【総説解説論文】

プログラミング言語のコンパイラの構文解析

Parsing theory of compilers of programming languages

山根　智
Satoshi Yamane[1]

[1]下関市立大学
Shimonoseki City University

要旨

私たちの世界を動かすプログラムは事実上すべて，人間が Python や Java，C などの高級プログラミング言語で記述し，それをコンパイルして低レベルのコードにまとめて実行する．現代のプログラミング言語のためのコンパイルを行う技術，すなわち，コンパイラの多くは，Alfred V. Aho と Jeffrey D. Ullman の貢献が大きく，それにより 2020 年度に，コンピュータ科学の最高峰のチューリング賞を受賞した [1]．コンパイラでは字句解析，構文解析，コード生成などのための技術やアルゴリズムが重要である． 本解説記事では，コンパイラの構造及び，コンパイラで最重要で最難関な構文解析，特に LR 構文解析 ［7］ について，類書にない構文解析手法の関係を明確化して，有効な事例を新規に追加して，解説を行う．これにより，現在と未来の社会を支えるコンピュータ・ソフトウェアの理解を深めることができる．さらに重要なことは，構文解析の理論，技術，実装は他の多くのソフトウェアの原理となっており，今後のソフトウェアの作成に大いに有益である．なお，本解説記事はドラゴンブック ［2］ や優れた教科書 ［3,4,5,6］ などを参考にしている．

キーワード：　プログラミング言語，コンパイラ，構文解析，LR 構文解析

Abstract

All programs that run our world are written by humans in high-level programming languages such as Python, Java, and C, which are then compiled into low-level code and executed. Much of the technology for compiling for modern programming languages, i.e., compilers, is due to the contributions of Alfred V. Aho and Jeffrey D. Ullman, which earned them the Turing Prize, the pinnacle of computer science, in FY2020 [1]. Techniques and algorithms for lexical analysis, parsing, and code generation are

important for compilers. This article describes the structure of compilers and the most important and most difficult part of compilers, parsing, especially LR parsing [7] , by clarifying the relationship between parsing methods and adding new effective examples not found in other books. This will deepen your understanding of computer software that supports present and future society. More importantly, the theory, techniques, and implementation of parsing are the principles of many other software, and will be of great benefit in the development of future software. This article is based on Dragon Book [2] and excellent textbooks [3,4,5,6].

Keywords: programming language, compiler, parser, LR parser

１.はじめに

コンピュータ・ソフトウェアは,私たちが関わるほとんどすべての技術に力を与える.携帯電話や自動車で動作するものから，大手ウェブ企業内の巨大サーバーファームで動作するものまで,私たちの世界を動かすプログラムは事実上すべて，人間が Python や Java，C などの高級プログラミング言語で記述し，それをコンパイルして低レベルのコードにまとめて実行する．現代のプログラミング言語のためのコンパイルを行う技術，すなわち，コンパイラの多くは，Alfred V. Aho と Ullman D. Jeffrey の貢献が大きく，それにより 2020 年度に，コンピュータ科学の最高峰のチューリング賞を受賞した［1].

また，Alfred V. Aho と Jeffrey D. Ullman らの共著『Compilers principles, techniques & tools』はコンパイラ技術に関する決定的な本であり，形式言語理論と構文指向翻訳技術をコンパイラ設計プロセスに統合したものである［2]．カバーデザインからしばしば「ドラゴンブック」と呼ばれており，高レベルのプログラミング言語をマシンコードに変換する段階を明晰に示し，コンパイラ構築の全プロセスをモジュール化している．この書籍には，字句解析，構文解析，コード生成のための効率的な技術に対して，彼らが行ったアルゴリズム的な貢献も含まれている.ドラゴンブックの最新版は 2007 年に出版され，コンパイラ設計に関する標準的な教科書として世界で広く読まれており，日本語の翻訳もあり，大学や大学院の講義や研究において有効活用されている.

本解説記事では，コンパイラの構造及び，コンパイラで最重要で最難関な構文解析について，類書にない構文解析手法の間の関連および新たな事例を盛り込んで解説を行う．これにより，現在と未来の社会を支えるコンピュータ・ソフトウェアのコンパイラの構文解析手法の理解を深めることができる．さらに重要なことは，構文解析の理論，技術，実装は他の多くのソフトウェアの原理となっており，今後のソフトウェアの作成に大いに有益である．なお，ドラゴンブックを手本に，日本語の優れた教科書も多数あり、その代表的な書籍［3,4,5,6］を参考文献にあげており，本解説記事の作成にも参考にしている.

２.コンパイラの位置づけと構造

最初に，原始プログラムを読み込んで，機械コードを出力するまでの典型的な流れを図１

に示す．以下では，ドラゴンブック［2］に従って簡単に概要を説明する．

まず，コンパイラは高級プログラミング言語で書かれた原始プログラムを読み込んで，必要に応じて前処理を行い，コンパイル可能な原始プログラムを出力する．なお，前処理では，別のファイルに格納している原始プログラムやライブラリなどを取り込んで原始プログラムに展開して，コンパイル可能な原始プログラムを作る．次に，その原始プログラムをコンパイラが読み込んで，ターゲットマシンに対応したアセンブリプログラムに変換する．次に，アセンブリプログラムをアセンブラに入力して，再配置可能な機械コードを出力する．最後に，リンカまたはローダに再配置可能な機械コードを入力して，リンカまたはローダはライブラリファイルや再配置可能目的ファイルを取り込んで，最終的に実行可能な機械コードを出力する．

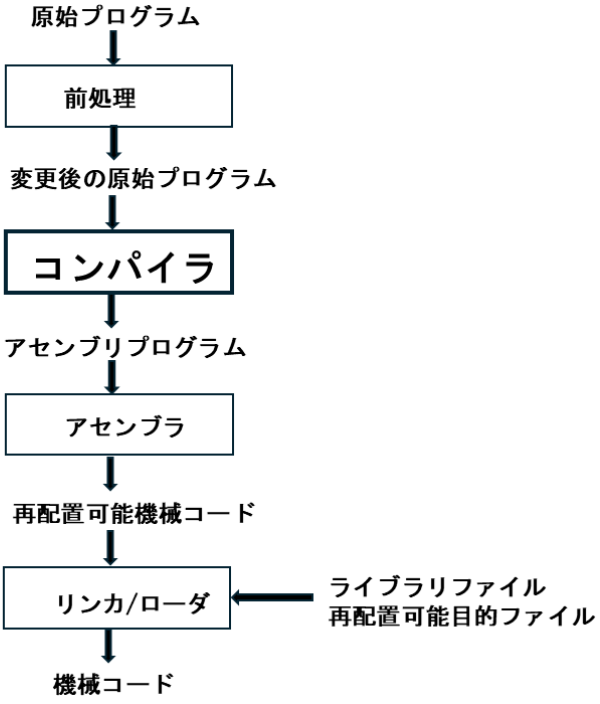

図１　コンパイラの位置づけ

次に，コンパイラの典型的な内部の構造を図２に示す．以下では，ドラゴンブック［2］に従って簡単に概要を説明する．コンパイラの内部の構造は，大まかには，解析と合成の２つの部分に分かれている．

(1)解析部は原始プログラムを構成要素に分解し，それらに文法的な構造を与える．この構造を利用して原始プログラムの中間コードを作り出す．また，原始プログラムのデータなどの情報を収集し，記号表に格納する．なお，解析部は字句解析，構文解析，意味解析，中間コード生成からなる．

① 字句解析は原始プログラムの文字ストリーム(文字列)から，名前やキーワードなどのプログラムとして意味のある最小単位のトークンストリーム(トークンの列)を作り出す．

② 構文解析はプログラミング言語の文法に従って，トークンストリームから構文木を作る．

③ 意味解析は型の整合性などをチェックする.

④ 中間コード生成は構文木からアセンブリの命令に近い中間コードを生成する.コンパイラによっては，中間コードを経ずに，直接にアセンブリプログラムを生成して，その後，最適化するものもある.

(2)合成部は中間コードと記号表の情報からアセンブリプログラムを作り出す.なお,合成部は中間コードの最適化とコード生成からなる.

① 中間コードの最適化は中間コードの合成において無駄な中間コードを削除する.並列処理などのハードウェアの進化により，多数の研究がなされている.

② コード生成は中間コードからアセンブリプログラムを生成する.

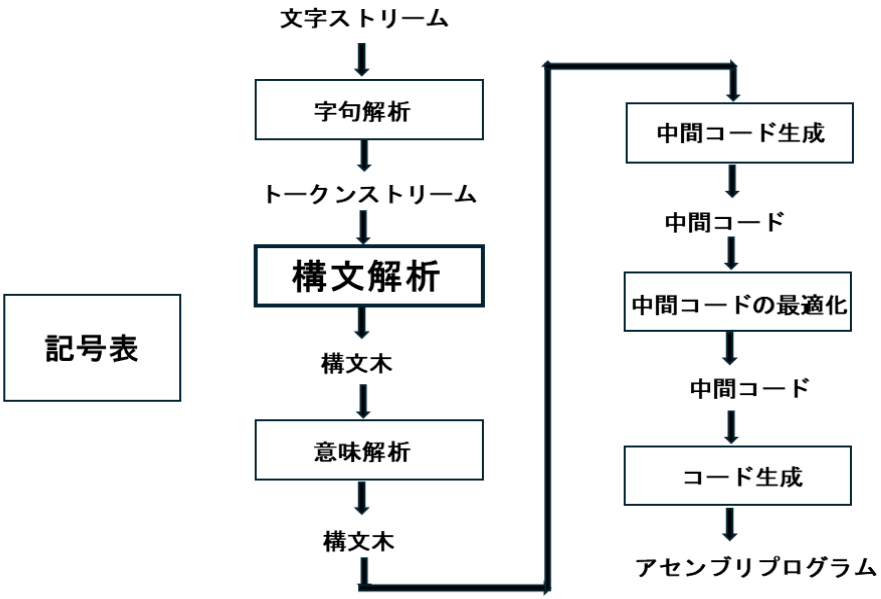

図２　コンパイラの構造

３.プログラミング言語の文法

構文解析の説明の前に，ドラゴンブック［2］や和文教科書［5］に従い，プログラミング言語の文法について説明する.

まず,構文解析するためには,プログラミング言語の文法を数学的に定義する必要がある.プログラミング言語は文法的には文脈自由文法であり，以下の４つ組$\langle V_N, V_T, P, S\rangle$で定義する.

（定義）文脈自由文法は４つ組$\langle V_N, V_T, P, S\rangle$である.

ここで，各記号は以下である：

非終端記号(Non-terminal Variable)の集合　V_N

終端記号(Terminal Variable)の集合 V_T

生成規則(Production Rule)の集合　P

開始記号(Start Symbole) $S \in V_N$

ただし，$V_N \cap V_T = \{\}$

また，生成規則は”$A \rightarrow \alpha$”$\in P$ であり，左辺は一つの終端記号 $A \in V_N$，右辺は $\alpha \in (V_N \cup V_T)^*$である．ただし，生成規則の集合 P は S を左辺とする生成規則を必ず含む．

なお，左辺が同じで右辺が異なる２つ以上の生成規則を以下のように書く．

$A \rightarrow \alpha_1, \cdots, A \rightarrow \alpha_n$ は $A \rightarrow \alpha_1 | \cdots | \alpha_n$ とまとめて書く．

非終端記号 V_N は変数のようなものであり，終端記号 V_T はプログラムに出現する記号である．重要なことは，生成規則の集合により文法を定義するものであり，生成規則は左辺の非終端記号から右辺の非終端記号または終端記号の連結された文字列に書き換える規則である．

次に，プログラミング言語に現れる典型的な文法の一部(多くの教科書で出現する文法)を文脈自由文法で記述してみる．

$G=\langle V_N, V_T, P, E\rangle$の各組は以下である．

$V_N=\{E\}$

$V_T=\{+, *, (,), i\}$

$P=\{E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow i\}$

開始記号 $E \in V_N$

次に,文法 G から i+i*i に対する構文木を作るが,図３のように２つの構文木が存在する．ここで，⇒は生成規則を１回適用して左辺から右辺が得られることを意味する．

① E ⇒ E + E ⇒ E + E * E ⇒ E + E * i ⇒ E + i * i ⇒ i + i * i

② E ⇒ E * E ⇒ E * i ⇒ E + E * i ⇒ E + i * i ⇒ i + i * i

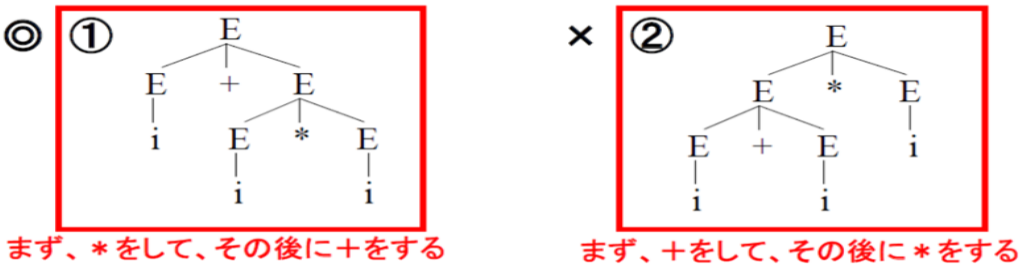

図３　i+i*i に対する２つの構文木

この文法は２つの構文木が生成でき，曖昧性を持つ文法である．掛け算が足し算よりも優先度が高いという演算子の優先順位を考慮すると，①が正しい構文木である．文法の曖昧性を除去するために，演算子の優先順位と結合性を生成規則に埋め込む必要がある．その結果，以下の文法が作れ，この文法がプログラム言語の文法として採用されており，以降の章でもこの文法 G’を用いる．

$G'=\langle V_N, V_T, P, E\rangle$の各組は以下である．

V_N={E, T, F}

V_T={+, ＊ (,), i}

P={E→E+T

E→T

T→T*F

T→F

F→(E)

F→i}

開始記号 E∈V_N

４.構文解析の概要

構文解析はプログラミング言語の文法に従って，原始プログラムのトークンの列から構文木を作るものである.構文解析の手法として,代表的なものが２つあり,図４に示すように,根からトップダウンに構文木を作る手法（再帰的下向き構文解析手法）と葉からボトムアップに構文木を作る手法（LR 構文解析手法）がある [2].

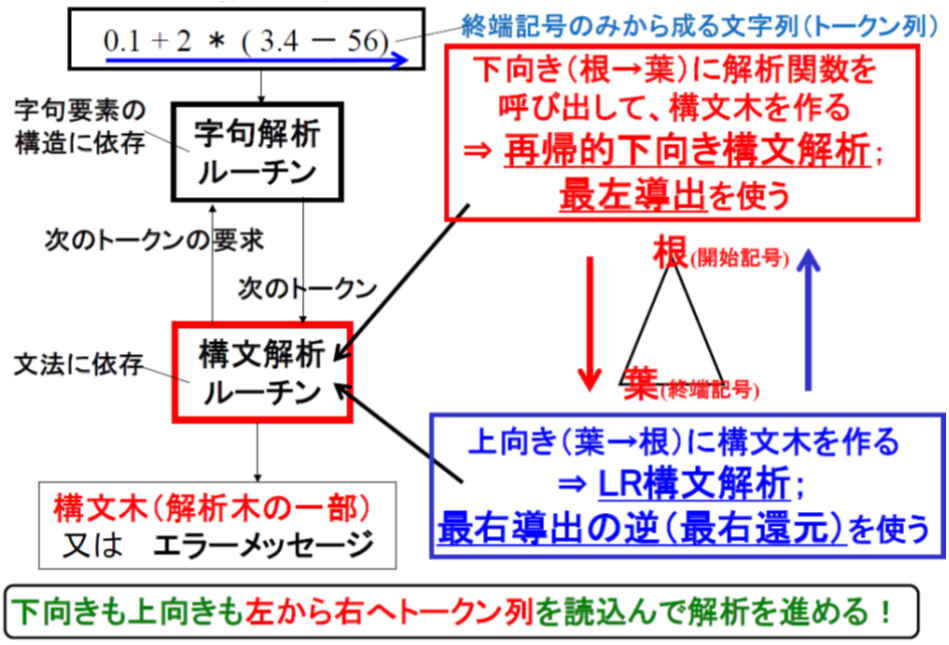

図４　代表的な２つの構文解析手法

再帰的下向き構文解析手法は最左導出を繰り返して，根から葉に向かって構文木を作る．一方，LR 構文解析手法は最右導出の逆，つまり最右還元を繰り返して，葉から根に向かって構文木を作る．２つの構文解析手法ともにトークンの列を左から右へバックトラックなしに一度読み込んで解析して構文木を作る．また，再帰的下向き構文解析器を作るには，生成規則毎に解析関数を手作り，再帰的に解析関数を呼び出して構文木を作る必要がある．一方，LR 構文解析器は文法から自動的に構文解析プログラム生成器できて，構文解析可能な文法クラ

スも広く，実用上から多用されている．

５．再帰的下向き構文解析手法

５．１　基本的考え方

この章では，３章で導入した以下の文法を対象とする．

G’=<V_N, V_T, P, E>の各組は以下である．

V_N={E, T, F}

V_T={+, ＊ (,), i}

P={E→E+T, E→T, T→T*F, T→F, F→(E), F→i}

開始記号 E∈V_N

まず，再帰的下向き構文解析の考え方に従い，図５に示すように，トークン列 i*(i+i)から，最左導出により，木の根から葉に向けゴールまでの構文木を生成する．ここで，図６に示すように，無限ループとバックトラッキングが発生する．G’は左再帰性のある文法（例えば，E→E+T）なので，E の解析関数は E から E+T の構文木を作るが，トークンへのポインタが前進しないで無限に呼び出され，無限ループを発生させる．一方，バックトラッキングが発生すると，構文木の作成時間が大きくなったり，プログラムが複雑になるので，実用上から大きな問題となる．

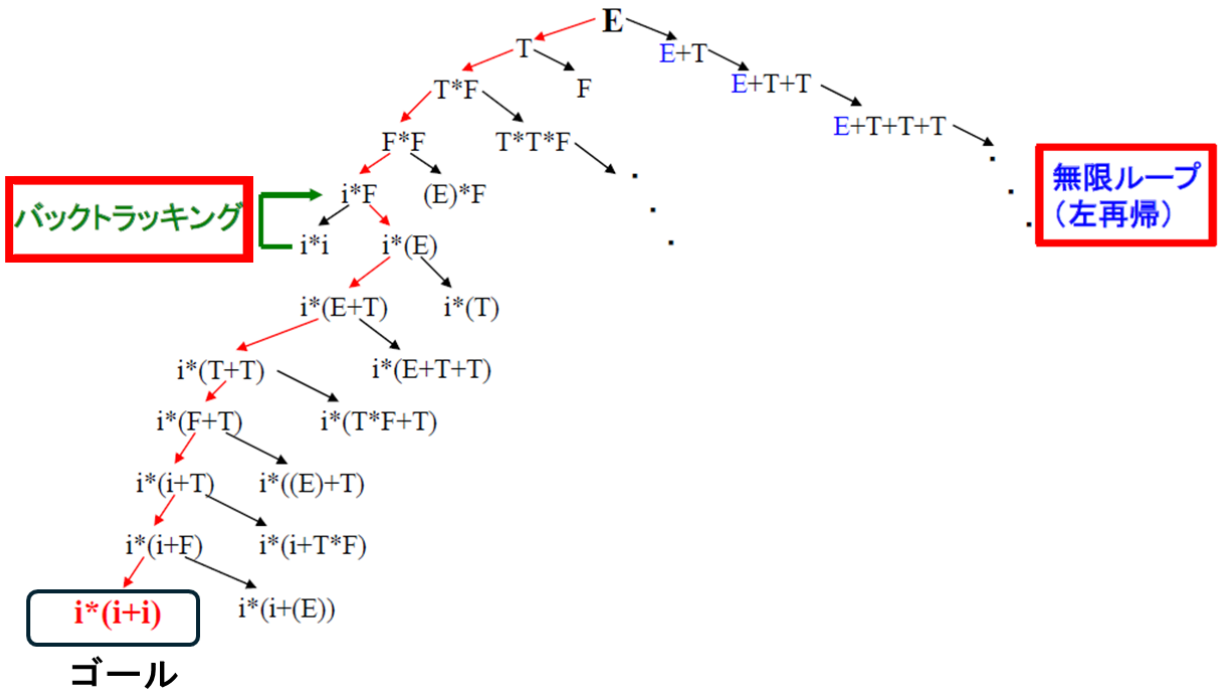

図５　再帰的下向き構文解析手法の流れ

再帰的下向き構文解析では，無限ループを回避するために，左再帰性のある文法を右再帰性の文法に変換して構文解析を行い，左再帰性の文法の構文木を生成する．一方，バックトラッキングを回避するために，トークンを先読みすると，呼び出すべき解析関数がユニーク

に決まるような文法を構文解析の対象とする．再帰的下向き構文解析の流れを図６に示す．

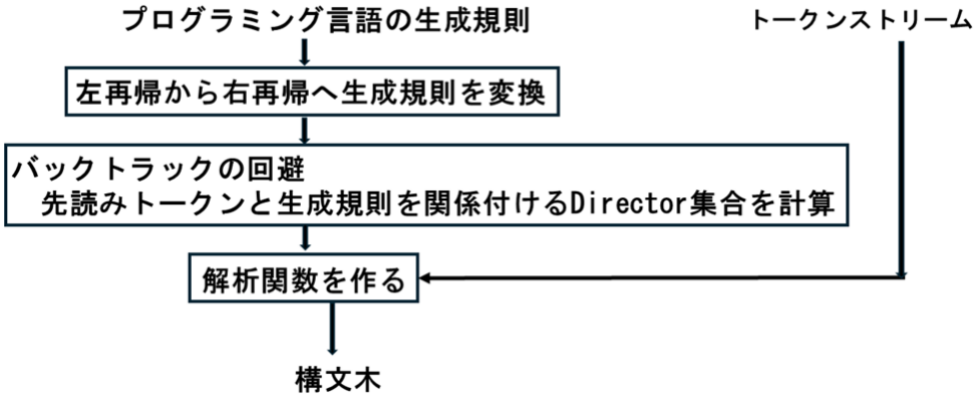

図６　再帰的下向き構文解析手法の流れ

５．２　左再帰性から右再帰性への変換

G’は左再帰性のある文法（例えば，E→E+T）なので，E の解析関数はトークンへのポインタが前進しないで無限に呼び出され,無限ループを発生させる.再帰的下向き構文解析では，無限ループを回避するために，左再帰性のある文法を右再帰性の文法に変換する．

例えば，左再帰性を持つ次の単純な生成規則を考える．

A→Aa|b

なお，A は非終端記号，a と b は終端記号であり，b は A 以外の記号とする．A から最左導出により，$ba^n(n\geqq 0)$が導出できる．ここで，新しい非終端記号 A’を導入して，右再帰性を持つ次の生成規則を作る．

A→bA’

A’→Aa’｜ε

最右導出を用いると，A→Aa|b と同じ終端記号列 $ba^n(n\geqq 0)$が導出できる．以上より，左再帰性を有する文法の構文解析は右再帰性の文法に変換して構文解析すればよいことがわかる．

左再帰性のある文法 G’=<V_N, V_T, P, E>から左再帰性を除去して，

V_N={E, T, F}

V_T={+, * (,), i}

P={E→E+T, E→T, T→T*F, T→F, F→(E), F→i}

開始記号 E∈V_N

以下の文法 G’’ =<V_N’, V_T, P’, E>が得られる．

V_N’={E, E’, T, T’, F}

V_T={+, * (,), i}

P’={E→TE’, E’→+TE’｜ε, T→FT’, T’→*FT|ε, F→(E), F→i}

開始記号 E∈V_N

５．３　Director 集合によるバックトラックの回避

再帰的下向き構文解析では，開始記号から最左導出により，生成規則毎に解析関数を実装

して，それらを再帰的に呼び出して，構文木を完成させる．このときに，ある非終端記号の生成規則が複数存在するときに，k 個のトークンを先読みし，どの生成規則，つまり，どの解析関数を呼び出せばよいかを決定する方法を用いることである．これが LL(k)構文解析法である．ここで，LL とは，Left to right scanning（左から右へトークンストリームをスキャンする）と Leftmost derivation（最左導出）である．以下では，最も簡単で実用上重要な LL(1)を説明する．以降では，ドラゴンブック ［2］ や和文教科書 ［4,5］ などに従い，Director 集合の計算方法を定義する．

Director 集合を計算するためには，まずは First 集合と Follow 集合を計算する必要がある．以降では，一般のプログラミング言語の文法 $G=\langle V_N, V_T, P, S\rangle$とその記号列 $\alpha \in (V_N \cup V_T)^*$に対して，αから生成される記号列の先頭に現れる終端記号の全体をαの First 集合と呼び，First(α)と表記する．非終端記号 A に対して，文法 G の文形式において A の直後に現れる可能性のある$を含む終端記号の全体を A の Follow 集合と呼び，Follow(A)と表記する．以下の図７に形式的定義と概念図を示す．ここで，⇒*は生成規則を０回以上適用して，左辺から右辺が得られることを表す．なお，$はトークン列の最後尾の記号であり，開始記号 S の Follow 集合には$が必ず含まれる．

$First(\alpha)=\{a \mid a\in V_T,\ \alpha \overset{*}{\Rightarrow} a\cdots\}$　　$Follow(A)=\{a \mid a\in V_T,\ S\$ \overset{*}{\Rightarrow} \cdots Aa\cdots\}$

図７　First 集合と Follow 集合

以上の First 集合と Follow 集合を用いて，図８で Director 集合を定義する．生成規則 A→αに対して，終端記号の集合 Director(A,α)を以下のように定義し，A→αの Director 集合は生成規則 A→α を使った場合に，入力トークン列の先頭に現れる可能性のある終端記号の集合である．ここで，αからεが生成される場合には，Follow(A)を使う必要がある．なお，First 集合や Follow 集合,Director 集合の計算方法は書籍 ［2,3,4,5］ に詳しく書いており，紙面の都合上から本解説では省略する．

生成規則A→αに対して，終端記号の集合Director(A, α)を以下のように定義し，
A→αのdirector集合と呼ぶ．

$$\mathrm{Director}(A,\alpha)=\begin{cases}\mathrm{First}(\alpha) & (\alpha \overset{*}{\Rightarrow} \varepsilon \text{ でないとき})\\ \mathrm{First}(\alpha)\cup\mathrm{Follow}(A) & (\alpha \overset{*}{\Rightarrow} \varepsilon \text{ のとき})\end{cases}$$

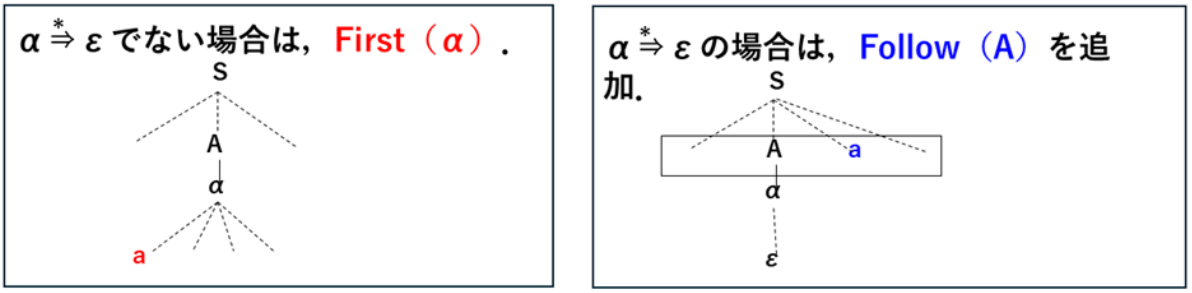

図８　Director 集合

５．４　再帰的下向き構文解析器の実現

Director 集合を計算するには，事前に計算した Null 関数値や First 集合，Follow 集合が必要である．なお，Null 関数値は生成規則の右辺に出現する記号列 $\alpha \in (V_N \cup V_T)^*$ から ε が最左導出されるかを表す関数である．図９に示すように，５．２で左再帰性文法から右再帰性文法へ変換した生成規則の集合 P に対して，事前に計算した Null 関数値や First 集合，Follow 集合を用いて，Director 集合を計算した結果を示す．

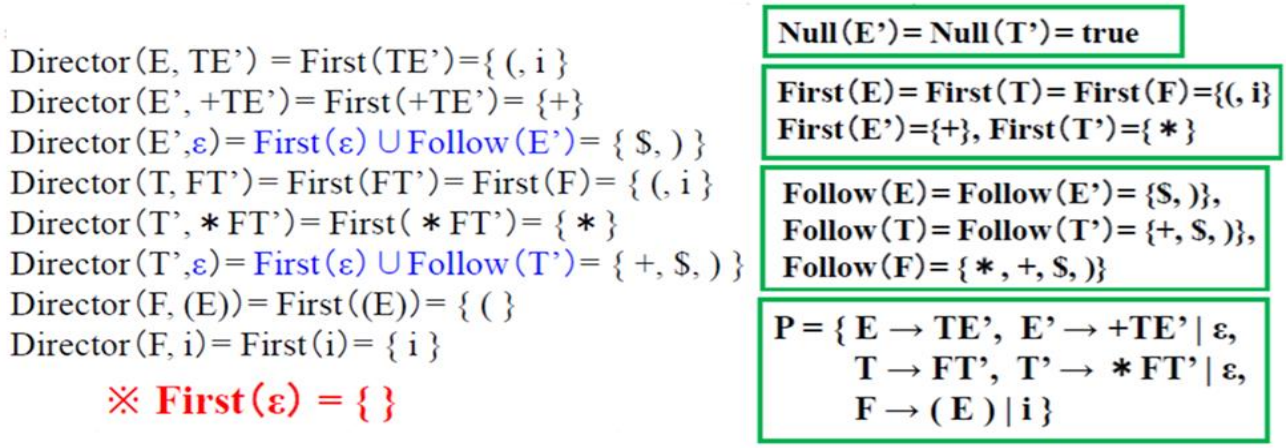

図９　Director 集合の計算

この Director 集合を用いて，この文法が LL(1)かどうかを判定する．左辺が同じで右辺のみが異なる，生成規則の部分集合{E'→+TE',E'→ε}，{T'→*FT'，T'→ε}，{F→(E)，F→i}に対して，

Director(E',+TE')∩Director(E',ε)={}

Director(T',*FT')∩Director(T',ε)={}

Director(F,(E))∩Director(F,i)={}

なので，この文法は LL(1)である．つまり，１つのトークンを先読みすれば，最左導出の対

象となる非終端記号の生成規則を決定できる．

次に，再帰的下向き構文解析により，LL(1)文法の入力トークン列 w=i*i+i$からの構文木の作成の例を図１０に示す．

(1)開始記号 E から下向きに構文木を作成する．

(2)左辺を E とする生成規則は E→TE’のみであり，トークンの先頭は i である．

(3)Director(E,TE’)={(,i)}であり，i∈Director(E,TE’)={(,i)}なので，①の構文木を作成する．

(4)最左導出の対象は T であり，T を左辺とする生成規則は T→FT’のみである．解析対象のトークンは依然，i である．理由は，まだ先頭のトークンの i が構文木になっていないからである．

(5)i∈Director(T,FT’)={(,i}なので，T の子供に②の木を追加する．

(6)最左導出の対象は F であり，F を左辺とする生成規則は F→(E)と F→i の２つある．解析対象のトークンは依然 i であり，i∈Director(F,i)，Director(F,(E))は i を含まない．よって，F→i を用いて構文木を作り，③ができる．

(7)最左導出の対象は T’であり，解析対象のトークンは*である．T’を左辺とする生成規則は T’→*FT’のみであり，*∈Director(T’,*FT’)なので，④の構文木を作る．

以上を続けて，①，②，…，⑪の順番に構文木を作り，最終的に，図１１の構文木ができ，構文解析が終了する．

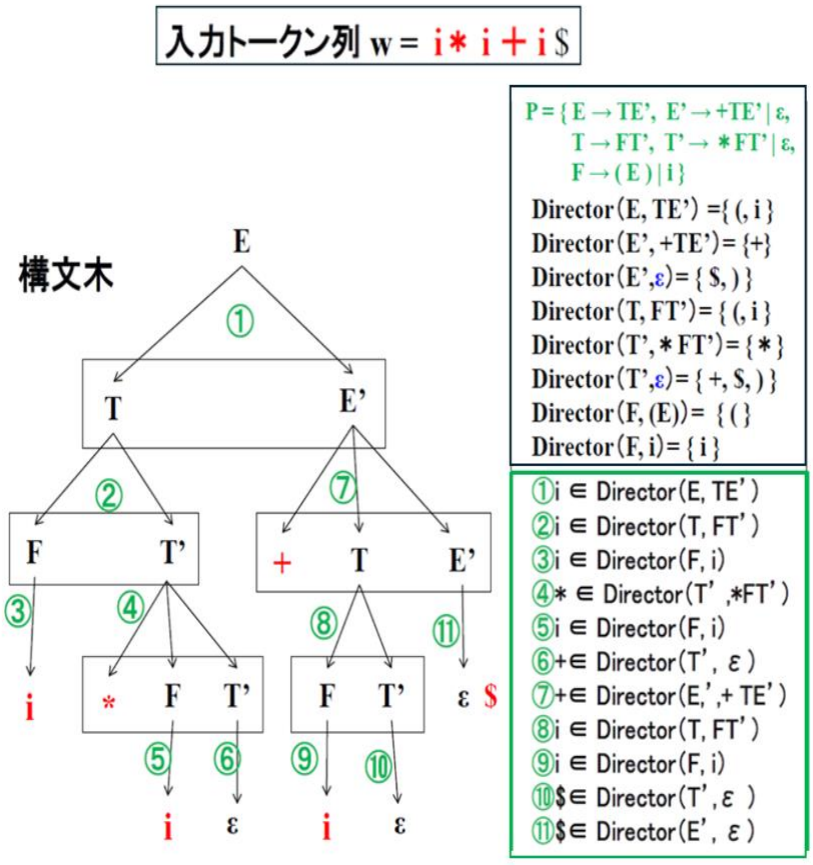

図１０　再帰的下向き構文解析による LL(1)文法の構文木の作成例

６.LR 構文解析手法

６．１　基本的考え方

LR 構文解析手法は最右導出の逆，つまり最右還元を繰り返して，葉から根に向かって構文木を作る．再帰的下向き構文解析手法と同様に，LR 構文解析手法もトークンの列を左から右へバックトラックなしに一度読み込んで解析して構文木を作る．これらの様子はすでに図５に示している．LR 構文解析手法の最大の特徴は文法から自動的に構文解析プログラムが生成できて，構文解析可能な文法クラスも広く，実用上から多用されている．さらに，重要なことは，最右導出の逆，つまり最右還元を用いるために，左再帰性のある文法でも問題なく解析でき，文法をそのまま構文解析できる．なお，LR 構文解析の名前は，Left-to-right-scanning（トークンの列を左から右へバックトラックなしに一度読み込んで構文解析を行うことを意味している)，Right-most-derivation in reverse（最右導出の逆の順序，つまり最右還元で構文解析を行うことを意味している）の頭文字をとったものである．以降では，ドラゴンブック［2］や和文教科書［3,4,5,6］に従い，解説する．

６．２　最も基本的な LR(0)構文解析手法

まず，構文解析の途中の状態を考える．たとえば

A→xyz

という生成規則で作られたトークン列を構文解析して構文木を作っている途中で，x を読み終わった状態で，y を読もうとしている状態であるとする．その状態を

A→x・yz

と書く．ここで，y が終端記号であれば y を読もうとしている状態であり，y が非終端記号 B であり，

$B \rightarrow x_1 y_1 z_1$

という生成規則があれば，x_1 を読もうとしている状態

$B \rightarrow \cdot x_1 y_1 z_1$

でもある．以上より，LR(0)項とその集合の閉包を導入する．以降においては，５章で導入した文法 G’=<V_N, V_T, P, E>を用いる．

ここで，V_N={E, T, F}

V_T={+, ＊ (,), i}

P={E→E+T, E→T, T→T*F, T→F, F→(E), F→i}

開始記号 E∈V_N

(定義)文法 G’の LR(0)項とは，P の任意の生成規則に対して，その右辺の左端，右端および記号の間のどれか一箇所に”・”をつけたものである．

例えば，生成規則 E→E+T に対して，以下の４つの LR(0)項が存在する．

E→・E+T　　E→E・+T　　E→E+・T　　E→E+T・

E→・E+T のように，右辺の先頭にドットがある LR(0)項を導入項と呼ぶ．また，E→E+T・の

ように，ドットが末尾にあるLR(0)項を完全項と呼ぶ．

また，先ほどの例で示したように，A→x・yzにおいて，yが非終端記号Bであり，B→$x_1y_1z_1$という生成規則があれば，x_1を読もうとしている状態B→・$x_1y_1z_1$でもあるので，LR(0)項の集合として{A→x・Bz， B→・$x_1y_1z_1$}を考える必要がある．ゆえに，LR(0)項の集合Ｉに対して，Closure(I)を考える．

（定義）文法G’のLR(0)項の集合Ｉの閉包Closure(I)とは次のように得られる．

1. Closure(I):=I
2. A→α・Bβ∈Closure(I)に対して，B→γという生成規則があればB→・γをClosure(I)に加える。
3. 上記2.を新たに加わるものがなくなるまで繰り返す．

例えば，I={E→E+・T}のClosure(I)は{ E→E+・T， T→・T*F， T→・F， F→・(E)， F→・i}となる．

また，一般に，LR(0)項の集合Ｉから記号Ｘによって遷移する先をGOTO(I,X)と書いて，以下に定義する．

（定義）ＩをLR(0)項の集合，X∈(V_N∪V_T)=Vとしたとき

GOTO(I,X)=Closure({A→αX・β｜ A→α・Xβ∈I})

例えば，I={T→T・*F}のとき，GOTO(I,*)は以下になる．

GOTO({T→T・*F},*)

=Closure({T→T*・F})

={ T→T*・F， F→・(E)， F→・i}

文法G’=<V_N， V_T， P， E>において，このようなLR(0)項の集合を求めるには，まず，構文解析の終了判定を容易にするために，新たな非終端記号E’を追加しE’→Eなる生成規則を文法G’の生成規則の集合Pに追加して，以下のアルゴリズムを実行する．

（アルゴリズム）

初期設定　C:={Closure({E’→・E })}

以下を新たな集合がCに付け加えられなくなるまで繰り返す：

任意のI∈CとX∈Vに対し，GOTO(I,X)が空集合でなく，Cに入っていなかったらCに加える

文法G’に適用すると，ドラゴンブック［2］や和文教科書［3][5］のオートマトンと同様な図１１のオートマトン得られる．このオートマトンを正準オートマトンと呼ぶ．特に，ここではLR(0)を用いているのでLR(0)オートマトンとも呼ぶ．なお，状態の中の完全項は青色，構文解析の終了を意味するE’→E・は赤字で示す．この正準オートマトンを用いて，与えられた入力トークン列の構文解析を進める．

一般に，入力トークン列ｘが与えられると，最初に・x$から出発し，正準オートマトンを用いて，構文木を作りながら，構文解析中ではu・v$の形になり，時点と呼ぶ．なお，u∈($V_N$∪$V_T$)*，v∈$V_T$*である．ここで，uは読み込み済みのトークン列から構成した部分的な構文木であり，vはまだ読み込んでいないトークン列である．最終的に，開始記号Ｓの時点Ｓ・$になれば，入力トークン列ｘの構文木ができて，構文解析は正常に終わる．構文解析中では，正準オートマトンの状態遷移に従い，以下のいずれかを行う．

(1)還元(reduce)：正準オートマトンのある状態の中の LR(0)項 A→α・を使って，時点 uα・v$の α を A に還元して，時点 uA・v$に移行する．

(2)シフト(shift)：正準オートマトンの a による状態遷移を使って，入力トークン a を読み込んで，時点 u・av$から時点 ua・v$に移行する．

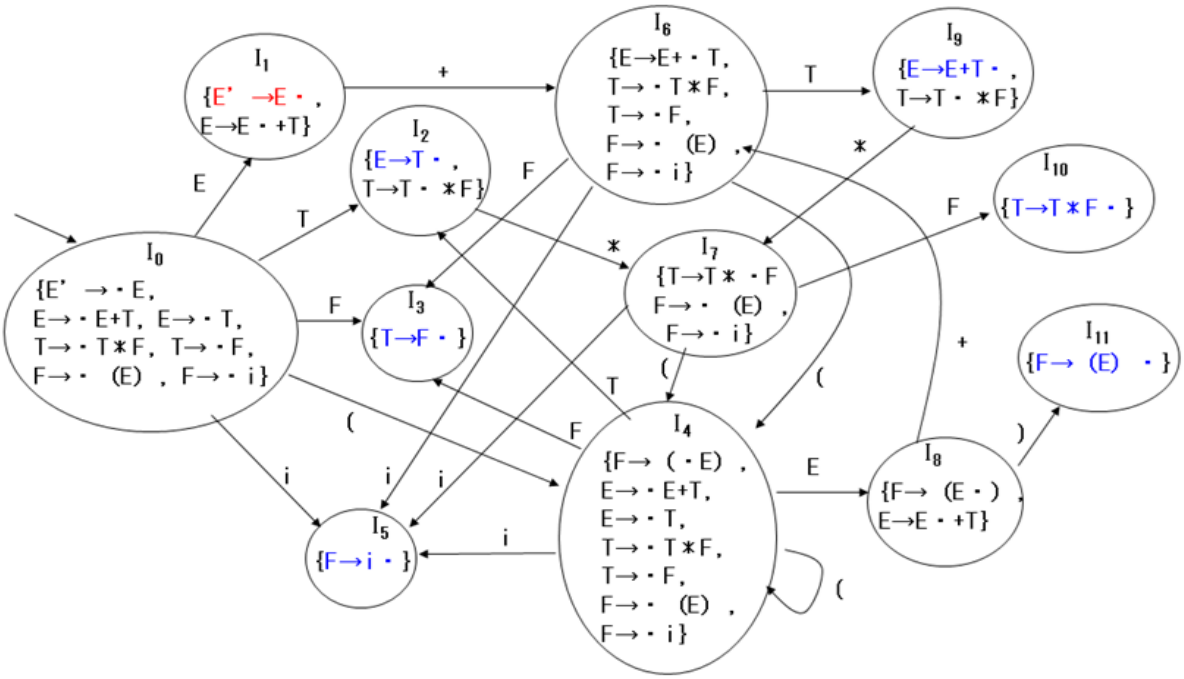

図１１　正準オートマトン（LR(0)オートマトン）

LR 構文解析の動作を説明するために，配置を導入する．まず，前述の時点 u・v$，u∈$(V_N \cup V_T)^*$，v∈$V_T^*$は，

$a_1a_2\cdots\cdots a_n \cdot x_kx_{k+1}\cdots\cdots x_n\$$

と書ける．ここで，$a_i \in V_N \cup V_T$ (i=1,…,n)，$x_j \in V_T$ (j=k,…,m)である．次に，時点 u・v$は正準オートマトンの状態 I_i(i=1,…,n)を用いて，配置

$(I_0a_1I_1a_2I_2\cdots\cdots a_nI_n \cdot x_kx_{k+1}\cdots\cdots x_n\$)$

が定義できる．ここで，I_0 は正準オートマトンの初期状態であり，I_i=GOTO(I_{i-1},a_i)(i=1,..,n)である．

次に，i+i*i$の構文解析を行い，図１２のように配置で示す．

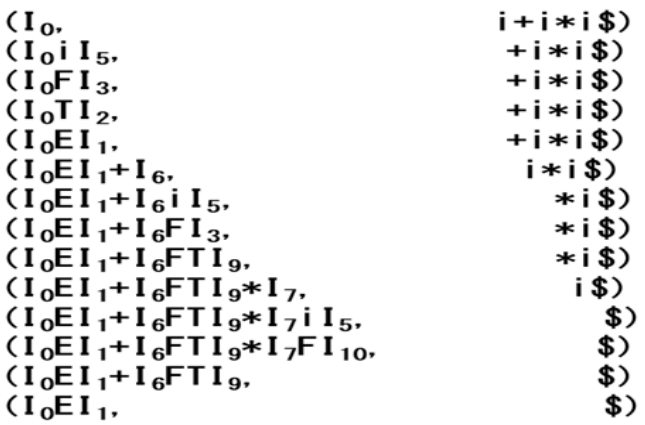

図１２　配置による i+i*i$の LR 構文解析の例

ここで，配置($I_0EI_1+I_6TI_9$,*i$)において，$I_9$={E→E+T・,T→T・*F}であり，E→E+T・による還元と T→T・*F によるシフトの２つの可能性があり，不都合な状態と呼ぶ。図１２では，シフトを優先させて，シフトを行った．この不都合な状態を回避する解決方法として，一文字先読みする SLR(1)や LR(1)，LALR(1)がよく知られている．

６．３　SLR(1)構文解析手法

LR 構文解析において，次の入力トークンを一つだけ先読みし，Follow 集合を使って不都合な状態を回避するのが SLR(1)構文解析手法である．以下に，次の入力トークンを一つだけ先読みし，Follow 集合を使って不都合な状態を回避する方法を定義する．

（定義）時点 u・v$=u・$v_1$v’$において，u に対し A→α・を使って還元するときは以下である．ただし，v_1は終端記号である．

(1)u=u’α

(2)S・$ $\Rightarrow^*_{rm}$ u’A・v_1v’$ $\Rightarrow^*_{rm}$ u’α・v_1v’$かつ $v_1\in$Follow(A)の場合にのみ還元する

ここで，$\Rightarrow^*_{rm}$は 0 回以上の最右導出を行うことを表す．

もし $v_1\notin$Follow(A)ならば，LR(0)項 B→α_1・aα_2に対して，v_1=a であれば」シフトする．

もし $v_1\notin$Follow(A)かつ $v_1\neq$a ならば，構文エラーである．

ただし，Follow(A)={a∈V_T|S $\Rightarrow^*_{rm}$ $u’Aa…}

SLR(1)構文解析手法では，動作表と行先表を使用する．６．２の LR(0)構文解析の正準オートマトンから作成した動作表，行先表，生成規則の番号，Follow 集合を表１に示す．

(1)動作表は状態 I において，次の入力が a∈V_Tで会った場合の動作を，I の行の a の列に記述したものである．

① 動作表中の sj は，次の入力を読み込んで状態 I_jにシフトすることを表す．

② 動作表中の rj は，j 番目の生成規則で還元することを表す．

(2)行先表は還元動作 rj において，還元後に，状態 I と非終端記号 a に対して行き先の状態を記述したものである．

表１　動作表，行先表，生成規則の番号，Follow 集合

動作表

	i	+	*	(	)	$
I_0	s5			s4		
I_1		s6				受理
I_2		r2	s7		r2	r2
I_3		r4	r4		r4	r4
I_4	s5			s4		
I_5		r6	r6		r6	r6
I_6	s5			s4		
I_7	s5			s4		
I_8		s6			s11	
I_9		r1	s7		r1	r1
I_{10}		r3	r3		r3	r3
I_{11}		r5	r5		r5	r5

行先表

	E	T	F
I_0	I_1	I_2	I_3
I_4	I_8	I_2	I_3
I_6		I_9	I_3
I_7			I_{10}

生成規則の番号

1. E→E+T
2. E→T
3. T→T*F
4. T→F
5. F→(E)
6. F→i

Follow集合

Follow（E'）={ $ }
Follow（E）={ $, +,) }
Follow（T）={ $, +,), * }
Follow（F）={ $, +,), * }

次に，トークンの列 i*i+i$の SLR(1)構文解析を行う．図１３に構文解析の配置を示す．まず，配置の図の１行目では，(F→・i)∈I_0と動作表の１行目１列目の(I_0,i)より s5 となり，２行目が得られる．次に，２行目では，(F→i・)∈I_5と動作表の６行目３列の(I_5,*)より r6 となり，生成規則の番号６の F→i で還元して，行先表の(I_0,F)でI_3になる．同様の還元とシフトを続けて，配置の図の最後の行は，動作表の(I_1,$)より受理となり，構文木が生成できた．

（読込み済み，未読）／動作

(I_0, i*i+i$)　／action[$I_0$][i] = s5 より，i をシフトし，$I_5$をpush
(I_0iI_5, *i+i$)　／action[$I_5$][*] = r6 より，i$I_5$をpopし，F とgoTo[$I_0$][F]をpush
(I_0FI_3, *i+i$)　／action[$I_3$][*] = r4 より，F$I_3$をpopし，T とgoTo[$I_0$][T]をpush
(I_0TI_2, *i+i$)　／action[$I_2$][*] = s7 より，* をシフトし，$I_7$をpush
(I_0TI_2*I_7, i+i$)　／action[$I_7$][i] = s5 より，i をシフトし，$I_5$をpush
(I_0TI_2*I_7iI_5, +i$)　／action[$I_5$][+] = r6 より，i$I_5$をpopし，F とgoTo[$I_7$][F]をpush
(I_0TI_2*I_7FI_{10}, +i$)　／action[$I_{10}$][+] = r3 より，T$I_2$*$I_7FI_{10}$をpopし，T とgoTo[$I_0$][T]をpush
(I_0TI_2, +i$)　／action[$I_2$][+] = r2 より，T$I_2$ をpopし，E とgoTo[I_0][E]をpush
(I_0EI_1, +i$)　／action[$I_1$][+] = s6 より，+ をシフトし，$I_6$をpush
(I_0EI_1+I_6, i$)　／action[$I_6$][i] = s5 より，i をシフトし，$I_5$をpush
(I_0EI_1+I_6iI_5, $)　／action[$I_5$][$] = r6 より，iI_5 をpopし，F とgoTo[I_6][F]をpush
(I_0EI_1+I_6FI_3, $)　／action[$I_3$][$] = r4 より，FI_3 をpopし，T とgoTo[I_6][T]をpush
(I_0EI_1+I_6TI_9, $)　／action[$I_9$][$] = r1 より，EI_1+I_6TI_9をpopし，E とgoTo[I_0][E]をpush
(I_0EI_1, $)　／action[$I_1$][$] = 受理 より，構文解析終了（エラーなし）．

図１３　i*i+i$の SLR(1)構文解析の配置

６．４　LR(1) 構文解析手法

SLR(1)構文解析では，不都合な状態を解決するために Follow 集合だけを用いた．すなわち，ある状態 I_1 の中の A→α・とそれ以外があったとき，次の入力記号 a が a∈Follow(A)であるとき A→α による還元をすることにした．Follow(A)={a|a∈V_T, S$⇒*…βAa…}とすると，Follow は A の右側の a を考慮しているが，A の左側にあるβは考慮していない．ここで，A の左側と右側の両方を考慮し，Follow 集合を細分化したものが LR(1)文法である．A の左側βが違えば次にくる a も違う可能性がある．そこで，入力記号も一緒にして，[A→α・,a]とう LR(1)項を使用する．さらに，この LR(1)項を一般化して，[A→x・y,a]を考える．左側の A→x・y は LR(0)項であり，LR(1)項の核と呼ばれる．a は先読み記号と呼ばれる終端記号である．LR(1)項は構文解析が進んで，もし完全項を核とする LR(1)項[A→xy・,a]になれば，次の入力が x のときに，生成規則 A→xy による還元ができることを意味する．なお，LR(1)項の集合については，核が同じである LR(1)項[A→x・y,a_1],…，[A→x・y,a_n]をまとめて，[A→x・y,a_1/…/a_n]と書く．

LR(1)項の集合 I に対して，その閉包 Closure(I)は次のアルゴリズムで得られる．

（アルゴリズム）

1. Closure(I):=I
2. [A→α・Bβ,a]∈Closure(I)に対して，B→γという生成規則があれば，b∈First(βa)なるすべての b について，[B→・γ,b]を Closure(I)に加える．
3. 上記 2. を新たに加えるものがなくなるまで繰り返す．

この閉包が LR(1)構文解析における正準オートマトンの状態である．正準オートマトンの状態遷移は次の GOTO(I,X)によって決められる．

（定義）I を LR(1)項の集合，X∈(V_N∪V_T)=V としたとき

GOTO(I,X)=Closure({[A→αX・β,a]|[A→α・Xβ,a]∈I})

文法 G’=<V_N, V_T, P, E>において，このような LR(1)項の集合を求めるには，まず，構文解析の終了判定を容易にするために，新たな非終端記号 E’を追加し E’→E なる生成規則を文法 G’の生成規則の集合 P に追加して，以下のアルゴリズムを実行する．

（アルゴリズム）

初期設定　C:={Closure({E’→・E })}

以下を新たな集合が C に付け加えられなくなるまで繰り返す：

任意の I∈C と X∈V に対し，GOTO(I,X)が空集合でなく，C に入っていなかったら C に加える

LR(1)構文解析の動作表は SLR(1)の動作表と同様に作られるが，次の点のみが異なる．

SLR(1)の場合は

A→α・∈I_1 ならば a∈Follow(A)に対して A[I_1,a]は A→α で還元する

であったが，LR(1)の場合は

[A→α・,a]∈I_1 ならば A[I_1,a]は A→α で還元する

となる．LR(1)の正準オートマトンの一部を図１４に示す。

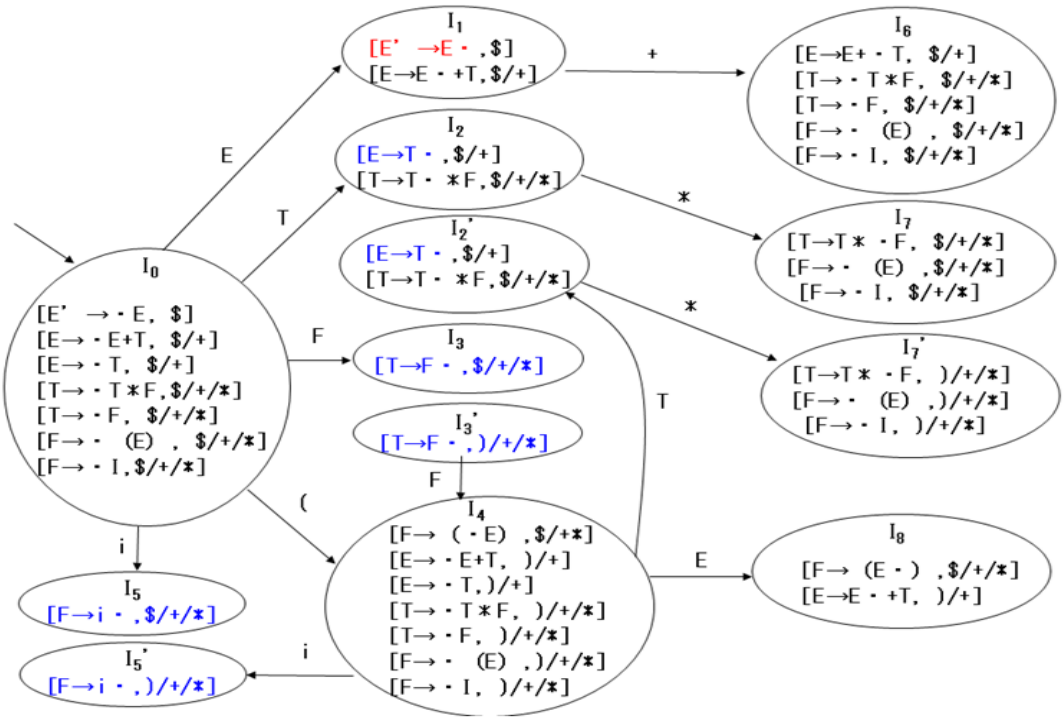

図１４　LR(1)の正準オートマトンの一部

６．５　LALR(1) 構文解析手法

LR(1)構文解析法は SLR(1)よりも詳細な構文解析が可能であるが，正準オートマトンの状態数が多すぎ，現実のプログラミング言語では実用的ではない．そこで，核が同じである LR(1)項を同一視することによって，状態数を減らすのが LALR(1)構文解析法である．これによって，LALR(1)の構文解析の正確さは LR(1)より劣るが，Follow 集合を使う SLR(1)よりも正確な構文解析が実現でき，しかも状態数は SLR(1)と全く同じになり，実用的な構文解析が可能となる．なお，LALR は Lookahead（先読み）LR の略である．LALR(1)の正準オートマトンを図１５に示す．C 言語は LALR(1)構文解析で実現できることが知られている．

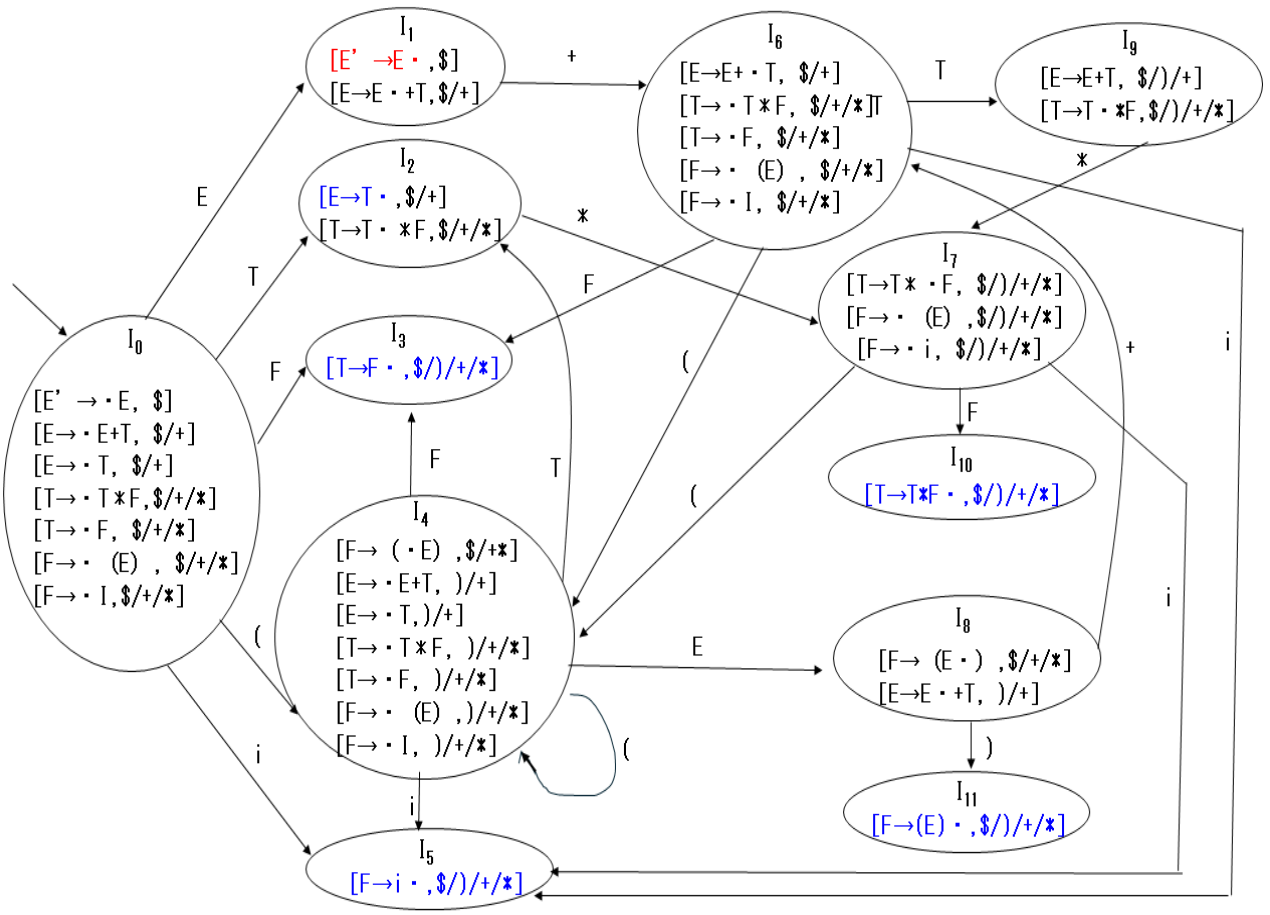

図１５　LALR(1)の正準オートマトン

６．６　LR 構文解析の理論的背景

ここでは，和文教科書［6］と LR 構文解析の論文［7］に従い，LR 構文解析の理論的背景を説明する．

まず，プログラミング言語の文法を表す文脈自由文法を G0=(N, Σ, P, S)とする．ここで，N は非終端記号の集合，Σは終端記号の集合，P は生成規則の集合，S は開始記号である．なお，不要な非終端記号は含まれないで，ε生成規則もないとする．次に，LR(0)の場合と同様な考え方により，以下のような新たな開始記号 S’を追加した文法 G0’を考える．なお，$はトークン列の最後尾を表す記号である．

$G0' = (N \cup \{S'\}, \Sigma \cup \{\$\}, P \cup \{S' \rightarrow S\$\}, S')$

次に，G0’によって定まる文字列の集合 C_G を以下のように定義する．

$C_G = \{\alpha\ \beta \mid S' \Rightarrow^*_{rm} \alpha Aw \Rightarrow_{rm} \alpha\ \beta w\}$

ここで，記号列αβはハンドルβが生成規則 A→βで還元できる記号列であり，C_G は還元できる記号列の集合である．なお，ハンドルとは，記号列の中で生成規則の本体と合致する部分列のことで，その生成規則による還元が最右導出を逆順に辿ったときの１ステップに対応するものを呼ぶ．もっとも重要なことは C_G が正規言語であることである．ゆえに，C_G を生成する文法は正規文法であるので，有限オートントンで受理できる［7］．以上より，C_G を受理する還元動作をする非決定性有限オートマトンが構成できることがわかる．

次に，和文書籍［6］に従い，C_G を受理する還元動作をする非決定性有限オートマトンを

構成する.

G0’=(N∪{S’},Σ∪{$},P∪{S’→S$},S’)とし,C_G={α β|S’$\Rightarrow^*_{rm}$α Aw$\Rightarrow_{rm}$α β w}を受理する非決定性有限オートマトン NG を次のように構成する.

NG=(Q, Σ, δ, S0, F)

ここで,

Q={[A→α・β]|A→α β∈P}　　(LR(0)項)

Σ=N∪Σ　　(語彙=非終端記号の集合∪終端記号の集合)

S0=[S→・S$]　　(開始記号の導入項)

F={[A→α・]|A→α∈P}　　(完全項（CG の受理状態（還元）))

δは以下の関数である：

δ([A→α・vβ],v) = [A→αv・β],ここで v∈N∪Σ

δ([A→α・Bβ], ε)= [B→・γ],ここで B→γ∈P

ここで，集合の包含関係より，L(NG)= {α β|S’$\Rightarrow^*_{rm}$α Aw$\Rightarrow_{rm}$α β w}= C_Gが示せる．非決定性有限オートマトン NG は部分集合構成法 [8] により，決定性オートントンである正準オートマトンが構成できる.

以上により,最右導出の最後の１ステップは有限オートマトンを使い実現できる.つまり，有限オートマト，すなわち正準オートマトンを繰り返し使うことにより，ある範囲の文脈自由文法の解析ができることがわかる．さらに，配置を用いて構文解析を進めてきたが，これは正準オートマトンの状態遷移をスタックに記憶しており，構文解析の効率化を実現している [6].

６．７　LR 構文解析のまとめ

LR 構文解析のクラスを図１６にまとめる．以下のことが知られている.

1. まず，LR 構文解析は再帰的下向き構文解析よりも広いクラスの文法が解析できる.
2. 次に，LR(K)は k が大きくなるほど，広いクラスの文法が解析できる.
3. 次に，LR(1)を細かく分けると，LR(0)<SLR(1)<LALR(1)<LR(1)の順番で，より広いクラスの文法が解析できる.
4. 次に，LALR(1)によりほとんどのプログラミング言語が構文解析できる.
5. 最後に，LR 構文解析器は構文解析器生成系 Yacc(Yet Another CompilerCompiler) [2] により，自動生成できる．これは文法から正準オートマトンが自動生成できることにより理解できるであろう.

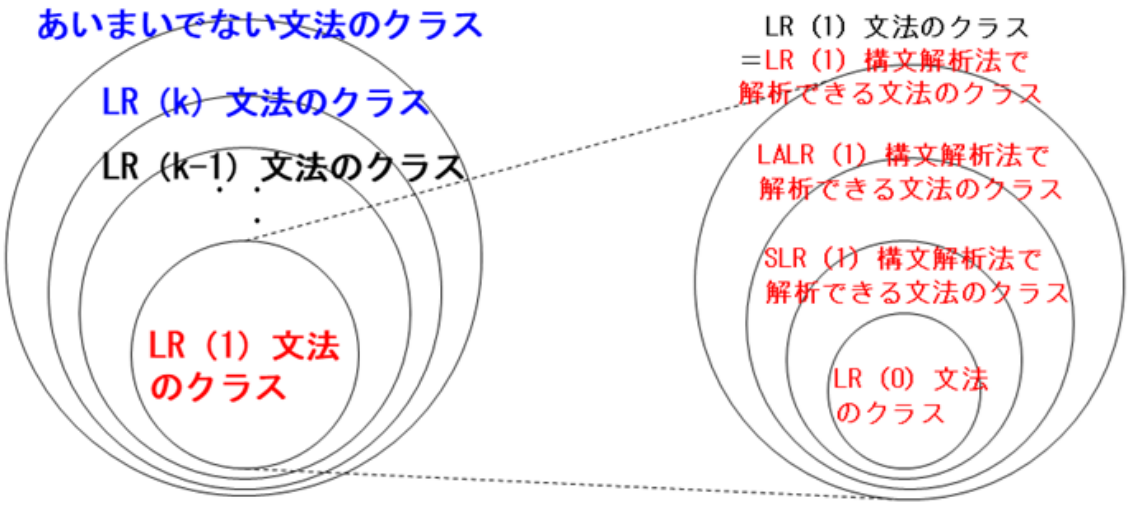

図１６　LR 構文解析の文法のクラス

７．まとめ

私たちの世界を動かすプログラムは，人間が Python や Java，C などの高級プログラミング言語で記述し，それをコンパイルして低レベルのコードにまとめて実行する．現代のプログラミング言語のコンパイラの多くは，Alfred V. Aho と Jeffrey D. Ullman の貢献が大きい[1]．コンパイラでは字句解析，構文解析，コード生成，最適化などのための効率的な技術やアルゴリズムが重要である． 本解説記事では，コンパイラの構造及び，コンパイラで最重要で最難関な構文解析，特に LR 構文解析［7］について理論や技術などの解説を行った．特に，類書にない構文解析手法の間の関連および新たな事例を盛り込んで解説を行ったので理解を深める助けとなる．もっとも重要なことは，LR 構文解析［7］の本質は，文脈自由言語のプログラミング言語の構文解析に対して，驚くべきことに有限オートマトンを繰り返し実行していることである．これの解説記事により，現在と未来の社会を支えるコンピュータ・ソフトウェアの理解を深めることができる．さらに重要なことは，コンパイラの理論，技術，実装は他の多くのソフトウェアの理論や設計，実装などの原理となっており，今後の新しいソフトウェアの設計や実装に有益である．

今後，人工知能や分散処理， 組込みシステムなどの多数の新しいアプリケーションの開発が必要であり，それらを合理的に記述して設計・実装できる新しいプログラミング言語が生まれるので，コンパイラの理論と技術，実装などの理解は必修であり，本解説はそれらに有益であろう．また， 今後， 組込みシステムや人工知能などにおいて， 新しいハードウェアが作られるので， コンパイラの最適化技術などの進展が期待される． なお，ドラゴンブック[2] を手本に，日本語の優れた教科書も多数あり， その代表的な書籍［3,4,5,6］などを参考文献にあげており，本解説記事の作成にも参考にしている．

引用・参考文献

[1] ACM(2020)”ACM Turing Award Honors Innovators Who Shaped the Foundations of Programming Language Compilers and Algorithms”,ACM(https://awards.acm.org/about/2020-turing (accessed on 31 December 2024)).

[2] Alfred V. Aho, Monica S. Lam, Jeffrey D. Ullman(2007)”Compilers principles, techniques & tools”, pearson Education, pp.1-1038.
（訳本：A.V.エイホ （著), 原田 賢一 （翻訳)(2009)『コンパイラ 第2版: 原理・技法・ツール』,サイエンス社,pp.1-1090.）

[3] 中田育男(2009)『コンパイラの構成と最適化 第2版』,朝倉書店,pp.1-601.

[4] 佐々 政孝(1989)『プログラミング言語処理系』,岩波書店,pp.1-602.

[5] 湯淺 太一(2014) 『コンパイラ』,オーム社,pp.1-248.

[6] 大堀 淳(2021)『コンパイラ: 原理と構造』,共立出版,pp.1-185.

[7] Donald E. Knuth(1965)”On the Translation of Languages from Left to Right”, Information and Control, 8(6), pp.607-639.

[8] Jeffrey D. Ullman, John Hopcroft, Rajeev Motwani(2006)” Introduction to Automata Theory, Languages, and Computation”, Addison Wesley,pp.1-750.

謝辞

本解説記事は金沢大学理工学域電子情報学類の３年生向けコンパイラの講義資料をまとめたものです。コンパイラ技術全般において，中田先生，佐々先生，湯浅先生，大堀先生のセミナーや書籍などは大変有益であり，ここに感謝します。また，TA として講義資料作成にご協力いただいた金沢大学大学院生の小寺広志さんに深く感謝します。最後に，丁寧に本解説記事を査読いただいた査読者各位に感謝します。