Compilers Course
Lecture 5: Top-Down Parsing

A top-down parser:

- Scans the input (token sequence) left-to-right.
- Attempts to discover a left-most derivation from the start symbol that matches the input (always expand left-most non-terminal).
- Thus constructs the parse tree top-down, left-to-right.
- Two main implementation models:
  - Predictive
  - Backtracking

Predictive Top-Down Parsing

- Goal: a queue of grammar symbols $X_1, ..., X_B$. Initially, n=1 and X_1=S.
- Input: a list of terminal symbols $T_1, ..., T_m$.
- If the current goal symbol is a terminal T', compare it with the current input terminal T: if they match, remove them and continue parsing, otherwise signal a parsing failure.
- Otherwise the current goal symbol is a non-terminal A, with productions $A \rightarrow \alpha_1 \mid ... \mid \alpha_j$.
- Consider the current input terminal T, and every α_i:
  - If α_i can start with T, then choose $A \rightarrow \alpha_i$.
  - If α_i can derive the empty string, and T can come after an A, then choose $A \rightarrow \alpha_i$.
  - There must be at most one possible production for the terminal T and the non-terminal A. Implication: never has to go back and change earlier choices.
  - Replace the goal symbol A with the chosen α_i, then continue parsing.

nullable, FIRST, FOLLOW

To construct a decision table we need these functions:

- nullable(X) = true if X is a variable and $X \Rightarrow^* \varepsilon$ nullable($X_1..X_n$) = true if nullable(X_i) for $1 \leq i \leq n$.
- $FIRST(\alpha) = \{ a \mid a \text{ is a terminal and } \alpha \Rightarrow^* a\beta \}$
- $FOLLOW(X) = \{ a \mid a \text{ is a terminal and } S \Rightarrow^* \alpha X a\beta \}$

LL(1) decision table M

- Rows are non-terminals
- Columns are terminals
- Table entries are productions: $\forall$ production $A \rightarrow \alpha$:
  - for every $t \in FIRST(\alpha)$, $M[A, t] = A \rightarrow \alpha$
  - if nullable(α), then for every $t \in FOLLOW(A)$, $M[A, t] = A \rightarrow \alpha$
- Error if any table entry has more than one definition.

Computing nullable, FIRST, FOLLOW

nullable:

1. Set nullable(X) = false for all variables X
2. ∀ production $A \rightarrow B_1..B_n$: if nullable($B_1..B_n$), then set nullable(A) = true, in particular, $A \rightarrow \varepsilon$ (n=0) implies nullable(A)
3. Repeat step 2 until no changes occur

FIRST:

1. Set FIRST(X) = {} for all variables X
2. If $A \rightarrow \alpha$ is a production, then FIRST(A) includes FIRST(α)
   2.1. If $\alpha = a\beta$, a is a terminal, then FIRST(α) includes a
   2.2. If $\alpha = B\beta$, B is a variable:
      2.2.1. FIRST(α) includes FIRST(B)
      2.2.2. If nullable(B), then FIRST(α) includes FIRST(β)
3. Repeat step 2 until no changes occur

FOLLOW:

1. Set FOLLOW(X) = {} for all variables X
2. If $A \rightarrow ...\ B\ \beta$ is a production:
   2.1.1. FOLLOW(B) includes FIRST(β)
   2.1.2. If nullable(β), then FOLLOW(B) includes FOLLOW(A)
3. Repeat step 2 until no changes occur

Example

$Z \rightarrow d \mid XYZ$
$Y \rightarrow c \mid \varepsilon$
$X \rightarrow Y \mid a$

nullable:
$Y \rightarrow \varepsilon$ implies Y is nullable
$X \rightarrow Y$ and Y is nullable implies X is nullable
Z is not nullable

FIRST:
$X \rightarrow a$ implies a ∊ FIRST(X)
$Y \rightarrow c$ implies c ∊ FIRST(Y)
$Z \rightarrow d$ implies d ∊ FIRST(Z)

$X \rightarrow Y$ and c ∊ FIRST(Y) implies c ∊ FIRST(X)
$Z \rightarrow XYZ$ and a,c ∊ FIRST(X) implies a,c ∊ FIRST(Z)

$Z \rightarrow XYZ$, nullable(X), and c ∊ FIRST(Y) implies c ∊ FIRST(Z); no change
$Z \rightarrow XYZ$ and nullable(XY) implies FIRST(Z) includes FIRST(Z); no change

So: FIRST(X) = {a, c}, FIRST(Y) = {c}, FIRST(Z) = {a, c, d}

FOLLOW:
Z → XYZ and c ϵ FIRST(Y) implies c ϵ FOLLOW(X)
Z → XYZ, nullable(Y), and a, c, d ϵ FIRST(Z) implies a, c, d ϵ FOLLOW(X)

Z → XYZ and a, c, d ϵ FIRST(Z) implies a, c, d ϵ FOLLOW(Y)

X → Y and a, c, d ϵ FOLLOW(X) implies a, c, d ϵ FOLLOW(Y)

So: FOLLOW(X) = FOLLOW(Y) = {a, c, d}, FOLLOW(Z) = {}

M[X, a]: X → a (FIRST-rule)
X → Y (FOLLOW-rule, nullable(Y), a ϵ FOLLOW(X))
M[X, c]: X → Y (FIRST-rule)
M[X, d]: X → Y (FOLLOW-rule, nullable(Y), d ϵ FOLLOW(X))
M[Y, a]: Y → ε (FOLLOW-rule, a ϵ FOLLOW(Y))
M[Y, c]: Y → c (FIRST-rule)
Y → ε (FOLLOW-rule, c ϵ FOLLOW(Y))
M[Y, d]: Y → ε (FOLLOW-rule, d ϵ FOLLOW(Y))
M[Z, a]: Z → XYZ (FIRST-rule, a ϵ FIRST(XYZ))
M[Z, c]: Z → XYZ (FIRST-rule, c ϵ FIRST(XYZ))
M[Z, d]: Z → d (FIRST-rule)
Z → XYZ (FIRST-rule, d ϵ FIRST(XYZ))

In this example the table M is ambiguous, so the grammar is not LL(1).

LL(1) stands for:
- Left-to-right scan of the input
- Leftmost derivation
- 1 token lookahead used for decision making

Recursive Descent Parsing

A Recursive Descent (RD) parser is an LL(1) parser implemented using a set of recursive procedures.

For well-behaved grammars, constructing an RD parser is easy:
- One procedure per variable.
- Each procedure starts by inspecting the current token and choosing a production to parse.
- Parsing a production becomes a sequence of statements to match tokens or parse variables (via recursive procedure calls).

Example:

```
S → if E then S else S
    | id := E
    | begin S ; S end
E → id | num

(* code for a 1-token buffer *)
var t:token := undefined

advance() =
        t := undefined

token() =
        if t = undefined then t := scanner()
        return t

match(expected_token) =
  if token() = expected_token then advance() else error()

(* recursive descent parser *)
S() =
  case token() of
     IF ==> (match(IF); E(); match(THEN); S(); match(ELSE); S())
     ID ==> (match(ID); match(ASSIGN); E())
     BEGIN ==> (match(BEGIN); S(); match(SEMI); S(); match(END))

E() =
   case token() of
     ID ==> (match(ID))
     NUM ==> (match(NUM))

(* the right-hand sides above can be optimized to call advance()
   instead of match() on the first terminal *)
```

<u>Non-LL(1) Problem #1: ambiguous productions</u>

Grammar G:
S → if E then S else S
| if E then S

Both productions have the same FIRST set: { if }. Given the goal S, how do we choose the correct production?

Solution: delay the decision by *factoring out* the common prefix of the two productions:

S → if E then S S'
S' → else S | ε

This grammar rewrite process is called *left-factoring*. Now the decision is easy: For goal S with input token "if" there is only one alternative. For goal S' we choose "else S" for input token "else", and ε for all other input tokens.

The left-factored grammar recognizes the same set of strings as the original grammar, but the parse trees change.

In RD code:

```
S() =
        case token() of
                IF ==> (match(IF); E(); match(THEN); S(); S'())
                ...

S'() =
        case token() of
                ELSE ==> (match(ELSE); S())
                default ==> ()
```

<u>Non-LL(1) Problem #2: left-recursive productions</u>

Grammar G:
E → E + T | T
T → ...

FIRST(E+T) = FIRST(E) = FIRST(T), so given the goal E, how can we choose the correct production?

Assume we want to parse E+T:

```
        E() =
                case token() of
                ??? ==> (E(); match(PLUS); T())
```

A recursive call from E() to E() without consuming any input == infinite loop. Very bad.

The grammar G is said to be *left-recursive*. Top-down parsers loop on such grammars.

We must rewrite the grammar G to an equivalent grammar G' that recognizes *the same set of input strings*, without being left-recursive.

Consider derivations from E:
$E \Rightarrow E + T \Rightarrow E + T + T \Rightarrow \ldots \Rightarrow E + T + \ldots + T \Rightarrow T + \ldots + T$

That is, E simply derives sequences of T's separated by "+" signs. An equivalent non left-recursive formulation is:

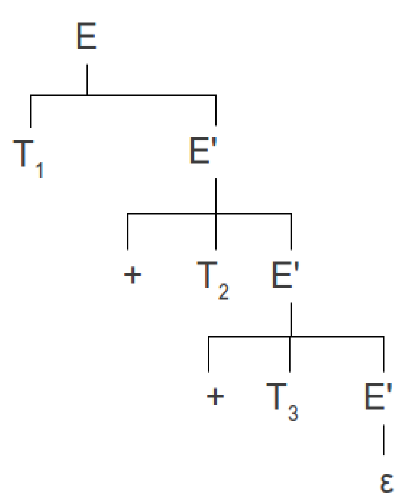
$E \rightarrow T\ E'$
$E' \rightarrow +T\ E' \mid \varepsilon$

Now:
$E \Rightarrow T\ E' \Rightarrow T + T\ E' \Rightarrow T + T + T\ E' \Rightarrow T + T + T$

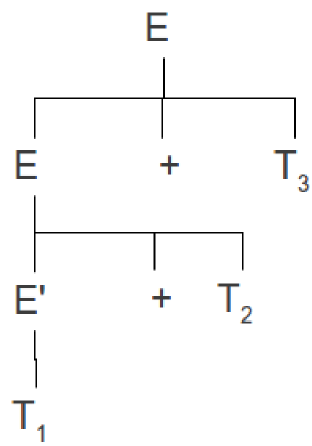
In RD code:
```
E() =
  (T(); E'())

E'() =
        case token() of
              PLUS ==> (match(PLUS); T(); E'())
              default ==> ()
```

G' recognizes the same set of input strings as G, unfortunately the parse trees are very different. For T1+T2+T3 we now get:



where before we had:



A consequence of rewriting grammars to work with top-down parsing methods is that the parse trees must be post-processed to recover the original intended structure.

<u>Top-Down Parsing with Deep Backtracking</u>

1. Goal: a queue of grammar symbols X_1, ..., X_n. Initially, n=1 and X_1=S.
2. Input: a list of terminal symbols T_1, ..., T_m.
3. Choice stack: stack of <goal queue, input list, integer i> triples. Initially the choice stack is empty.
4. Algorithm:
   4.1. If Goal = ε and Input = ε return success.
   4.2. Is X_1 a terminal? Is $X_1=T_1$? **Yes:** remove X_1 and T_1, go to step 1. **No:** signal an error (go to step 4.10).
   4.3. X_1 must be a variable A.
   4.4. Assume A has the productions $A \rightarrow \alpha_1 \mid ... \mid \alpha_j$
   4.5. If j=1: only one choice: replace X_1 with α_1, go to step 4.1.
   4.6. If j>1: multiple choices: start by setting i=1.
   4.7. Push <goal, input, i> on goal stack.
   4.8. Replace X_1 with α_i.
   4.9. Go to step 4.1.

   When an error is signaled:
   4.10. If the choice stack is empty: return error.
   4.11. Pop <goal, input, i> from choice stack. Now goal = Aγ and $A \rightarrow \alpha_1 \mid ... \mid \alpha_j$.
   4.12. If i=j: tried all choices: signal an error (go to step 4.10).
   4.13. If $i<j$: more choices to try: set i to i+1, go to step 4.7.

Backtracking Example

S → aBcD
B → b | bc
D → d | cd

Input "abcd".

I: abcd
G: S

S ⇒ aBcD is the only option

I: abcd
G: aBcD

a=a so move ahead

I: bcd
G: BcD

Remember this state, try B ⇒ b

I: bcd
G: bcD

b=b so move ahead

I: cd
G: cD

c=c so move ahead

I: d
G: D

Remember this state, try D ⇒ d

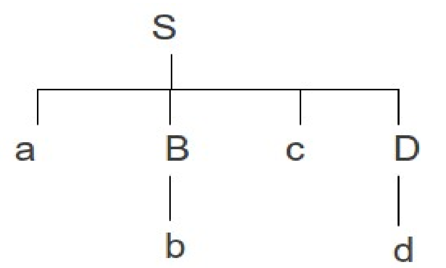
I: d
G: d

d=d so move ahead

I:
G:

Both I and G are empty. Success.

The parse tree is:



Now consider the input "abcccd".

I: abcccd
G: S

S ⇒ aBcD

I: abcccd
G: aBcD

a=a so move ahead

I: bcccd
G: BcD

Remember this state (1a) where i=1, try B ⇒ b

I: bcccd
G: bcD

b=b so move ahead

I: cccd
G: cD

c=c so move ahead

I: ccd
G: D

Remember this state (2a) where i=1, try D ⇒ d

I: ccd
G: d

d ≠ c, signal failure

Restore state (2a), set i=2, save state (2b), try D ⇒ cd

I: ccd
G: cd

c=c so move ahead

I: cd
G: d

d ≠ c, signal failure

Restore state (2a), i=j so signal failure.

Restore state (1a), set i=2, save state (1b), try B ⇒ bc

I: bcccd
G: bccD

b=b so move ahead

I: cccd
G: ccD

c=c so move ahead

I: ccd
G: cD

c=c so move ahead

I: cd
G: D

Remember this state (3a) where i=1, try D ⇒ d

I: cd
G: d

d ≠ c, signal failure

Restore state (3a), set i=2, save state (3b), try D ⇒ cd

I: cd
G: cd

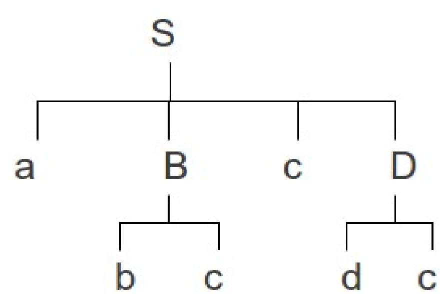
c=c so move ahead

I: d
G: d

d=d so move ahead

Both I and G are empty. Success.

The parse tree is:



This is called deep backtracking because after the first successful derivation of B we got failures later on, forcing us to go back and try other alternatives for B.

> + intuitive, fairly simple to illustrate
> - complicated to implement, slow parser, rarely used in compilers