Bottom up Parsing

Handle is defined as the substring that matches RHS of a production.

A left-to-right, bottom-up parse works by iteratively searching for a handle, then reducing the handle to the left hand side of the production. The process is repeated until we get the Start Symbol and is called **Handle Pruning.**

Searching for Handles:
When using a shift/reduce parser, we must decide whether to shift or reduce at each point.
●We only want to reduce when we know we have a handle.
●Question:How can we tell that we might be looking at a handle?

Item : Production with a dot in RHS of the producttion. Example : A -> XY.Z
. indicates where we are in this production.
When . Is at the end of the production its called a final item or a kernel item and indicates that the handle is on TOS and we can perform reduction.

At any instant in time, the contents of the left side of the parser can be described using the following process:
●Trace out, from the start symbol, the series of productions that have not yet been completed and where we are in each production.
●For each production, in order, output all of the symbols up to the point where we change from one production to the next.

Idea: At each point, track
- ●Which production we are in, and
- ●Where we are in that production.

At each point, we can do one of two things:
- ●Match the next symbol of the candidate left-hand side with the next symbol in the current production, or
- ●If the next symbol of the candidate left-hand side is a nonterminal, nondeterministically guess which production to try next.

An Important Result
- ●There are only finitely many productions, and within those productions only finitely many positions.
- ●At any point in time, we only need to track where we are in one production.
- ●There are only finitely many options we can take at any one point.
- ●We can use a finite automaton as our recognizer.

Different LR Parsers:
●L: Left-to-right scan
●R: Rightmost derivation in Reverse
1) LR(0)
2) SLR (S – Simple) aka SLR(1)
3) LALR(LA – lookahead) aka LALR(1)
4) CLR(C – canonical) aka CLR(1) aka LR aka LR(1)

Why LR(0) is Weak
●LR(0) only accepts languages where the handle can be found with no right context
●Our shift/reduce parser only looks to the left of the handle, not to the right.
●How do we exploit the tokens after a possible handle to determine what to do?

Analysis of SLR(1)
●Exploits lookahead in a small space.
●Small automaton – same number of states as in as LR(0).
●Works on many more grammars than LR(0)
●Too weak for most grammars: lose context from not having extra states.

A Powerful Parser: LR(1)
Bottom-up predictive parsing with
●L: Left-to-right scan
●R: Rightmost derivation in Reverse
●(1): One token lookahead
●Substantiallymore powerful than the other methods we've covered so far.
●Tries to more intelligently find handles by using a lookahead token at each step

The Intuition behind LR(1)
●Guess which series of productions we are reversing.
●Use this information to maintain information about what lookahead to expect.
●When deciding whether to shift or reduce, use lookahead to disambiguate.

The Power of LR(1)
●Any LR(0) grammar is LR(1).
●Any LL(1) grammar is LR(1).
●Any deterministic CFL (a CFL parseable by a deterministic pushdown automaton) has an LR(1) grammar.

LR(1) Automata are Huge; Rarely used in Practice
●In a grammar with n terminals, could in theory be O(2n) times as large as the LR(0) automaton.
●Replicate each state with all O(2n) possible lookaheads.
●LR(1) tables for practical programming languages can have hundreds of thousands or even millions of states.
●Consequently, LR(1) parsers are rarely used in practice.

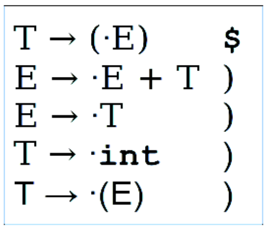
Why is LR(1) so powerful?
Intuitively, for two reasons:
1) Lookahead makes handle-finding easier.
●The LR(0) automaton says whether there could be a handle later on based on no right context.
●The LR(1) automaton can predict whether it needs to reduce based on more information.

2) More states encode more information.
●LR(1) lookaheads are very good because there's a greater number of states to be in.
●Goal: Incorporate lookahead without increasing the number of states.

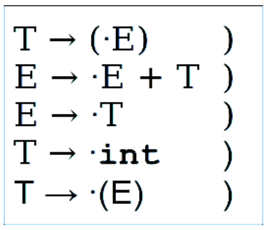
Why is SLR(1) Weak?
●With LR(1), incredible contextual information.
Lookaheads at each state only possible after applying the productions that could get us there.
●With SLR(1), minimalcontext.
FOLLOW(A) means “what could follow A somewhere in the grammar?,” even if in a particular state A couldn't possibly have that symbol after it.

LR(1) and SLR(1)
●SLR(1) is weak because it has no contextual information.
●LR(1) is impractical because its contextual information makes the automaton too big.
●Can we retain the LR(1) automaton's contextual information without all its states?

Review of LR(1) : LALR
●Each state in an LR(1) automaton is a combination of an LR(0) state and lookahead information.
●Two LR(1) items have the same core if they are identical except for lookahead.



$T \to (\cdot E)$	$
$E \to \cdot E + T$	)
$E \to \cdot T$	)
$T \to \cdot$**int**	)
$T \to \cdot(E)$	)



$T \to (\cdot E)$	)
$E \to \cdot E + T$	)
$E \to \cdot T$	)
$T \to \cdot$**int**	)
$T \to \cdot(E)$	)

In an LR(1) automaton, we have multiple states with the same core but different lookahead.
●What if we merge all these states together?
●This is called LALR(1)

Advantages of LALR(1)
●Maintains context.
●Lookup sets based on the fine-grained LR(1) automaton.
●Each state's lookup relevant only for that state.
●Keeps automaton small.
●Resulting automaton has same size as LR(0) automaton.

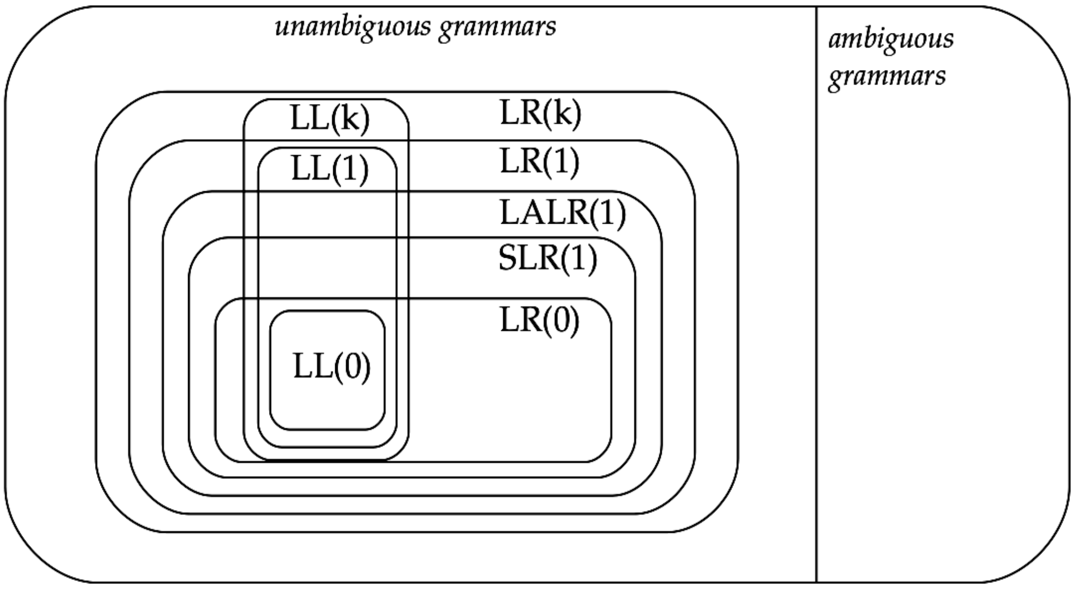
LALR(1) is Powerful
●Every LR(0) grammar is LALR(1).
●Every SLR(1) grammar is LALR(1)
●Most(but not all) LR(1) grammars are LALR(1).

LALR Merge Conflicts
●Merging LR(1) states cannot introduce a shift/reduce conflict. Why?
Since the items have the same core, a shift/reduce conflict in a LALR(1) state would have to also exist in one of the LR(1) states it was merged from.
●Merging LR(1) states can introduce a reduce/reduce conflict.
Often these conflicts appear without any good reason; this is one limitation of LALR(1).

The Limits of LR Parsers in general.
LR parsers use shift and reduce actions to reduce the input to the start symbol.
●LR parsers cannot deterministically handle shift/reduce or reduce/reduce conflicts.
●However, they can nondeterministically handle these conflicts by guessing which option to choose.

Error Handling
●What should the parser do when it encounters an error?
●Could just say "syntax error," but we'd like more detailed messages.
●How do we resume parsing after an error? [error-recovery strategies discussed in class]

Summary



Note this diagram refers to **grammars**, not **languages**, e.g. there may be an equivalent LR(1) grammar that accepts the same language as another non-LR(1) grammar. No ambiguous grammar is LL(1) or LR(1), so we must either rewrite the grammar to remove the ambiguity or resolve conflicts in the parser table or implementation.

The hierarchy of LR variants is clear: every LR(0) grammar is SLR(1) and every SLR(1) is LALR(1) which in turn is LR(1). But there are grammars that don't meet the requirements for the weaker forms that can be parsed by the more powerful variations.

We've seen several examples of grammars that are not LL(1) that are LR(1). But every LL(1) grammar is guaranteed to be LR(1). A rigorous proof is fairly straightforward from the definitions of LL(1) and LR(1) grammars. Your intuition should tell you that an LR(1) parser uses more information than the LL(1) parser since it postpones the decision about which production is being expanded until it sees the entire right side rather than attempting to predict after seeing just the first terminal

LL(1) vs. LALR(1)
Error repair: Both LL(1) and LALR(1) parsers possess the *valid prefix* property. What is on the stack will always be a valid prefix of a sentential form. Errors in both types of parsers can be detected at the earliest possible point without pushing the next input symbol onto the stack. LL(1) parse stacks contain symbols that are predicted but not yet matched. This information can be valuable in determining proper repairs. LALR(1) parse stacks contain information about what has already been seen, but do not have the same information about the right context that is expected. This means deciding possible continuations is somewhat easier in an LL(1) parser.

Efficiency**:** Both require a stack of some sort to manage the input. That stack can grow to a maximum depth of n, where n is the number of symbols in the input. If you are using the runtime stack (i.e. function calls) rather than pushing and popping on a data stack, you will probably pay some significant overhead for that convenience (i.e. a recursive descent parser takes that hit). If both parsers are using the same sort of stack, LL(1) and LALR(1) each examine every non-terminal and terminal when building the parse tree, and so parsing speeds tend to be comparable between the two.