Top-Down Parsing

Compiler Design Syntax Analysis
s.l. dr. ing. Ciprian-Bogdan Chirila
chirila@cs.upt.ro
http://www.cs.upt.ro/~chirila
Outline

- Recursive-Descent Parsing
- FIRST and FOLLOW
- LL(1) Grammars
- Non-recursive Predictive Parsing
- Error Recovery in Predicting Parsing
Top Down Parsing

- constructing a parse tree from the input string
  - starting from the root
  - creating the nodes in preorder
- finding the left-most derivation for an input string
Grammar Example

- E -> TE'
- E' -> +TE'|ε
- T -> FT'
- T' -> *FT'|ε
- F -> (E) | id
Derivation Example for $id + id \ast id$
LL(k) Grammars

- **LL(k)** – class of grammar for which we can construct predictive parsers looking k symbols ahead
- **LL(1)**
- **FIRST and FOLLOW** sets
  - are used to construct predictive parsing tables
  - make explicit the choice of production
  - are useful for bottom-up parsing
Recursive Descendant Parsing Program

- set of procedures
- one procedure for each non-terminal
- the start symbol
  - launches the execution
  - announces success if the body scans its input string
Recursive Descendant Parsing

```c
void A()
{
    choose an A-production, A->X_1X_2...X_k;
    for(i=1 to k)
    {
        if (X_i is a non-terminal)
            call procedure X_i();
        else if (X_i equals the current symbol a)
            advance the input to the next symbol;
        else /* an error has occurred */
    }
}
```
Recursive Descendent Parsing

**Pseudocode**

- non-deterministic
  - the manner in which the A-production is chosen is not specified
- generally requires backtracking
  - repeated scans over the input
  - rarely needed to parse programming language constructs
  - not very efficient – tabular methods such as dynamic programming are preferred
void A()
{
    choose an A-production, A—>X₁X₂…Xₖ;
    for(i=1 to k)
    {
        if (Xᵢ is a non-terminal)
        call procedure Xᵢ();
        else if (Xᵢ equals the current symbol a)
        advance the input to the next
        else /* an error has occurred */
    }
}
Top-Down Parse Tree

- $S \rightarrow cA d$
- $A \rightarrow a b | a$
- $w=cad$
Step 1

- S has only one production
- we expand S

- first character of input $w=\text{cad}$ matches the leftmost leaf in the tree $c$
Step 2

- we expand A->a b
- we have a match for second input character a

![Diagram]

- we go to next symbol d
- b does not match d
- we report failure
- we go back to A to try another alternative
- we reset input pointer to position 2
Step 3

- the second alternative for A is $A \rightarrow a$

- leaf a matches second symbol
- leaf d matched the third symbol
- we halt with successful parsing message
FIRST and FOLLOW Functions

- Two functions useful in creating parsers for both
  - top-down
  - bottom-up
- Helps which production to apply based on next input symbol
- In panic mode error recovery tokens produced by FOLLOW are used for synchronization
The FIRST Function

- FIRST(α)
  - set of terminals that begin strings derived from α
  - α is any string of grammar symbols
  - if αǂ→ε then ε is in FIRST(α)
- Aǂ→cγ
  - c is in FIRST(A)
How FIRST function works?

- \( A \rightarrow \alpha | \beta \)
- \( \text{FIRST}(\alpha) \) and \( \text{FIRST}(\beta) \) are disjoint sets
- input symbol \( a \) can be in one of the two sets
- if \( a \) is in \( \text{FIRST}(\beta) \) we can choose the production \( A \rightarrow \beta \)
The FOLLOW Function

- FOLLOW(A)
  - the set of terminals $a$ that can appear immediately to the right of $A$ in some sentential form
  - the set of terminals $a$ such that $S => \alpha A a \beta$ for some $\alpha$ and $\beta$
How to compute FIRST?

- if \( X \) is terminal then \( \text{FIRST}(X) = \{X\} \)
- if \( X \) is non-terminal \( X \rightarrow Y_1Y_2 \ldots Y_k \) is a production for some \( k \geq 1 \)
  - place \( a \) in \( \text{FIRST}(X) \) if for some \( i \)
    - \( a \) is in \( \text{FIRST}(Y_i) \) and
    - \( \epsilon \) is in \( \text{FIRST}(Y_1) \ldots \text{FIRST}(Y_{i-1}) \)
  - if \( \epsilon \) is in all \( \text{FIRST}(Y_j) \) \( j = 1, \ldots, k \)
    - then add \( \epsilon \) to \( \text{FIRST}(X) \)
- if \( X \rightarrow \epsilon \) is a production
  - then add \( \epsilon \) to \( \text{FIRST}(X) \)
How to compute FIRST?

- input string $X_1 X_2 \ldots X_n$
- add to FIRST($X_1 X_2 \ldots X_n$)
  - all non-$\varepsilon$ symbols of FIRST($X_1$)
  - all non-$\varepsilon$ symbols of FIRST($X_2$) if $\varepsilon$ is in FIRST($X_1$)
  - all non-$\varepsilon$ symbols of FIRST($X_3$) if $\varepsilon$ is in FIRST($X_1$) and in FIRST($X_2$)
  - …
  - $\varepsilon$, if $\varepsilon$ is in all FIRST($X_i$) $i=1,\ldots,n$
How to compute FOLLOW?

- place $ in FOLLOW(S)
  - S is the start symbol
  - $ is the right end-marker
- if there is a production A -> $B\beta$
  - everything in FIRST($\beta$) except $\epsilon$ is in FOLLOW(B)
- if there is a production A -> $\alpha B$ or A -> $\alpha B\beta$ where first($\beta$) contains $\epsilon$
  - everything in FOLLOW(A) is in FOLLOW(B)
Example

- FIRST(F)={(:, id}
- FIRST(T)=FIRST(F)={(:, id}
- FIRST(E)=FIRST(T)=FIRST(F)={(:, id}
- FIRST(E')={+, ε}
- FIRST(T')={*, ε}
Example

- FOLLOW(E)={},$
  - E is the start symbol so it must include $
  - the body (E) tells that the ) symbol must be included

- FOLLOW(E')={},$
  - E->TE' so what follows after E will follow after E'
  - FOLLOW(E')=FOLLOW(E)
Example

- $\text{FOLLOW}(T)= (+, ), $)
  - $E \rightarrow TE' \text{ so } \text{FOLLOW}(T) \text{ includes } \text{FIRST}(E')=\{+\} \text{ (except } \varepsilon \text{)}$
  - $E \rightarrow TE' \text{ and } E' \text{ includes } \varepsilon, \text{ so } \text{FOLLOW}(E)=\{\}, $) \text{ is included in } \text{FOLLOW}(T)$

- $\text{FOLLOW}(T')= (+, ), $)
  - $T \rightarrow FT' \text{ so } \text{FOLLOW}(T') \text{ is included in } \text{FOLLOW}(T)$
Example

- FOLLOW(F)={+,*,),$}
  - T’->*FT’ so FOLLOW(F) includes
    FIRST(T’)={*} (except $)
  - T->FT’ and T’->ε so FOLLOW(T’) includes
    FOLLOW(T)={+,),$}
LL(1) Grammars

- predictive parsers
  - recursive descendant with no backtracking
- can be constructed for LL(1) grammar class
  - first L stands for scanning the input from left to right
  - second L for producing leftmost derivation
  - the 1 is for using one input symbol of lookahead at each step to make parsing actions decisions
Transition Diagrams for Predictive Parsers

- useful for visualizing predictive parsers
  - E -> TE'
  - E' -> + TE' | $\varepsilon$

![Transition Diagrams](image)
Building a Transition Diagram

- eliminate left recursion
- left factor the grammar
- for each non-terminal
  - create an initial and a final state
  - for each production $A \rightarrow X_1 X_2 \ldots X_k$
    - create a path from initial state to final state with edges labeled $X_1, X_2, \ldots, X_k$
    - if $A \rightarrow \epsilon$ the path is an edge labeled $\epsilon$

- label of edges can be tokens or non-terminals
- $\epsilon$-transitions are the default choice
LL(1) Grammar Definition

- rich enough to cover most programming constructs
- a grammar $G$ is LL(1) iff $A \rightarrow \alpha | \beta$
  - for no terminal $a$ do both $\alpha$ and $\beta$ derive strings beginning with $a$
  - at most one of $\alpha$ and $\beta$ can derive the empty string
  - if $\beta \Rightarrow^* \varepsilon$
    - $\alpha$ does not derive any string beginning with a terminal in FOLLOW($A$)
LL(1) Grammar Definition

- FIRST(\(\alpha\)) and FIRST(\(\beta\)) are disjoint sets
- If \(\varepsilon\) is in FIRST(\(\beta\)) then FIRST(\(\alpha\)) and FOLLOW(A) are disjoint sets
- vice versa if \(\varepsilon\) is in FIRST(\(\alpha\))
Example

- control flow constructs having distinguishable keywords generally satisfies the LL(1) constraints
- stmt -> if (expr) stmt else stmt
  | while(expr) stmt
  | {stmt_list}
- keywords like: if, while, { tells which alternative to take in order to succeed in finding a statement
The Construction of a Predictive Parsing Table

- to collect information from FIRST and FOLLOW
- to store them into a predictive parsing table $M[A,a]$ – two dimensional array
  - $A$ – non-terminal
  - $a$ – terminal or the $\$ \text{ end marker}$
- main idea
  - $A\rightarrow\alpha$ is chosen if the next input symbol $a$ is in $\text{FIRST}(\alpha)$
  - if $\alpha=\varepsilon$ or $A\rightarrow\alpha$ production $A\rightarrow\alpha$ is chosen when the current input symbol or $\$ \text{ is in } \text{FOLLOW}(A)$
The Construction Algorithm

- **Input**
  - Grammar G

- **Output**
  - Parsing table M

- **Method**
  - for each production A→α
    - for each terminal a in FIRST(A) add A→α to M[A,a]
    - if ε is in FIRST(α) then for each terminal b in FOLLOW(A) add A→α to M[A,b]
    - if ε is in FIRST(α) and $ is in FOLLOW(A) the add A→α to M[A,$]
  - after filling the table if there is no production in M[A,a] then set M[A,a] to error, represented by an empty entry in the table
Example

- $E \rightarrow TE'$
- $E' \rightarrow +TE'|\varepsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT'|\varepsilon$
- $F \rightarrow (E) | id$

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>E-&gt;TE'</td>
<td></td>
<td></td>
<td>E-&gt;TE'</td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>$E'$</td>
<td>E'-&gt;+TE'</td>
<td></td>
<td></td>
<td></td>
<td>E'-&gt;\varepsilon</td>
<td>E'-&gt;\varepsilon</td>
</tr>
<tr>
<td>$T$</td>
<td>T-&gt;FT'</td>
<td></td>
<td></td>
<td>T-&gt;FT'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T'$</td>
<td></td>
<td></td>
<td>T'-&gt;\varepsilon</td>
<td>T'-&gt;*FT'</td>
<td></td>
<td>T'-&gt;\varepsilon</td>
</tr>
<tr>
<td>$F$</td>
<td>F-&gt;id</td>
<td></td>
<td></td>
<td></td>
<td>F-&gt;(E)</td>
<td></td>
</tr>
</tbody>
</table>
Example

- **E'->TE'**
  - $\text{FIRST}(TE') = \text{FIRST}(T) = \{ (, id) \}$
  - added to $M[E, (]$ and $M[E, id]$

- **E'->+TE'**
  - $\text{FIRST}(+TE') = \{ + \}$
  - added to $M[E', +]$

- **E'->ε**
  - $\text{FOLLOW}(E') = \{ , \}, \$ \}$
  - added to $M[E', )]$ and $M[E', \$]$

- **FIRST(TE') = FIRST(T) = \{(, id)\}**
- **FIRST(+TE') = \{ + \}**
- **FOLLOW(E') = \{(, ), \$\}**
Example 2

- $S \rightarrow iEtSS' \mid a$
- $S' \rightarrow eS \mid \varepsilon$
- $E \rightarrow b$

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>e</th>
<th>i</th>
<th>t</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>S\rightarrow a</td>
<td></td>
<td></td>
<td></td>
<td>S\rightarrow iEtSS'</td>
<td></td>
</tr>
<tr>
<td>$S'$</td>
<td></td>
<td></td>
<td>$S'\rightarrow \varepsilon$</td>
<td>$S'\rightarrow eS$</td>
<td></td>
<td>$S'\rightarrow \varepsilon$</td>
</tr>
<tr>
<td>$E$</td>
<td></td>
<td>E\rightarrow b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Non-recursive Predictive Parsing

- to maintain a stack explicitly
- rather than implicitly by recursive calls
- the parser simulates the leftmost derivation
- if w is the input matched so far
  - then the stack holds a sequence of grammar symbols α such that $S_{lm}^* \Rightarrow w\alpha$
Model of a Table Driven Predictive Parser
Model of a Table Driven Predictive Parser

- **input buffer**
  - string to be parsed
  - end marker $$

- **stack containing grammar symbols**
  - it’s bottom is marked by $$

- **parsing table**

- **output stream**
Model of a Table Driven Predictive Parser

- \( X \) is the symbol on top of the stack
- \( a \) is the current input symbol

If \( X \) is non-terminal
- the parser chooses a production by consulting the entry \( M[X,a] \)
- semantic actions can be added to build a node in the parse tree

If \( X \) is a terminal
- a match is checked between \( X \) and input symbol \( a \)
Model of a Table Driven Predictive Parser

- parser configurations
  - stack content
  - remaining input
Table Driven Predictive Parsing

• Input
  ◦ a string w
  ◦ parsing table M for a grammar G

• Output
  ◦ if w is in L(G) then
    • a leftmost derivation of w
    • otherwise error indication

• Method
  ◦ initially the parser has
    • w$ in the input buffer
    • start symbol S of G on the stack top, above $
Predictive Parsing Algorithm

set ip to point the first symbol $a$ of $w$
set $X$ to the top stack symbol
while($X!=$$)$
{
    if ($X$ is $a$) then pop the stack and advance ip
    elseif ($X$ is a terminal) error();
    elseif ($M[X,a]$ is an error entry) error();
    elseif ($M[X,a]=X\rightarrow Y_1,Y_2,...,Y_k$)
    {
        output the production $X\rightarrow Y_1,Y_2,...,Y_k$
        pop the stack
        push $Y_k,Y_{k-1},...,Y_1$ onto the stack with $Y_1$ on top
    }
    set $X$ to the top stack symbol
}
Example

- $E \rightarrow TE'$
- $E' \rightarrow +TE'|\varepsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT'|\varepsilon$
- $F \rightarrow (E) \mid id$

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$E \rightarrow TE'$</td>
<td></td>
<td></td>
<td>$E \rightarrow TE'$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E'$</td>
<td></td>
<td>$E' \rightarrow +TE'$</td>
<td></td>
<td></td>
<td>$E' \rightarrow \varepsilon$</td>
<td>$E' \rightarrow \varepsilon$</td>
</tr>
<tr>
<td>$T$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$T \rightarrow FT'$</td>
<td></td>
</tr>
<tr>
<td>$T'$</td>
<td></td>
<td>$T' \rightarrow \varepsilon$</td>
<td>$T' \rightarrow *FT'$</td>
<td>$T' \rightarrow \varepsilon$</td>
<td></td>
<td>$T' \rightarrow \varepsilon$</td>
</tr>
<tr>
<td>$F$</td>
<td>$F \rightarrow id$</td>
<td></td>
<td></td>
<td>$F \rightarrow (E)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Moves Made by a Predictive Parser on id+id*id

<table>
<thead>
<tr>
<th>Matched</th>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>id + id * id$</td>
<td></td>
<td>output $E \rightarrow TE'$</td>
</tr>
<tr>
<td>$TE'$</td>
<td>id + id * id$</td>
<td></td>
<td>output $T \rightarrow FT'$</td>
</tr>
<tr>
<td>$FT'E'$</td>
<td>id + id * id$</td>
<td></td>
<td>output $F \rightarrow id$</td>
</tr>
<tr>
<td>id</td>
<td>$T'E'$</td>
<td>+ id * id$</td>
<td>match id</td>
</tr>
<tr>
<td>id</td>
<td>$E'$</td>
<td>+ id * id$</td>
<td>output $T' \rightarrow \epsilon$</td>
</tr>
<tr>
<td>id</td>
<td>+ $TE'$</td>
<td>+ id * id$</td>
<td>output $E' \rightarrow + TE'$</td>
</tr>
<tr>
<td>id +</td>
<td>$TE'$</td>
<td>id * id$</td>
<td>match +</td>
</tr>
<tr>
<td>id +</td>
<td>$FT'E'$</td>
<td>id * id$</td>
<td>output $T \rightarrow FT'$</td>
</tr>
<tr>
<td>id +</td>
<td>id $T'E'$</td>
<td>id * id$</td>
<td>output $F \rightarrow id$</td>
</tr>
<tr>
<td>id + id</td>
<td>$T'E'$</td>
<td>* id$</td>
<td>match id</td>
</tr>
<tr>
<td>id + id</td>
<td>* $FT'E'$</td>
<td>* id$</td>
<td>output $T' \rightarrow * FT'$</td>
</tr>
<tr>
<td>id + id *</td>
<td>$FT'E'$</td>
<td>id$</td>
<td>match *</td>
</tr>
<tr>
<td>id + id *</td>
<td>id $T'E'$</td>
<td>id$</td>
<td>output $F \rightarrow id$</td>
</tr>
<tr>
<td>id + id * id</td>
<td>$T'E'$</td>
<td>$</td>
<td>match id</td>
</tr>
<tr>
<td>id + id * id</td>
<td>$E'$</td>
<td>$</td>
<td>output $T' \rightarrow \epsilon$</td>
</tr>
<tr>
<td>id + id * id</td>
<td>$</td>
<td>$</td>
<td>output $E' \rightarrow \epsilon$</td>
</tr>
</tbody>
</table>
Error Recovery in Predicting Parsing

- Error recovery refers to the stack of the table driven predictive parser
- It makes explicit the terminals and non-terminals the parser hopes to match
- The techniques can be used with recursive-descendant parsing
- An error is detected when:
  - Stack top terminal does not match the next input symbol
  - $M[A,a]$ is error (empty)
    - $A$ is on the non-terminal on the top of the stack
    - $a$ is the next input symbol
Panic Mode

- skipping input symbols until a set of synchronizing symbols appear
- effectiveness depend on the set of the chosen set
- the sets should be chosen so the parser recovers quickly from errors that are likely to occur in practice
Some Heuristics

- all symbols in FOLLOW(A) will be added to the synchronizing set for A non-terminal
  - skip tokens until an element of FOLLOW(A) is seen
  - pop A from the stack
  - is likely the parsing can continue
Some Heuristics

- only FOLLOW(A) set is not enough
- semicolons terminate statements in C
- keywords that begin statements may not appear in the FOLLOW set for expression non-terminal
- a missing semicolon after an assignment may result in the keyword beginning next statement to be skipped
- expressions appear within statements
- we need to add to the synchronizing symbols of lower level constructs the synchronizing symbols of higher level constructs
- we can add symbols that begin statements to the synchronizing sets for the non-terminals generating expressions
Some Heuristics

- all symbols from FIRST(A) will be added to the synchronizing set for A non-terminal
  - it is possible to resume parsing according to A
  - if a symbol from FIRST(A) appears
- if a non-terminal can generate the empty string
  - then the production deriving in ε can be used by default
  - we may postpone some error detection
  - cannot cause an error to be missed
  - reduces the number of non-terminals to be considered during error recovery
Some Heuristics

- if a terminal on the top cannot be matched
  - pop the terminal
  - issue a message
  - continue parsing
  - the synchronization set of a token consists in all other tokens
Phrase Level Recovery

- filling in the blank cells pointers to error routines
  - change, insert, delete symbols
  - pop from the stack
- stack alteration is questionable
  - modifying the stack might not enable derivation at all
  - risk of infinite loop
  - to check the stack size after modifying it
    - it should decrease
Bibliography