Top Down Parsing

- When we are parsing, we produce a *unique* syntax tree from a legal sentence.
  - An unambiguous grammar gives rise to a single leftmost derivation for any sentence in the language.
- So, if we are trying to recognise a sentence, what we are trying to do is grow a parse tree corresponding to that sentence
  - We are trying to find the leftmost derivation.
- A top-down parser constructs a leftmost parse
  - We will always be looking at the *leftmost* nonterminal.
- This follows the push down automaton model of the previous lecture
- The parser must choose the *correct* production from the set of productions that correspond to the current state of the parse.
- If at any time there is no candidate production corresponding to the state of the parse, we must have made a wrong turn at some earlier stage and we will need to *backtrack*.

Example

Suppose we have a grammar:

S → E
E → T | E+T
T → F | T*F
F → unit | (E)

and the expression:
1 + 2 * 3

A legal parse would be:

1. S	1 + 2 * 3
only one rule: S → E	
2. E	1 + 2 * 3
choose E → E+T	
3. E+T	1 + 2 * 3
choose E → T → F→ unit	
4. unit+T	<u>1 +</u> 2 * 3
match 1 and +	
5. 1+T	2 * 3
choose T → F→ unit	
6. 1+unit	<u>2</u> * 3
match 2	
7. 1+2	*3
✷WRONG	

This means that to parse this sentence some backtracking is required, i.e., put input symbols back! Backtracking in compilers is nontrivial and to be avoided!!

Example #2

Suppose we have a grammar:

S → E
E → T | E+T
T → F | T*F
F → unit | (E)

and the expression:
1 + 2 * 3

A legal parse would be:

1. S	1 + 2 * 3
only one rule: S → E	
2. E	1 + 2 * 3
choose E → E+T	
3. E+T	1 + 2 * 3
choose E → E+T	
4. E+T+T	1 + 2 * 3
choose E → E+T	
5. E+T+T+T	1 + 2 * 3
choose E → E+T	
...you can see what happens!	

The problem is simple: Left Recursion!

Solutions?

- We *could* rearrange the productions so that the left recursive ones come at the end, and always choose the *first matching* production.
- For the previous examples, this has already been done. The left recursive ones are at the end of the list!
- Note that this is not an easy task in general since mutually recursive grammars have the same problems:

  A → B | C D
  C → E | A F
- In general, rearranging productions will not help – the parser will still have problems.
  - Even if it does help, a parser which needs to backtrack an arbitrary distance is *inefficient*.
- What we need is a way to ***deterministically*** parse a grammar in a top down fashion <u>without backtracking</u>.

Eliminating left recursion

- An algorithm to eliminate arbitrary left recursion (by replacing it with right recursion) is as follows:

1. Arbitrarily order the non-terminals: N_1, N_2, N_3, ...
2. Apply the following steps to the productions for N_1, then N_2, ...
3. For N_i:
   a) For all productions $N_i \rightarrow N_k\ \alpha$, where $k < i$ and if the productions for N_k are $N_k \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$ then expand the reference to N_k, i.e. replace the production $N_i \rightarrow N_k\ \alpha$ by $N_i \rightarrow \beta_1\ \alpha \mid \beta_2\ \alpha \mid \ldots$
   b) If the productions for N_i are now
      $N_i \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid N_i\ \beta_1 \mid N_i\ \beta_2 \mid \ldots$
      (where the first few are not left recursive while the latter are) then replace them with
      $N_i \rightarrow \alpha_1\ N_i' \mid \alpha_2\ N_i' \mid \ldots$
      $N_i' \rightarrow \varepsilon \mid \beta_1\ N_i' \mid \beta_2\ N_i' \mid \ldots$

Example of eliminating left recursion

- Consider the productions:
  $A \rightarrow a \mid Ba$ $\quad B \rightarrow b \mid Cb$ $\quad C \rightarrow c \mid Ac$

1. Arbitrarily order the non-terminals: A, B, C
2. Consider the productions for A: no change
2. Consider the productions for B: no change
3. Consider the productions for C:
   a) Replace $C \rightarrow Ac$ by $C \rightarrow ac \mid Bac$
   a) Replace $C \rightarrow Bac$ by $C \rightarrow bac \mid Cbac$

   Productions for C are now: $C \rightarrow c \mid ac \mid bac \mid Cbac$

   b) Replace the productions for C by:
      $C \rightarrow cC' \mid acC' \mid bacC'$
      $C' \rightarrow \varepsilon \mid bacC'$

A Workable Solution

Observation

- The trouble which gives rise to nondeterminacy and backtracking in top down parsers shows itself in only one place – that is when a parser has to choose between several alternatives with the same left hand side.
- The only information which we can use to make the *correct decision* is the input stream itself.
  - —In the example, we (humans) could see which alternative to choose by looking at the input yet-to-be-read.
- If we are going to *look ahead* in order to make the correct decision, we need a buffer in which to store the next few symbols.
- In practice, this buffer is of a fixed length.

Definitions

- A parser which can make a deterministic decision about which alternative to choose when faced with one, if given a buffer of k symbols, is called a LL(*k*) parser.
  - —Left to right scan of input
  - —Left most derivation
  - —*k* symbols of look-ahead
- The grammar that an LL(k) parser recognizes is an LL(k) grammar and any language that has an LL(k) grammar is an LL(k) language.
  - —We are constructing an LL(1) compiler that recognises LL(1) grammars.
  - —So the question is *How do we know when we have an LL(1) grammar?*
- We also have LR(*k*) grammars and other variations, but our focus is currently on LL(1) grammars.

Definition of LL(1)

- When faced with a production such as:

  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3$

- We chose one of the α_i *uniquely* by looking at the ***next input symbol***.
- We employ two sets: first and follow, to help us.

Recall:

- First(X) is the set of all terminal symbols that can "start" the production X
- Follow(X) is the set of terminal symbols that can follow an "X"

Definition of First

- To compute FIRST(X) for all grammar symbols X, apply the following algorithm until no more terminals or ε can be added to any FIRST set.

1. If X is a terminal, then FIRST(X) is {X}
2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X)
3. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_n$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ε is in all of FIRST(Y_1), ..., FIRST (Y_{i-1}); that is $Y_1 Y_2 \ldots Y_{i-1} \Rightarrow^* \varepsilon$.
   If ε is in FIRST(Y_j) for all j = 1, 2, ... n, then add ε to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ε, then we add nothing more to FIRST(X), but if $Y_1 \Rightarrow \varepsilon$ then we add FIRST(Y_2) and so on.

Definition of Follow

- To compute FOLLOW(A) for all nonterminals A, apply the following algorithm until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker.
2. If there is a production $A \rightarrow \alpha B \beta$ then everything in FIRST(β) except for ε is placed in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B\beta$ where FIRST(β) contains ε (i.e., $\beta \Rightarrow^* \varepsilon$), then everything in FOLLOW(A) is in FOLLOW(B).

Definition of LL(1) property

Definition: A grammar G is LL(1) if and only if for all rules

$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$

- director(α_i) $\cap$ director(α_k) = $\emptyset$ $\forall$ i $\neq$ k

where:

$$\text{director}(\alpha_i) = \begin{cases} \text{first}(\alpha_i) \cup \text{follow}(A) & \text{if } \alpha_j \Rightarrow^* \varepsilon \\ \text{first}(\alpha_i) & \text{otherwise} \end{cases}$$

Making Grammars LL(1)

- We can't always make a grammar which is not LL(1) into an *equivalent* LL(1) grammar.
- Some tricks to help are factorisation and substitution.

Consider the grammar
S → T
T → L B | L C **array**
L → **long** | ε
C → B | ε
B → **real** | **integer**

Transform the grammar by factorisation:
S → T
T → L X
X → B | C **array**
L → **long** | ε
C → B | ε
B → **real** | **integer**

✷ Note that it still is not LL(1)!

Fixing the problem

S → T
T → L B | L C **array**
L → **long** | ε
C → B | ε
B → **real** | **integer**

1

Transform the grammar by factorisation:
S → T
T → L X
X → B | C **array**
L → **long** | ε
C → B | ε
B → **real** | **integer**

2

Substitute for C wherever it occurs:
S → T
T → L X
X → B | B **array** | **array**
L → **long** | ε
B → **real** | **integer**

3

Factorisation of B in X gives:
S → T
T → L X
X → B Y | **array**
Y → **array** | ε
L → **long** | ε
B → **real** | **integer**