LR 構文解析に関する分り易い解説集
（解説１）「構文解析の話をしよう（http://www.kmonos.net/wlog/83.html）」を元に少し変更

LL(1) とか LR(1) って要するに何？ っていうのを、思いっきり大ざっぱに紹介してみたいと思います。

まず構文解析ってなに？

単語の列を、文法と照らし合わせて解釈する作業のことです。 プログラミング言語の場合、要するに、ソースのこっからここまでが if 文で、こっからここまでが １個の関数定義で、この部分はクラス定義で、ここからここは足し算してる式で…みたいな、 ソースコードの構造を完全に判定する作業です。

プログラミング言語的には「単語」のことを「トークン」といいます。「if」とか「else」とか 「+」とか「<=」とかがトークン。

d & d って何？

プログラミング言語の構文解析には、"directional" で "deterministic" な手法が好まれます。 "directional" というのは左から右に順番にトークンを読みながら順番に構造を判定する、ということ。 「ちょっと突然一番最後のトークンが知りたくなった」とか言ってジャンプしたりしない。 "deterministic" というのは、一度判定した構造を絶対取り消さないこと。「ここ if 文っぽいから とりあえず if 文と思って解析続けてみてダメだったら取り消し」とか軟弱なこと言わない。 好まれるのは何でかというと、速いから。あとメモリもあんまり喰わない。

で、"directional" で "deterministic" な手法は、大きく分けて２つあります。LL と LR です。

LL って何？
LL(1) 法というのは、構文のはじめの １ トークンで構造を決定していく方法。

if
っていうトークンを見たら即「ここから if 文がはじまる！」と判定する。 もう即座に new IfStatement() とか言って構文木作っちゃう。

def
っていうトークンを見たら「関数定義ｷﾀｺﾚ！」と決めつけちゃう（Ruby とか Python の場合ですね)。

LL(2) 法なら、はじめの 2 トークンまで見て決める。LL(3) なら、はじめの 3 トークン。以下略。 LL(0) は、要するに 1 文字も見ずに「次絶対 if 文来るね。マジ。間違いないね」とか予言しないと いけないのでつまり絶対無理なので、LL(0) という方法はありません。

LL(k)法は、構文の開始より k トークン先まで読んで構文を予測するので、 「k トークン予測先読み (prediction look-ahead) する」などと言います。 あんまり言わないかもしれない。

逆に、LL(1) じゃない文法も見てみよう。C 言語。

foo
って先頭のトークン見ても、これから foo = 100; っていう代入文が始まるのか、 foo myfunc(); っていう関数宣言が始まるのか、はたまた foo[1]("hello"); という配列アクセスした上に関数呼び出しという怪しげな文が 始まるのか、まったく予測できない。こういうのは LL(1) じゃない。

実際のところ、四則演算の数式の文法が LL(1) で書けるとか、はじめの 1 トークンで「予測できる」 「予測できない」っていうのはそのままの直感とは外れるところもあるかとは思うんですけど、 基本的には、「最初の 1 トークンで何の文かわかるようになってる言語/文法」は LL(1)っぽいといえる。

LR って何？

LR(0) 法は、構文のおわりのタイミングで 「あ今 if 文終わったちょっとストップ」とかいう雰囲気で構文を判定する手法。
閉じ括弧を読んだそのタイミングで、構造をひとつ決定する。

while(...) { ... }
の閉じ中括弧を見た時点で、「while ｵﾜﾀ」って言って、「バランスする中括弧の手前の while から 今の中括弧まで while 文でした」と決める。

一応構文をおわりまで読んでから決定するので、LL 系の方法よりも LR 系の方法の方が、何かと 判定能力は高かったりします。が、LR(0) はやっぱりちょっと弱いので、ここでも「先読み」 が活躍します。

LR(1) 法は、構文のおわりのさらに次の 1 トークンまで見て決断を下します。 LR(0) は先読みできないので例えば

if(...) { ... }
ここまで読んだ時点で、「今 else なし if 文終わったよ！」と宣言しちゃってよいものか、それとも、次 else が来る可能性があるのでここはスルーして

if(...) { ... } else { ... }
になるのを待ってから「これぞ if 文！if 文完成！」と叫ぶべきなのかわからない。 ちなみに「今〇〇文終わったよ！」と宣言することを専門用語で「reduce」といい、 スルーして次のトークンを読み込むことを「shift」といいます。 reduce すべきか shift すべきかわからなくて困ることを「shift/reduce conflict」と言います。

って、そうだ、LR(1) の話をしてたんだった。LR(1) の場合は、

if(...) { ... }
この時点で、その次のトークンまでチラ見してから判定します。次が例えば「while」だったら、「else 無し if 文でした！」って宣言しちゃっていいし、次が「else」だったら、「else 無し if 文の後ろに else が来るとか常識的に考えてあり得ない」ので、ここはスルーして良い。 このように、チラ見した次のトークンを使って、「〇〇文の後ろに△△が来ることがあり得るかどうか」 を考えて判定するのが LR(1) です。

LR(2) なら構文の後ろの 2 トークンを見て判定するし、LR(3) なら 3 トークンを見て判定する。 構文のおわりのタイミングで判定するので、1 トークンも先を読まないで判定する LR(0) はあり得ます。 LL(0) があり得ないのと対照的です。

+++

（解説２）最左導出と最右導出の解説
（http://www.ieice-hbkb.org/files/06/06gun_02hen_03.pdf）から引用

規則

S → AB,
A → aA | Ab | ε,
B → c
で定まる 文脈自由文法 を考える.
以下は，すべてこの文法に おける導出である.

S ⇒ AB ⇒ aAB ⇒ aAbB ⇒ abB ⇒ abc (3・1)
S ⇒ AB ⇒ Ac ⇒ aAc ⇒ aAbc ⇒ abc (3・2)
S ⇒ AB ⇒ aAB ⇒ aAc ⇒ aAbc ⇒ abc (3・3)
S ⇒ AB ⇒ AbB ⇒ aAbB ⇒ abB ⇒ abc (3・4)
これらはすべて同じ終端記号列 abc を導出しており，
式 (3・1)，式 (3・2)，式 (3・3) に対応する導出木はすべて等しい（図 3・1(a)）. 式 (3・1) では導出途中の文型において常に一番左の 非終端記号に規則を適用しているが，式 (3・2) では常に一番右の非終端記号に規則を適用し ている. 前者のような導出を最左導出（leftmost derivation），後者のような導出を最右導出 （rightmost derivation）と呼ぶ. もちろん式 (3・3) のように最左導出でも最右導出でもないような導出もある. 一方，式 (3・1)，式 (3・2)，式 (3・3) に対応する導出木と式 (3・4) に対応する導出木は異なる（それぞれ図 3・1(a) と (b)）. ちなみに，式 (3・4) も最左導出である. この例のように，一つの導出木に複数個の導出が対応することがある. しかし，導出木と最左導出，導出木と最右導出は 1 対 1 に対応する. 導出木はプログラミング言語や自然言語の構文構造を表現するのに用いられる，また最左導出及び最右導出は cfg の構文解析において重要な概念である. 異なる導出木をもつような終端記号列が L(G) に一つでも存在するとき，G はあいまいで
あるという.

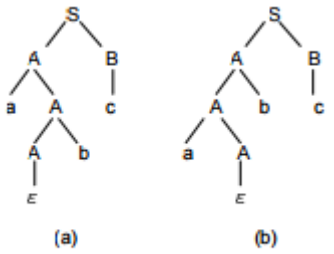

図 3・1 導出木

＋＋＋＋
（解説３）LR(ｋ)法とシフト・レデュース（シフト・還元）
LR 法は、「入力を左(Left)から右に読んでいき、最右導出(Rightmost derivation)を行う」。ということになっているが、「何で左から読んでいくのに最右（導出）になるのか分りにく

い」
（古谷イメージ）＊＊＊
ドミノ倒しのイメージで言うと、入力を左から読んでいきながら、（分ったところに）ドミノ板を立てていき、文末に行って最後の板を立てた後、最後の板を倒すと今まで立ててきた板がパタパタと倒れて文法の最初（S）に辿り付けば構文解析成功、って感じ。
＊＊＊

もう少しフォーマルに、「独習コンピュータ科学基礎 III、p.105」をベースに若干変更。
次の文法を用いて、上向き構文解析（LR）のアイデアを紹介する。
S ⇒ aSB|d
B ⇒ b

｛注：aadbb の構文解析とは、aadbb が上の文法に合っているかの解析。元々aadbb は次の最右導出で作ったものであるから、文法に合っている。
S ⇒ aSB ⇒ aSb ⇒ aaSBb ⇒ aaSbb ⇒ aadbb
｝

文字列 aadbb を例にとって、aadbb が導出された上の過程を aadbb から逆向きに辿って S に戻れるのかインフォーマルな記述を行う。
導出ステップは、表駆動“シフト還元（shift-reduce）”構文解析器を用いて求めることができる。
構文解析器は、次の様にスタック、入力、動作、の 3 列の表で表現できる。
スタックのトップ（TOS）はスタックの右端である。現在の入力記号は入力の最左の記号である。表の「スタックと入力」は、直前の行の「実行する動作」を行うことによって」得られる。シフト動作とは、現在の入力記号をスタックへプッシュすることをいう。還元動作とは、TOS 近辺の記号（複数のことがある）によって表現される生成規則（文法の規則）の右辺を生成規則の左辺で書き換えることをいう。 $ は空なスタックと空な入力記号を示す。
解析表

スタック	入力	実行する動作
$	aadbb$	シフト
$a	adbb$	シフト
$aa	dbb$	シフト
$aad	bb$	S→d　で還元
$aaS	bb$	シフト
$aaSb	b$	B→b で還元

$aaSB	b$	S→aSB で還元
$aS	b$	シフト
$aSb	$	B→b で還元
$aSB	$	S→aSB で還元
$S	$	受理

（注：スタックに積まれた上の方から還元が行われていることが分る。シフトで取り出した入力の右端から還元が進められている）

+++

LR(k)文法、ハンドル、状態遷移図、構文解析、に関しては、

「独習コンピュータ科学基礎 III、p.106~112」が、例も良く、よく分る。

＋＋＋

LR(1)を自然言語に適用する説明は、（あまり、深追いしてもな、って感じで）雰囲気だけでも「はじめての自然言語処理（p.68 から）」でいいんじゃないかしら。