# 50006 - Compilers - (Dr Dulay) Lecture 2

Oliver Killane

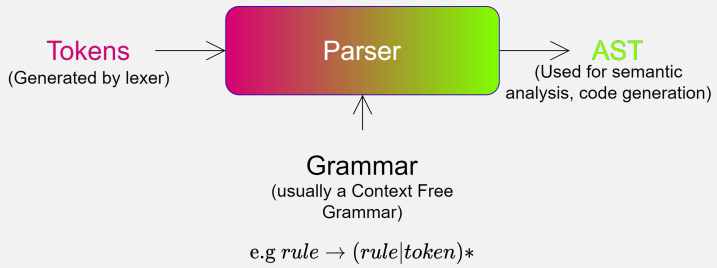
07/04/22

**Lecture Recording**

Lecture recording is available here

# Parsing

**Definition: Parsing (Static Analysis)**




Transforming a sequence of tokens into an **abstract syntax tree** using a language's **grammar**.

## Chomsky Hierarchy

A hierarchy of grammars and the machines required to parse them.

Given that:

| | |
|---|---|
| $R$ | non-terminal |
| t | sequence of tokens |
| $\alpha, \beta, \varphi$ | sequences of terminals and non-terminals |

We can express the rules as:

| | | | |
|---|---|---|---|
| Type 3 | $R \rightarrow t$ | Regular | DFA |
| Type 2 | $R \rightarrow \alpha$ | Context Free | Pushdown automata (DFA with a stack and read-only tape) |
| Type 1 | $\alpha\ R\ \beta \rightarrow \alpha\ \varphi\ \beta$ where $R \rightarrow \varphi$ | Context Sensitive | Linear Bounded Automata (Turing machine with limited tape) |
| Type 0 | $\alpha \rightarrow \beta$ | Unrestricted/recursively enumerable | Turing Machine |

regular $\subset$ context-free $\subset$ context-sensitive $\subset$ recursively enumerable

## Parsers for Context Free Grammars

**Context-free grammars** are parsed with complexity $O(n^3)$, **LL** and **LR** grammars are subsets of **CFG** that can be parsed in $O(n)$. Note that **LL(n)** is a subset of **LR(n)**.

Higher lookahead allows for more powerful & complex grammars, at the cost of performance (particularly memory).

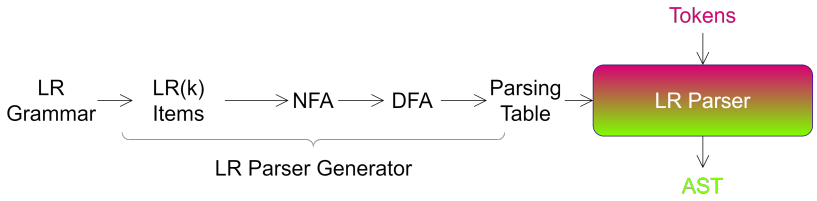
| | |
|---|---|
| **LL/Top Down Parsers** (Builds **AST** from root to leaves) | |
| **LL(k)** | **L**eft-to-right scan, **L**eftmost-derivation, **k**-token look-ahead. |
| **LL(0)** | Does not exist (needs to have lookahead of at least 1). |
| **LL(1)** | Most Popular. Can be implemented using recursive descent, r as an automaton. |
| **LL(2)** | Sometimes useful. |
| **LL(k > 2)** | Rarely Needed. |
| **LR/Bottom-Up Parsers** (Builds **AST** from leaves to root) | |
| **LR(k)** | **L**eft-to-right scan, **R**ightmost-derivation (in reverse) with **k**-token lookahead. |
| **LR(0)** | Weak & used in education. |
| **SLR(1)** | Stronger than **LR(0)**, but superseded by **LALR(1)** |
| **LALR(1)** | Fast, popular and comparable power to **LR(1)** |
| **LR(1)** | Powerful but has high memory usage. |
| **LR(k ≥ 2)** | Possible but rarely used. |

**Programming Language Grammars**

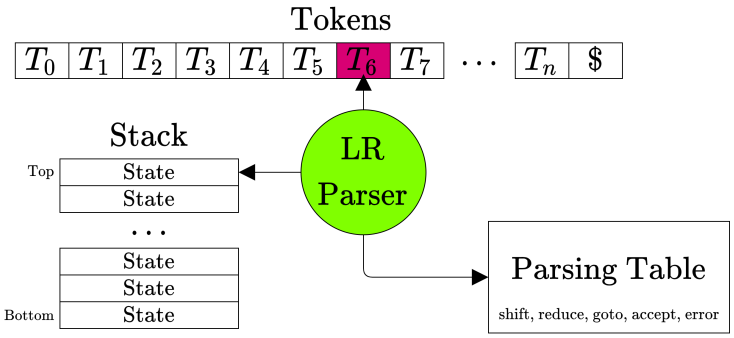
Most programming languages' syntaxs are described using **Context-Free Grammars**. Almost all **context-free** programming language grammars can be described as an **LR grammar** and implemented using an **LR parser**.

C++ is much more complex, as it is not a context-free grammar (very well explained here.)

# LR Parser



## LR Parser Operation




| | |
|---|---|
| *shift Sn* | Push state $n$ onto the stack and move to the next token. |
| *reduce Rn* | 1. Use rule $n$ to remove $m$ states from the stack (where $m$ = length of RHS of rule $n$).<br>2. Get the rule associated with the new-top of stack (i.e perform the goto action).<br>3. Push back the LHS of rule $n$.<br>4. Generate the new AST node for the rule. |
| *accept a* | Parse was successful |
| *error* | Report an error. Note that errors may be added to the AST under construction so multiply syntax errors can be reported. |
| *goto Gn* | Used in the reduce case, not directly. |

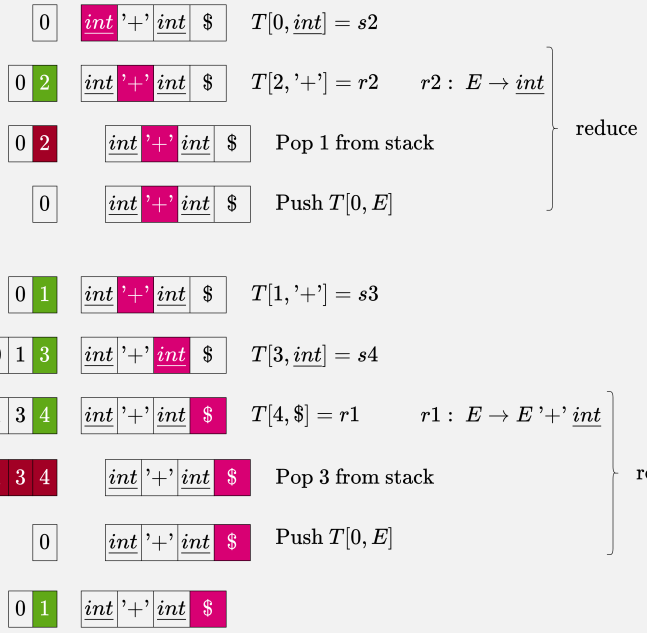
## Example: Basic Parser Operations

With a basic grammar:

$$E' \to E\ \$$$
$$r1: \quad E \to E\ \text{'+'}\ \underline{int}$$
$$r2: \quad E \to \underline{int}$$

We have an **LR(1) Parse Table**:

| State | Action | | | GOTO |
|---|---|---|---|---|
| | $\underline{int}$ | '+' | $ | $E$ |
| 0 | s2 | | | g1 |
| 1 | | s3 | a | |
| 2 | | r2 | r2 | |
| 3 | s4 | | | |
| 4 | | r1 | r1 | |




## End of Input

For **LR** parsers, an end of input symbol $ rule is always required.

$$E' \to E\ \$$$

## LR(0) Parsers

**Definition: LR(0) Items**

Instances of the rules of the grammar with $\bullet$ (representing the current position of the parser) in some position on the right hand side of the rule.

$$X \to \underbrace{AB\ldots}_{\text{Has been matched}} \bullet \underbrace{\ldots YZ}_{\text{To be matched}}$$

| | |
|---|---|
| $X \to \bullet ABC$ | **Initial Item** (has not yet matched any part of the rule) |
| $X \to A \bullet BC$ | |
| $X \to AB \bullet C$ | |
| $X \to ABC\bullet$ | **Complete Item** (Has matched the whole rule) |

We can use the **LR(0) items** as states in our **NFA** as they track thr progress of the parser through a rule.

We can create transitions between **LR(0) Items**:

$$\underbrace{(X \to A \bullet BC)}_{\text{current state}} \overbrace{\to_B}^{\text{If encountering a } B} \underbrace{(X \to AB \bullet C)}_{\text{new state}}$$

If $B$ is a non-terminal, for each rule $B \to \bullet D$ we add a $\epsilon$ transition (no token needed to match):

$$(X \to A \bullet BC) \to_\epsilon (B \to \bullet D)$$

**Definition: LR(0) Parsing Table**

The parsing table describes the rules to apply, and states to move to when a given item is encountered.

It is generated from a **DFA** using the rules:

| | |
|---|---|
| Terminal translation $X \rightarrow_T Y$ | Add $P[X, T] = sY$ (shift $Y$, we cannot reduce yet) |
| Non-terminal translation $X \rightarrow_N Y$ | Add $P[X, N] = gY$ (Goto $Y$) |
| State $X$ containing item $R' \rightarrow \cdots \bullet$ | Add $P[X, \$] = a$ (accept) (end of input) |
| State $X$ containing item $R \rightarrow \cdots \bullet$ | Add $P[X, T] = rN$ (reduce) (use rule $rN$ to reduce) |

| **State** | **Action** | | | **GOTO** |
|---|---|---|---|---|
| | *int* | '+' | $ | *E* |
| 0 | s3 | | | |
| 1 | | s4 | a | |
| 2 | r1 | r1 | r1 | |
| 3 | r3 | r3 | r3 | |
| ⋮ | | | | |

- 

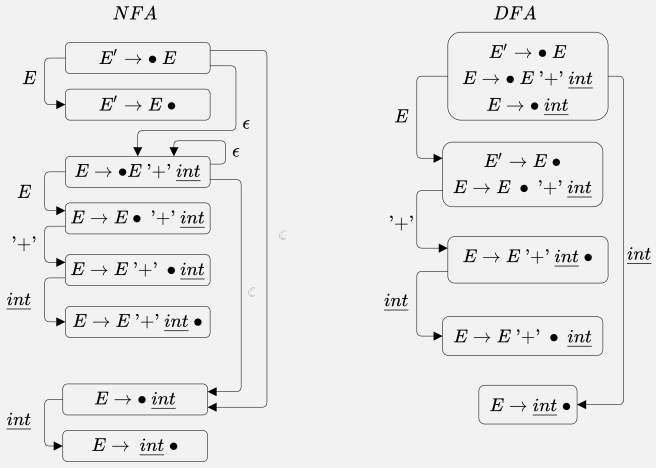
To create an **LR(0)** parser the steps are:

1. To generate an **LR(0)** parser the context free grammar is converted into **LR(0) items**.
2. The items are used as states in an **NFA**, with $\epsilon$ transitions for non-terminals.
3. The **NFA** is converted to a **DFA**.
4. Generate a **parsing table** from the **DFA**

Example: Basic **LR(0)** Example

For a language allowing only addition expressions.

| **Grammar** | **LR(0) Items** |
|---|---|
| $E' \to E\ \$$ | $E' \to \bullet E$ (Initial Item, the \$ is implied)<br>$E' \to E\bullet$ |
| $E \to E\text{'+'}\underline{int}$ | $E \to \bullet E\text{'+'}\underline{int}$ (Initial Item)<br>$E \to E \bullet \text{'+'}\underline{int}$<br>$E \to E\text{'+'} \bullet \underline{int}$<br>$E \to E\text{'+'}\underline{int} \bullet$ (Complete Item) |
| $E \to \underline{int}$ | $E \to \bullet\underline{int}$ (Initial Item)<br>$E \to \underline{int} \bullet$ (Complete Item) |




| **State** | **Action** | | | **GOTO** |
|---|---|---|---|---|
| | *int* | '+' | \$ | *E* |
| 0 | s3 | | | g2 |
| 1 | | s4 | a | |
| 2 | r1 | r1 | r1 | |
| 3 | r3 | r3 | r3 | |
| ⋮ | | | | |

## LR(1) Parsers

Lecture Recording

Lecture recording is available here

**Notation**

| | |
|---|---|
| $\rightarrow^*$ | Zero or more derivations. |
| $\rightarrow^+$ | One or more derivations. |
| $A \rightarrow \epsilon$ | Non-terminal $A$ is **nullable**. |
| $A \rightarrow AB \vert BC$ | $AB$ and $BC$ are **alternatives** of $A$. |

**Definition: First Set**

The **first set** for a sequence of rules (non-terminals) and tokens (terminals) $\alpha$ is the set of all tokens that could start a derivation $\alpha$.

$$\text{first}(\alpha) = \{t : \alpha \rightarrow^* t\beta\} \cup [\begin{cases} \{\epsilon\} & \text{if } \alpha \rightarrow *\epsilon \\ \{\} & otherwise \end{cases}]$$

- For a token $T$ the $\text{first}(T) = \{T\}$
- $\text{first}(\epsilon) = \{\epsilon\}$

We can construct **first set** from the rule by going through each alternative. In each alternative we iterate through each terminal/non-terminal $B$, if $\epsilon \in \text{first}(B)$ then we include $\text{first}(B) \setminus \{\epsilon\}$ in the **first set** and move on to the next in the sequence.

```
1  def get_first_set(non_terminal):
2      first_set = []
3      # go through each alternative
4      for alternative in non_terminal:
5          # iterate through all terminal/non-terminals in the alternative until
6          # one does not have an epsilon.
7
8          # If it has an epsilon, it means we could potentially skip through to
9          # the next production (e.g in "i*j" the first if i has epsilon, as we
10         # can skip it, hence we must also consider j).
11
12         for b in alternative:
13             b_first_set = get_first_set(b)
14             first_set += b_first_set - EPSILON
15
16             # if no epsilon in the next terminal/non-terminal, we stop adding
17             # to the first set.
18             if EPSILON not in get_first_set(b):
19                 break
```

**Definition: Follow Set**

The follow set for a non-terminal rule is the set of all tokens (terminals) that could immediately follow, and \$ if the end of the input could follow.

$$\text{Follow Set}(A) = \{t : S \to^+ \alpha A t \beta\} \cup \begin{cases} \{\$\} & \text{if } S \to *\alpha A \\ \{\} & otherwise \end{cases}$$

Hence we must check each rule which contains the non-terminal.

$$\text{Given rule } B \to C\ A\ D \text{ we have follow}(A) = \text{first}(D) \setminus \{\epsilon\} \cup \begin{cases} \text{follow}(B) & \text{if } D \to^* \epsilon \\ \{\} & otherwise \end{cases}$$

- $C$ can be empty here, since it is at the start of the rule, it has no bearing on the **follow set**.
- If $A$ can end the input, we include \$ in the **follow set**.

**Definition: LR(1) Items**

An **LR(1) Item** is a pair:

$$[\text{LR(0) item},\ \text{lookahead token } t]$$

Hence given an **LR(1) item**

$$[X \to A\ \bullet B\ C,\ t]$$

- $A$ is on top of the stack.
- We only want to recognise $B$ if it is followed by a string derivable from $Ct$.
- $B$ is followed by $Ct$ if the current token $\in$ first$(Ct)$.

### NFA Transitions

Given a state $[X \to A\ \bullet\ B\ C,\ t]$ we add the transition:

$$[X \to A\ \bullet\ B\ C,\ t] \to_B [X \to A\ B\ \bullet\ C,\ t]$$

If $B$ is a rule/non-terminal then for every rule of form $B \to D$ we add an $\epsilon$ transition and a new state for every token $u$ in first$(Ct)$ (Add a transition iff $B$ derivable from $Ct$).

$$[X \to A\ \bullet\ B\ C,\ t] \to_\epsilon [B \to \bullet\ D,\ u]$$

We also need an initial item for the rule $[X' \to \bullet\ X, \$]$ (start of text, has end of input token \$).

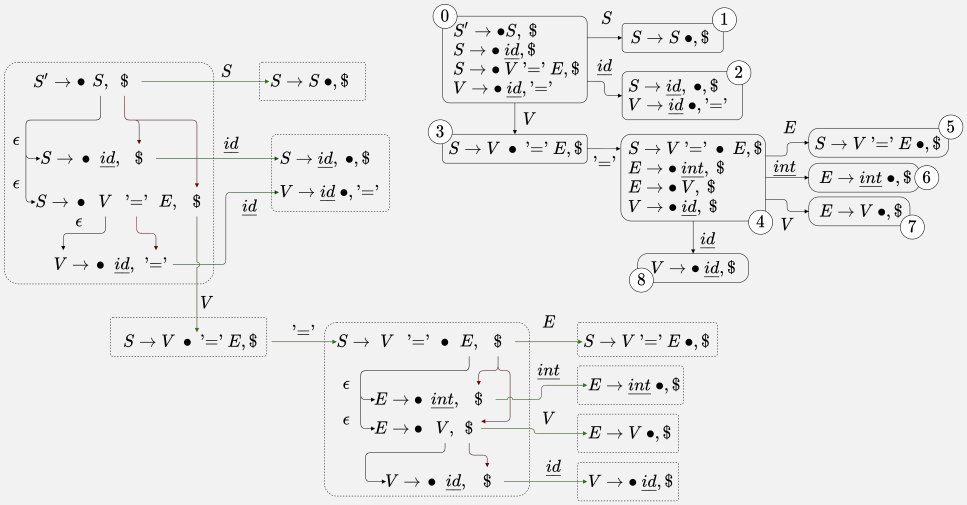
**Definition: LR(1) Parsing Table**

Our parsing table can now contain several different rules per row (same state, different current token).

We only perform reduction of a rule $[A \to A\ \bullet,\ t]$ when the current token is $t$.

## Example: Basic LR(1) Grammar

| Grammar | Expanded |
|---|---|
| | $S' \rightarrow S\ \$$ |
| $S \rightarrow \underline{id} \mid V\text{'='}E$ | $r1 : S \rightarrow \underline{id}$ |
| | $r2 : S \rightarrow V\text{'='}E$ |
| $V \rightarrow \underline{id}$ | $r3 : V \rightarrow \underline{id}$ |
| $E \rightarrow V \mid \underline{int}$ | $r4 : E \rightarrow V$ |
| | $r5 : E \rightarrow \underline{int}$ |




| State | Action | | | | GOTO | | |
|---|---|---|---|---|---|---|---|
| | *id* | *int* | '=' | $ | *E* | *V* | *S* |
| 0 | s2 | | | | | g3 | g1 |
| 1 | | | | a | | | |
| 2 | | | r3 | r1 | | | |
| 3 | | | s4 | | | | |
| 4 | | s8 | s6 | | | g5 | g7 |
| 5 | | | | r2 | | | |
| 6 | | | | r5 | | | |
| 7 | | | | r4 | | | |
| 8 | | | | r3 | | | |

# LALR(1) Parsers

**Definition: LALR(1) Items**

**LALR(1)** parsers are similar to **LR(1)** parsers, however **LR(1)** states that have the same **LR(0) items** and only differ in the lookahead token are merged. This reduces the memory usage of **LALR(1) Parsers**.

e.g LR(1) items $\begin{matrix}[R \to A\ B\ \bullet\ C, t_0] \\ [R \to A\ B\ \bullet\ C, t_1] \\ \vdots \\ [R \to A\ B\ \bullet\ C, t_n]\end{matrix}$ become LALR(1) Item $[R \to A\ B\ \bullet\ C, \{t_0, t_1, \ldots, t_n\}]$

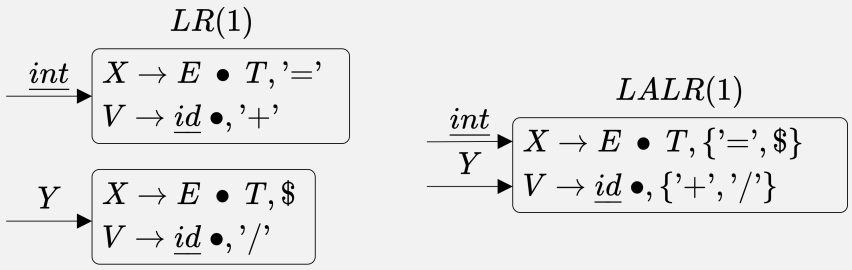
Due to this merging, some reductions can occur before an error is detected, which would've been immediately detected by an **LR(1)** parser.

$$\text{Grammar } \begin{matrix}A \to \text{'('}\ A\ \text{')'} \\ A \to +\end{matrix} \text{ given input string "+)"}$$

With **LR(1)** as soon as we get to + and shift it, we immediately have an error.

However for **LALR(1)** as we would two states $A \to$ '+'$\bullet$, \$ and $A \to$ '+'$\bullet$, ')' merged, we would detect the error after reaching the ')'.

**Example: LALR(1) Items**




# Conflicts and Ambiguity

**Definition: Ambiguous Grammars**

Grammars for which more than one parse tree can be created for some input.

Ambiguous grammars cannot be **LR(k)** as during parsing it will reach a state where it cannot decide wether to shift or reduce.

Ambiguity can be resolved two ways:

1. Rewrite the grammar to remove ambiguity.
2. Augment grammar with additional rules (e.g associativity or precedence)

For **LR** parser generators shifting is given priority, when a shoft-reduce confligt occurs, a warning is typically given.

**Definition: Shift-Reduce Conflict**

Caused by an ambiguity where the parser cannot decide if to reduce the tokens to the LHS of a rule, of continue to shift another token.

For example the grammar below would cause a **shift-reduce conflict**.

$$Expr \rightarrow \underline{Num} \quad \text{and} \quad Expr \rightarrow \underline{Num} \text{ '+' } \underline{Num}$$

**Example: Shift-Reduce Conflict**

Given the grammar:

$$S \rightarrow \text{'if' } E \text{ 'then' } S \text{ 'else' } S$$
$$S \rightarrow \text{'if' } E \text{ 'then' } S$$
$$S \rightarrow \underline{id}$$

We can generate **LR(1) items**:

$$[S \rightarrow \text{'if' } E \text{ 'then' } S \bullet, \text{'else'}]$$
$$[S \rightarrow \text{'if' } E \text{ 'then' } S \bullet \text{ 'else' } S, \underline{any}]$$

Given some input *"if a then if b then c else d"* we can parse:

$$\underset{\text{Shift}}{if\ a\ then\ \underbrace{if\ b\ then\ c\ else\ d}_{\text{inner if}}} \qquad \underset{\text{reduce}}{if\ a\ then\ \underbrace{if\ b\ then\ c}_{\text{inner if}}\ else\ d}$$

We can resolve this using a modified grammar:

$$S \rightarrow MatchedS$$
$$MatchedS \rightarrow \text{'if' } E \text{ 'then' } MatchedS \text{ 'else' } MatchedS$$
$$MatchedS \rightarrow other$$

$$S \rightarrow UnMatchedS$$
$$UnMatchedS \rightarrow \text{'if' } E \text{ 'then' } S$$
$$UnMatchedS \rightarrow \text{'if' } E \text{ 'then' } MatchedS \text{ 'else' } UnMatchedS$$

This grammar ensures only matched statements can come before the 'else' in an if statement, hence the ambiguity is resolved.

**Definition: Reduce-Reduce Conflict**

Caused by two rules having the same RHS. Cannot determine which rule should be applied.

For example the following grammar has a **reduce-reduce conflict**.

$$Expr \rightarrow \underline{id} \quad \text{and} \quad Var \rightarrow \underline{id}$$

A common disambiguation rule added for LR parser generators is to prioritise the earliest rule from the grammar source.

**Example: Conflict with Precedence**

For example the grammar:

$$Expr \rightarrow Expr \text{ '+' } Expr \mid Expr \text{ '-' } Expr \mid Expr \text{ '*' } Expr \mid \text{'(' } Expr \text{ ')'} \mid \underline{int}$$

Hence if given a source *"3 + 2 * 5"* we have a conflict

$$(3 + 2) * 5 \text{ or } 3 + (2 * 5)$$

Hence we can diambiguate by using several rules.

$$\begin{aligned} Expr &\rightarrow Expr \text{ '+' } Term \mid Term \\ Term &\rightarrow Term \text{ '*' } Factor \mid Factor \\ Factor &\rightarrow \text{'(' } expr \text{ ')'} \mid \underline{int} \end{aligned}$$

## Parse Tree

A parse tree is produced by the parser. It is broadly similar to the AST, but can contain much redundant information associated with the grammar.

- Leaf nodes built on a shift operations
- Non-Leaf nodes created on a reduce.

To produce the AST we can either build it in a pass over the completed parse tree, or associate AST construction code with rules in the grammar to build the AST during parsing.