50.054 Top Down Parsing

ISTD, SUTD

Learning Outcomes

1. Apply left-recursion elimination and left-factoring
2. Construct a `LL(1)` predictive parsing table
3. Explain first-first conflicts and first-follow conflicts

Recap

$$
\begin{array}{rcll}
(\texttt{JSON}) & J & ::= & i \mid {'s'} \mid [] \mid [IS] \mid \{NS\} \\
(\texttt{Items}) & IS & ::= & J, IS \mid J \\
(\texttt{Named Objects}) & NS & ::= & N, NS \mid N \\
(\texttt{Named Object}) & N & ::= & {'s'} : J
\end{array}
$$

Recap

```
enum Json {
    case IntLit(v:Int)
    case StrLit(v:String)
    case JsonList(vs:List[Json])
    case JsonObject(flds:Map[String,Json])
}
val input = List(LBRace,SQuote,StrTok("k1"),SQuote
    ,Colon,IntTok(1),Comma,SQuote
    ,StrTok("k2"),Colon,LBracket, RBracket,RBrace)
val expected = Some(JsonObject(
    Map(
        "k1" -> IntLit(1),
        "k2" -> JsonList(Nil)
    )
))
```

Recap

```
def parse(toks:List[LToken]):Option[Json] = toks match {
    case Nil => // Done? what to return?
    case (t:ts) if t is digit => {
        val i = parse_an_int(toks); Some(IntLit(i)) }
    case (t:ts) if t is '\'' => {
        val s = parse_a_str(toks); Some(StrLit(s)) }
    case (t:ts) if t is '[' => {
        val l = parse_a_list(toks); Some(JsonList(l)) }
    case (t:ts) => {
        val m = parse_a_map(toks); Some(JsonObject(m)) }
} // Can we always decide which path to go by checking t?
```

Breaking Down the Grammar

<<Grammar 1>>

$$\begin{array}{rrcl}
(\texttt{JSON}) & J & ::= & i \mid {'s'} \mid [\,] \mid [IS] \mid \{NS\} \\
(\texttt{Items}) & IS & ::= & J, IS \mid J \\
(\texttt{Named Objects}) & NS & ::= & N, NS \mid N \\
(\texttt{Named Object}) & N & ::= & {'s'} : J
\end{array}$$

```
(1) J ::= i
(2) J ::= 's'
(3) J ::= []
(4) J ::= [IS]
(5) J ::= {NS}
(6) IS ::= J,IS
(7) IS ::= J
(8) NS ::= N,NS
(9) NS ::= N
(10) N ::= 's':J
```

Top Down Parsing - naive algorithm

Besides input tokens, `toks`, we need

- the current grammar rule being considered, `N ::= RHS`.
- remaining to-be-parsed symbols from the `RHS`, say `symbols`
- the (partially) constructed parse tree.

Base case: `symbols` is `Nil`

1. the parse tree for the current rule `N ::= RHS` must have been constructed and we just return it.

Top Down Parsing - naive algorithm

Recursive case: `symbols` is `symbol::symbols1`

1. the leading symbol `symbol` is a terminal
   1.1 if `toks` is `tok::toks1` and `tok` matches with `symbol`, construct the leaf of the parse tree. Move on to the next token/symbol, i.e. `toks1` and `symbols1`.
   1.2 otherwise signals a failure
2. the leading symbol `symbol` is a non-terminal `M`
   2.1 if `toks` is `Nil` and a rule `M::= RHS2`
      2.1.1 if `RHS2` accepts empty tokens, construct the empty parse tree leaf w.r.t `M`. Move on to the next symbol, i.e. parsing `Nil` with `symbols`.
      2.1.2 if `RHS2` does not accept empty tokens, signals a failure.
   2.2 if the input token list is `tok::toks`, **pick an alternative `M::= RHS'`**, apply recursion with the rule `M::= RHS'` and `tok::toks`. Keep trying until one alternative succeeds in parsing `tok::toks`.
   2.3 otherwise signal a failure.

TopDown Parsing - Example

```
(1) J ::= i
(2) J ::= 's'
(3) J ::= []
(4) J ::= [IS]
(5) J ::= {NS}
(6) IS ::= J,IS
(7) IS ::= J
(8) NS ::= N,NS
(9) NS ::= N
(10) N ::= 's':J
```

- input: `{ ' k1 ' : 1 , ' k2 ' : [ ] }`
- rule ID: 5
- symbols: `{ NS }`
- parse tree:

```
   J
 / | \
{  NS }
```

- input: `' k1 ' : 1 , ' k2 ' : [ ] }`
- rule ID: 5
- symbols: `NS }`
- parse tree:

```
   J
 / | \
{  NS }
```

- recursive call
- input: `' k1 ' : 1 , ' k2 ' : [ ] }`
- rule ID: 8
- symbols: `N , NS`
- parse tree:

```
   J
 / | \
{  NS }
   /|\
  N , NS
```

TopDown Parsing - Example

```
(1) J ::= i
(2) J ::= 's'
(3) J ::= []
(4) J ::= [IS]
(5) J ::= {NS}
(6) IS ::= J,IS
(7) IS ::= J
(8) NS ::= N,NS
(9) NS ::= N
(10) N ::= 's':J
```

TopDown Parsing - Example

```
(1) J ::= i
(2) J ::= 's'
(3) J ::= []
(4) J ::= [IS]
(5) J ::= {NS}
(6) IS ::= J,IS
(7) IS ::= J
(8) NS ::= N,NS
(9) NS ::= N
(10) N ::= 's':J
```

TopDown Parsing - Example

```
(1) J ::= i
(2) J ::= 's'
(3) J ::= []
(4) J ::= [IS]
(5) J ::= {NS}
(6) IS ::= J,IS
(7) IS ::= J
(8) NS ::= N,NS
(9) NS ::= N
(10) N ::= 's':J
```

TopDown Parsing - Example

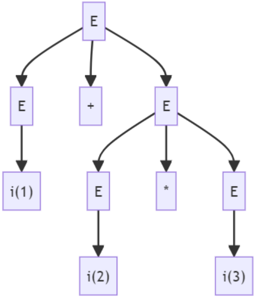
```
(1) J ::= i
(2) J ::= 's'
(3) J ::= []
(4) J ::= [IS]
(5) J ::= {NS}
(6) IS ::= J,IS
(7) IS ::= J
(8) NS ::= N,NS
(9) NS ::= N
(10) N ::= 's':J
```

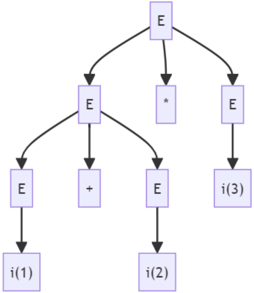
Top Down Parsing Issue 1 - Ambiguous Grammar

<<Grammar 3>>

$$
\begin{aligned}
E &::= E + E \\
E &::= E * E \\
E &::= i
\end{aligned}
$$

input $=$ 1 + 2 * 3





Top Down Parsing Issue 1 - Ambiguous Grammar

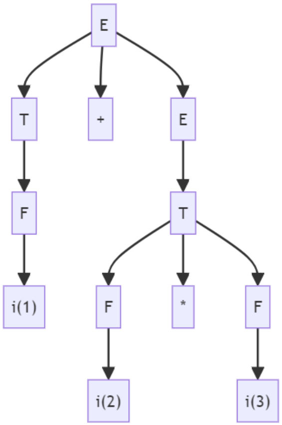
- Parsing with an ambiguous grammar leads to non-determinsm, or some greedy approach must be adopted, e.g. favor the first successful parse.
- No general ambiguity detection algorithm exists.
- Language designers need to check and rewrite the grammar if it's ambiguous.

`<<Grammar 3>>`

$$
\begin{array}{lcl}
E & ::= & E + E \\
E & ::= & E * E \\
E & ::= & i
\end{array}
$$

into `<<Grammar 4>>`

$$
\begin{array}{lcl}
E & ::= & T + E \\
E & ::= & T \\
T & ::= & T * F \\
T & ::= & F \\
F & ::= & i
\end{array}
$$



Top Down Parsing Issue 2 - Left Recursive Grammar

- When the grammar is left recursive, the algorithm will not terminate (or stack over flow). `<<Grammar 5>>`

$$\begin{array}{lcl} E & ::= & E + T \\ E & ::= & T \\ T & ::= & i \end{array}$$

e.g. the 1st rule is always picked.

Top Down Parsing Issue 2 - Left Recursive Grammar

- Idea
  1. Convert the left recursive grammar G into a non-left-recursive G'.
  2. Parse using G' with the input, to obtain parse tree D'
  3. Convert D' of grammar (type) G' to D of grammar (type) G.

Top Down Parsing Issue 2 - Left Recursive Grammar

Let `N` be a (left-recursive) non-terminal, α_i and β_j be sequences of symbols (consist of terminals and non-terminals)

Left recursive grammar rules

$$
\begin{array}{rcl}
N & ::= & N\alpha_1 \\
 & \dots & \\
N & ::= & N\alpha_n \\
N & ::= & \beta_1 \\
 & \dots & \\
N & ::= & \beta_m
\end{array}
$$

can be transformed into

$$
\begin{array}{rcl}
N & ::= & \beta_1 N' \\
 & \dots & \\
N & ::= & \beta_m N' \\
N' & ::= & \alpha_1 N' \\
 & \dots & \\
N' & ::= & \alpha_n N' \\
N' & ::= & \epsilon
\end{array}
$$

Top Down Parsing Issue 2 - Left Recursive Grammar

Let N be a (left-recursive) non-terminal, α_i and β_j be sequences of symbols (consist of terminals and non-terminals)

<<<Grammar 5>>>

$$
\begin{array}{rcl}
E & ::= & E + T \\
E & ::= & T \\
T & ::= & i
\end{array}
$$

- N is E,
- α_1 is $+T$,
- β_1 is T

can be transformed into <<<Grammar 6>>>

$$
\begin{array}{rcl}
E & ::= & TE' \\
E' & ::= & +TE' \\
E' & ::= & \epsilon \\
T & ::= & i
\end{array}
$$

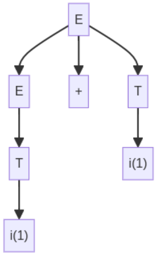
Top Down Parsing Issue 2 - Left Recursive Grammar

Let `N` be a (left-recursive) non-terminal, α_i and β_j be sequences of symbols (consist of terminals and non-terminals)

`<<<Grammar 5>>>`

$$\begin{aligned} E &::= E + T \\ E &::= T \\ T &::= i \end{aligned}$$

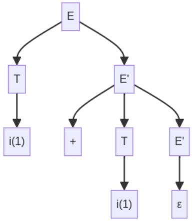
convert from parse tree on the right



can be transformed into `<<<Grammar 6>>>`

$$\begin{aligned} E &::= TE' \\ E' &::= +TE' \\ E' &::= \epsilon \\ T &::= i \end{aligned}$$

parse using `<<Grammar 6>>`



Top Down Parsing Issue 2 - Left Recursive Grammar

Let N be a (left-recursive) non-terminal, α_i and β_j be sequences of symbols (consist of terminals and non-terminals)

<<<Grammar 5>>>

$$
\begin{array}{rcl}
E & ::= & E + T \\
E & ::= & T \\
T & ::= & i
\end{array}
$$

```
enum E {
    case Plus(e:E, t:T)
    case Term(t:T)
}
case class T(v:Int)
// converted from u
val v = E.Plus(E.Term(T(1)),
        E.Term(T(1)))
```

can be transformed into <<<Grammar 6>>>

$$
\begin{array}{rcl}
E & ::= & TE' \\
E' & ::= & +TE' \\
E' & ::= & \epsilon \\
T & ::= & i
\end{array}
$$

```
case class E1(t:T, ep:EP)
enum EP {
    case Plus(t:T, ep:EP)
    case Eps
}
// parse using grammar 6
val u = E1(T(1), EP.Plus(T(1), EP.Eps)
```

Recall Top Down Parsing - naive algorithm

- Base case: `symbols` is `Nil`
  1. the parse tree for the current rule `N ::= RHS` must have been constructed and we just return it.
- Recursive case: `symbols` is `symbol::symbols'`

1. if the leading `symbol` is a terminal
   1.1 if the input token list is `tok::toks` and `tok` matches with `symbol`, construct the leaf of the parse tree. Move on to the next token/symbol, i.e. `toks` and `symbols'`.
   1.2 otherwise signals a failure
2. if the leading `symbol` is a non-terminal `M`
   2.1 if the input token list is `Nil` and a rule `M::= RHS2` exists in the grammar
      2.1.1 if `RHS2` accepts empty tokens, construct the empty parse tree leaf w.r.t `M`. Move on to the next symbol, i.e. parsing `Nil` with `symbols`.
      2.1.2 if `RHS2` does not accept empty tokens, signals a failure.
   2.2 if the input token list is `tok::toks`, **pick an alternative `M::= RHS'`**, apply recursion with the rule `M::= RHS'` and `tok::toks`. Keep trying until one alternative succeeds in parsing `tok::toks`.
   2.3 otherwise signal a failure.

Can we find such an alternative `M::= RHS'` based on the leading token?

Predictive Top Down Parsing

Objective: pick the “right” alternative production rule without trial-and-error.

- Undecideable in general.
- But we can restrict to a sub class of grammar that always work.
- `LL(k)` grammar
  - `k` refers to the number of leading symbols from the input we need to check
- We consider `LL(1)` grammar

Checking for $\texttt{LL(1)}$

- G denotes a grammar to be verified.
- $\overline{\sigma}$ denotes a sequence of symbols
- $null(\overline{\sigma}, G)$ checks whether the language denoted by $\overline{\sigma}$ contains the empty sequence.

$$
\begin{array}{rcl}
null(t, G) & = & false \\
null(\epsilon, G) & = & true \\
null(N, G) & = & \bigvee_{N::=\overline{\sigma} \in G} null(\overline{\sigma}, G) \\
null(\sigma_1 ... \sigma_n, G) & = & null(\sigma_1, G) \wedge ... \wedge null(\sigma_n, G)
\end{array}
$$

Recall `<<Grammar 6>>`

$$
\begin{array}{rcl}
E & ::= & TE' \\
E' & ::= & +TE' \\
E' & ::= & \epsilon \\
T & ::= & i
\end{array}
$$

$$
\begin{array}{l}
null(E) = null(TE') = null(T) \wedge null(E') = false \wedge null(E') = false \\
null(E') = null(+TE') \vee null(\epsilon) = null(+TE') \vee true = true \\
null(T) = null(i) = false
\end{array}
$$

Checking for $\texttt{LL(1)}$

- $\mathit{first}(\overline{\sigma}, G)$ computes the set of leading terminals from the language denoted by $\overline{\sigma}$.

$$
\begin{array}{rcl}
\mathit{first}(\epsilon, G) & = & \{\} \\
\mathit{first}(t, G) & = & \{t\} \\
\mathit{first}(N, G) & = & \bigcup_{N ::= \overline{\sigma} \in G} \mathit{first}(\overline{\sigma}, G) \\
\mathit{first}(\sigma\overline{\sigma}, G) & = & \begin{cases} \mathit{first}(\sigma, G) \cup \mathit{first}(\overline{\sigma}, G) & \texttt{if } \mathit{null}(\sigma, G) \\ \mathit{first}(\sigma, G) & \texttt{otherwise} \end{cases}
\end{array}
$$

Recall `<<Grammar 6>>`

$$
\begin{array}{rcl}
E & ::= & TE' \\
E' & ::= & +TE' \\
E' & ::= & \epsilon \\
T & ::= & i
\end{array}
$$

$$
\begin{array}{l}
\mathit{first}(E) = \mathit{first}(TE') = \mathit{first}(T) = \{i\} \\
\mathit{first}(E') = \mathit{first}(+TE') \cup \mathit{first}(\epsilon) = \mathit{first}(+TE') = \{+\} \\
\mathit{first}(T) = \{i\}
\end{array}
$$

Checking for $\texttt{LL(1)}$

- $\mathit{follow}(\sigma, G)$ finds the set of terminals that immediately follows symbol σ in any derivation derivable from G.

$$\mathit{follow}(\sigma, G) \quad = \quad \bigcup_{N::=\overline{\sigma}\sigma\overline{\gamma}\in G} \left[\begin{array}{ll} \mathit{first}(\overline{\gamma}, G) \cup \mathit{follow}(N, G) & \texttt{if } \mathit{null}(\overline{\gamma}, G) \\ \mathit{first}(\overline{\gamma}, G) & \texttt{otherwise} \end{array} \right.$$

Recall `<<Grammar 6>>`

$$\begin{array}{rcl} E & ::= & TE' \\ E' & ::= & +TE' \\ E' & ::= & \epsilon \\ T & ::= & i \end{array}$$

$$\begin{array}{l} \mathit{follow}(E) = \{\} \\ \mathit{follow}(E') = \mathit{follow}(E) \cup \mathit{follow}(E') = \{\} \cup \mathit{follow}(E') \\ \mathit{follow}(T) = \mathit{first}(E') \cup \mathit{follow}(E') = \{+\} \cup \mathit{follow}(E') \end{array}$$

Checking for LL(1)

- Construct a predictive parsing table
- each row is indexed a non-terminal, and each column is indexed by a terminal.

	i	+
E		
E'		
T		

Checking for `LL(1)`

For each production rule $N ::= \overline{\sigma}$, we put the production rule in

- cell (N, t) if $t \in \mathit{first}(\overline{\sigma})$
- cell (N, t') if $\mathit{null}(\overline{\sigma})$ and $t' \in \mathit{follow}(N)$

Recall `<<Grammar 6>>`

$$
\begin{array}{rcl}
E & ::= & TE' \\
E' & ::= & +TE' \\
E' & ::= & \epsilon \\
T & ::= & i
\end{array}
$$

$$
\begin{array}{lll}
\mathit{null}(E) = \mathit{false} & \mathit{first}(E) = \{i\} & \mathit{follow}(E) = \{\} \\
\mathit{null}(E') = \mathit{true} & \mathit{first}(E') = \{+\} & \mathit{follow}(E') = \{\} \\
\mathit{null}(T) = \mathit{false} & \mathit{first}(T) = \{i\} & \mathit{follow}(T) = \{+\}
\end{array}
$$

	i	+
E	E ::= TE'	
E'		E' ::= + TE'
T	T ::= i	

- `<<Grammar 6>>` is `LL(1)`

Another Example

`<<Grammar 9>>`

$$
\begin{array}{lcl}
S & ::= & Xb \\
S & ::= & Yc \\
X & ::= & a \\
Y & ::= & a
\end{array}
$$

$$
\begin{array}{lll}
null(S) = null(Xb) = false & first(S) = first(Xb) \cup first(Yc) = \{a\} & follow(S) = \{\} \\
null(X) = null(a) = false & first(X) = first(a) = \{a\} & follow(X) = \{b\} \\
null(Y) = null(a) = false & first(Y) = first(a) = \{a\} & follow(Y) = \{c\}
\end{array}
$$

	a	b	c
S	S::=Xb, S::=Yc		
X	X::=a		
Y	Y::=a		

- It's not in LL(1),
- It has a first-first conflict.

Left-factoring

`<<Grammar 9>>`

$$
\begin{array}{lcl}
S & ::= & Xb \\
S & ::= & Yc \\
X & ::= & a \\
Y & ::= & a
\end{array}
$$

Substitute a/X and a/Y to the two rules of S

`<<Grammar 10>>`

$$
\begin{array}{lcl}
S & ::= & ab \\
S & ::= & ac
\end{array}
$$

Merge the two production alternatives of S by introducing a non terminal Z

`<<Grammar 11>>`

$$
\begin{array}{lcl}
S & ::= & aZ \\
Z & ::= & b \\
Z & ::= & c
\end{array}
$$

- `Grammar 11` is in `LL(1)`

One more example

<<Grammar 12>>

$$
\begin{array}{lcl}
S & ::= & Xd \\
X & ::= & C \\
X & ::= & Ba \\
C & ::= & \epsilon \\
B & ::= & d
\end{array}
$$

$$
\begin{array}{lll}
null(S) = false & first(S) = first(Xd) = first(X) \cup first(d) = \{d\} & follow(S) = \{\} \\
null(X) = true & first(X) = first(C) \cup first(Ba) = \{d\} & follow(X) = \{d\} \\
null(C) = true & first(C) = \{\} & follow(C) = follow(X) = \{d\} \\
null(B) = false & first(B) = \{d\} & follow(B) = \{a\}
\end{array}
$$

	a	d
S		S::=Xd
X		*X::=Ba*, **X::=C(S::=Xd)**
C		C::=epsilon (S::=Xd)
B		B::=d

- First-follow conflict

This example can be fixed

<<Grammar 12>>

$$\begin{array}{rcl} S & ::= & Xd \\ X & ::= & C \\ X & ::= & Ba \\ C & ::= & \epsilon \\ B & ::= & d \end{array}$$

Substitute $[d/B]$ and $[\epsilon/C]$

<<Grammar 13>>

$$\begin{array}{rcl} S & ::= & Xd \\ X & ::= & \epsilon \\ X & ::= & da \end{array}$$

Substitute $[\epsilon|da]/X$

<<Grammar 14>>

$$\begin{array}{rcl} S & ::= & d \\ S & ::= & dad \end{array}$$

- Can apply left factoring to turn <<Grammar 14>> into LL(1)
- This is not always possible in general.

Incorporating everything

1. Disambiguiate the grammar if it is ambiguous
2. Eliminate left-recursion if it contains any
3. Apply left-factoring to eliminate first-first conflict
4. Apply substitution to eliminate first-follow conflict
5. Repeat step 3, (but might not converge)

- It is ok to stop at step 2 and allow some small scale backtracking exists in the parser.

From this point onwards

Summary

- The roles and functionalities of lexers and parsers in a compiler pipeline
- There are two major types of parser, top-down parsing and bottom-up parsing (next week)
- How to eliminate left-recursion from a grammar,
- How to apply left-factoring
- How to construct a `LL(1)` predictive parsing table