Plan for Today

Context Free Grammars
- Create derivation and parse tree for some examples.
- Use syntax-directed translation of parse tree to evaluate examples.

Top-down Predictive Parsing

Logistics
- Office hours Friday 3-4.
- PA2a due Wednesday Feb 18th. Partners posted on Canvas just now due to button pressing issue.

Derivation, Parse Tree, and Interpretation for Example

Grammar
Stm --> id := Exp
Exp --> num
Exp --> ( Stm, Exp )

Parse Tree

String
x := ( y := 42, 7 )

Recall Semantic Rules from Monday’s Example

Grammar
(1) exp --> exp * exp
(2) exp --> exp + exp
(3) exp --> NUM

String
3 + 4 + 5
**Example:**

```
Parser
S → SS  
S → aSb  
S → bSa  
S → ε
```

**input**

```
aabb
```

**derivation**

```
aabb
```

**Exhaustive Search**

```
S → SS | aSb | bSa | ε
```

**Phase 1:**

- $S \Rightarrow SS$
- $S \Rightarrow aSb$
- $S \Rightarrow bSa$
- $S \Rightarrow ε$

*All possible derivations of length 1*
### Phase 2

\[ S \rightarrow SS | aSb | bSa | \epsilon \]

- \[ S \Rightarrow SS \Rightarrow SSS \]
- \[ S \Rightarrow SS \Rightarrow aSbS \]
- \[ S \Rightarrow SS \Rightarrow bSaS \]
- \[ S \Rightarrow SS \Rightarrow \]

### Phase 1

- \[ S \Rightarrow SS \Rightarrow S \]
- \[ S \Rightarrow aSb \Rightarrow aSbS \]
- \[ S \Rightarrow aSb \Rightarrow aSSb \]
- \[ S \Rightarrow aSb \Rightarrow aaSbb \]
- \[ S \Rightarrow aSb \Rightarrow abSab \]
- \[ S \Rightarrow aSb \Rightarrow ab \]
- \[ aabb \]

### Final result of exhaustive search

(\textit{top-down parsing})

- \[ S \Rightarrow SS \]
- \[ S \Rightarrow aSb \]
- \[ S \Rightarrow bSa \]
- \[ S \Rightarrow \epsilon \]

### For general context-free grammars:

The exhaustive search approach is extremely costly: \( O(|P||w|) \)

There exists a parsing algorithm that parses a string \( w \) in time \( |w|^3 \) for any CFG (Earley parser)

For LL(1) grammars, a simple type of CFGs that we will meet soon, we can use Predictive parsing and parse in \( |w| \) time.
**Predictive Parsing**

Predictive parsing, such as recursive descent parsing, creates the parse tree TOP DOWN, starting at the start symbol, and doing a LEFT-MOST derivation.

For each non-terminal \( N \) there is a function recognizing the strings that can be produced by \( N \), with one (case) clause for each production. Consider:

\[
\text{can each production clause be uniquely identified by looking ahead one token? Let’s predictively build the parse tree for:}
\]

\[
\text{if } t \{ \text{ while } b \{ x = 6 \} } \$
\]

**Recursive Descent Parsing**

Each non-terminal becomes a function that mimics the RHSs of the productions associated with it and chooses a particular RHS:
- an alternative based on a look-ahead symbol
- and throws an exception if no alternative applies

When does this work?

**Example Predictive Parser: Recursive Descent**

```java
void start() { switch(m_lookahead) {
    case IF, WHILE, EOF:
        stmts(); match(Token.Tag.EOF); break;
    default:    throw new ParseException(…);
}}
void stmts() { switch(m_lookahead) {
    case IF,WHILE:
        stmt(); stmts(); break;
    case EOF:       break;
    default:        throw new ParseException(…);
}}
void stmt() { switch(m_lookahead) {
    case IF:
        ifStmt();break;
    case WHILE:
        whileStmt(); break;
    default:    throw new ParseException(…);
}}
void ifStmt() {switch(m_lookahead) {
    case IF: match(id); match(OPENBRACE);
        stmts(); match(CLOSEBRACE); break;
    default: throw new ParseException(…);
}}
void whileStmt() {switch(m_lookahead) {
    case IF: ifStmt();break;
    case WHILE: whileStmt(); break;
    default:    throw new ParseException(…);
}}
```

**First**

Given a phrase \( \gamma \) of terminals and non-terminals (a RHS of a production), \( \text{FIRST}(\gamma) \) is the set of all terminals that can begin a string derived from \( \gamma \).

\[
\text{FIRST(T*F)} = ? \text{ FIRST(F)} = ? \text{ FIRST(XYZ) = FIRST(X) } ?
\]

\textbf{NO!} X could produce \( \epsilon \) and then \( \text{FIRST(Y)} \) comes into play

we must keep track of which non terminals are \textbf{NULLABLE}
### FIRST example

<table>
<thead>
<tr>
<th>Rule</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>stmts EOF</td>
</tr>
<tr>
<td>stmts</td>
<td>ε</td>
</tr>
<tr>
<td>stmt</td>
<td>ifStmt</td>
</tr>
<tr>
<td>ifStmt</td>
<td>IF id { stmts }</td>
</tr>
<tr>
<td>whileStmt</td>
<td>WHILE id { stmts }</td>
</tr>
</tbody>
</table>

### Follow

It also turns out to be useful to determine which terminals can directly follow a non terminal X (to decide parsing X is finished).

terminal t is in FOLLOW(X) if there is any derivation containing Xt.

This can occur if the derivation contains XYZt and Y and Z are nullable

### Constructive Definition of nullable, first and follow

for each terminal t, FIRST(t) = \{t\}

Another Transitive Closure algorithm:
keep doing STEP until nothing changes

**STEP:**
for each production X → Y₁ Y₂ ... Yₖ
0: if Yᵢ to Yₖ nullable (or k = 0) nullable(X) = true
   for each i from 1 to k, each j from i+1 to k
1: if Yᵢ₁...Yᵢₙ nullable (or i=1) FIRST(Yᵢ) += FIRST(Yᵢₙ) //+: union
2: if Yᵢ₊₁...Yₖ nullable (or i=k) FOLLOW(Yᵢ) += FOLLOW(X)
3: if Yᵢ₊₁...Yₖ₊₁ nullable (or i+1=j) FOLLOW(Yᵢ) += FIRST(Yⱼ)

We can compute nullable, then FIRST, and then FOLLOW

### FIRST and FOLLOW sets

**NULLABLE**
- X is a nonterminal
- nullable(X) is true if X can derive the empty string

**FIRST**
- FIRST(z) = \{z\}, where z is a terminal
- FIRST(X) = union of all FIRST(rhsᵢ), where X is a nonterminal and X → rhsᵢ is a production
- FIRST(rhsᵢ) = union all of FIRST(sym) on rhs up to and including first nonnullable

**FOLLOW(Y), only relevant when Y is a nonterminal**
- look for Y in rhs of rules (lhs → rhs) and union all FIRST sets for symbols after Y up to and including first nonnullable
- if all symbols after Y are nullable then also union in FOLLOW(lhs)
Class Exercise

Compute nullable, FIRST and FOLLOW for

\[ Z \rightarrow d \mid XYZ \]
\[ X \rightarrow a \mid Y \]
\[ Y \rightarrow c \mid \varepsilon \]

Constructing the Predictive Parser Table

A predictive parse table has a row for each non-terminal X, and a column for each input token t. Entries table[X,t] contain productions:

- for each X -> gamma
- for each t in FIRST(gamma)
  
  \[ \text{table}[X,t] = X \rightarrow \gamma \]
- if gamma is nullable
- for each t in FOLLOW(X)
  
  \[ \text{table}[X,t] = X \rightarrow \gamma \]

Compute the predictive parse table for

\[ Z \rightarrow d \mid XYZ \]
\[ X \rightarrow a \mid Y \]
\[ Y \rightarrow c \mid \varepsilon \]

<table>
<thead>
<tr>
<th>(d)</th>
<th>(a)</th>
<th>(c)</th>
<th>(d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X(\rightarrow a)</td>
<td>X(\rightarrow Y)</td>
<td>X(\rightarrow Y)</td>
</tr>
<tr>
<td>Y</td>
<td>Y(\rightarrow \varepsilon)</td>
<td>Y(\rightarrow \varepsilon)</td>
<td>Y(\rightarrow \varepsilon)</td>
</tr>
<tr>
<td>Z</td>
<td>Z(\rightarrow XYZ)</td>
<td>Z(\rightarrow XYZ)</td>
<td>Z(\rightarrow XYZ)</td>
</tr>
</tbody>
</table>

Multiple entries in the Predictive parse table: Ambiguity

An ambiguous grammar will lead to multiple entries in the parse table.

Our grammar is ambiguous, e.g.

\[ Z \rightarrow d \]

but also

\[ Z \rightarrow XYZ \rightarrow YZ \rightarrow d \]

For grammars with no multiple entries in the table, we can use the table to produce one parse tree for each valid sentence. We call these grammars LL(1): Left to right parse, Left-most derivation, 1 symbol lookahead.

A recursive descent parser examines input left to right. The order it expands non-terminals is leftmost first, and it looks ahead 1 token.

One more time

Balanced parentheses grammar 1:

\[ S \rightarrow ( S ) \mid SS \mid \varepsilon \]

1. Augment the grammar with EOF $S$

2. Construct Nullable, First and Follow

3. Build the predictive parse table, what happens?
One more time, but this time with feeling …

Balanced parentheses grammar 2:

\[ S \rightarrow ( S )S | \varepsilon \]

1. Augment the grammar with EOF/$

2. Construct Nullable, First and Follow

3. Build the predictive parse table

4. Using the predictive parse table, construct the parse tree for

   \( ( ) ( ) $ \)

   and

   \( () () $ \)