Parsing

- Parsing involves:
  - determining if a string belongs to a language, and
  - constructing structure of string if it belongs to language.
- Two approaches to constructing parsers:
  1. Top down parsing: Our focus is on table-driven predictive parsing.
     Perform a left most derivation of string.
  2. Bottom up: SLR(1), Canonical LR(1), and LALR(1).
     Perform a right-most derivation in reverse.
- Common theme: Finite state controller for stack automaton. In top down parsing, the states were implicit. In LR parsing the states are all explicit.

Top Down Parsing

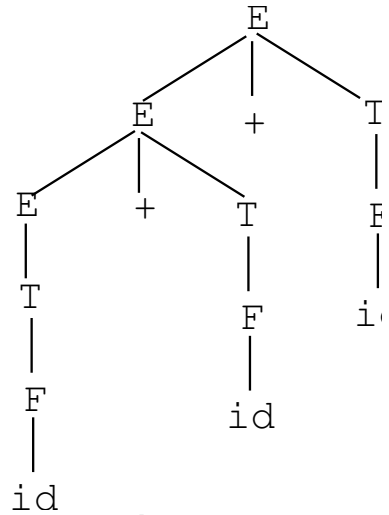
Introduction

- Top-down parser starts at the top of the tree, and tries to construct the tree that led to the given token string.
  Can be viewed as an attempt to find *left-most* derivation for an input string.
- Constrains on a top-down parser:
  1. Start from the root, and construct tree solely from tokens and rules.
  2. It must scan tokens left to right.
- Recursive descent as well as non-recursive predictive parsers.
- Approach for a table driven parser:
  - Construct a CFG. CFG must be in a certain specific form. If not, apply transformations. (We will do these last).
  - Construct a table that uniquely determines what productions to apply given a nonterminal and an input symbol.
  - Use a parser driver to recognize strings.

Grammar Transformations

- Left recursion:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow \textbf{id} \mid (E)$$



- Top down parser will start by expanding E to $E + T$, then expand E to $E + T$ again. It may therefore keep expanding on it infinitely often. It could have expanded using rule $E \rightarrow T$, but given the input string. Also, since it has not consumed any input, it must keep expanding using some rule, in this case the same rule.

  No top down parsers can handle left-recursive grammars.
- Solution: rewrite the grammar so that left recursion can be eliminated. Note: two kinds of left recursion: Immediate and indirect.

  $A \rightarrow B\alpha \mid \cdots$
  $B \rightarrow A\beta \mid \cdots$

Behavior of Parser

- Parser may create the parse tree by applying all possible rules until a parse tree is constructed.
- Consider grammar G1 (shown below) and string *bcde*.

  $S \rightarrow ee \mid bAc \mid bAe$

  $A \rightarrow d \mid cA$
- Behavior of parser as it tries to build the parse tree:

  $S \Rightarrow bAc \Rightarrow bcAc \Rightarrow bcdc$

  ***Show the parse tree.**
- Parser must backtrack now. Here it must backtrack all the way up to the root and try rule $S \rightarrow bAe$

  What kind of search?
- Problems: As the parser creates the parse tree, it uses up the tokens. When it backtracks, it must be able to go back to tokens it has already consumed. If scanner is under control of parser, scanner also must backtrack to produce the tokens again or parser must have a separate buffer.
- Backtracking slows down parsing and hence not an attractive approach.
- Chage the grammar:

  $S \rightarrow ee \mid bAQ$

  $Q \rightarrow c \mid e$

  $A \rightarrow d \mid cA$

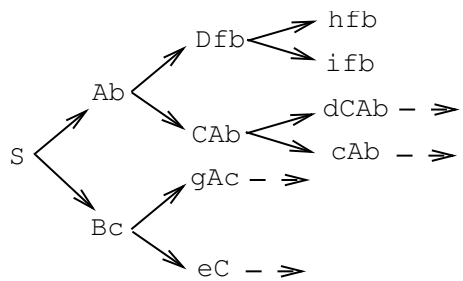
  Factor out the common prefix. Now parser can grow the tree without backtracking.

Predictive Parsers

- Compute the terminal symbols that a terminal can produce. Use this information to select a rule during derivation.
- Consider grammar:

$$
\begin{aligned}
S &\rightarrow Ab \mid Bc \\
A &\rightarrow Df \mid CA \\
B &\rightarrow gA \mid e \\
C &\rightarrow dC \mid c \\
D &\rightarrow h \mid i
\end{aligned}
$$

- For an input string *gchfc*, a simple parser may have to do great deal of backtracking before it finds the derivation. Backtracking can be avoided if parser can look ahead in grammar to anticipate what symbols are derivable. Consider possible leftmost derivation starting from S:



  Choose $S \rightarrow Ab$ if string begins with c, d, h, or i.
  Choose $S \rightarrow Bc$ if string begins with g, e.
- First: terminals that can begin strings derivable from a nonterminal.

  $\texttt{First}(Ab) = \{c, d, h, i\}$
  $\texttt{First}(Bc) = \{e, g\}$
- Parsers that use `First` are known as *predictive parsers*.

Non-recursive Predictive Parsers for LL(1) grammars

- Skip recursive descent predictive parsers (hopefully you wrote one in ECS 140A).
- Consists of a simple control procedure that runs off a table:
  1. Input buffer
  2. Stack
  3. Parsing table, M[A, a].
  4. Output stream
- Constructing parse $\Longrightarrow$ constructing table. We will look at it later.
- Table: for each nonterminal, specify the rule that should be used to expand the nonterminal for a given input symbol.
- Example table for Grammar:
  1. $E \rightarrow TQ$
  2. $T \rightarrow FR$
  3. $Q \rightarrow +TQ \mid -TQ \mid \epsilon$
  4. $R \rightarrow *FR \mid /FT \mid \epsilon$
  5. $F \rightarrow (E) \mid \textbf{id}$

	id	+	−	*	/	(	)	$
E	TQ					TQ		
Q		$+TQ$	$-TQ$				ϵ	ϵ
T	FR					FR		
R		ϵ	ϵ	$*FR$	$/FR$		ϵ	ϵ
F	**id**					(E)		

Blanks: error conditions.

Behavior of Parser

- Initially stack contains $S\$$ with S at top, and input contains $w\$$.
- Behavior of parser at each step: let X be at the top of stack, and a be a symbol:
  1. $X = a \neq \$$: pop X off stack and advance to next symbol.
  2. X is NT, consult table $M[X, a]$. If $M[X, a] = Y_1 Y_2 \cdots Y_n$:
     2.1 pop X off
     2.2 Push $Y_1 Y_2 \cdots Y_n$ on stack with Y_1 on top.

     If no entry, issue error. At this step, parser has determined the rule that can be applied and uses that for derivation.
  3. $X = a = \$$: Parser halts. That is, we have matched all symbols.

Stack	Input	Production
$\$E$	$(\mathbf{id} + \mathbf{id}) * \mathbf{id}\ \$$	
$\$QT$	$(\mathbf{id} + \mathbf{id}) * \mathbf{id}\ \$$	$E \rightarrow TQ$
$\$QRF$	$(\mathbf{id} + \mathbf{id}) * \mathbf{id}\ \$$	$T \rightarrow FR$
$\$QR)E($	$(\mathbf{id} + \mathbf{id}) * \mathbf{id}\ \$$	$F \rightarrow (E)$
$\$QR)E$	$\mathbf{id} + \mathbf{id}) * \mathbf{id}\ \$$	Pop token
$\$QR)QT$	$\mathbf{id} + \mathbf{id}) * \mathbf{id}\ \$$	$E \rightarrow TQ$
$\$QR)QRF$	$\mathbf{id} + \mathbf{id}) * \mathbf{id}\ \$$	$T \rightarrow FR$
$\$QR)QR\mathbf{id}$	$\mathbf{id} + \mathbf{id}) * \mathbf{id}\ \$$	$F \rightarrow \mathbf{id}$
$\$QR)QR$	$+\mathbf{id})*\mathbf{id}\ \$$	Pop token
$\$QR)Q$	$+\mathbf{id})*\mathbf{id}\ \$$	$R \rightarrow \epsilon$
$\$QR)QT+$	$+\mathbf{id})*\mathbf{id}\ \$$	$Q \rightarrow +TQ$
$\$QR)QT$	$\mathbf{id})*\mathbf{id}\ \$$	pop token
$\$QR)QRF$	$\mathbf{id})*\mathbf{id}\ \$$	$T \rightarrow FR$
$\$QR)QR\mathbf{id}$	$\mathbf{id})*\mathbf{id}\ \$$	$F \rightarrow \mathbf{id}$
$\$QR)QR$	$)*\mathbf{id}\ \$$	Pop token
$\$QR)Q$	$)*\mathbf{id}\ \$$	$R \rightarrow \epsilon$
$\$QR)$	$)*\mathbf{id}\ \$$	$Q \rightarrow \epsilon$
$\$QR$	$*\mathbf{id}\ \$$	Pop token
$\$QRF*$	$*\mathbf{id}\ \$$	$R \rightarrow *FR$
$\$QRF$	$\mathbf{id}\ \$$	Pop token
$\$QR\mathbf{id}$	$\mathbf{id}\ \$$	$F \rightarrow \mathbf{id}$
$\$QR$	$\$$	Pop token
$\$Q$	$\$$	$R \rightarrow \epsilon$
$\$$	$\$$	$Q \rightarrow \epsilon$
		Accept

LL(1) Parser

- Example:

Stack	Input	Production
$\$E$ (**id** $*$)$		
$\$QT$	(**id** $*$)$	$E \rightarrow TQ$
$\$QRF$	(**id** $*$)$	$T \rightarrow FR$
$\$QR)E($	(**id** $*$)$	$F \rightarrow (E)$
$\$QR)E$	**id** $*$)$	Pop token
$\$QR)QT$	**id** $*$)$	$E \rightarrow TQ$
$\$QR)QRF$	**id** $*$)$	$T \rightarrow FR$
$\$QR)QR$**id**	)	**id** $*$)$ $F \rightarrow cfgId$
$\$QR)QR$	$*$)$	Pop token
$\$QRF*$	$*$)$	$T \rightarrow +FR$
$\$QRF$	)$	Error

- A correct, leftmost parse is guarateed.
- All grammars in the LL(1) class are unambiguous: ambiguity implies two or more distinct leftmost parse. This means more than one correct predictions possible.
- LL(1) parsers operate in linear time, and at most, linear space (relative to the length of the input being parsed).

First

- First: terminals that can begin strings derivable from a nonterminal.
- Algorithm for evaluating First(α):
  1) α is a single character or ϵ:
     - terminal or $\epsilon \implies \text{First}(\alpha) = \alpha$
     - Nonterminal and $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \implies$ $\text{First}(\alpha) = \cup_k \text{First}(\beta_k)$
  2) $\alpha = X_1 X_2 \cdots X_n$:

```
First(α) = {};
j := 0;
repeat
   j := j + 1;
   include First(X_j) in First(α)
until X_j does not derive ε or j = n;
if X_n derives ε, add {ε} to First(α).
```

- Example:

$$
\begin{aligned}
S &\rightarrow ABCd \\
A &\rightarrow e \mid f \mid \epsilon \\
B &\rightarrow g \mid h \mid \epsilon \\
C &\rightarrow p \mid q
\end{aligned}
$$

$$
\begin{aligned}
\text{First}(ABCd) = \{e, f\} &\cup \\
\{g, h\} &\cup \\
\{p, q\} &= \{e, f, g, h, p, q\}
\end{aligned}
$$

Follow

- Sometimes `First` does not contain enough information for the parser to choose the right rule for derivation, especially when grammar contains $\epsilon-$productions:

  $$\begin{aligned} S &\rightarrow XY \\ X &\rightarrow a \mid \epsilon \\ Y &\rightarrow c \end{aligned}$$

  What does X do when it sees on input c?
- Follow tells us when to use ϵ productions: check whether the forthcoming token is in the `First` set. If it is not and there is a ϵ production, check if the token is in `Follow`. If it is, then use the ϵ production. Else there is a parsing error.
- Follow(A): set of all *terminals* that can come right after A in any sentential form.
- Assume that end of string denoted by \$.
- Algorithm for evaluating Follow(A):
  1. If A is starting symbol, put \$ in Follow(A).
  2. For all productions of form $Q \rightarrow \alpha A \beta$:
     a) if β begins with a terminal a, add a to Follow(A) otherwise Follow(A) includes First(β)$-\{\epsilon\}$.
     b) if $\beta = \epsilon$ or if β derives ϵ, add Follow(Q) in Follow(A).

Example: First and Follow

- Grammar:
  1. $E \rightarrow TQ$
  2. $T \rightarrow FR$
  3. $Q \rightarrow +TQ \mid -TQ \mid \epsilon$
  4. $R \rightarrow *FR \mid /FT \mid \epsilon$
  5. $F \rightarrow (E) \mid \textbf{id}$
- First:

  $\texttt{First}(E) = \texttt{First}(T)$
  $= \texttt{First}(F)$
  $= \{(, \textbf{id}\ \}$
  $\texttt{First}(Q) = \{+, -, \epsilon\}$
  $\texttt{First}(R) = \{*, /, \epsilon\}$
- Follow:

  $\texttt{Follow}(E) = \{\$,)\}$
  $\texttt{Follow}(Q) = \texttt{Follow}(E)$
  $\texttt{Follow}(T) = \{+, -,), \$\}$
  $\texttt{Follow}(R) = \{+, -,), \$\}$
  $\texttt{Follow}(F) = \{+, -,), *, /, \$\}$

Constructing Parser Table

- Motivation: For a symbol X on stack, and a as input symbol, select a right hand i) which begins with a or ii) can lead to a sentential form beginning with a:
- Algorithm:
  forall productions of the form $X \rightarrow \beta$
  1. for all terminal a in First(β) except ϵ
     $M[X, a] = \beta$
  2. if $\epsilon \in$ First(β),
     for all terminal $b \in$ Follow(X),
     $M[X, b] = \beta$
     if \$$\in$ Follow(X),
     $M[X, \$] = \beta$
- Example:

  First(E) = First(T) = First(F) = {(, **id** }.
  First(Q) = $\{+, -, \epsilon\}$ First(R) = $\{*, /, \epsilon\}$
  First($+TQ$) = $\{+\}$, First($-TQ$) = $\{-\}$
  First($*RF$) = $\{*\}$, First($/RF$) = $\{/\}$
  Follow(E) = $\{\$,)\}$, Follow($T$) = $\{+, -,), \$\}$
  Follow(Q) = Follow(E) = {\$,)}
  Follow(R) = Follow(T) = $\{+, -,), \$\}$
  Follow(F) = $\{+, -, *, /,), \$\}$

Paritally filled table:

	id	+	−	*	/	(	)	\$
E	TQ					TQ		
Q		$+TQ$	$-TQ$				ϵ	ϵ

LL(1) Grammars

- Grammars for which `First` and `Follow` can be used to uniquely determine which production to apply are called `LL(1)` grammars. Also, if all entries in `M` contain a unique prediction.
  `1` denotes 1 character look-ahead: One character lookahead tells us that every incoming token uniquely determines which production to choose
- Not always easy to write LL(1) grammars because LL(1) requires a unique derivation for each combination of nonterminal and lookahead symbol.
- Most conflicts arise due to existence of the following two in grammars: *common prefixes* and *left recursion*. Simple grammar transformations can be used to eliminate them.

Transforming common prefixes

- Common prefix:
  `S` $\rightarrow$ `if E then S else S` | `if E then S`
  On seeing `if`, we cannot decide which rule to use.
- Algorithm:
  Replace each production of form $A \rightarrow \alpha\beta_1|\alpha\beta_2|\cdots|\alpha\beta_n|\gamma$ by

$$A \rightarrow \alpha A'|\gamma$$
$$A' \rightarrow \beta_1|\beta_2|\cdots|\beta_n$$

Algorithm for removing left recursion

- Two kinds of left recursion: Immediate and indirect:
  $E \rightarrow E + T$
  $A \rightarrow B\alpha \mid \cdots$
  $B \rightarrow A\beta \mid \cdots$
- Removal of immediate left recursion:
  for each nonterminal
  1. Separate left-recursive productions from others
     $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots$
     $A \rightarrow \delta_1 \mid \delta_2 \mid \cdots$
  2. Introduce a new NT A', and change non-left-recursive rules:
     $A \rightarrow \delta_1 A' \mid \delta_2 A' \mid \cdots$
  3. Remove left recursive productions and Add: $A' \rightarrow \epsilon \mid \alpha_1 A' \mid \alpha_2 A' \mid \cdots$
- Example:
  `Original grammar:`
  $A \rightarrow Ac \mid Ad \mid e \mid f$
  `Modified grammar:`
  $A \rightarrow eA' \mid fA'$
  $A' \rightarrow \epsilon \mid cA' \mid dA'$
- Skip removal of indirect left recursion. (Algorithm for this one works by removing all indirect recursion and applying the previous algorithm)

LL(1) Grammars

- Factoring and left recursion removal are primary transforms used to make grammars LL(1). However, in certain cases more transformations needed: Example: following grammar used in a language that allows idetifiers as labels.

  `Stmt` $\rightarrow$ `Label UnlabeledStmt`
  `Label` $\rightarrow$ **id** `:` $\mid \epsilon$
  `UnlabeledStmt` $\rightarrow$ **id** `:= Expr`

  Problem: Symbol **id** predicts both `Label` productions.
  Solution: Factor **id** from productions:

  `Stmt` $\rightarrow$ **id** `IdSuffix`
  `IdSuffx` $\rightarrow$ `:` `UnlabeledStmt` $\mid$ `:= Expr`
  `UnlabeledStmt` $\rightarrow$ **id** `:= Expr`

- Another example: Array declaration in ADA:

  `ArrayBound` $\rightarrow$ `Expr ..` `Expr` $\mid$ **id**

  Problem: **id** can be generated from `Expr` as well. Factoring **id** from `Expr` can be tedious as it may define many other expressions.
  Solution:
  `ArrayBound` $\rightarrow$ `Expr BoundTrail`
  `BoundTrail` $\rightarrow$ `..` `Expr` $\mid \epsilon$

  If a single `Expr`, it must generate **id** . Checked during semantic analysis phase.

LL(1) Grammars

- Dangling else problem: Most constructs specified by LL(1) grammars, except for `if-then-else` construct of Algol 60 and Pascal: there may be more `then` parts than smtt else parts.
- Can model as a matching problem: treat `if Expr then Stmt` as open bracket, and `else Stmt` as optional closing bracket. The following represents the language:

$$L = \{[^i \ ]^j \mid i \geq j \geq 0\}$$

- L is not LL(1), in fact, is not LL(k) for any k.
- Technique used to handle dangling else problem: Use an ambiguous grammar along with special case rules to resolve any non-unique predictions that arise.

$G \rightarrow S;$
$S \rightarrow$ **if** $S\ E \mid$ **Other**
$E \rightarrow$ **else** $S \mid \epsilon$

	if	**else**	**Other**	;
S	**if** $S\ E$		**Other**	
E		**else** S ϵ	ϵ	
G	$S;$	$S;$		

Multiple entries due to ambiguity in grammar.

- Use Auxiliary rule: `else` associates with with the nearest `if`. That is, in predicting E, if we see `else` as a lookahead, we will match it immediately . Thus $M[E,$ **else** $\ ] =$ **else** $\ S$.
- A language design issue. Can be easily resolved by specifying that all if statements are terminated with **endif** :

$S \rightarrow$ **if** $\ S\ E \mid$ **Other**
$E \rightarrow$ **else** $\ S$ **endif** $\mid$**endif**

LL(k) Grammars

- Grammar G is LL(k) iff the three conditions:
  1. $S \overset{*}{\Rightarrow} wA\alpha \Rightarrow w\beta\alpha \overset{*}{\Rightarrow} wx$
  2. $S \overset{*}{\Rightarrow} wA\alpha \Rightarrow w\gamma\alpha \overset{*}{\Rightarrow} wy$
  3. $First_k(x) = First_k(y)$

  imply that $\beta = \gamma$.
  LL(k) grammar: Lookahead of k symbols, that is G is LL(k) iff, knowing w symbols to be expanded, A, and the next k input symbols, $First_k(x) = First_k(y)$ is always sufficient to uniquely determine the next prediction.
  Are there LL(k) grammars that are not LL(1)?
  Example:

  $$\begin{array}{lcl} G & \rightarrow & S \\ S & \rightarrow & aAa \mid bAba \\ A & \rightarrow & b \mid \epsilon \end{array}$$

  Grammar is not LL(1) because input symbol b predicts both productions of A: consider context aAa. A look ahead of ba predicts $A \rightarrow b$, and $a\$$ predicts $A \rightarrow \epsilon$.
- The following containment results hold: LL(k) $\subset$ LL(k+1)
- LL(k), $k > 1$ grammars are primarily of academic interest, as only LL(1) parsers are used in practice.