# SWEN430 - Compiler Engineering

## Lecture 4 - Parsing

Lindsay Groves & David J. Pearce

*School of Engineering and Computer Science*
*Victoria University of Wellington*

# When does the Choice Condition Fail?

- $S \longrightarrow \mathtt{a}\ U \mid \mathtt{a}\ V$

- $S \longrightarrow \mathtt{a}\ U \mid T\ V$
  $T \longrightarrow \mathtt{a}\ \ldots \mid \ldots$

- $S \longrightarrow T \ldots \mid U \ldots$
  $T \longrightarrow \mathtt{a}\ \ldots \mid \ldots$
  $U \longrightarrow \mathtt{a}\ \ldots \mid \ldots$

- $S \longrightarrow \mathtt{a}\ T \mid S\ \mathtt{b}$

# Left factoring

Sometimes we can transform a non-LL(1) grammar into LL(1) form:

- $S \longrightarrow \mathtt{a}\ U \mid \mathtt{a}\ V$ $\qquad$ $S \longrightarrow \mathtt{a}\ (\ U \mid V\ )$

- $S \longrightarrow \mathtt{a}\ U \mid T\ \ V \mid c$ $\qquad$ $S \longrightarrow \mathtt{a}\ (\ U \mid X\ \ V) \mid T\ \ V \mid c$

  $T \longrightarrow \mathtt{a}\ X \mid \mathtt{b}\ Y$ $\qquad$ $T' \longrightarrow \mathtt{b}\ Y$

- $S \longrightarrow \mathtt{a}\ T \mid S\ \mathtt{b}$ $\qquad$ $S \longrightarrow \mathtt{a}\ T\ U$

  $U \longrightarrow \mathtt{b}\ U \mid \lambda$

Note: (i) We are using an extended grammar notation allowing nested alternatives.
(ii) The transformation changes the parse tree, especially in the last case.
(iii) Can't always do this!! Ex: Find a counterexample.

# Left factoring

Given productions $N \longrightarrow x\ \alpha$ and $N \longrightarrow x\ \beta$, where $x$ is either a terminal or non-terminal. We can rewrite this into $N \longrightarrow x\ M$ and $M \longrightarrow \alpha \mid \beta$.

- Example:

$$\begin{aligned} List &\longrightarrow (\ ) \mid (\ ListBody\ ) \\ ListBody &\longrightarrow ListElt \mid ListElt\ ,\ ListBody \\ ListElt &\longrightarrow N \mid List \end{aligned}$$

- Becomes [N=List, x=(, $\alpha$=), $\beta$=ListBody)] :

$$\begin{aligned} List &\longrightarrow (\ RestOfList \\ RestOfList &\longrightarrow )\ \mid ListBody\ ) \\ &\ldots \end{aligned}$$

# Eliminating Left Recursion

- Left recursion occurs frequently in PL grammars, e.g.:

$$E \longrightarrow E + T \mid T$$
$$T \longrightarrow n$$

- Using **right-recursion** instead gives:

$$E \longrightarrow T\ E'$$
$$E' \longrightarrow +\ T\ E' \mid \lambda$$
$$T \longrightarrow n$$

- Better to use an extended BNF grammar, allowing repetition as in regular expressions:

$$E \longrightarrow T\ (\ +\ T\ )^*$$
$$T \longrightarrow n$$

# When does it work? — Take two

- When else does the parser need to make a decision?
- Consider a grammar of the form:

$$\begin{array}{lcl} S & \longrightarrow & a\ T\ U \\ T & \longrightarrow & \lambda \mid b \\ U & \longrightarrow & b \end{array}$$

  Since $T$ can produce the empty string ($\lambda$), we need to decide whether a $b$ in the input is part of $T$ or part of $U$ (in which case $T$ is $\lambda$.
- If the grammar is:

$$\begin{array}{lcl} S & \longrightarrow & a\ T\ U \\ T & \longrightarrow & \lambda \mid b \\ U & \longrightarrow & c \end{array}$$

  There is no problem.

# Follow sets

Let $\gamma$ be a non-terminal symbol. Then *follow*($\gamma$) is the set of all terminal symbols which may follow a string derived from $\gamma$ in a sentence.

- For example:

$$
\begin{array}{lll}
S \longrightarrow T\,\mathtt{a} \mid U\,\mathtt{c} & & (1,2) \\
T \longrightarrow \mathtt{d}\,T \mid \mathtt{e} & & (3,4) \\
U \longrightarrow \mathtt{c}\,U \mid \lambda & & (5,6)
\end{array}
$$

follow(U) = { $\mathtt{c}$ }
follow(T) = { $\mathtt{a}$ }

- What does this tell us about production rules 5 and 6 ?
- Do we need *follow* sets if there are no $\lambda$ productions?

# LL(1) — Condition 2

Consider an $LL(1)$ Context-Free Grammar. For any non-terminal symbol $N$ where $N \Longrightarrow^{*} \lambda$, it must hold that $\mathit{first}(N) \cap \mathit{follow}(N) = \emptyset$.

- For example:

**(i)**

$$\begin{aligned} S &\longrightarrow \texttt{a} \mid T\ \texttt{b} \\ T &\longrightarrow \texttt{b} \mid \epsilon \end{aligned}$$

**(ii)**

$$\begin{aligned} S &\longrightarrow \texttt{(}\ T \mid \epsilon \\ T &\longrightarrow S\ \texttt{)}\ S \end{aligned}$$

- What are the follow() sets for these grammars?

# Recursive Descent Parsing with Extended BNF Grammars

- Extended BNF grammars allow:
  - Nested alternatives, using ( ... ) for grouping.
  - [*A*] to mean that *A* is optional.
  - $A^*$ to mean that *A* may occur 0 or more times.
  - $A^+$ to mean that *A* may occur 1 or more times.
- How can we extend the recursive descent to handle these forms?
- How do we extend the LL(1) conditions?

# Recursive Descent Parsing with Extended BNF Grammars

- To parse [$A$]:

```
if nextSym can start A then parseA;
```

- To parse $A^*$:

```
while nextSym can start A do parseA;
```

- To parse $A^+$:
  - Treat as $A\,A^*$

  ```
  parseA; while nextSym can start A do parseA;
  ```

  - Or:

  ```
  do parseA while nextSym can start A
  ```

- [$A$] and $A^*$ can produce $\lambda$, so need to apply $\lambda$ Condition to them.