システムソフトウェア特論'17 # 8 構文解析 (3)

久野　靖 [*]

2017.9.15

1　上向き解析

1.1　シフト還元解析器

前回までで、構文木を上の方 (出発記号の側) から組み立てて行く下向き解析を扱いました。今回は構文木を下の方から組み立てていく**上向き解析** (bottom-up parsing) を扱います。上向き解析を行う構文解析器は通常、**シフト還元解析器** (shift-reduce parser) の形を取ります。そこで具体的な解析方式の説明に先立ち、シフト還元解析器の枠組について説明しましょう。

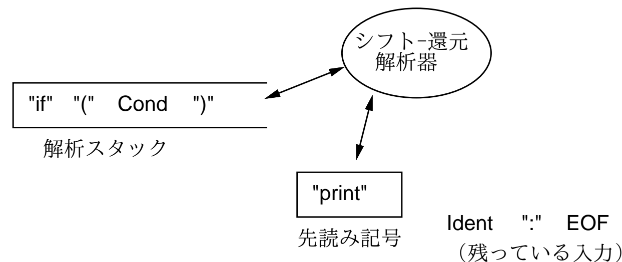

図 1: シフト還元解析器の枠組

シフト還元解析器の基本的な道具立てを図 1 に示します。解析には構文記号を積むスタック (解析スタック) 1 つと、先読み記号 1 個を使用します (より一般的には入力記号を n 個先まで見ることも考えられますが、動作の原理としては同じです)。そして、解析器の行う動作は必ず次の 2 つのうちいずれかです。

- シフト — 先読み記号をスタックに積み、入力を進める。
- 還元 — 生成規則 $A \rightarrow \alpha$ に対応し、α の長さ分スタックを取り降ろして代りに A を積む。

ただしこれでは終りがないので、文法で出発記号 S を左辺に持つ生成規則が 1 つだけになるようにし、その規則番号を 1 番とします。そして還元時に規則番号が 1 番だったら解析を終了します。以下では例題として次の文法を使用しますが、これも上記の条件を満たしています。

$$
\begin{array}{l}
prog ::= statlist \\
statlist ::= statlist\ stat \mid \epsilon \\
stat ::= ident\ \texttt{=}\ expr\ \texttt{;} \mid \texttt{read}\ ident\ \texttt{;} \mid \texttt{print}\ expr\ \texttt{;} \\
\quad\quad \mid \texttt{if}\ \texttt{(}\ cond\ \texttt{)}\ stat \mid \texttt{while}\ \texttt{(}\ cond\ \texttt{)}\ stat \mid \{\ statlist\ \} \\
cond ::= expr\ \texttt{<}\ expr \mid expr\ \texttt{>}\ expr \\
expr ::= ident \mid iconst
\end{array}
$$

[*]電気通信大学 情報理工学研究科

先の言語の最小限のプログラム「`read x; print x;`」をシフト還元解析器で解析する様子を図 2 に示します。これからわかるように、シフト還元解析器では入力は次々にスタックに移され、スタック上に生成規則の右辺と一致するものができると左辺の非端記号に置き換える、という形で解析が進んで行きます。

ここまでで「なるほど、スタック上に規則の右辺が現れたら還元、それ以外ではシフトをすればいいのか」と思われたかも知れませんが、それほど単純ではありません。上の例でも print 文の *ident* はスタックに積まれた直後に *expr* に還元されていますが、一方 read 文の *ident* はそのまま残されています。これはもちろん、read 文の規則の右辺が `read` *ident* `;` であるからですが、ともかく右辺が現れたらすぐ還元してよい、というものでないわけです。

このシフトと還元の選択を正しく行うところがシフト還元解析器の「きも」です。その手法としてここでは最も広く使われている **LR 解析器** (LR parser) を説明します。

LR の最初の L は LL と同様「left-to-right に 1 回入力を捜査するだけで解析を行う」という意味、2 文字目の R は「rightmost derivation の逆順の系列を生成する」という意味です。逆順なのは上向き、つまり出発記号から遠いところから構文木をつくっていくため出発記号が最後になるからで、逆順にする前で考えれば左側の導出が先に出てきます (入力を左から読むのでそれが普通です)。

実際、図 2 に現れる規則を下から順に並べて導出列を作ると、*prog* ⇒ *statlist* ⇒ *stat statlist* ⇒ *stat stat statlist* ⇒ *stat stat* ⇒ *stat* `print` *expr* `;` ⇒ *stat* `print` *ident* `;` ⇒ `read` *ident* `;` `print` *ident* `;` となり、確かに最右導出になっています。

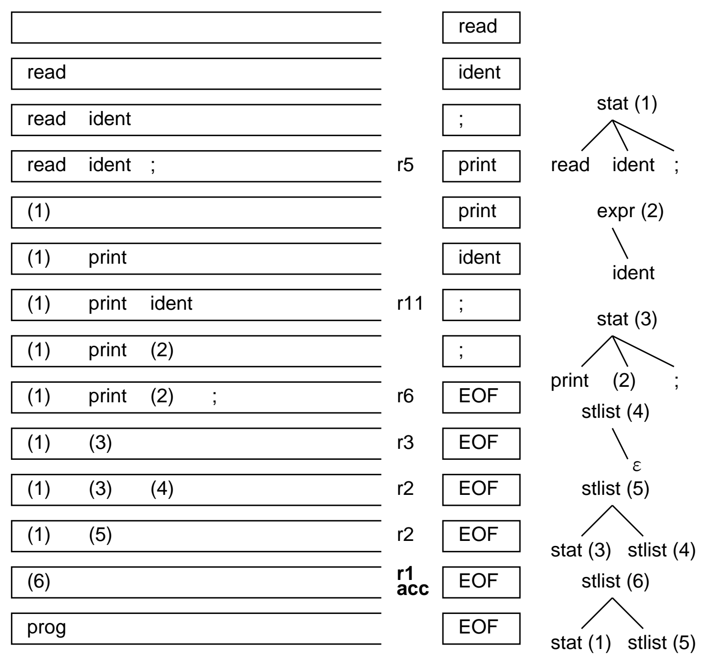

図 2: シフト還元解析器による構文解析

1.2 LR オートマトンと項

先の *ident* を還元するかどうかは、先に `read` をシフトしたか `print` をシフトしたかによって選択が違っていました。「置かれた状態によって同じ入力に対する動作が変化する」ことが問題ですが、この種の問題はオートマトンで表現するのが適しています。

ここでは **LR オートマトン**とよばれるものを使いますが、それは字句解析のときの有限オートマトンとはだいぶ違います。まず、LR オートマトンでは各状態が「構文規則の右辺のどの位置にいるか」に対応します。これを表すために、構文規則の右辺の任意の位置に • を記入して「現在いる位置」を

表します。これを**項** (item) と呼びます (より厳密には LR(0) 項。0 は項の中に先読み記号の情報が含まれていないことを意味する)。

例えば、規則 $stat \rightarrow \texttt{if}\ (\ cond\ )\ stat$ からは 6 つの項ができます (項は `[]` で囲んで表す)。

$[\ stat \rightarrow \bullet\ \texttt{if}\ (\ cond\ )\ stat\ ]$
$[\ stat \rightarrow \texttt{if}\ \bullet\ (\ cond\ )\ stat\ ]$
$[\ stat \rightarrow \texttt{if}\ (\ \bullet\ cond\ )\ stat\ ]$
$[\ stat \rightarrow \texttt{if}\ (\ cond\ \bullet\ )\ stat\ ]$
$[\ stat \rightarrow \texttt{if}\ (\ cond\ )\ \bullet\ stat\ ]$
$[\ stat \rightarrow \texttt{if}\ (\ cond\ )\ stat\ \bullet\ ]$

1.3 LR(0) オートマトンの作成

それでは、上向き構文解析のためのオートマトンの構成方法について説明しましょう。字句解析の決定性オートマトンでは、1 つの状態は複数の正規表現のそれぞれ特定の場所に「並行して」対応していました。ここでも同様なことを考える必要があります。例えば、次の状態にいるものとします。

$[\ stat \rightarrow \texttt{if}\ (\ \bullet\ cond\ )\ stat\ ]$

つまり $cond$ の直前にいるわけですが、そこで次の生成規則を参照します。

$cond \rightarrow expr\ \texttt{<}\ expr$
$cond \rightarrow expr\ \texttt{>}\ expr$

ということは、上の状態は同時に次の 2 つの状態にも「並行して」存在していることになります。

$[\ cond \rightarrow \bullet\ expr\ \texttt{<}\ expr\ ]$
$[\ cond \rightarrow \bullet\ expr\ \texttt{>}\ expr\ ]$

さらに次の規則も参照します。

$expr \rightarrow ident$
$expr \rightarrow ionst$

ということは、これらの状態にいるということは次の状態にもいることになります。

$[\ expr \rightarrow \bullet\ ident\ ]$
$[\ expr \rightarrow \bullet\ iconst\ ]$

このように「並行していることになる」項の集合を (LR(0) 項集合の) **閉包** (closure) と呼びます。したがって、LR オートマトンの各状態は項の集合で、なおかつ閉包になっている必要があります。そして、状態間の遷移については、$\bullet$ の直後にある記号 (端記号・非端記号の双方) をラベルに持つ遷移により、$\bullet$ がその記号の後ろに移動した項 (の閉包) の状態に移ることになります。

先の文法に対応する LR(0) オートマトンを作成すると、図 3 のようになります。中身の項集合は表 1 に別に示しています。

この LR オートマトンを用いた解析過程を手でシミュレートしてみましょう (図 4)。プログラム例は再び「`read x; print x;`」です。

今度は道具立てとして、解析スタックと先読み記号に加えて「現在の状態」が必要なので、解析スタックには構文記号と LR オートマトンの状態を交互に載せます。まず状態 S1 から開始し、入力記号を `read`、*ident*、`;`、とシフトしながら S3、S24、S25 と遷移すると同時に、これらの記号と状態をそれぞれスタックに載せていきます。

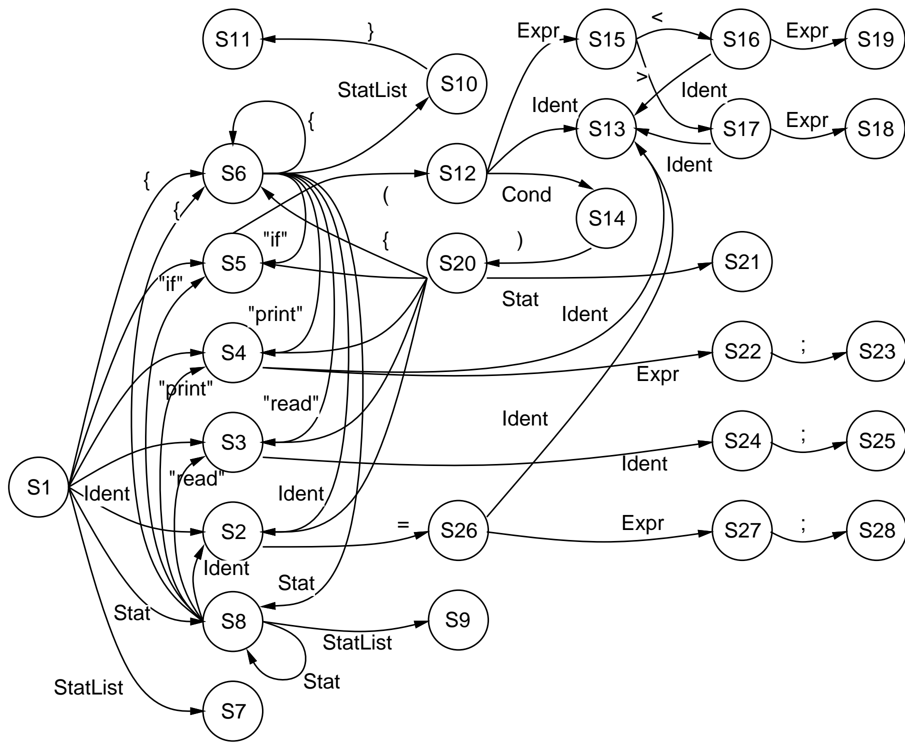

図 3: LR(0) オートマトン

スタック	動作	入力
S1	s3	read
S1 read S3	s24	ident
S1 read S3 ident S24	s25	;
S1 read S3 ident S24 ; S25	r5	print
S1 stat	s8	print
S1 stat S8	s4	print
S1 stat S8 print S4	s13	ident
S1 stat S8 print S4 ident S13	r11	;
S1 stat S8 print S4 expr	s22	;
S1 stat S8 print S4 expr S22	s23	;
S1 stat S8 print S4 expr S22 ; S23	r6	EOF
S1 stat S8 stat	s8	EOF
S1 stat S8 stat S8	r3	EOF
S1 stat S8 stat S8 stlist	s9	EOF
S1 stat S8 stat S8 stlist S9	r2	EOF
S1 stat S8 stlist	s9	EOF
S1 stat S8 stlist S9	r2	EOF
S1 stlist	s7	EOF
S1 stlist S7	**r1 acc**	EOF

図 4: LR 解析器による解析のようす

表 1: LR(0) オートマトンの各状態

状態	項
S1:	$[\mathit{prog} \to \bullet \mathit{statlist}]$
	$[\mathit{statlist} \to \bullet \mathit{stat}\mathit{statlist}]$
	$[\mathit{statlist} \to \bullet\ ]$
	$[\mathit{stat} \to \bullet \mathit{ident}\ \texttt{=}\ \mathit{expr}\ \texttt{;}]$
	$[\mathit{stat} \to \bullet\ \texttt{read}\ \mathit{ident}\ \texttt{;}]$
	$[\mathit{stat} \to \bullet\ \texttt{print}\ \mathit{expr}\ \texttt{;}]$
	$[\mathit{stat} \to \bullet\ \texttt{if}\ \texttt{(}\ \mathit{cond}\ \texttt{)}\ \mathit{stat}]$
	$[\mathit{stat} \to \bullet\ \{\ \mathit{statlist}\ \}]$
S2:	$[\mathit{stat} \to \mathit{ident}\bullet\ \texttt{=}\ \mathit{expr}\ \texttt{;}]$
S3:	$[\mathit{stat} \to\ \texttt{read}\ \bullet \mathit{ident}\ \texttt{;}]$
S4:	$[\mathit{stat} \to\ \texttt{print}\ \bullet \mathit{expr}\ \texttt{;}]$
	$[\mathit{expr} \to \bullet \mathit{ident}]$
	$[\mathit{expr} \to \bullet \mathit{iconst}]$
S5:	$[\mathit{stat} \to\ \texttt{if}\ \bullet\ \texttt{(}\ \mathit{cond}\ \texttt{)}\ \mathit{stat}]$
S6:	$[\mathit{statlist} \to \bullet \mathit{stat}\mathit{statlist}]$
	$[\mathit{statlist} \to \bullet\ ]$
	$[\mathit{stat} \to \bullet \mathit{ident}\ \texttt{=}\ \mathit{expr}\ \texttt{;}]$
	$[\mathit{stat} \to \bullet\ \texttt{read}\ \mathit{ident}\ \texttt{;}]$
	$[\mathit{stat} \to \bullet\ \texttt{print}\ \mathit{expr}\ \texttt{;}]$
	$[\mathit{stat} \to \bullet\ \texttt{if}\ \texttt{(}\ \mathit{cond}\ \texttt{)}\ \mathit{stat}]$
	$[\mathit{stat} \to \bullet\ \{\ \mathit{statlist}\ \}]$
	$[\mathit{stat} \to\ \{\ \bullet \mathit{statlist}\ \}]$
S7:	$[\mathit{prog} \to \mathit{statlist}\bullet\ ]$
S8:	$[\mathit{statlist} \to \bullet \mathit{stat}\mathit{statlist}]$
	$[\mathit{statlist} \to \mathit{stat} \bullet \mathit{statlist}]$
	$[\mathit{statlist} \to \bullet\ ]$
	$[\mathit{stat} \to \bullet \mathit{ident}\ \texttt{=}\ \mathit{expr}\ \texttt{;}]$
	$[\mathit{stat} \to \bullet\ \texttt{read}\ \mathit{ident}\ \texttt{;}]$
	$[\mathit{stat} \to \bullet\ \texttt{print}\ \mathit{expr}\ \texttt{;}]$
	$[\mathit{stat} \to \bullet\ \texttt{if}\ \texttt{(}\ \mathit{cond}\ \texttt{)}\ \mathit{stat}]$
	$[\mathit{stat} \to \bullet\ \{\ \mathit{statlist}\ \}]$
S9:	$[\mathit{statlist} \to \mathit{stat}\mathit{statlist}\bullet\ ]$
S10:	$[\mathit{stat} \to\ \{\ \mathit{statlist}\bullet\ \}]$
S11:	$[\mathit{stat} \to\ \{\ \mathit{statlist}\ \}\ \bullet\ ]$
S12:	$[\mathit{stat} \to\ \texttt{if}\ \texttt{(}\ \bullet \mathit{cond}\ \texttt{)}\ \mathit{stat}]$
	$[\mathit{cond} \to \bullet \mathit{expr}\ \texttt{<}\ \mathit{expr}]$
	$[\mathit{cond} \to \bullet \mathit{expr}\ \texttt{>}\ \mathit{expr}]$
	$[\mathit{expr} \to \bullet \mathit{ident}]$
	$[\mathit{expr} \to \bullet \mathit{iconst}]$
S13:	$[\mathit{expr} \to \mathit{ident}\bullet\ ]$
S14:	$[\mathit{stat} \to\ \texttt{if}\ \texttt{(}\ \mathit{cond}\bullet\ \texttt{)}\ \mathit{stat}]$
S15:	$[\mathit{cond} \to \mathit{expr}\bullet\ \texttt{<}\ \mathit{expr}]$
	$[\mathit{cond} \to \mathit{expr}\bullet\ \texttt{>}\ \mathit{expr}]$
S16:	$[\mathit{cond} \to \mathit{expr}\ \texttt{<}\ \bullet \mathit{expr}]$
	$[\mathit{expr} \to \bullet \mathit{ident}]$
	$[\mathit{expr} \to \bullet \mathit{iconst}]$
S17:	$[\mathit{cond} \to \mathit{expr}\ \texttt{>}\ \bullet \mathit{expr}]$
	$[\mathit{expr} \to \bullet \mathit{ident}]$
	$[\mathit{expr} \to \bullet \mathit{iconst}]$
S18:	$[\mathit{cond} \to \mathit{expr}\ \texttt{>}\ \mathit{expr}\bullet\ ]$
S19:	$[\mathit{cond} \to \mathit{expr}\ \texttt{<}\ \mathit{expr}\bullet\ ]$
S20:	$[\mathit{stat} \to \bullet \mathit{ident}\ \texttt{=}\ \mathit{expr}\ \texttt{;}]$
	$[\mathit{stat} \to \bullet\ \texttt{read}\ \mathit{ident}\ \texttt{;}]$
	$[\mathit{stat} \to \bullet\ \texttt{print}\ \mathit{expr}\ \texttt{;}]$
	$[\mathit{stat} \to \bullet\ \texttt{if}\ \texttt{(}\ \mathit{cond}\ \texttt{)}\ \mathit{stat}]$
	$[\mathit{stat} \to\ \texttt{if}\ \texttt{(}\ \mathit{cond}\ \texttt{)}\ \bullet \mathit{stat}]$
	$[\mathit{stat} \to \bullet\ \{\ \mathit{statlist}\ \}]$
S21:	$[\mathit{stat} \to\ \texttt{if}\ \texttt{(}\ \mathit{cond}\ \texttt{)}\ \mathit{stat}\bullet\ ]$
S22:	$[\mathit{stat} \to\ \texttt{print}\ \mathit{expr}\bullet\ \texttt{;}]$
S23:	$[\mathit{stat} \to\ \texttt{print}\ \mathit{expr}\ \texttt{;}\ \bullet\ ]$
S24:	$[\mathit{stat} \to\ \texttt{read}\ \mathit{ident}\bullet\ \texttt{;}]$
S25:	$[\mathit{stat} \to\ \texttt{read}\ \mathit{ident}\ \texttt{;}\ \bullet\ ]$
S26:	$[\mathit{stat} \to \mathit{ident}\ \texttt{=}\ \bullet \mathit{expr}\ \texttt{;}]$
	$[\mathit{expr} \to \bullet \mathit{ident}]$
	$[\mathit{expr} \to \bullet \mathit{iconst}]$
S27:	$[\mathit{stat} \to \mathit{ident}\ \texttt{=}\ \mathit{expr}\bullet\ \texttt{;}]$
S28:	$[\mathit{stat} \to \mathit{ident}\ \texttt{=}\ \mathit{expr}\ \texttt{;}\ \bullet\ ]$

S25 まで来ると「行き止り」になってしまいますが、この状態は生成規則 R6 の一番最後に • をもった項から成っているので、R6 に従って還元を行います。具体的には、R6 の右辺の記号数の倍 (状態と記号を対にして載せたから) だけスタックを取り降ろし (S1 だけが残る)、左辺 (つまり *stat*) を積みます。これは「S1 で *stat* が読めた」という状況に対応しています。そこで S1 から *stat* のラベルがついた遷移を行い、S8 へ来ます。

次は S8 で `print` なので S4 に行き、次が *ident* で S13 です。すると今度は先程と違い、ここで「行き止り」になって *ident* が *expr* に還元さて `S4` へ戻り、ここから改めて S22 へ進み、ここでようやく ; がシフトされて S23 へ進み、print 文全体が還元されて S8 へ戻ります。このように、「シフトすべきか、還元すべきか」という選択はオートマトンの状態を通じて的確に指示されるわけです。

このあとはもう次が EOF なので、ϵ を *statlist* に還元し、それと前に認識した *stat* を併せて *statlist* に還元することを繰り返していき、S1 まで戻ったところで *statlist* により S7 へ行きます。次の動作は最初の規則 R1 による還元なので、これで終了となります。

1.4 SLR(1) 解析器

先の説明では「行き止り」になったとき、• が最後にある項 (「行き止り」だから必ずそういう規則がある) を用いて還元していました。

しかし、行き止りではないが • が最後に持つ項も含んでいる状態もあり得ますし、還元に使える項が複数ある可能性もあります。これらの場合の動作はオートマトンだけでは決まらず、別の情報を用いなければなりません。

1 つの考え方として、選択を *Follow* 集合に基づいて行うという方針が挙げられます。つまり、ある状態に [$A \to \alpha\bullet$] なる項が含まれ、かつ次の先読み記号 t が $Follow(A)$ に含まれている時のみ、この項に対応する規則による還元を行うわけです。

入力が構文的に正しいならば A の後に来る端記号は $Follow(A)$ に含まれるはずなので、もしそうでないなら還元してはならないのは明らかです。この方針に基づく解析器を SLR(1) 解析器と呼びます (最初の S は Simlpe の意)。

LR 解析器の動作は一般に、状態と先読み記号の対から「シフトしてどの状態へ行く」か「どの規則で還元する」のいずれかを指示する**動作表** (action table) と、状態と非端記号の対から「還元した後どの状態に進むか」を指示する**行き先表** (goto table) の組からなる **LR 構文解析表** LR parsing table で表せます。先の例に対応する解析表を図 5 に示します。

先の図 4 の動作は、この解析表に従って解析した場合を反映したものです。どこに *Follow* 集合の情報が使われていたか分かりますか? それは、$statlist \to \epsilon$ による還元を行なうのは先読み記号が EOF か } のいずれかの時だけ、という部分です。この制約がないと、空列はいつでも *statlist* に還元できてしまうので、それをどこで適用すべきか分からないことになります。

図 4 についてもう 1 つ補足です。先にはわかりやすさのためスタック上に構文記号と状態を交互に積むとしていましたが、実際には構文記号は必要としません。その理由は、オートマトンの状態には「現在どのような入力列を読んだところか」という情報が必要なだけ含まれているためです。このため、実際には解析スタック上には状態だけ積めば十分です。実際、動作表ではシフト時には「どの状態に進む」かは直接示されていますし、還元時には還元された記号が分かるのでそれを状態と照らし併せて次の状態が決められます。

1.5 正準 LR(1) 解析器と LALR(1) 解析器

SLR(1) 構文解析では還元の選択を *Follow* 集合によっていますが、*Follow* はもともと構文全体を通してどこかで後続記号になっているかを見るものなので、細かい文脈の区別には不向きです。

例えば次のような文法を考えてみます。

$$
\begin{aligned}
prog &::= stat \\
stat &::= left \;\texttt{=}\; right \mid right
\end{aligned}
$$

	ident	=	;	(	)	{	}	<	>	eof	if	read	print	stlist	stat	cond	expr
S1	s2					s6	r3			r3	s5	s3	s4	7	8		
S2		s26															
S3	s24																
S4	s13																22
S5				s12													
S6	s2					s6	r3			r3	s5	s3	s4	9	8		
S7										accept							
S8	s2					s6	r3			r3	s5	s3	s4	9	8		
S9							r2			r2							
S10							s11										
S11	r8					r8	r8			r8	r8	r8	r8				
S12	s13															14	15
S13			r11		r11			r11	r11								
S14					s20												
S15								s16	s17								
S16	s13																18
S17	s13																18
S18					r10												
S19					r9												
S20	s2					s6					s5	s3	s4		21		
S21	r7					r7	r7			r7	r7	r7	r7				
S22			s23														
S23	r6					r6	r6			r6	r6	r6	r6				
S24			s25														
S25	r5					r5	r5			r5	r5	r5	r5				
S26	s13																27
S27			s28														
S28	r4					r4	r4			r4	r4	r4	r4				

動作表　行先表

図 5: SLR(1) 構文解析表

$$left \ ::= \ ident \ | \ * \ right$$
$$right \ ::= \ left$$

これはＣのように「式は単独でも文になり得、また代入の左辺はポインタ参照でもよい」ような言語の文法を簡略化したものになっています。これから先のプログラムにより SLR(1) 構文解析器をつくろうとすると、$left$ を認識したあとに「=」が先読み記号になった場合、シフトすべきか $right \to left$ で還元すべきかが判別できません。これは、$right$ =の前に来ることが可能なため $Follow(right)$ に=が含まれているからです。しかし実際には代入記号の直前にいるのだから、$left$ を $right$ に還元するのは誤りです。

このような $Follow$ による「おおざっぱな判定」の弱点を克服するため、最初から状態の中に先読み記号も含めて計算を行うことを考えます。そこで、今度は項の中に次のようにして先読み記号を含めることにします。

[$prog \to \bullet \ stat$; EOF]
[$right \to left \bullet$; =]

このような項を **LR(1) 項**と呼びます (1 というのは先読み記号の長さを示す)。そしてあとは、この項どうしの遷移により、これまで通りに解析表を作ります。これを正準 (cannonical)LR(1) 構文解析表、それに従って動作する解析器を正準 LR(1) 解析器と呼びます。

正準 LR(1) 文法は SLR(1) よりも広い文法クラスを含んでいます。実は、正準 LR(1) 文法は左から右へ後戻りなしで解析可能な最大の文法クラスとなっていることが知られています。

一方、正準 LR(1) 解析器の問題点はその状態数が非常に多くなるということです。それは文法における端記号の数が n として、項の数が SLR(1) の n 倍になっていることに起因します。このため、正準 LR(1) 解析器をそのままコンパイラに採用するのは実用的でないとされます。

実用のコンパイラでは、SLR(1) より受け入れられる文法の範囲が広く、しかも LR(1) ほどには状態数が多くない実用的な構文解析器として、LALR(1) 解析器が多く使われています。

これは、LR(1) でオートマトンを構成したあと、先読み記号部分 (; の後ろ) を取り除いて同じになる状態を互いに併合することで構成されます (併合をおこなうときに行き先となる状態が食い違って併合できないなどのことは起きないことが知られています)。

併合をおこなった後の状態数は LR(0) オートマトンと同じになるため、状態数の問題が解消され、なおかつ先読み記号の情報が使えるため、SLR(1) よりは広いクラスの文法が扱えます (ただし併合により正確さが減じるため、正準 LR(1) よりは狭くなる)。

なお、解析表の生成時だけとは言え、正準 LR(1) オートマトンをいったん作るのでは計算の手間が大きくなりますが LR(1) オートマトンを経ずに直接必要な状態のみを生成しながら合せて先読み記号の情報を計算する手順が知られています。

演習 8-1 例題のオートマトンや構文解析表で別のプログラムを与えたとき確かに解析ができることを確認してみなさい。解析のようすを記録すること。

演習 8-2 簡単な言語を BNF で記述し、その LR(0) オートマトンを構成してみなさい。SLR(1) 構文解析表までできるとなおよい。これらを使って簡単なプログラムを解析するようすを示すこと。

2 曖昧な文法の活用

ここまでは曖昧でない文法のみを想定して来ましたが、場合によっては曖昧な文法を許す方が人間、コンピュータ双方にとって有利なこともあります。その代表的な例として**ぶらさがり else**(dangling else) があります。多くの言語では `if` 文に対して次の構文定義を用いています。

$$stat \rightarrow \texttt{if}\ cond\ \texttt{then}\ stat\ \texttt{else}\ stat$$
$$stat \rightarrow \texttt{if}\ cond\ \texttt{then}\ stat$$

この場合、`if C1 then if C2then A else B` という文は次の 2 通りに解釈でき、曖昧です。

```
if C1 then ( if C2 then A else B )  -- (1)
if C1 then ( if C2 then A ) else B  -- (2)
```

そこで「ただし、おのおのの `else` はまだ対応する `else` をもたない最も近い `if` に対応させる」という規則を (自然言語で) 付すのが恒例です (これにより上の解釈 (2) は削除される)。

なお、この例の場合は代りに文法を次のように直しても曖昧さは解消できますが、明らかに繁雑になります。

$$balstat \rightarrow \texttt{if}\ cond\ \texttt{then}\ balstat\ \texttt{else}\ stat\ \texttt{|}\ \text{各種の文}$$
$$stat \rightarrow \texttt{if}\ cond\ \texttt{then}\ stat\ \texttt{|}\ balstat$$

このように文法が曖昧なまま構文解析器を作ろうとすると、LR 構文解析の場合に**シフト還元衝突** (shift-reduce conflict) や**還元還元衝突** (reduce-reduce conflict) が発生します。つまり、構文解析表を構成しているときに、特定の位置の動作表に「シフト」「還元」の両方の動作が記入できることになったり、複数の規則による「還元」が記入できるようになることを言います。

たとえば、ぶらさがり else の場合、文法の曖昧さに対応して「`then` 部だけから成る `if` 文を全て読み終った時点とも、`else` 部まである `if` 文の `else` の直前とも取れる」状態が現れ、そこでシフト還元衝突となります。

そこで、このような状態では常にシフトする—つまり、後者の解釈を取る—ことにして解析表を作っていまうことができます。これによって構成される解析器は上述の「`else` を最も近い `if` に対応させる」解析動作を行います。LL(1) 解析器の場合でもこれと同様なことが行えます。

曖昧な文法が役に立つもう 1 つの代表例は式と演算子の構文です。「$expr \rightarrow expr\ \texttt{+}\ expr$」という文法は「`1 + 2 + 3`」を「`(1 + 2) + 3`」とも「`1 + (2 + 3)`」とも解釈できます。

この曖昧さを除くには、これまでに見てきたように、式-項-因子のように演算子の強さごとに別の非端記号を使用すればよいわけですが、これも読みやすくはありません。ここでは+は左結合的なので、前者を選択し、還元を行うように解析表を作ればよいわけです。同様に、演算子間の順位や右結合なども、そのことを予め考慮して解析表を作ることで対応できるわけです。

曖昧な文法を使用することの利点は 1 つは、構文記号の数が少なく人間にとってわかりやすくなることですが、同時に解析器の側でも状態数が少なくてすみ、また余分な還元がなくなるので効率よく解析が行えるという利点があります。

3 構文解析器生成系

3.1 Cup とその構文定義

解析表とドライバが分かれた形の構文解析器では、コンパイラ作成者が自ら解析表の計算を行うことはほとんどなく、構文記述を入力すると解析表を作成してくれるツールを使用するのが普通です。これを構文解析器生成系 (parser generator) と呼びます。

たとえば、Unix に古くから備わっている **Yacc** やそのフリーソフト版である Bison は LALR(1) 解析器を生成する生成系で、出力は C 言語のコードになっています。ここでは言語として Java を使っていて、JFlex との相性もよいことから、Cup と呼ばれる生成系を使ってみます (細かい書き方は Yacc と違っていますが、大筋は同じです)。

まず先頭の `import` は CUP のライブラリにアクセスするため必須のもので、そのままコードの先頭の入ります。そのあと、端記号および非端記号として使う名前を宣言します。ここでは端記号を大文字、非端記号を小文字にしています。

その次ですが、上にのべたように曖昧な文法を使うため、優先順位の低いものから演算子の端記号を列挙します。また、`nonassoc`、`left`、`right` により結合のしかた (非結合、左結合、右結合) を指定します。左結合は「$x \star y \star z \rightarrow (x \star y) \star z$」の意味で、非結合は「$x \star y \star z$」のようにつなげては書けないことを意味します (比較演算子は通常そうですね)。なお、最後の `UMINUS` という端記号は最も優先順位が高くなりますが、実際には現れません。これについてはすぐ後で述べます。

```
import java_cup.runtime.*;

terminal READ, PRINT, IF, WHILE, ASSIGN, SEMI, LPAR, RPAR, LBRA, RBRA;
terminal GT, LT, PLUS, MINUS, ASTER, SLASH, UMINUS;
terminal ICONST, IDENT;

non terminal prog, statlist, stat, expr;

precedence nonassoc LT, GT;
precedence left PLUS, MINUS;
precedence left ASTER, SLASH;
precedence left UMINUS;

prog ::= statlist
         ;
statlist ::= stat statlist
         |
         ;
stat ::= IDENT ASSIGN expr SEMI
```

```
        | READ IDENT SEMI
        | PRINT expr SEMI
        | IF LPAR expr RPAR stat
        | WHILE LPAR expr RPAR stat
        | LBRA statlist RBRA
        ;
expr ::= expr PLUS expr
        | expr MINUS expr
        | expr ASTER expr
        | expr SLASH expr
        | expr GT expr
        | expr LT expr
        | MINUS expr  %prec UMINUS
        | IDENT
        | ICONST
        | LPAR expr RPAR
        ;
```

さて文法ですが、見て分かるとおり普通の BNF で、ただし上述のように式のところは曖昧な文法を使っています (なので簡潔です)。「MINUS exp %prec UMINUS」というところが謎ですが、これは単項のマイナス演算子で、端記号としては MINUS が使われますが、ただし順位は他の演算よりも高い UMINUS の強さにします、という指定です。

演習 8-3 上の例題を Cup で処理してみなさい。うまくいったら、自分独自の簡単な言語を BNF で定義し、次に Cup で構文記述を書いて変換してみなさい。エラーがあれば直すこと。

3.2 JFlex と Cup の連携

上記の文法指定を cup コマンドで処理すると、parser.java と sym.java という 2 つのファイルが生成されます。前者はパーサのファイルですが、後者は Token.java のように端記号の番号を定義するためのもので、使い方も同様です。これを用いてこの言語用の JFlex ファイルを作成します。

まず冒頭で Cup と同じライブラリの import を行ないます。これは、字句解析器が返すものが Symbol オブジェクトである必要があるためです。「%cup」という指定は字句解析用のメソッド名を Cup 用の名前にするために必要です。次の 3 行は、Symbol オブジェクトを生成するためのオブジェクトを用意するものです。%eofval の 3 行は EOF になったときに EOF シンボルを返す指定です。

```
import java_cup.runtime.*;
%%
%class Lexer
%cup
%{
  SymbolFactory sf = new DefaultSymbolFactory();
%}
%eofval{
  return sf.newSymbol("EOF", sym.EOF);
%eofval}
L = [A-Za-z]
```

```
D = [0-9]
Ident = {L}({L}|{D})*
Iconst = [-+]?{D}+
Blank = [ \t\n]+
%%
\+        { return sf.newSymbol("PLUS", sym.PLUS); }
\-        { return sf.newSymbol("MINUS", sym.MINUS); }
\*        { return sf.newSymbol("ASTER", sym.ASTER); }
\/        { return sf.newSymbol("SLASH", sym.SLASH); }
\<        { return sf.newSymbol("LPAR", sym.LT); }
\>        { return sf.newSymbol("RPAR", sym.GT); }
\(        { return sf.newSymbol("LPAR", sym.LPAR); }
\)        { return sf.newSymbol("RPAR", sym.RPAR); }
\{        { return sf.newSymbol("LPAR", sym.LBRA); }
\}        { return sf.newSymbol("RPAR", sym.RBRA); }
\;        { return sf.newSymbol("SEMI", sym.SEMI); }
\=        { return sf.newSymbol("ASSIGN", sym.ASSIGN); }
if        { return sf.newSymbol("IF", sym.IF); }
while     { return sf.newSymbol("WHILE", sym.WHILE); }
read      { return sf.newSymbol("READ", sym.READ); }
print     { return sf.newSymbol("PRINT", sym.PRINT); }
{Ident}   { return sf.newSymbol("IDENT", sym.IDENT, yytext()); }
{Iconst}  { return sf.newSymbol("ICONST", sym.ICONST, yytext()); }
{Blank}   { /* ignore */ }
```

その後は基本的にこれまでの JFlex と同じですが、ただし返す値は上述のように Symbol オブジェクトを返すようにします。そのときのパラメタとして、1 番目は表示用の文字列、2 番目がトークンの値 (クラス sym で定義されているものを使います)、3 番目は $Ident$ と $Const$ についてのみ、値としてトークンの文字列を渡しています。これは後で使います。

main は次のようになります。ここでは何も動作を指定していないので、構文エラーがあればエラーが出るというだけです (つまり認識器です)。

```
import java.util.*;
import java.io.*;
import java_cup.runtime.*;

public class Sam81 {
  public static void main(String[] args) throws Exception {
    parser p1 = new parser(new Lexer(new FileReader(args[0])));
    p1.parse();
  }
}
```

実行のようすを一応のせましょう。

```
% cup sam81.cup
(Cup のメッセージ)
% jflex sam81.jflex
```

```
(JFlex のメッセージ)
% javac Sam81.java
% java Sam81 test.min  ←後述
%                      ←出力なし: エラーがないことは分かる
```

3.3 属性とアクション

前節まででは、構文解析はしても何も動作がついていないので、出力が何もありませんでした。Cup で (Yacc や Bison もそうですが) 実際にコンパイラを作るにはどうするのでしょう。それには、次の 2 つの道具だてがあります。

(a) それぞれの構文記号 (端記号、非端記号) に**属性** (attribute) をつけられる。

(b) 構文規則に**動作** (action) を付随させられる。

(a) については、記号ごとにその記号が値を伴うことができる、ということです。これは実装としては、構文解析に使うスタックと並行してもう 1 つ**意味スタック** (semantic stack) というスタックを用意し、構文記号 (実際には対応する状態) を入れるのと同じ場所の意味スタック側に属性の値を格納します。Cup では属性値は Java のオブジェクト型である必要があります。

たとえば今回は、IDENT と ICONST は字句解析からの文字列 (既に `yytext()` の値を渡すように作ってありました) を属性値とし、端記号はすべて `Tree.Node` オブジェクトを属性値とすれば、木構造を組み立てることができます。

(b) については、それぞれの構文規則の右端に「`{:` … `:}`」という形で囲んだ Java コードを記述しておくと、そのコードはその構文規則が還元されるときに実行されます。上向き解析なので、下から順に木のノードを組み立てて行くのが自然な使い方です。

しかし、子の構文規則で組み立てたノードを親側から取り出すのにはどうすればいいでしょうか? そこで属性値を使います。Cup では構文規則の右辺に現れる端記号、非端記号には「`:`名前」という指定がつけられ、アクション中ではその名前の変数が、対応する記号の属性値を保持しています。そして、`RESULT` という特別な名前に代入したものが、その構文規則の左辺の記号の属性値となります。

では具体例を見てみましょう。文法は先の例と同じですが、属性とアクションが追加されています。まず属性の型として、IDENT と ICONST は `String`、非端記号は `Tree.Node` を指定しています。そして、規則の右辺で属性を受け渡す記号は`:x` とか`:y` など、受け渡す変数を指定しています。

prog の属性値は *statlist* の属性値そのままです。*statlist* は、空に対応するときは空の `Tree.Seq` を作り、そのあと 1 つ文が現れるごとにその文のノードを追加し、値としては `Tree.Seq` を受け渡して行きます。

```
import java_cup.runtime.*;

terminal READ, PRINT, IF, WHILE, ASSIGN, SEMI, LPAR, RPAR, LBRA, RBRA;
terminal GT, LT, PLUS, MINUS, ASTER, SLASH, UMINUS;
terminal String ICONST, IDENT;

non terminal Tree.Node prog, statlist, stat, expr;

precedence nonassoc LT, GT;
precedence left PLUS, MINUS;
precedence left ASTER, SLASH;
precedence left UMINUS;
```

```
prog ::= statlist:x            {: RESULT = x; :}
         ;
statlist ::= statlist:x stat:y {: RESULT = x; x.add(y); :}
           |                   {: RESULT = new Tree.Seq(); :}
           ;
stat ::= IDENT:x ASSIGN expr:y SEMI
         {: RESULT = new Tree.Assign(new Tree.Var(x), y); :}
       | READ IDENT:x SEMI  {: RESULT = new Tree.Read(new Tree.Var(x)); :}
       | PRINT expr:x SEMI  {: RESULT = new Tree.Print(x); :}
       | IF LPAR expr:x RPAR stat:y    {: RESULT = new Tree.If1(x, y); :}
       | WHILE LPAR expr:x RPAR stat:y {: RESULT = new Tree.While(x, y); :}
       | LBRA statlist:x RBRA {: RESULT = x; :}
       ;
expr ::= expr:x PLUS expr:y  {: RESULT = new Tree.Add(x, y); :}
       | expr:x MINUS expr:y {: RESULT = new Tree.Sub(x, y); :}
       | expr:x ASTER expr:y {: RESULT = new Tree.Mul(x, y); :}
       | expr:x SLASH expr:y {: RESULT = new Tree.Div(x, y); :}
       | expr:x GT expr:y    {: RESULT = new Tree.Gt(x, y); :}
       | expr:x LT expr:y    {: RESULT = new Tree.Lt(x, y); :}
       | MINUS expr:x        {: RESULT = new Tree.Neg(x); :} %prec UMINUS
       | IDENT:x  {: RESULT = new Tree.Var(x); :}
       | ICONST:x {: RESULT = new Tree.Lit(Integer.parseInt(x)); :}
       | LPAR expr:x RPAR {: RESULT = x; :}
       ;
```

それぞれの文や式はこれまでやってきたように、各種のノードを作って自分の値として返すだけです。そのとき子ノードの値は、構文規則の属性として受け渡されてくる値を使えばよいわけです。このようにして、パーサジェネレータを使うことでコンパクトに動作のついたパーサを構成し、抽象構文木を組み立てることができるわけです。

最後に `main()` ですが、組み立てた木構造は `parse()` が返す `Symbol` オブジェクトの変数 `value` に入っているので、それを `Tree.Node` 型にキャストして `eval()` することで実行できます。

```
import java.util.*;
import java.io.*;
import java_cup.runtime.*;

public class Sam82 {
  public static void main(String[] args) throws Exception {
    parser p1 = new parser(new Lexer(new FileReader(args[0])));
    Symbol s1 = p1.parse();
    ((Tree.Node)s1.value).eval();
  }
}
```

では、最大公約数のプログラムを実行してみましょう。

```
% cat test.min
read x; read y;
```

```
while(x - y) {
  if(x > y) { x = x - y; }
  if(x < y) { y = y - x; }
}
print x;
% java Sam82 test.min
x? 60
y? 18
6
%
```

演習 8-4 例題の文法の if 文に else 部をつけてみなさい。曖昧な文法と曖昧でない文法の両方でできるとなおよい (曖昧な文法は Cup のマニュアルをよく読む必要があるかも)。

演習 8-5 例題の文法の if 文や while 文を end のある形に直してみなさい。if 文については、if-elsif-elsif-else-end の構文もきちんとサポートすること。

演習 8-6 例題に自分の好きな言語機能 (制御構造でもそれ以外でも) を追加してみなさい。

演習 8-7 実行のされかたは例題と同じだが、書き方の見た目がまったく違う構文になっている言語を実装してみなさい。

4 課題 8A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (`y-kuno@uec.ac.jp`) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル – 「システムソフトウェア特論 課題# 7」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

  Q1. 上向き解析の原理は分かりましたか。LR オートマトンの構成は無理としても、オートマトンができたらそれを使って動作するやり方が分かればよいです。

  Q2. cup を使ってパーサを構築するのはどうでしたか。

  Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。