDG). CDG has similar format with context-free grammar, but offers new features such as intersection, exclusion, followed-by, not-followed-by, longest match. We expect the syntaxes of programming languages in lexical and hierarchical grammars to be rewritten in a single CDG.

This paper also proposes a parsing algorithm for CDGs. We have proved that the algorithm is correct and always terminates. The time and space complexity of the algorithm varies by the characteristics of the given grammar, but further studies are required.

REFERENCES

2017. J Parser. https://github.com/cdgrammar/jparser. (2017).

Jay Earley. 1968. *An efficient context-free parsing algorithm.* Ph.D. Dissertation. Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 111–122.

Dick Grune and Ceriel JH Jacobs. 2007. *Parsing techniques: a practical guide.* Springer Science & Business Media.

ISO/IEC 14882: 2011(E) 2011. *ISO/IEC 14882: 2011, Information technology – Programming languages – C++.* Standard. International Organization for Standardization.

ISO/IEC 30170: 2012(E) 2012. *ISO/IEC 30170: 2012, Information technology – Programming languages – Ruby.* Standard. International Organization for Standardization.