CEN G444: Syntactic Analysis (Parsing)

Cem Bozşahin

Cognitive Science Department, Informatics Institute,
Middle East Technical Univ. (ODTÜ), Ankara

Some time-honored maxims of compiling

1 Understand the internal structure of input (parse tree)

2 Error reporting (they must be easy to identify by the programmer, by location and by message content)

3 Keep the meaning of source code as intended (transparency and compositionality)

3.1 Resist the temptation to "fix" the errors of the programmer

3.2 It is not YOUR program!

3.3 It is usually a bad idea to second-guess a programmer, if you want to help.

- Syntax of PLs is (almost) context-free.

  Declare before use

  Not declare twice

  Type match of use of a variable

- These are not context-free properties.

- Semantics of a program must not be ambiguous; PLs are designed not to be ambiguous in this sense.

  The parser for a PL must cope with non-determinism in rule selection and application order, but it cannot allow global ambiguity.

- Development of deterministic parsing algorithms for context-free grammars.

  LL parsing: left-to-right scan of input: leftmost derivations.

  LR parsing: left-to-right scan of input: rightmost derivations.

- LL(k)/LR(k): an LL/LR grammar that can be parsed deterministically using k-symbol lookahead.

Some basics of formal grammars

Let G=(V, Σ, R, S) be a formal grammar G.

- rule definition: Every rule in R is $A \rightarrow w$
- rule use: if we reach the point uAv in the derivation of a string, and $A \rightarrow w \in R$, then we also have uwv

  $uAv \Rightarrow uwv$

- If A is the leftmost variable, this is a *leftmost derivation*
- if A is the rightmost variable, this is a *rightmost derivation*

L(G)=$\{w \in \Sigma^* \mid S \Rightarrow^* w\}$

- Intermediate forms reachable from S are *sentential forms*.
- Right-sentential forms:
  $\{x \in (\Sigma \cup V)^* \mid S \Rightarrow_R^* x \Rightarrow_R^* w, w \in \Sigma^*\}$
- Left-sentential forms: $\{x \in (\Sigma \cup V)^* \mid S \Rightarrow_L^* x \Rightarrow_L^* w, w \in \Sigma^*\}$
- Top-down parsers start with S, and reach w.
- Bottom-up parsers start with w, and reach S.

  That's why they end up using rightmost derivations in reverse order.

- *Lookahead:* looking at next tokens before deciding on the rule application.

  in $S \Rightarrow_* uAv, u \in \Sigma^*$, x is the lookahead string in ux

  $S \overset{L}{\Rightarrow}_* uAv$ and $u \in \Sigma^*$. If u is a prefix of p, which $A \rightarrow w$ rules to apply?

  if $p = uaq, a \in \Sigma$, then 1-symbol lookahead can reveal that A rules *not starting* with a cannot lead to p.

input: $aaba$ $G_1:$

$$\begin{aligned} S &\rightarrow aA \mid abC \\ A &\rightarrow aA \mid bA \mid C \mid \varepsilon \\ C &\rightarrow c \end{aligned}$$

- LOOKAHEAD SETS OF VARIABLES in grammar $G = (V, \Sigma, P, S)$

  $\mathrm{LA}(A) = \{x \mid S \Rightarrow_* uAv \Rightarrow_* ux; ux \in \Sigma^*\}$

  x is a terminal string after prefix u: this gives all terminals from Av when prefix is u
- LOOKAHEAD SETS OF RULES in the grammar:

  $\mathrm{LA}(A \rightarrow w) = \{x \mid wv \Rightarrow_* x \in \Sigma^*; S \Rightarrow_* uAv, u \in \Sigma^*\}$

  subset of LA(A) in which subderivation $Av \Rightarrow_* x$ are done by $A \rightarrow w$
- These are lookaheads when A is an unknown.

  $\mathrm{LA}(A)$: all the strings A can start

  $\mathrm{LA}(A \rightarrow w)$: all the strings that using the A rule can start

$$\mathrm{LA}(A) = \bigcup_{i=0}^{N} \mathrm{LA}(A \rightarrow w_i)$$

- if LA sets of rules are disjoint, rule selection can be done deterministically.

  G_2: $S \rightarrow \quad Aabd \mid cAbcd$
  $\quad\quad A \rightarrow \quad a \mid b \mid \varepsilon$

  LA$(S \rightarrow Aabd) = \{aabd, babd, abd\}$
  LA$(S \rightarrow cAbcd) = \{cabcd, cbbcd, cbcd\}$
  LA$(S) =$ union of the two LA sets
  LA$(A \rightarrow a) = \{aabd, abcd\}$
  LA$(A \rightarrow b) = \{babd, bbcd\}$
  LA$(A \rightarrow \varepsilon) = \{abd, bcd\}$

- How many lookaheads do we need for this grammar?
  $\Rightarrow$ choosing A : *a* can pick 1st or 3rd A rule
  $\Rightarrow$ choosing A : *ab* can pick 1st or 3rd A rule $\Rightarrow$ LL(3)

- If there are recursive rules, LA strings can be of arbitrary length; use $trunc_k(x)$ to truncate them to length k

  $\text{LA}_k(A) = trunc_k(LA(A))$

- FIRST, FOLLOW and LA sets. Critical for LR and LL.

- Direct construction of LA sets is not straightforward in a large grammar; use FIRST and FOLLOW sets.

  $\text{FIRST}_k(A) =$ prefixes of terminal strings derivable from A

  $\text{FOLLOW}_k(A) =$ prefixes of terminal strings that can follow the strings derivable from A

for any string $u \in (V \cup \Sigma)^*$
$\mathrm{FIRST}_k(u) = trunc_k(\{x \mid u \Rightarrow_* x; \quad x \in \Sigma*\})$
$\mathrm{FOLLOW}_k(A) = trunc_k(\{x \mid S \Rightarrow_* uAv; \quad x \in \mathrm{FIRST}_k(v)\}$
Then, $\mathrm{LA}_k(A) = trunc_k(\mathrm{FIRST}_k(A)\mathrm{FOLLOW}_k(A))$
and, $\mathrm{LA}_k(A \to w) = trunc_k(\mathrm{FIRST}_k(w)\mathrm{FOLLOW}_k(A))$

ex: for G_2
$\Rightarrow \mathrm{FIRST}_3(S) = \{aab, bab, abd, cab, cbb, cbc\}$
$\Rightarrow \mathrm{FOLLOW}_3(S) = \{\varepsilon\}$

- STRONG LL(k) GRAMMARS: if the $\mathrm{LA}_k(A)$ sets are partitioned by sets $\mathrm{LA}_K(A \to w_i)$, then G is strong LL(k).
- if G is strong LL(k), G is unambiguous.

Some fine points of string calculus

- Is stack simply a conventional data structure for parsing strings?

  No. It follows from property of concatenation and substitution

- If $uAv \Rightarrow uwv$ because of $A \rightarrow w$, then

  if FIRST(w)=B, we have $uAv \Rightarrow uwv \Rightarrow uB\cdots v \Rightarrow u\beta\cdots v$ because of $B \rightarrow \beta$

- B's material is always before $\cdots v$

- Asymmetry in FOLLOW sets:

  If we have $A \rightarrow \alpha B \beta$ where FIRST(β) contains 'ε'

- then FOLLOW(B) contains FOLLOW (A):

  $S \Rightarrow^* uAv \Rightarrow u\alpha B\beta v \Rightarrow \alpha Bv$

- but FOLLOW(A) does not necessarily contain FOLLOW(B)

  because we don't know whether $S \Rightarrow^* xBy \Rightarrow^* xAy$

- Making FOLLOW sets non-empty: design an end marker and a rule $S' \rightarrow S\$$
- Place \$ in FOLLOW(S). Nothing follows S'. $\$ \notin \Sigma$
- For $A \rightarrow \alpha B \beta$, everthing in FIRST($\beta$) is in FOLLOW(B) if $\beta \neq \varepsilon$
- For $A \rightarrow \alpha B \beta$ where $\beta \Rightarrow^* \varepsilon$ then everything in FOLLOW(A) is in FOLLOW(B) because:
  $S \Rightarrow^* uAv \Rightarrow u\alpha B\beta v \Rightarrow^* \alpha Bv$

Bottom-up parsing

- LR(k): deterministic bottom-up parsing using rightmost derivations with k-symbol lookahead.
- Advantages of LR parsers:

  More grammars are LR parsable than LL parsable

  Earliest error detection in the string

  Can be automated, just like LL parsers.

- Disadvantage: too much work to design manually: use a parser generator (hence the name *compiler-compiler*) eg: non-determinism in bottom-up parse

$$
\begin{aligned}
S \rightarrow &\quad aAb \mid BaAa \\
A \rightarrow &\quad ab \mid b \\
B \rightarrow &\quad Bb \mid b
\end{aligned}
$$

input: $aabb \Rightarrow aAb \qquad aaAb \qquad aaBb$?

- Towards determinism in BUP: How to locate what to reduce, and how to decide by which rule to do the reduction.

Bottom-up shift-reduce parsers work on *right* sentential forms; they obtain these forms in reverse order.

- *handle*: use of a rule $A \rightarrow \beta$ at certain position in a right sentential form.

  $S \overset{R}{\Rightarrow}_* \alpha A w \overset{R}{\Rightarrow} \alpha \beta w$

  ($A \rightarrow \beta$ is a handle at position after $\alpha; w \in \Sigma^*$).

- There may be more than one handle if the grammar is ambiguous.

$$
\begin{aligned}
S \rightarrow &\quad A \\
A \rightarrow &\quad T \mid A + T \\
T \rightarrow &\quad b \mid (A)
\end{aligned}
$$

```
input b+(b+b)       (T->b at positions 1,4,6)
      T+(b+b)       (A->T at 0; T->b at 4,6)
      A+(T+b)       (not S->A at 1; A->T at 4; T->b at 6)
      A+(A+b)       (S->A at 4; T->b at 6; not S->A at 1)
      A+(A+T)       (A->A+T at 4; A->T at 6; not S->A  1,4)
      A+(A)         (S->A at 4; not S->A at 1; T->(A) at 3)
      A+T           (A->T at 3; not S->A at 1; A->A+T at 1)
      A
      S
```

The ones that can lead to S are handles.

- Bottom-up parsing can be seen as 'handle pruning':
  Start with input
  locate a handle
  use handle to reduce the sentential form
  do until S is reached

- Problems: 1) how to locate the handle; 2) reduce by which rule?

1 Using a stack to keep sentential forms solves the first problem: the right end of the handle is always on top of the stack. The left end must be found.

2 Knowledge of context in which a rule can lead to a viable alternative solves the second problem: don't use the rule if it's context is not satisfied.

- *viable prefix (LR(0) context)*: the set of prefixes of a right-sentential form that can appear on stack.

$$S \overset{R}{\Rightarrow}_* uAv \overset{A \to w}{\Rightarrow} uwv \qquad v \in \Sigma^*$$

uw is a viable prefix

$$S \overset{R}{\Rightarrow}_* uAv \overset{A\to w}{\Rightarrow} uwv \qquad v \in \Sigma^*$$

uw is a viable prefix

- Why LR(0) context? no look into v
- Why look at a PREFIX if we're parsing with rightmost derivations in reverse?

  'L' in LR is for left-to-right scan.

  We are trying to ensure BEFOREHAND that only viable prefixes appear on stack (deterministic parsing).

- Viable prefixes may contain patterns if there's recursion in the grammar. Finding them on-the-fly is costly.

$$
\begin{aligned}
S &\rightarrow \quad aA \mid bB \\
A &\rightarrow \quad abA \mid bB \\
B &\rightarrow \quad bBc \mid bc
\end{aligned}
$$

$$S \Rightarrow aA \Rightarrow^{i}_{*} a(ab)^{i}A \Rightarrow a(ab)^{i}bB \Rightarrow^{j}_{*} a(ab)^{i}bb^{j}Bc^{j} \Rightarrow a(ab)^{i}bb^{j+1}c^{j+1}$$

$$S \Rightarrow bB \Rightarrow^{j}_{*} bb^{j}Bc^{j} \Rightarrow bb^{j+1}c^{j+1}$$

$$
\begin{aligned}
&\mathrm{LR0C}(S \rightarrow aA) = \{aA\} \\
&\qquad \mathrm{LR0C}(A \rightarrow abA) = \{a(ab)^{i}A \mid i > 0\} \\
&\qquad \mathrm{LR0C}(\mathrm{B} \rightarrow bBc) = \{a(ab)^{i}b^{j+1}Bc, b^{j+1}Bc \mid i \geq 0, j > 0\}
\end{aligned}
$$

$\text{LR0C}(S \to aA) = \{aA\}$

$\text{LR0C}(A \to abA) = \{a(ab)^i A \mid i > 0\}$

$\text{LR0C}(\text{B} \to bBc) = \{a(ab)^i b^{j+1} Bc, b^{j+1} Bc \mid i \geq 0, j > 0\}$

why not c^j at the end? Viable prefix includes up to and including RHS of the rule.

- Solutions to the viable prefix problem:

1. Shift-reduce parsing with stack search to locate the left-end of the handle.

2. Shift-reduce parsing with no stack search; design a recognizer to keep progress over the RHSs so that we can tell whether the handle is a viable prefix for a particular rule ⇒LR parsing

LR & Shift-reduce: find both ends of a RHS without search

In the first alternative, the stack contains right sentential forms.

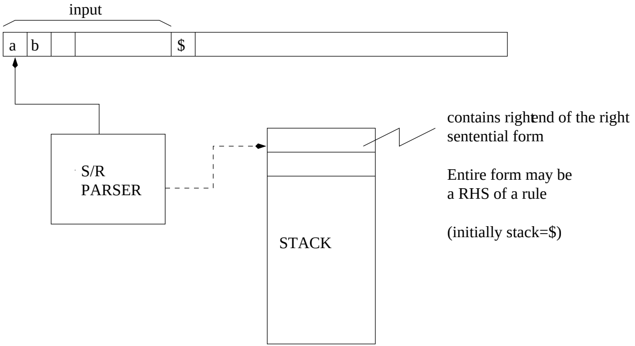
In the second alternative, the stack contains right sentential forms and states which give a summary of viable prefixes. No search in stack.

- These are the famous SLR/LALR/LR tables.

- $S \overset{R}{\Rightarrow}_* uAv \overset{R}{\Rightarrow} uwv \qquad v \in \Sigma^*$

  $A \rightarrow w$ is a handle at the end of u

  uw is the viable prefix (LR0C).



Only viable prefixes appear in the stack

- S/R parsing:
  repeat
  if a handle is on top, reduce by the handle (pop)
  otherwise shift (put in stack)
  until accept or input exhausts

$$
\begin{array}{lll}
G_{AE}: & S' \rightarrow & S\$ \\
 & S \rightarrow & A \\
 & A \rightarrow & T \mid A+T \\
 & T \rightarrow & b \mid (A)
\end{array}
$$

stack	input	action
$	b+(b+b)$	sh
$b	+(b+b)$	red by T->b
$T		r A->T
$A		sh
$A+	(b+b)$	sh
$A+(	b+b)$	sh
$A+(b	+b)$	r T->b
$A+(T		r A->T
$A+(A		sh
$A+(A+	b)$	sh
$A+(A+b	)$	T->b
$A+(A+T		A->A+T (conflict; but A->T is not a handle)
$A+(A		sh
$A+(A)	$	r T->(A)
$A+T		r A->A+T (conflict)
$A		r S->A (shift-red conflict)
$S		sh
S	-	r S'->S$
$S'		accept

A+A would not be a handle

- Reduce-reduce conflict shows the significance of the viable prefix problem. How to choose the right one?
- Shift-reduce conflict shows the significance of handle manipulation and structure of derivations. If the handle is not reduced, can we still get a derivation?

eg.

```
E -> E * E | E + E | id

stack     input          action

$         id1+id2*id3    sh
$id1      +id2*id3       r E->id1
$E                       sh
$E+       id2*id3        sh
$E+id2    *id3           r E->id2
$E+E                     conflict: shift or reduce?
```

- If shift-reduce conflict is not resolved by grammar re-writing, shift seems to work better for PL grammars.

```
S-> if E then S | if E then S else S

input: if b then if S1 then S2 else S3

stack              action

$                  sh
..
$if E .. S2        ?
```

- Shift will favor innermost attachment of 'else'; reduce, outermost.
- This may be context-free, but is it deterministic context-free? (DCFL). No. That's why it is usually stated separately in PL grammars.
  Not all CFLs are DCFLs (ex. Parikh languages)

- Does the presence of conflicts mean non-LRness? not necessarily. If the grammar is ambiguous, it can't be LR(k) for any k. But if the conflict is local, it may still be LR.
- Going from S/R parsing to LR parsing: keep track of "progress" over the RHS of rules to select the right handle without a search for the left-end of the handle.
- no look beyond the RHS $\Rightarrow$Simple LR (SLR)

  lookahead $\Rightarrow$LALR(K) and LR(k) parsing
- SLR is more informative than simple LL.

- side note: Efficiency of left-recursion vs. right-recursion in S/R parsing

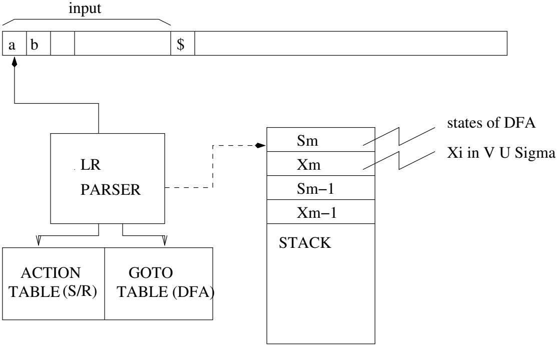
```
Xs -> Xs , X | X       vs.     Xs -> X , Xs | X
X  -> id                       X  -> id

input: id1,id2,id3
```

- Left recursion makes minimal use of the stack
- Right recursion makes maximal use of the stack

in S/R bottom up parsing.

- Simple LR: use of LR(0) items (no look past the RHS).
- construct a DFA for recognizing the progress over the RHSs

eg. $A \rightarrow \cdot \underbrace{B\cdot}\underbrace{C\cdot}\underbrace{a\cdot}\underbrace{D\cdot}$
$C \rightarrow \cdot \underbrace{B\cdot}\underbrace{C\cdot}$



- SLR table construction:
  - Expand the grammar with $S' \rightarrow S\$$
  - find LR(0) items
  - find closure of items
  - find set of items from closures
  - construct the DFA from sets of items
  - find FIRST and FOLLOW sets
  - construct ACTION and GOTO tables

- LR(0) ITEM: progress of passing over the RHS of a rule
  LR(0)-item$(A \to uv) = \{A \to \cdot uv, A \to u \cdot v, A \to uv\cdot\}$
  LR(0)-item$(A \to \varepsilon) = \{A \to \cdot\}$

  eg. for the grammar G_{AE}:

$S \to \cdot A$		$S \to A\cdot$
$A \to \cdot T$		$A \to T\cdot$
$A \to \cdot A + T$		$A \to A \cdot + T$
$A \to A + \cdot T$		$A \to A + T\cdot$
$T \to \cdot b$	$T \to b\cdot$	$T \to \cdot(A)$ ...

- CLOSURE OF an LR(0) ITEM: All the states of parsing that can be reached from a state.

  closure$(S \to \cdot A) = \{S \to \cdot A, A \to \cdot T, A \to \cdot A + T, T \to \cdot b, T \to \cdot(A)\}$

- goto(item,symbol)= the set of states one can go from the item by consuming the symbol (note: this is not the GOTO table!)

  goto$(A \rightarrow A \cdot + T, +) = \{A \rightarrow A + \cdot T, T \rightarrow \cdot b, T \rightarrow \cdot (A)\}$

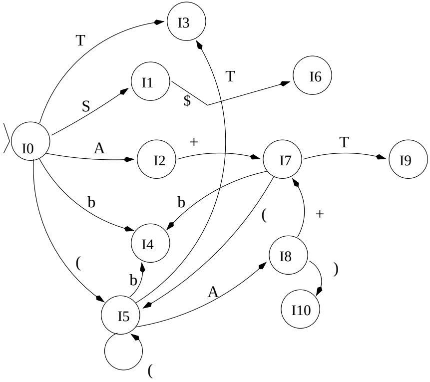
- SETS OF ITEMS= possible states of parsing. Since there are finitely many RHS, there will be finitely many combinations.

  initially: $S' \rightarrow \cdot S\$$.

  Find out where to go from goto(item,X) for all $X \in V \cup \Sigma$. Add this to sets of items.

  Each set is a possible state of SLR parsing (GOTO/ACTION states).

- eg.: $I_0 = \{S' \rightarrow \cdot S\$, S \rightarrow \cdot A, A \rightarrow \cdot T,$
  $A \rightarrow \cdot A + T, T \rightarrow \cdot b, T \rightarrow \cdot (A)\}$
  $I_1 = goto(I_0, S) = \{S' \rightarrow S \cdot \$\}$
  $I_2 = goto(I_0, A) = \{S \rightarrow A \cdot, A \rightarrow A \cdot + T\}$
  $I_3 = goto(I_0, T) = \{A \rightarrow T \cdot\}$
  $I_4 = goto(I_0, b) = \{T \rightarrow b \cdot\}$
  $I_5 = goto(I_0, () =$
  $\{T \rightarrow (\cdot A), A \rightarrow \cdot T, A \rightarrow \cdot A + T, T \rightarrow \cdot b, T \rightarrow \cdot (A)\}$
  $I_6 = goto(I_1, \$) = \{S' \rightarrow S\$ \cdot\}$
  $I_7 = goto(I_2, +) = \{A \rightarrow A + \cdot T, T \rightarrow \cdot b, T \rightarrow \cdot (A)\}$
  $I_8 = goto(I_5, A) = \{T \rightarrow (A \cdot), A \rightarrow A \cdot + T\}$
  $goto(I_5, T) = I_3 \quad goto(I_5, b) = I_4 \quad goto(I_5, () = I_5$
  $I_9 = goto(I_7, T) = \{A \rightarrow A + T \cdot\}$
  $goto(I_7, b) = I_4 \; goto(I_7, () = I_5 \; goto(I_8, +) = I_7$
  $I_{10} = goto(I_8,)) = \{T \rightarrow (A) \cdot\}$



The GOTO function as a FA

(GOTO table is a subset which only contains variable transitions)

$\text{FOLLOW}(S) = \{\$\}$ $\quad$ $\text{FOLLOW}(A) = \{), +, \$\}$
$\text{FOLLOW}(T) = \{\$, +,)\}$

- setting up the ACTION table
  for each I_i, if $A \rightarrow \alpha \cdot a\beta$ is in I_i, ACTION[i,a]=shift j (where goto(I_i,a)=I_j)
  for each $A \rightarrow \alpha\cdot$, ACTION[$I_i$,a]=reduce by $A \rightarrow \alpha$ for all $a \in FOLLOW(A)$

- Since the action depends on a, there may be a mixture of shift and reduce actions in the same state if the state contains both kinds of LR(0) items.

	ACTION					GOTO		
	b	(	)	+	$	S	A	T
0	sh4	sh5				1	2	3
1					accept			
2				sh7	rS->A			
3			rA->T	rA->T	rA->T			
4			rT->b	rT->b	rT->b			
5	sh4	sh5					8	3
6					accept			
7	sh4	sh5						9
8			sh10	sh7				
9			rA->A+T	rA->A+T	rA->A+T			
10			rT->(A)	rT->(A)	rT->(A)			

EVERY empty slot can be used for error reporting.

- LR parsers catch errors as early as possible

  because the empty table entries are invalid derivations (and we are parsing deterministically), so that we don't need to read input further to find the error.

parsing (b+b) with SLR table

stack	input	action
0	(b+b)$	sh5
0(5	b+b)$	sh4
0(5b4	+b)$	rT->b
0(5T3		rA->T
0(5A8		sh7
0(5A8+7	b)$	sh4
0(5A8+7b4	)$	rT->b
0(5A8+7T9		rA->A+T
0(5A8		sh10
0(5A8)10	$	rT->(A)
0T3		rA->T
0A2		rS->A
0S1		accept

- A grammar with no conflict in the SLR tables is SLR(1).
- What is a lookahead if we are not looking past the righthand sides? the next symbol decides local action (shift/reduce).

  because LA($A \to w$)=FIRST(w)FOLLOW(A)

- Reductions and new top determine the next viable prefix.
- In general, direct DFA construction is cumbersome. Design a NFA with empty transitions and convert to DFA.
- General LR(k) parsing is a generalization of SLR where LR(k)-items are used.
- Historically, SLR and LALR came after LR, to reduce its complexity (in class we go from SLR to LALR)

LALR = LA()LR()

- LR(0) items are crucial to LALR

  because LALR(1)=LA(1)LR(0)

- We get LR(1) items from LR(0) items:

  $LR1\text{-}context(A \rightarrow w) = LR0\text{-}context(A \rightarrow w) FOLLOW_1(A)$

- LALR is the most widely used LR technique.
- Every PL on the planet wants to be LALR.
- The ones which are not don't last very long.

- General LR(k) machine construction:
  method 1: find out closure/goto.., with LR(k) items (Dragon book)
  method 2: find LR(k) items and construct NFA from them (Sudkamp's book and EAC book)

- Finding LR(1) items

  LR(0) items are of the form $[A \rightarrow \alpha \cdot \beta]$
  shows 'local' parsing configuration in the RHS

  LR(1) items are $[A \rightarrow \alpha \cdot \beta, a]$
  a is the next symbol after RHS of A (non-local LA)

Clarifying the idea of lookahead

- There is no SLR(0). SLR means LR(0) with 1-symbol lookahead.
- This lookahead is not part of table elements in SLR:

  LR(0) items: $[A \rightarrow \alpha \cdot \beta]$ LR(1) items $[A \rightarrow \alpha \cdot \beta, a]$

  When the dot is at the end, SLR and LALR do same action on a: $[A \rightarrow \alpha\beta \cdot]$ $[A \rightarrow \alpha\beta \cdot, a]$
- But, SLR does it for all x in FOLLOW(A) including a, whereas LR does it only for a, which must be in FOLLOW(A).

- LALR also has $[A \rightarrow \alpha \cdot \beta, a]$; it creates more states reachable from this if next symbol is in FIRST(β).

- The first component is called CORE (or kernel)

- SLR $\subset$ LALR $\subset$ LR

1 A grammar which is not SLR but LALR (from Dragon book)

$$
\begin{aligned}
S \rightarrow &\quad L = R \\
S \rightarrow &\quad R \\
L \rightarrow &\quad *R \\
L \rightarrow &\quad id \\
R \rightarrow &\quad L
\end{aligned}
$$

- hint: FOLLOW(R) contains '='.

- exercise: try SLR construction for this grammar.

  in LR(0) item $I = [\{S \rightarrow L \cdot = R\} \cup \{R \rightarrow L \cdot\}]$ we have a conflict.

2 A grammar which is not LALR but LR (from stackoverflow)

$$\begin{aligned} S \rightarrow &\quad aEa \mid bEb \mid aFb \mid bFa \\ E \rightarrow &\quad e \\ F \rightarrow &\quad e \end{aligned}$$

- will give reduce/reduce conflict with LR(1) item: $[\{E \rightarrow e\cdot, a/b\} \cup \{F \rightarrow e\cdot, a/b\}]$
- Grammar 1 seems like a reasonable PL construction.
- Grammar 2 doesn't.

- LALR idea: LR(1) items with common core can be merged, along with their second elements.

  Let $I_1 = [C \to d\cdot, \{c, d\}]$ and $I_2 = [C \to d\cdot, \$]$.

- Merge them into one state: $I_{12} = [C \to d\cdot, \{c, d, \$\}]$
- If a state wanted to go to I_1 or I_2, it can now go to I_{12}
- It might reduce d to C in some examples where the original states would declare error,

  BUT that error will be caught by LALR soon enough (before any shift)

- LALR gives error whenever LR gives error, IF the grammar is LALR.
- Otherwise, they produce the same result.
- Sketch of proof: Suppose we merged two sets to get
  $I_{12} = [\{A \rightarrow \alpha \cdot, a\} \cup \{B \rightarrow \beta \cdot a\gamma, b\}]$ **S/R conflict**

  say I_1 is from a set that contains $[A \rightarrow \alpha \cdot, a]$. Since THAT set has common core,

  it must also have $[B \rightarrow \beta \cdot a\gamma, c]$ for some c. (We decide merge on cores; c not in it)
  It wasn't LR (1) anyway.

- *valid* LR(1) items: things to appear on stack top

  $S \Rightarrow uAv \Rightarrow uw_1w_2av \quad (v \in \Sigma^*)$

  $[A \rightarrow w_1 \cdot w_2, x]$ is valid if uw_1 is a viable prefix and $x \in \Sigma \cup \{\$\}$

- Why try SLR if LR is more general? LR tables are two or three orders of magnitude larger than SLR tables. But not every LR grammar is SLR. Lookahead LR (LALR) combines LR(1) states with common 'core' (LR(0) states) hence reducing the number of states. LALR seems to be practical for PL grammars.

- if the merged states do not introduce conflicts, the grammar is LALR(1)

- Recursive-descent parsing: Grammars rules as procedures

  $A \rightarrow B_1 \cdots B_i \cdots \ B_n$

  - A : head of function
  - B_i : if syntactic rule, then define function B_i and call it from RHSs with B_i.
  - otherwise, call $\texttt{match}(B_i)$
- Very intuitive
- Hand-coded error reporting
- Rapid prototyping

```
match(T): returns true if next token is of type T

advance(): consumes the lookahead token

procedure S;
begin
   ID;
   if match(ASSIGN) then advance() else error();
   E;
   if match(STOP) then advance() else error();
end;

procedure ID;
begin
   if match(ID) then token:=advance();
   install_id(token);
end;

procedure E;
begin
   T;
   Eprime;
end;

procedure Eprime;
begin
   if match(OP) then {advance(); T; Eprime;}
   else /* no consumption */
end
```

Syntactic Analysis:LL(k) parsing

eg: BUILDING a STRONG LL(1) GRAMMAR

- G_{AE} :
$$\begin{aligned} S' &\rightarrow S\$ \\ S &\rightarrow A \\ A &\rightarrow T \mid A+T \\ T &\rightarrow b \mid (A) \end{aligned}$$

- A is left-recursive: re-write
$$\begin{aligned} A &\rightarrow TA' \\ A' &\rightarrow +TA' \mid \varepsilon \end{aligned}$$

- FIRST sets:
$\text{FIRST}_1(S) = \{b, (\} \Rightarrow \text{FIRST}_1(A) = \{b, (\}$
$\text{FIRST}_1(A') = \{+, \varepsilon\} \Rightarrow \text{FIRST}_1(T) = \{b, (\}$

- FOLLOW sets:
  $\text{FOLLOW}_1(S) = \{\$\} \Rightarrow \text{FOLLOW}_1(A) = \{\$,)\}$
  $\text{FOLLOW}_1(A') = \{\$,)\}$
  $\text{FOLLOW}_1(T) = \{\$,), +\} = \text{FIRST}_1(A')\text{FOLLOW}_1(A')$
- LA sets:
  $\text{LA}_1(S \rightarrow A) = \{b, (\}$
  $\text{LA}_1(A \rightarrow TA') = \{b, (\}$
  $\text{LA}_1(T \rightarrow b) = \{b\}$
  $\text{LA}_1(T \rightarrow (A)) = \{(\}$
  $\text{LA}_1(A' \rightarrow \varepsilon) = \{\$,)\}$
  $\text{LA}_1(A' \rightarrow +TA') = \{+\}$

- A strong LL(k) parser

```
input: LL(k) grammar;
       input string p;
       LAk sets

1. Start with S (q=S)
2. repeat
     let q=uAv    (A is leftmost variable;
         p=uyz     u= prefix of q;
                   y is the lookahead string;
                   length(y)=k)

     if y is in LAk set of a rule A->w
         q=uAv => uwv  can de done deterministically
         let q=uwv
   until q=p or y is not in any LAk set
3. if q=p then accept, otherwise reject
```

- TABLE-DRIVEN LL. rows: variables. columns: LA_k sets
- The table predicts which rule to apply (predictive parsing)
- if the table has multiple entries, it is not LL(k)

- Constructing the LL(k) table (uses LA sets directly)

```
input:  G=(Alphabet,V,P,S)
        LAk sets for all rules in P
 output: parsing table M

 1. For all A->w in P

     2. M[A,u] contains A->w ; for each terminal
                                string u in LAk(A->w)

     3. M[A,$] contains A->w if $ is in LAk(A->w)

 4. Make all empty entries of M error states
```

	b	(	)	+	$
S'	S'->S$	S'->S$			
S	S->A	S->A			
A	A->TA'	A->TA'			
A'			A'-> null	A'->+TA'	A'->null
T	T->b	T->(A)			

parsing: start with S', find the right rule from LA sets. if next symbol is $, see if $q = p$ after last rule.

exercise: try input `(b+b)` on LL(1) version of G_{AE}

- A no-lookahead algorithm would backtrack 3 times for the same input.
- What about left-recursive rules?
  - Can we parse them with table-driven LL?
  - Knowing that we can't parse them with RDP?
- No ambiguous or left-recursive grammar can be LL(1)

Because $FIRST_1(w)FOLLOW_1(A)$ would be the same for all left-recursive rules.

- Reconsider the left recursive grammar:

$$
\begin{aligned}
G_{AE}: \quad S' &\rightarrow \quad S\$ \\
S &\rightarrow \quad A \\
A &\rightarrow \quad T \mid A + T \\
T &\rightarrow \quad b \mid (A)
\end{aligned}
$$

```
        b         (          )           +          $

   --------------------------------------------------
A  | A->T    A->T
   | A->A+T  A->A+T
```

- LL(k) versus strong LL(k) grammars:

  A grammar is LL(k) but not strong LL(k) if the LA sets of rules are not necessarily disjoint but the LA sets of any sentential form is unique.

$$
\begin{aligned}
S \rightarrow &\quad Aabd \mid cAbcd \\
A \rightarrow &\quad a \mid b \mid \varepsilon
\end{aligned}
$$

$\mathrm{LA}_2(A \rightarrow a) = \{aa, ab\}$
$\mathrm{LA}_2(A \rightarrow \varepsilon) = \{ab, bc\} \Rightarrow$ not strong LL(2)

- $\text{LA}_k(uAv, A \to w) = \text{FIRST}_k(wv)$

  $\text{LA}_2(Aabd, A \to a) = \{aa\}$
  $\text{LA}_2(Aabd, A \to b) = \{ba\}$
  $\text{LA}_2(Aabd, A \to \varepsilon) = \{ab\}$

  Similarly LA_2 sets of *cAbcd* also form a partition. $\Rightarrow$ LL(2)

- In general though, number of sentential forms in a grammar is quite large (sometimes can only be described by a pattern).

- LA sets must be calculated on-the-fly $\Rightarrow$ costly

Adaptive LL

- in 2014, Parr, Harwell and Fisher introduced Adaptive LL(), in SIGPLAN Notices of ACM.
- ALL() can handle DIRECT left-recursion because it handles rule choice at run-time (i.e. not in tables)
- Indirect left recursion is still a problem, and not ALL-parsable.
- antLR4 implements ALL()
- a bit of history: yacc can handle precedence at parse-time out of grammar: prefer shift to reduce etc.
- They are one-step closer to their onomastic nemesis, aka. LR!

BOTTOM LINE: We do all this math to ensure that the source code is translated to internal representation

1) unambiguously,

2) predictably (i.e. by an algorithm)

- How about ambiguous code? Now that's AI we haven't bargained for. Somebody has to make code choice, and that needs autonomous decision making. But it's humans who have to live with its consequences!

- I think we should start looking at computers and ethics from this angle before it's too late.