信州大学工学部

学士論文

演算子の文字列表現を自由に定義できる構文解析器の実装

指導教員　西新 幹彦 准教授

学科　電気電子工学科
学籍番号　09T2065B
氏名　西山 達矢

2013 年 3 月 15 日

目次

1　はじめに

1.1　研究の目的と設計

C++ 言語などでは演算子をプログラマ独自のデータ型に対して使用する際に，その演算子に特有の意味を定義することができ，これを演算子の多重定義と呼んでいる．演算子の多重定義では，既存の演算子にデータ型特有の別の意味を与えるため，型が保有する演算子の種類は既存の演算子の数に限られている．本研究では，プログラム文法内に定義された演算子のほかに，+，−，*，/の 4 つの文字の組み合わせを新たな演算子として使用でき，演算子の数に制限がないプログラム文法を作ろうと考えた．その実現のため，プログラム文法内に定義された演算子と新たに定義された演算子を用いて記述されたソースコードが，プログラム文法に従っているか確認するための構文解析器を作成した．

通常，プログラム言語を設計する際に，演算子の扱いはプログラム文法内に定義され，プログラム文法内に演算子の文字列表現と，その文字列表現の演算子の扱いを記述する．本研究では，演算子の文字列表現を自由にすることを考えたため，どのような文字列表現の演算子を用いるかは文法の設計者ではなく，文法に従いプログラムを記述するプログラマが定義する．そのため，プログラム文法内に，演算子の文字列表現を直接記述することはできず，演算子の定義は，プログラム文法の書かれたファイルとは別のファイルに記述するように設計した．構文解析器はプログラム文法とプログラマが記述した演算子の定義に従いソースコードの構文を解析するように設計した．

作成した構文解析器は「プログラム文法の定義ファイル」,「演算子の定義ファイル」,「ソースコード」の 3 つを用いて，ソースコードの構文解析を行う．「プログラム文法の定義ファイル」はプログラム文法の設計者が記述するファイルであり,「演算子の定義ファイル」と「ソースコード」は文法に従いプログラマが記述する．本研究の段階では,「ソースコード」内に演算子の定義を記述することはできない．ソースコード内に演算子の定義を記述した場合,「ソースコード内に演算子が定義されている」というソースコードの意味を解析する必要がある．これは意味解析器の処理であるが，現段階では構文解析器まで実装したため，意味解析を行うことができない．そのため,「ソースコード」と「演算子定義ファイル」の 2 つのファイルに分け，構文解析器でソースコードの解析ができるように設計をした．

1.2　本論文の構成

本論文は次のように構成されている．2 章では，ソースコードからトークンを切り出す方法を説明する．3 章ではソースコードの構文解析法を説明する．4 章では作成した構文解析器の概要と構文解析器に入力する「プログラム文法の定義ファイル」,「演算子の定義ファイル」の

2 つのファイルの記述方法を説明する．5 章では構文解析器の実装に使用したデータ構造と関数群の説明をする．6 章ではプログラム文法に従って記述されたソースコードとプログラム文法に従っていないソースコードを解析したときの構文解析結果を示す．7 章においてまとめを示す．

2　字句解析

ソースコードから言語を構成する意味のある最小単位であるトークン（たとえば，変数や演算子など）に分解する動作を字句解析という．トークンを定義する際には，トークンの名前と文字の並びを記述したパターンが必要であり，正規表現を用いて表 1 のように定義する．

表 1　トークンの名前とパターン 1

トークンの名前	パターン
ID	`[a-zA-Z]([a-zA-Z]|[0-9])*`
SPACE	`\s+`
EQUAL	`==`
DIGIT	`[0-9]+`

表 2　トークンの切り出し 1

トークンの名前	トークン
ID	`digit`
SPACE	␣
EQUAL	`==`
DIGIT	`100`

ソースコードをトークンに分解する際には，表 1 に示したパターンについて上から照合していき，最初にマッチしたトークンを切り出していく．表 1 に従って，ソースコード `digit␣==100` をトークンに分解すると表 2 のようなトークンが得られる．分解の様子を図 1 に示す．

`digit␣==100`　`digit`(ID) の切り出し
`␣==100`　␣ (SPACE) の切り出し
`==100`　`==`(EQUAL) の切り出し
`100`　`100`(DIGIT) の切り出し

図 1　トークンの切り出し

表 3 は表 1 に代入演算子 (=) を追加したものである．表 3 によってソースコードをトークンに分解すると, 表 4 に示すように==は=が 2 つ並んでいると解析される．これは代入演算子の解析順位が等号演算子よりも高いためであり，==を一つのトークンとして分解するためには，==の解析順位を=の解析よりも先に行うように記述しなければならない．

表 3　トークンの名前とパターン 2

トークンの名前	パターン
ID	`[a-zA-Z]([a-zA-Z]|[0-9])*`
SPACE	\s+
ASSIGNMENT	=
EQUAL	==
DIGIT	`[0-9]+`

表 4　トークンの切り出し 2

トークンの名前	トークン
ID	`digit`
SPACE	␣
ASSIGNMENT	=
ASSIGNMENT	=
DIGIT	100

3　構文解析

ソースコードより得られたトークン列からもとのソースコードがどのように構成されているか調べる動作を構文解析という．プログラム言語の記号の並べ方を記述した規則を構文規則といい，構文規則によって, プログラムの文法を記述する．構文解析ではソースコードの構造を解析し，ソースコードが構文規則に従って記述されているかどうかを判定する．

3.1　構文解析法の種類

構文解析の手法にはいくつかの種類があり，例えば，演算子順位法，LL 法，LR 法などがある．本研究では，LR 法による構文解析器の作成を行ったが，LR 法をさらに分類すると LR(1) 構文解析，LALR(1) 構文解析などがある．LALR(1) 法は LR(1) 法に比べて扱えるプログラム文法の範囲が狭いが，LR(1) 法に比べて構文解析表がずっと小さく収まり，プログラミング言語に共通の構文要素は LALR(1) 文法で表現できるので，実用上もよく使用されている [1]. LALR(1) 法では C や JAVA のプログラム文法が記述できる [1][2].

本研究では，C 言語を拡張し，ソースコード中に新たな演算子を定義できるようなプログラミング言語の実装を目指している．しかし，本研究で実現を目指すプログラム文法が LALR(1) 法で解析できるかどうかは不明である．そのため，LALR(1) 法よりも適用範囲の広い LR(1) 法を用いて構文解析器を実装した.

3.2　構文規則

字句解析で得られたトークンは，構文解析では終端記号と呼ばれる．終端記号は表 1 や表 3 のようなトークンの名前によって区別される．したがって，表 1 によってトークンに分解するとき，ソースコードに書かれた `if` や `digit` は文字の並びは違うが，構文解析を行う上では，

どちらも ID という終端記号であるとみなされる．終端記号は記号の一部であり，終端記号でない記号を非終端記号という．非終端記号の意味は以下の説明の中で述べる．

生成規則とは，左辺 $\rightarrow$ 右辺の形をしたもので，左辺に非終端記号，右辺に非終端記号と終端記号の列が並ぶ．これは，左辺の非終端記号が右辺の記号列に対応していることを表している．たとえば，$S \rightarrow E = E$ という生成規則は非終端記号 S が記号列 $E = E$ に対応している．$E = E$ の記号列を S に書き換えることを還元と呼ぶ．反対に，S から $E = E$ の記号列に書き換えることは導出という．

生成規則の集合を構文規則といい，構文規則によってプログラム言語のもつ階層構造を表現する．

特別な意味を持つ開始記号という記号が非終端記号の中に一つだけ存在する．開始記号は構文規則の中のただ一つの生成規則の左辺に現れる．図 2 に開始記号 $Start$ を含んだ構文規則の例を示す．終端記号を生成規則を利用して非終端記号に還元し，その非終端記号を生成規則によって別の非終端記号に還元していくと，最後には $Start$ に書き換わる．反対に，$Start$ から生成規則を何回か利用して導出していくと，終端記号の列が得られる．

$$
\begin{aligned}
&Start \rightarrow S \\
&S \rightarrow E = E \\
&S \rightarrow ID \\
&E \rightarrow E + T \\
&E \rightarrow T \\
&T \rightarrow ID
\end{aligned}
$$

図 2　構文規則 1

構文規則によってプログラム文法を記述するため，構文規則とプログラム文法は同じ意味で用いる．構文解析とは，与えられた記号列が文法によって導出できるかどうかを判定することである．

3.3　LR(1) 項

ソースコードを構文解析をしている途中の状態を考える．たとえば，A と B が非終端記号で，x と y が終端記号である生成規則

$$A \rightarrow x\ B\ y$$

から作られるものを構文解析している途中で，x を読み終わった後だとする．この状態を，

$$A \rightarrow x \bullet B\ y$$

と表し，• をマーカーと呼ぶことにする．B が非終端記号で，$B \to \beta$ という生成規則があったとき，マーカーが B の前にあるということは，β を読む前であるとも考えられる．β が終端記号であれば，β から導出される生成規則はないから，A と B に関する 2 つの式を，

$$A \to x \bullet B\ y$$
$$B \to \bullet\beta$$

とまとめることができる．x を読み終わった後，次に β を読んだときの状態は，マーカーをひとつ右にずらし

$$B \to \beta\bullet$$

とかける．マーカーが末尾にあるとき，右辺は左辺に還元される．上の例では，βは B に還元される．A の生成規則をみると，B を読んだ後の記号は y だから，y が終端記号の場合，次の入力記号は y である．生成規則の末尾にマーカーが来たとき，次の入力記号が y であることを表現するために，

$$B \to \beta\bullet,\ y$$

と書くことにする．

上のように，生成規則と，その生成規則の直後にくる終端記号を添えたものを LR(1) 項と呼ぶ．上の例の y は先読み文字と呼ばれる．LR(1) 項はマーカーの位置が生成規則の末尾になくてもよく，

$$A \to x \bullet y,\ z$$

のように，• が生成規則の端にない場合も LR(1) 項と呼ぶ．また，先読み文字は複数あってもよい．

今は，ソースコードを構文解析している途中の状態を考えたが，次に，途中の状態からではなく，最初の状態から構文解析することを考える．最初の構文解析の状態は

$$Start \to \bullet S,\ \$$$

と書くことができる．$ はソースコードの終わりを表す入力である．$Start$ は開始記号であり，全ての生成規則は $Start$ に還元されるのだから，$Start$ はソースコード全体を表している．今，ソースコードの解析を何もしていないから，ソースコード全体を表す $Start$ の前にマーカーをつけている．

ソースコードを読み終わり，

$$Start \to S\bullet,\ \$$$

から，次に $ を読み込めば，そのソースコードは構文規則に従って記述されていると言える．

3.4 LR(1) 項集合集成の作成

ソースコードを解析する前の $Start \to \bullet S,\ \$$ からは，

I_0
$S \to \bullet E = E,\ \$$
$S \to \bullet ID,\ \$$
$E \to \bullet E + T,\ = / +$
$E \to \bullet T,\ = / +$
$T \to \bullet ID,\ = /+$

の LR(1) 項の集合 I_0 が得られる．$= /+$ は $=$ と $+$ の 2 つの先読み文字があることを表している．このような LR(1) 項の集合を求める演算を，LR(1) 項集合の閉包演算という．

I_0 を求めるためには，まず，$[Start \to \bullet S,\ \$]$ から $[S \to \bullet E = E,\ a_0]$, $[S \to \bullet ID,\ a_1]$ を導出する．2 つの項の先読み文字 a_0 と a_1 は，$[Start \to S\bullet,\ \$]$ のマーカーが指している文字である．マーカーは生成規則の末尾にあるから，a_0 と a_1 は $[Start \to S\bullet,\ \$]$ の先読み文字である $ となり，$[S \to \bullet E = E,\ \$]$, $[S \to \bullet ID,\ \$]$ が得られる．

次に，$[S \to \bullet E = E,\ \$]$ から $[E \to \bullet E + T,\ =]$, $[E \to \bullet T,\ =]$ を導出する．導出元の LR(1) 項のマーカーをひとつ右に進めた結果から先読み文字は $=$ であることがわかる．導出元のマーカーが末尾にないときには，導出された LR(1) 項からさらに導出を繰り返し，先読みを追加する．$[E \to \bullet E + T,\ =]$ から導出を行うと，$[E \to \bullet E + T,\ = /+]$, $[E \to \bullet T,\ = /+]$ となる．先読み文字に新たに $+$ が追加されている．この導出結果からさらに導出を行い，先読み文字を求めても，新たに追加される先読み文字はないから，これ以上導出を行わない．

最後に，$[E \to \bullet T,\ = /+]$ から $[T \to \bullet ID,\ a_5]$ が導出される．導出元のマーカーをひとつ右に進めれば，マーカーは生成規則の末端にあるから，a_5 は $= /+$ である．

今は $[Start \to \bullet S,\ \$]$ の 1 つの LR(1) 項から閉包演算を行ったが，LR(1) 項の集合に対しても閉包演算をおこなうことができる．

閉包演算をする際には，LR(1) 項の先読み文字を求める必要がある．先読み文字を求めるためには，次の FIRST 演算を使用する．

定義 1（FIRST 演算） 文法記号 X に対して，FIRST(X) を求めるには，次のステップを，どの FIRST 集合にも終端記号が追加されなくなるまで繰り返す．

1. X が終端記号であれば，FIRST(X) $= \{X\}$ である．
2. X が非終端記号であり，$X \to \alpha$ ならば，FIRST(α) を加える．

LR(1) 項集合の閉包演算は FIRST 演算を利用し，次のように定義することができる．

定義 2（LR(1) 項集合の閉包演算） I を文法の LR(1) 項の集合とするとき，I の閉包 closure(I) とは，I から次のようにして得られる LR(1) 項の集合である．

1. I 中のすべての LR(1) 項は closure(I) に含まれる．
2. closure(I) 中に含まれ，マーカーの次が非終端記号であるような LR(1) 項 $[A \to \alpha \bullet B\ \beta,\ a]$ があるなら，B を左辺とする各生成規則 $B \to \gamma$ と FIRST($\beta\ a$) 中の各終端記号 b から定まる $[B \to \bullet\gamma,\ b]$ を，それが closure(I) に含まれていなければ，closure(I) に加える．この操作を closure(I) に新たに付け加えられる項がなくなるまで繰り返す．

closure($[Start \to \bullet S,\ \$]$) によって求めた I_0 に対して，各文法記号を読んだとき，次のように，4 つの LR(1) 項集合 I_1，I_2，I_3，I_3 が作られる．

S を読んだとき，

I_1
$S \to S\bullet,\ \$$

E を読んだとき，

I_2
$S \to E\bullet = E,\ \$$
$E \to E \bullet +T,\ =/+$

ID を読んだとき，

I_3
$S \to ID\bullet,\ \$$
$T \to ID\bullet,\ =/+$

T を読んだとき，

I_4
$E \to T\bullet,\ =/+$

このような LR(1) 項集合を求める操作を goto 演算といい，次の定義に従って求める．

定義 3（LR(1) 項集合の goto 演算） I が LR(1) 項の集合，X が文法記号のとき，それに goto 演算を施した goto(I, X) は，次によって決まる LR(1) 項の集合である．

$$\text{goto}(I, X) = \text{closure}(\{[A \to \alpha\ X \bullet \beta,\ a] | [A \to \alpha \bullet X\ \beta,\ a] \in I\})$$

解析器の初期状態は $[Start \to \bullet S,\ \$]$ から閉包をとった LR(1) 項集合である．初期状態から到達する可能性のある LR(1) 項集合をまとめたものを LR(1) 項集合集成と呼び，次のアル

ゴリズムによって求められる．

アルゴリズム 1（LR(1) 項集合集成の作成） 次によって得られる C が LR(1) 項集合の集成である．
初期条件：$C = \{\text{closure}(\{[Start \to \bullet S,\ \$]\})\}$
C に新たな LR(1) 項集合が付け加えられなくなるまで以下を続ける．
C 中の LR(1) 項集合 I と，$\text{goto}(I, X)$ が空でないような各文法記号 X について，LR(1) 項集合 $\text{goto}(I, X)$ が C に含まれていなければ，それを C に加える．

3.5 LR(1) 構文解析表の作成

3.4 節によって作成した LR(1) 項集合集成から，次に LR(1) 項構文解析表を作成する．構文解析表は各文法記号を読んだときの解析器の状態を表にしたものである．構文解析表には，goto 演算と以下で説明する action 演算の情報が格納されている．

goto 演算は非終端記号の文法記号を読んだときに，現在の LR(1) から，どの LR(1) 項集合に飛ぶかまとめたものである．定義 3 の goto は各文法記号について求めたが，構文解析表での goto 演算は非終端記号についてのみまとめたものである．

action 演算はシフト，還元，受理の 3 つの命令で構成されている．シフトは終端記号に対する goto 演算に相当する．還元は生成規則の右辺値を左辺値にまとめる操作であり，受理はソースコードがプログラム文法に従って書かれていることを示している．

アルゴリズム 2（LR(1) 構文解析表の作成） アルゴリズム 1 により LR(1) 項集合集成 $C = \{I_0, I_1, \cdots, I_n\}$ を作り，それから LR(1) 構文解析表を作成する．

1. 状態 i を I_i に対応させる．状態 i に対する構文解析表の action 表のエントリは次のように定める．下の規則で二重で定められる表エントリがあれば，衝突あるいは競合があるといい，文法は LR(1) ではない．
   (a) マーカーの次に終端記号が来るような LR(1) 項 $[A \to \alpha \bullet a\ \beta,\ b]$(a は終端記号) が I_j に含まれ，$\text{goto}(I_i, a) = I_j$ なら，action$[i, a]$=シフト j とする．
   (b) LR(1) 項 $[A \to \alpha\bullet,\ a]$(ただし，$A \neq Start$) が I_i に含まれるなら，action$[i, a]$=$A \to \alpha$ で還元とする．
   (c) $[Start \to S\bullet,\ \$]$ が I_i に含まれるなら，action$[i, \$]$=受理とする．
2. 状態 i に対する goto 表のエントリは，各非終端記号 A について ($Start$ については不要)，$\text{goto}(i, A) = I_j$ なら，$\text{goto}[i, A] = j$ とする．
3. 上の (1)，(2) で定義されない表エントリは誤りとする．
4. 構文解析表の初期状態は，項 $[Start \to \bullet S,\ \$]$ を含む項集合に対応する状態とする．

3.6 LR(1) 構文解析表による構文解析

前節によって作成された LR(1) 構文解析表とソースコードを解析し得られるトークンを用いて，実際にソースコードが文法に従って記述されているか確かめるためのアルゴリズムを下に示した．

アルゴリズム 3（構文解析表による構文解析） ソースコードからトークンを一つずつ解析器に入力し，解析器の状態をスタックに積み，ソースコードの解析を行う．スタックトップを s_m とし，解析器に入力されるトークンを a_i とする．

1. action$[s_m, a_i]$=シフト s なら，スタックに s をプッシュする．
2. action$[s_m, a_i]$=$A \to \beta$ で還元なら，還元に用いた生成規則の右辺 β の長さ r だけポップする．するとスタックの先頭には s_{m-r} がみえる．次に，goto 表を引いて s=goto$[s_{m-r}, A]$ をプッシュする．入力は進めない．
3. action$[s_m, a_i]$=受理なら，構文解析は成功して止まる．

構文解析をする際には，まず最初に構文規則から LR(1) 項集合集成を作成する．LR(1) 項集合集成はアルゴリズム 1 により作成する．次に LR(1) 項集合集成からアルゴリズム 2 に従い，構文解析表を作成する．ソースコードからトークンを切り出し，アルゴリズム 3 に従い，構文解析表をもとに，ソースコードの構文解析を行う．

文法 1 の構文解析表を図 3 に示した．図中の命令は，(状態，文法記号)=動作と読む．sj 動作はシフトして状態 j をプッシュすること．rj 動作は番号 j の生成規則で還元すること．acc は受理を意味している．gj は状態 i において goto$(i,$文法記号$) = j$ という goto 演算を意味している．

図 3 に従い，`value=left+right` を解析すると，図 4 のように構文解析される．最初，スタックの状態は 0 であり，先読み記号は `ID` なので，構文解析表から $(0, ID)$ を引くと，$s12$ 命令である．そこで，スタックに 12 をプッシュする．解析器の状態は 12 になる．次に，状態 12 で，先読み記号が`=`なので，$(12, =)$ を引くと，$r5$ 命令である．$r5$ は $T \to ID$ で還元することを意味する．この生成規則の右辺の長さだけポップする．スタックの先頭は 0 となり，ここで，$(0, T) = g13$ を引き，状態 13 をプッシュする．以下同様にして，構文解析を続けると，`value=left+right` は acc(受理) される．

4 構文解析器の概要と入力ファイルの記述方法

ソースコードが LR(1) のプログラム文法に従って記述されているか確認するための構文解析器の概要を説明する．また，構文解析器に渡すファイルの種類とその記述方法を説明する．

(0, S) = g1	(3, T) = g8	(7, $) = r5	(11, EQUAL) = r5
(0, E) = g2	(3, ID) = s7	(7, PLUS) = r5	(11, PLUS) = r5
(0, ID) = s12	(4, $) = r1	(8, $) = r4	(12, $) = r2
(0, T) = g13	(4, PLUS) = s5	(8, PLUS) = r4	(12, EQUAL) = r5
(1, $) = acc	(5, T) = g6	(9, T) = g10	(12, PLUS) = r5
(2, EQUAL) = s3	(5, ID) = s7	(9, ID) = s11	(13, EQUAL) = r4
(2, PLUS) = s9	(6, $) = r3	(10, EQUAL) = r3	(13, PLUS) = r4
(3, E) = g4	(6, PLUS) = r3	(10, PLUS) = r3	

図 3　文法 1 の構文解析表

スタック	残り入力	動作
0:	`value=left+right$`	シフトし，12 をプッシュ
0:12:	`=left+right$`	$T \to ID$ で還元し，13 をプッシュ
0:13:	`=left+right$`	$E \to T$ で還元し，2 をプッシュ
0:2:	`=left+right$`	シフトし，3 をプッシュ
0:2:3:	`left+right$`	シフトし，7 をプッシュ
0:2:3:7:	`+right$`	$T \to ID$ で還元し，8 をプッシュ
0:2:3:8:	`+right$`	$E \to T$ で還元し，4 をプッシュ
0:2:3:4:	`+right$`	シフトし，5 をプッシュ
0:2:3:4:5:	`right$`	シフトし，7 をプッシュ
0:2:3:4:5:7:	`$`	$T \to ID$ で還元し，6 をプッシュ
0:2:3:4:5:6:	`$`	$E \to E + T$ で還元し，4 をプッシュ
0:2:3:4:	`$`	$S \to E = E$ で還元し，1 をプッシュ
0:1:	`$`	受理

図 4　value=left+right の構文解析例

4.1　構文解析器の概要

構文解析器は図 5 に示すように，「ソースコード」「字句文法ファイル」「演算子定義ファイル」の 3 つのファイルを入力とする．

演算子定義ファイルには，演算子の文字列表現と，演算子の種類が記述されている．字句文法ファイルには，トークンの名前とトークンのパターンが定義され，プログラム文法が定義さ

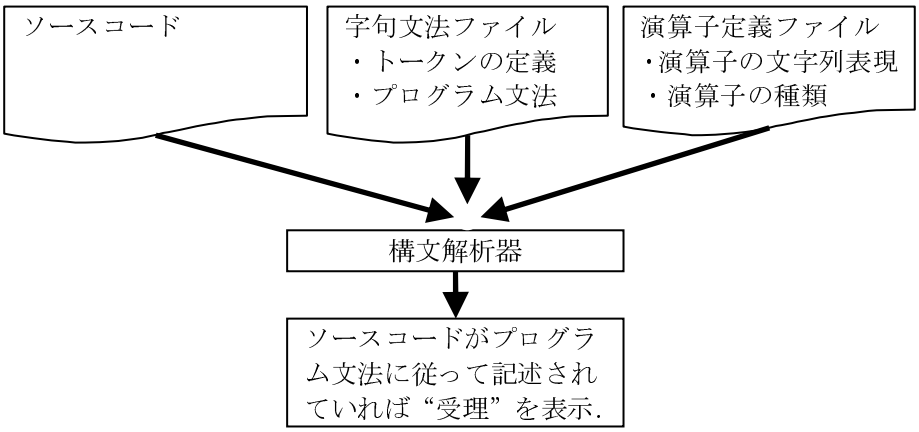


図 5　構文解析器の引数と結果

ている．ソースコードにはプログラム文法に従ったプログラムが書かれている．ソースコードが，プログラム文法に従っていれば，構文解析器は“受理”を表示する．

構文解析器の大まかな処理の流れを図 6 に示した．まず，構文解析器は字句文法ファイルに記述されたプログラム文法から LR(1) 構文解析表を作成する．次に演算子定義ファイルに記述された演算子のパターンにマッチするトークンを切り出す．演算子のパターンに一致するトークンが存在しなかったら，字句文法ファイルに記述されたトークンのパターンに従いソースファイルからトークンを切り出す．トークンの種類と構文解析表に従い，現在の解析状態をスタックに積みながらソースコードを構文解析する．

4.2　演算子定義ファイルの記述方法

演算子のパターンは `+,-,*,/`の文字の並びで表し，演算子は 1 から 6 のフラグを持つ．それぞれの演算子には，二項演算子 (4)，前置単項演算子 (2)，後置単項演算子 (1) のフラグが設定されている．演算子が二項演算子と前置単項演算子の 2 つの属性を持つ場合には，論理和である 6 のフラグをもつ．演算子定義ファイルの記述例を図 7 に示した．`**`演算子はフラグが 6 だから，二項演算子と前置単項演算子の両方の意味で使用でき，`*+*-`演算子は前置単項演算子として使用できる．

演算子は二項演算子，前置単項演算子，後置単項演算子の 3 つのうち 2 つまでをフラグとして持つことができる．これは演算子が 3 つ全てのフラグをもつと，プログラムの構文を一通りに定めることができないためである．

演算子が 3 つのフラグのうち 1 つを持つとき，フラグを一つしか持っていないから，その演算子が二項，前置単項，後置単項のうち，どの種類であるか必ず特定できる．

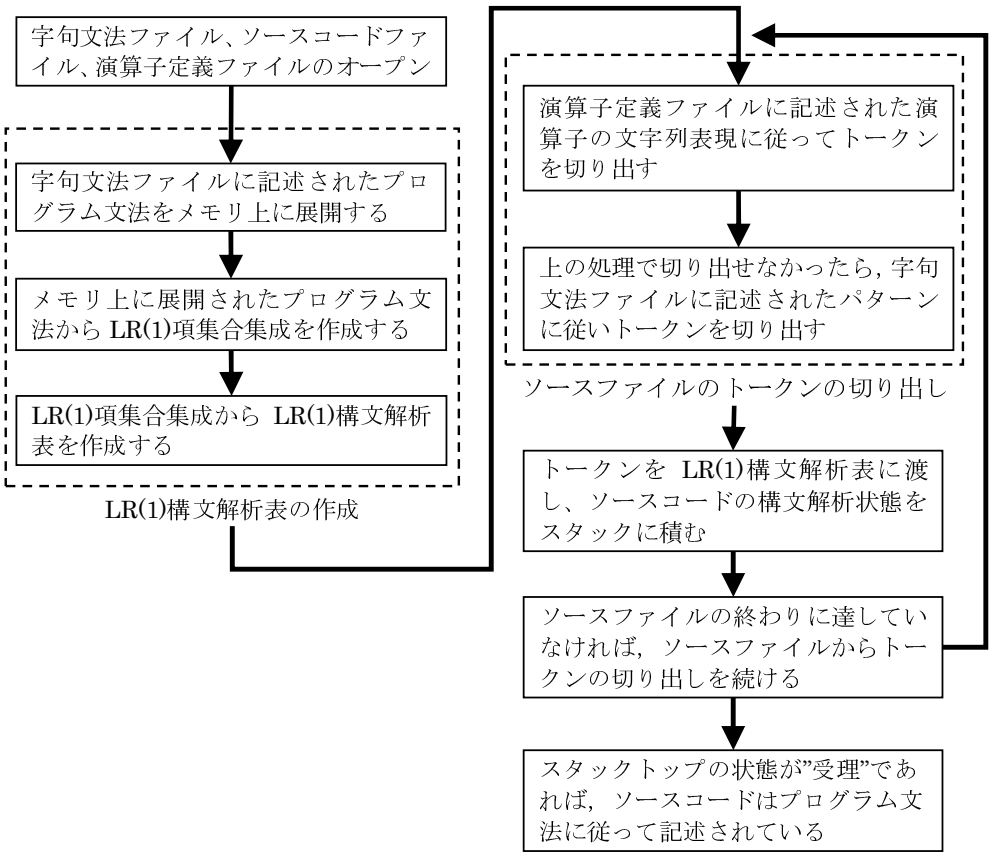


図 6　構文解析器の処理の概要

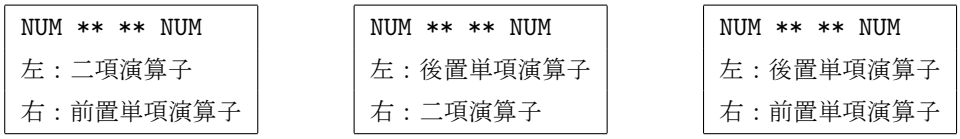
```
**     6   **の演算子は二項演算子であり前置単項演算子
*+*-   2   *+*-の演算子は前置単項演算子
```

図 7　演算子定義ファイルの記述例

3 つのフラグのうち，2 つを持つとき，その組み合わせは二項と前置単項，後置単項と二項，前置単項と後置単項の 3 パターンが存在する．**が 2 つのフラグをもった演算子だと仮定し，文字列 NUM ** ** NUM の解釈を考える．図 8 に示すように，どの場合も，構文は一通りに定まる．

演算子が 3 つ全てのフラグをもつときには，構文解析の結果は図 9 に示すように 2 つの解釈の仕方が存在し，プログラムの構文を一通りに定めることができない．

プログラムの構文を一通りに定めるために，演算子は 2 つまでのフラグを持つことがで



(a) 二項と前置単項　(b) 後置単項と二項　(c) 後置単項と前置単項

図 8　2 つのフラグを持つ演算子の解析

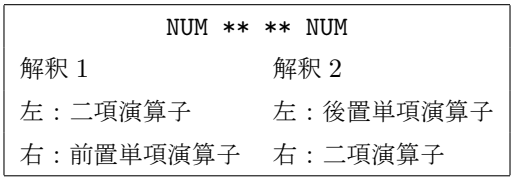


図 9　3 つのフラグをもつ演算子の解析

きる．

4.3　字句文法ファイルの記述方法

字句文法ファイルの記述例を図 10 に示した．ファイルには [VOCAB] 以下にトークンの名前とトークンのパターンを記述する．[SYNTAX] 以下にプログラム文法を記述する．ここでは例として構文規則 1 の構文規則を記述した．トークンの名前は英字の並びで記述し，トークンのパターンは正規表現で記述し，ダブルクォーテーションで囲む．プログラム文法の記述方法は，生成規則の左辺をコロンの前に記述し，生成規則の右辺はコロン以下から記述していく．コロンは生成規則の → に相当する．生成規則の文法記号は英字の並びで記述する．セミコロンは左辺に還元される生成規則の区切りを表す．左辺に還元される生成規則が複数行存在する場合は，右辺同士の区切りのためにカンマを使用する．図の例では E EQUAL T と ID はカンマで区切られて記述されており，どちらも S に還元される．なお，構文規則を記述する際には，開始記号を含む生成規則を最初に記述する．図 10 では Start が開始記号である．

自由演算子は +,-,*,/の文字の並びで表現されるから，すべて同じパターン [\+-*/]+ で表される．よって，演算子のフラグが 1 であるトークン NEWOP1 と，演算子のフラグが 2 であるトークン NEWOP2 は

```
NEWOP1     [\+-\*/]+
NEWOP2     [\+-\*/]+
```

と書くことができ，これを字句文法ファイルのトークンの定義部に記述したとする．上の定義に従ってソースコードからトークンを切りだしたとき，[\+-*/]+ の文字列はすべて NEWOP1

```
[VOCAB]          トークンの定義
ID               "[a-zA-Z]([a-zA-Z]|[0-9])*"
PLUS             "\+"
EQUAL            "="

[SYNTAX]         プログラム文法の定義
Start:           コロンは左辺と右辺の区切りを表す
   S;            セミコロンは右辺 Start に還元される生成規則の区切りを表す
S:
   E EQUAL E,    カンマは右辺が複数行にわたる場合に使用する
   ID;
E:
   E PLUS T,
   T;
T:
   ID;
```

図 10　字句文法ファイルの記述

に判定される．そのため，自由演算子はトークンのパターンを書くことができない．構文規則中に自由演算子の文法記号を記述するときは，次の定義済みの文法記号 NEWOP1，NEWOP2，NEWOP3，NEWOP4，NEWOP5，NEWOP6 を使用する．NEWOP に続く数字は演算子のフラグを表している．NEWOP 記号はパターンを字句文法ファイル中に記述しないが，終端文字として扱われる．二項演算子と前置単項演算子を表す記号 NEWOP6 と前置単項演算子を表す NEWOP2 を使用した構文規則の例を図 11 に示した．

ここで，たとえば，フラグが 2 である自由演算子は全て NEWOP2 の文法記号にまとめているから，同じフラグを持つ自由演算子は同じ優先順位をもつことがわかる．

5　構文解析器の実装

今まで説明した構文解析器を C 言語で実装した．実装の際に使用したデータ構造と関数群の説明を行う．

```
[VOCAB]
ID              "[a-zA-Z]([a-zA-Z]|[0-9])*"
PLUS            "\+"
EQUAL           "="

[SYNTAX]
Start:
   S;
S:
   E EQUAL E,
   ID;
E:
   E PLUS T,
   E NEWOP6 T,
   T;
T:
   NEWOP6 ID,
   NEWOP2 ID,
   ID;
```

図 11　自由演算子を含む構文規則

5.1　構文規則をメモリ上に展開

メモリ上に展開されたプログラム文法は下に示す matrix 構造体に格納される．matrix 構造体は integer 構造体のパラメータを持っている．matrix 構造体に構文規則を格納する際には，予め matrix 構造体の array パラメータを動的に確保し，integer 型の配列を用意する．matrix 構造体の array 配列には生成規則を一行ごと格納するため，array 配列は生成規則の行数だけ確保する必要がある．

```
struct integer{
    int *array;     //int 配列
    int size;       //配列の総行数
    int end;        //配列の使用行数
};
```

```
struct matrix{
    integer *array; //integer 配列
    int size;       //配列の総行数
    int end;        //配列の使用行数
};
```

生成規則は文法記号の並びによって構成されているが，matrix 構造体に生成規則を格納する際には，文法記号にその文法記号固有の番号を関連付け，その関連付けられた番号を文法記号の代わりに使用する．文法記号を番号に関連付けるためには，生成規則に現れた文法記号を登録する必要があり，下に示す dict 構造体に文法記号を登録していく．

```
struct dict{
    char *contents;     //辞書の内容
    int size;           //印字可能領域
    int tail;           //印字された領域の大きさ
    int pageCount;      //辞書のページ数（単語数）
};
```

プログラム文法が何行にわたって記述されているかはプログラムの実行前にはわからないので，matrix 構造体の array パラメータは領域が確定できない．そのため，array パラメータは動的に確保し，確保した領域がすべて使われたら拡張する．また，構文規則に使用されている文法記号の数も不明なので，dict 構造体の contents パラメータも動的に確保する．

構文規則 1 をメモリ上に展開したときの matrix 構造体と dict 構造体の内容を図 12，図 13 に示した．ここで，matrix 構造体の array パラメータは integer 型の領域を 4096 個動的に確保し，dict 構造体の contents パラメータは char 型の領域を 4096 個動的に確保した場合を考えた．

dict 構造体には構文規則 1 に使用された文法記号が格納されており，各文法記号はヌル文字によって区切られている．文法記号を dict 構造体に登録するとき，既に同じ綴りの文法記号が登録されていたら登録しない．よって dict 構造体内には構文規則 1 に使用された文法記号が一個ずつ登録される．文法記号に関係づけられた番号は，contents 内で何番目に文法記号が登録されているかを数えることによってわかる．たとえば，Start の文法記号に関係づけられた番号は 0 であり，EQUAL のトークン番号は 3 である．この番号を文法記号の代わりに使用し，図 13 のように matrix 構造体にプログラム文法が格納される．生成規則の終わりには，必ず文の終わりを表す-1 を入力する．

パラメータ名	パラメータの値														
contents	Start	\0	S	\0	E	\0	EQUAL	\0	ID	\0	PLUS	\0	T	\0	
size	4096														
tail	26														
pageCount	7														

図 12　構文規則 1 をロードしたときの dict 構造体の内容

パラメータ名	パラメータの値				
array[0]	0	1	-1		
array[1]	1	2	3	2	-1
array[2]	1	4	-1		
array[3]	2	2	5	6	-1
array[4]	2	6	-1		
array[5]	6	4	-1		
size	4096				
end	6				

図 13　構文規則 1 をロードしたときの matrix 構造体の内容

5.2　LR(1) 項集合集成のデータ構造

LR(1) 項はマーカーと先読み集合がひとまとめになっているので，LR(1) 項は item 構造体で表される．item 構造体の marker パラメータは line と offset の 2 つのパラメータを持っており，line が何行目の生成規則か意味し，offset は line 行の生成規則内でのマーカーの位置を意味する．マーカーが line=1，offset=2 のとき，構文規則 1 では，$S \rightarrow E\bullet = E$ を指している．LR(1) 項の先読み文字の数はプログラム実行前にはわからない．先読み文字を格納する領域は，itenger 構造体を用い動的に確保し，都合によっては領域を拡張できるようにする．

LR(1) 項集合は unit 構造体で表される．array パラメータは LR(1) 項の集合を格納する．また，LR(1) 項集合同士の goto 関係を表現するために，integer 型の arrow のパラメータをもっている．arrow は偶数列目に文法記号に関連付けられた番号が格納され，奇数列目にその文法記号を読んだ時に飛ぶ LR(1) 項集合の番号を格納している．arrow の例を図 14 に示す．図 14 の例では，4 の文法記号を読んだら 12 番目の LR(1) 項集合に飛び，6 の文法記号を読んだら 13 番目の LR(1) 項集合に飛ぶ．

```
struct point{
```

パラメータ名	パラメータの値							
array	1	1	2	2	4	12	6	13

図 14　LR(1) 項集合の goto 関係

```
    int line;       //生成規則の番号
    int offset;     //オフセット
};

struct item{
    point marker;       //マーカー
    integer lookahead;  //先読み
};

struct unit{
    item *array;        //LR(1) 項集合
    int size;           //array のサイズ
    int end;            //array の使用行数
    integer arrow;      //goto 関係
};
```

LR(1) 項集合集成は以下の coll 構造体によって表される．unit 配列をパラメータとして持つことで，LR(1) 項集合の集成を表現する．

```
struct coll{
    unit *array;        //LR(1) 項集合集成
    int size;           //array のサイズ
    int end;            //array の使用行数
};
```

5.3　FIRST 関数

FIRST 演算を行う FIRST 関数は LR(1) 項を受け取ると，その LR(1) 項から終端記号列を計算する．

```
void FIRST(
    integer *terminal,
    integer *checked,
    point marker,
    const integer *lookahead,
```

```
    const matrix *syntax
)
```

terminal 引数には終端記号列が格納される．checked 引数は，どの生成規則を導出したか記憶するために使用する．marker 引数と lookahead 引数は LR(1) 項のマーカーと先読み文字を渡し，syntax 引数は構文規則をメモリ上に展開した際のデータ構造を渡す．

terminal 引数は FIRST 関数に渡す前に動的に確保する必要がある．terminal には終端記号が格納されるので，構文規則に現れた文法記号の数だけ領域を確保すればよい．

checked 引数も terminal 引数と同様に，FIRST 関数に渡す前に int 配列を動的に確保し，構文規則の行数だけ領域を確保する．FIRST 関数は LR(1) 項から生成規則を導出し，その生成規則からさらに生成規則を導出する．

FIRST 関数は導出を繰り返し行い，この部分は再帰処理によって実現している．再帰の終了条件は，ある生成規則から導出を行ったとき，すでに別の過程でその生成規則が導出されていたときである．これは，例えば，$E \to \bullet E + T$ からの導出を考えると，

$$
\begin{aligned}
&E \to \bullet E + T \\
&E \to \bullet E + T \\
&E \to \bullet E + T \\
&\quad\vdots
\end{aligned}
$$

となり，際限なく導出が繰り返されることを防ぐためである．

5.4 Closure 関数

Closure 演算を行うには，Closure 関数と ClosureSingle 関数を使用する．ClosureSingle 関数は Closure 関数によって呼び出されるため，閉包演算を行う際には，Closure 演算のみに引数を与えれば良い．

```
void Closure(
    integer *terminal,
    integer *checked,
    unit *set,
    const unit *seed,
    const matrix *syntax
)
```

閉包演算は，FIRST 演算を行うため，terminal と checked を必要とする．set 引数には，次に説明する seed 引数によって作成された LR(1) 項集合が格納される．seed 引数は Closure 演算を行うための種となる LR(1) 項集合が格納されている．syntax 引数には構文規則のデー

タ構造を渡す.

terminal と checked は Closure 関数に渡す前に領域を動的に確保する必要があり，確保する領域の大きさは FIRST 関数の説明で述べたとおりである．Closure 関数内で，terminal と checked の領域を確保しないのは，Closure 関数は LR(1) 項集合集成を作成する際に，何度も呼び出され，そのたびに動的の領域の確保と解放をすることを嫌ったためである.

set 引数の array パラメータは Closure 関数に渡す前に動的に領域を確保する必要がある．確保する領域の大きさは，seed 引数に格納されている LR(1) 項の数と構文規則の行数の和である．これは，seed に格納されている LR(1) 項から導出される生成規則は構文規則の数を超えることはないためである.

Closure 関数はまず，set 配列に seed 配列の LR(1) 項を登録するので，set と seed の内容は同じになる．Closure 関数内では，seed に記載された LR(1) 項の数だけ ClosureSingle 関数を呼び出す．seed から一つずつ LR(1) 項を ClosureSingle 関数に渡し，その項から導出し，導出した生成規則の終端文字列を計算する．ClosureSingle 関数に LR(1) 項を渡すとき，item 構造体として渡すのではなく，set 配列の何番目に登録された LR(1) 項から計算をせよ，というように配列の番号を渡す.

```
void ClosureSingle(
    integer *terminal,
    integer *checked,
    unit *set,
    int itemNumber,
    const matrix *syntax
)
```

terminal，checked，set，syntax 引数は Closure 関数と同じ意味で用いられている．第 4 引数の itemNumber は set 配列の何番目に記載された LR(1) 項に対して演算を行うかを意味している．ClosureSingle 関数は FIRST 関数と同じように，導出を繰り返す関数なので，再帰処理により実現している．再帰処理の終了条件は，itemNumber で指定された LR(1) 項から導出された新たな LR(1) 項とその先読み文字が set 内に既に登録されていた時である.

5.5 LR(1) 項集合集成の作成関数

LR(1) 項集合集成の作成を行うには，MakeCollection 関数を使用する.

```
void MakeCollection(
    coll *collection,
    const matrix *syntax
)
```

collection 引数には LR(1) 項集合集成が格納される．syntax 引数には構文規則をメモリ上に展開した際のデータ構造を渡す．

MakeCollection 関数はその関数内で複数の関数を呼び出す．関数同士の関係を図 15 に示した．MakeCollection 関数は左辺値が開始記号である生成規則の閉包をとり，その初期 LR(1) 項集合から一つ記号を読み進めた初期シード（LR(1) 項集合集成）を MakeSeed 関数を用いて作成する．MakeCollection 関数は初期シードから LR(1) 項集合を一つずつ MakeCollectionRec 関数に渡し，MakeCollectionRec 関数内では，渡されたから LR(1) 項集合の閉包を取り，その閉包を取った LR(1) 項集合からシードを作成し，そのシードを MakeCollectionRec 関数に渡す．このとき，MakeArrow 関数によって，LR(1) 項集合集成の goto 関係を表す領域を作成し，AppendArrow 関数で goto 関係を記述していく．

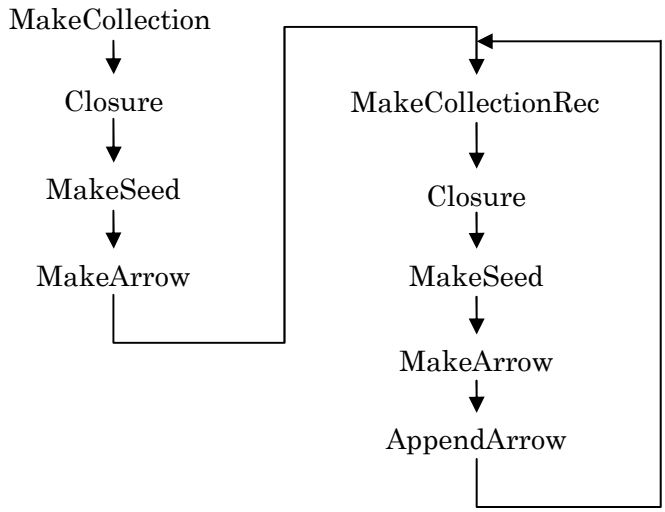


図 15　LR(1) 項集合集成を作成する関数

5.6　LR(1) 構文解析表の作成関数

構文解析表を作成する際は，MakeTable 関数を使用する．LR(1) 項集合集成を受け取ると，構文解析表の作成アルゴリズムに従い，構文解析表を作成する．

```
void MakeTable(
    integer *table,
    const coll *collection,
    const matrix *syntax
)
```

table 引数はサイズ 4 の配列のポインタを表している．table は action 表と goto 表を格納するために使用する．collectoin 引数は MakeCollection 関数で作成した LR(1) 項集合集成を格納したデータ構造を渡す．syntax 引数には構文規則のデータ構造を渡す．

table 引数は MakeTable 関数に渡す前に動的に確保する必要がある．table の領域が不足した場合には関数内でその都度拡張されるので，動的に確保する領域の大きさは適当でよい．action(0,7)=s6 というシフト命令を table に格納するときには，シフト命令を構成する 4 つの数字を図 16 のように格納する．table[0] には LR(1) 項集合の番号を格納し，table[1] には構文規則中に現れた文法記号と対応付けられた数字を格納する．table[2] は命令の種類を格納する．各命令にはシフト (0)，還元 (1)，受理 (2)，goto(3) の数字が割り当てられている．table[3] には動作によって飛ぶ LR(1) 項集合の番号を格納する．

table[0]	table[1]	table[2]	table[3]
0	7	0	6

図 16　action(0,7) = s6

5.7　構文解析表による構文解析

ソースファイルが構文規則に従って記述されているか判定するためには Accepter 関数を使用する．構文解析表と，ソースファイルのトークンの並びを渡すと，そのソースファイルの構文解析状態をスタックに積み，ソースファイルの構文解析を行う．

```
void Accepter(
    integer *stack,
    const dict *vocab,
    const char *word,
    const matrix *syntax,
    const integer *table
)
```

関数内でスタックを使用するために stack 引数にスタックを指定する．vocab 引数には構文規則内に現れた文法記号が収められた辞書を渡す．word 引数はソースファイルからトークンを切り出し，そのトークンの名前を渡す．syntax には構文規則を渡し，table 引数は MakeTable 関数によって作られた構文解析表を渡す．

stack 引数は，Accepter に渡す前に動的に領域を確保しておく．スタックの深さが足りなくなった場合には，関数内で，スタックの深さをその都度拡張するため，予め動的に確保する領域の大きさは適当でよい．

6　ソースファイルの構文解析例

図 11 の文法のもと，演算子として図 7 を使用する場合を考える．ソースコードが図 11 に従った `id = ***-a** **b` であった場合，構文解析は図 17 のように，最後に acc(受理) が表示される．ソースコードが `id = a***- ** **b` の場合，前置単項演算子である`***-`がまるで後置単項演算子のように記述されているから，これを構文解析すると図 18 のように解析の途中に，演算に一致しないと表示され，構文解析に失敗する．

(0, ID) = s24	(3, E) = g4
(24, EQUAL) = r8	(4, NEWOP6) = s12
(0, T) = g25	(12, NEWOP6) = s7
(25, EQUAL) = r5	(7, ID) = s8
(0, E) = g2	(8, $) = r6
(2, EQUAL) = s3	(12, T) = g13
(3, NEWOP2) = s9	(13, $) = r4
(9, ID) = s10	(3, E) = g4
(10, NEWOP6) = r7	(4, $) = r1
(3, T) = g14	(0, S) = g1
(14, NEWOP6) = r5	(1, $) = acc

図 17　受理されたソースコード

(0, ID) = s24
(24, EQUAL) = r8
(0, T) = g25
(25, EQUAL) = r5
(0, E) = g2
(2, EQUAL) = s3
(3, ID) = s11
演算に一致しない

図 18　受理されなかったソースコード

7　まとめ

プログラム文法内に定義された演算子のほかに，+,-,*,/の 4 つの文字の組み合わせを新たな演算子として使用でき，演算子の数に制限がないプログラム文法を考えた．また，プログラム文法内に定義された既存の演算子と新たに定義された演算子を用いて記述されたソースコードが，プログラム文法に従って書かれているか確認するための構文解析器を C 言語で作成した．二項演算子，前置単項演算子，後置単項演算子の 3 種類の演算子の内，新たな演算子は 2 つまでを属性として持てば，新たな演算子を用いてもプログラムの構文を一通りに定められることがわかった．既存の演算とし新たな演算子の区別をするために，演算子同士を括弧や空白を用いて，演算子を明確に分離できるようにする制約が生まれた．実装した構文解析器は不具合があり，構文規則が少ない文法は解析することができるが，C のような構文規則が多い文法に対しては，構文解析することができない．構文解析器の不具合を取り除くことが次の最初の

課題である．構文解析までは実装したが，意味解析においても，新たな演算子を使用できる文法を解析できるか調べることが今後の課題である．

謝辞

本研究を行うにあたって，細かく指導してくださった指導教員の西新幹彦准教授に感謝の意を表する．

参考文献

[1] A.V. エイホ，M.S. ラム，R. セシィ，J.D. ウルマン，“コンパイラ—原理 · 技法 · ツール—”，サイエンス社，p.287，2009.
[2] 中田育男，“コンパイラの構成と最適化”，朝倉書店，p.116，2009.
[3] 佐々政孝，“プログラミング言語処理系”，岩波書店，2004.