Top-Down Parsing

Parser Classes

- ***Top-down*** LL (read Left to right, follow Leftmost derivation)
- ***Bottom-up*** LR (read Left to right, follow Rightmost derivation)
- Named after how the parse tree is built: top to bottom or bottom to top
- Also differ and power

Top-Down Parser

Power and Implementation

- Less powerful
  - o There are grammars that can be parsed with bottom-up parsers but not with top-down parsers with the same lookaheads
- More complicated for back-end operations unless some limitations are in place
- Easier to implement, especially *recursive descent*

Operations and Properties

- Builds the parse tree from the root down
- Follows leftmost derivation, therefore, it is called LL(k)
- Uses stack memory
  - o Systems stack in recursive descent
  - o Expands the topmost symbol on the stack (leftmost in derivation)
- Tokens from program being parsed are **never** on the stack
  - o the stack contains tokens from the grammar that are **expected** in the program

Push-Down Algorithm

- Utilize stack memory
- Start with pushing initial the nonterminal S
- At any moment
  - o if **a terminal** is on the top of the stack
    - ▪ if it matches the incoming token
      - the terminal is popped
        and the token is consumed (next token is requested)
    - ▪ error otherwise
  - o if a <nonterminal> is on the top of the stack
    - ▪ the nonterminal is replaced with one of its RHS productions
    - ▪ if more than one production, must *predict* the correct one
    - ▪ errors possible in a predictive parser if none predicted

- Successful termination
    - Stack is empty and **EOFtk** is the remaining token

Example

Use the unambiguous expression grammar to top-down parse the following program: `id+id*id`

Problems in Top-Down Parsing

1. **Left recursive productions** (direct or indirect)
    - infinite stack growth
    - can always be handled by a removal procedure that does not alter the grammar just writes it differently
2. **Non-deterministic productions** (more than one production for a <nonterminal>)
    - determinism is verified by *First* and *Follow* sets
    - sets must be pairwise disjoint for the same <nonterminal>
    - may be handled (make deterministic), but not guaranteed
        - *left-factorization* is the general technique for that

Preparing Grammar for LL(1)

- Not every grammar can be converted to LL(k=1). The objective is to avoid the above problems. The steps are:
    1. Remove left recursion (guaranteed)
    2. Verify k=1 using First and Follow sets. If k<=1, done. Otherwise
        - Left factorize and verify again
        - This step is not guaranteed to succeed

Left Recursion

- Top-down parsers cannot handle left-recursion, direct or indirect
- Any direct left-recursion can be removed with equivalent grammar
- Indirect left-recursions can be replaced by direct, and subsequently removed
- This process is guaranteed to succeed but care must be taken if removing multiple recursions
- Removing direct left recursion
    - separate all left recursive from the other productions for each nonterminal

      $\boxed{A \to A\alpha \mid A\beta \mid \quad \gamma \mid \delta}$
    - introduce a new nonterminal X
    - change nonrecursive productions to

      $A \to \gamma X \mid \delta X$
    - replace recursive productions by

      $X \to \epsilon \mid \alpha X \mid \beta X$

Example

Remove left recursion from {E->E+T | T}. Result is {E->TA', A'->+TA' | ϵ}

Nondeterminism

- Grammar requires the number of lookaheads equal to the lookahead needed for the worst nonterminal (max k)
  - Thus computing k (the number of lookaheads) needed for a grammar comes down to computing k for each nonterminal and then taking the max value
- Nonterminal is deterministic when there is only one choice of a production
  - No lookahead needed to make decision thus k=0
- Nonterminal is non-deterministic when there is more than one production thus k>0
- The objective is to ensure that 1 lookahead (k=1) will be enough to make the right decision for each nonterminal

  There are two parts here
    - *First* and *Follow* sets verify whether k=1 or k>1
    - *Left factorization* potentially reduces k
  - There are LL(k>1) grammar that cannot be expressed as LL(k-1) and thus converting grammar to LL(1) is not always possible

Example

if then [else] is an example of not-LL(1) construct and cannot be reduced to LL(1), but it may be solved in LL(1)-parser using other techniques such as by ordering productions (not covered here).

Grammar Verification for LL(1)

1. Remove all left recursions first
2. Nonterminals without choice have k=0

   N -> α
3. Nonterminals with multiple productions must be verified whether k=1 or k>1

   N -> α | β

   N -> α | ϵ

   N -> α | β | ϵ
4. The entire grammar is LL(max k over all nonterminals).

   Thus, each nonterminal with multiple productions must be LL(1) for the entire grammar to be LL(1)

Nondeterministic Nonterminal Verification for LL(1)

1. Computer *First* sets for each RHS choice
2. Compute *Follow* set of LHS if there is the empty production
3. All the above sets must be pairwise disjoint (excluding the empty symbol) for the nonterminal to be LL(1)
   - otherwise the nonterminal is LL(k>1) and thus the entire grammar is LL(k>1)

Example

Verify LL(1) or LL(k>1)

S -> α k=0

N -> β k=0

M -> α | β k=1 (suppose)

Then the entire grammar is LL(1).

Left Factorization

- Combines alternative productions starting with the same prefixes
  - this delays decisions about predictions until new tokens are seen
  - this is a form of extending the lookahead by utilizing the stack
  - bottom-up parsers extend this idea even further
  - from implementation perspective, it may not be necessary to perform left-factorization but rather to delay decision in the code

Example

Assume grammar

S -> **ee** | **b**A**c** | **b**A**e**

A -> **d** | **c**A

When the parsed program is **bcde**, it is impossible to decide production on S while looking only at the incoming 1 token (**b** here)

Rewrite by combining common prefix **b**A

S -> **ee** | **b**A(**c** | **e**) -> **ee** | **b**AX

X -> **c** | **e**

A -> **d** | **c**A

First Set

The name *First* comes from the fact that the set contains token which may show up first on the stack when the production is used and before a token is consumed.

We compute *First* of right hand sides of non-deterministic productions (more than one choice) and not individual nonterminals unless needed in the algorithm.

N -> α | β

Must computer First(α) and First(β)

FIRST(α) algorithm:

α is a single element (terminal/token or nonterminal) or ϵ

if α=terminal y then FIRST(α)={y}

if α=ϵ then FIRST(α)={ϵ}

if α is nonterminal and we have productions $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \ldots$ then $FIRST(\alpha) = \bigcup FIRST(\beta_i)$

$\alpha = X_1X_2 \ldots X_n$

set $FIRST(\alpha)=\{\}$

for j=1..n include $FIRST(X_j)$ in $FIRST(\alpha)$, but STOP when X_j is not *nullabl*e

X is nullable if X -> ϵ directly or indirectly

X = ϵ or ϵ is a member of FIRST(X)

terminal is of course never nullable

if X_n was reached and is also nullable then include ϵ in $FIRST(\alpha)$

Example

S -> A**b** | B**c** First(A**b**) = {**h**,**i**,**c**,**d**} First(B**c**) = {e,**g**} thus k=1
A -> D**f** | CA
B -> **g**A | **e**
C -> **d**C | **c**
D -> **h** | **i**

Follow Set

The name Follow comes from the fact that the set contains tokens that physically follow the given nonterminal on the stack (are below).

A -> α | ϵ

Note that Follow set never contains ϵ.

FOLLOW(A) algorithm:

If A=S then put end marker (*e.g.*, EOFtk) into FOLLOW(A) and continue

Find all productions with A on rhs: Q -> αAβ

- if β begins with a terminal q then q is in FOLLOW(A)
- if β begins with a nonterminal then FOLLOW(A) includes FIRST(β) - {ϵ}
- if β=ϵ or when β is nullable then include FOLLOW(Q) in FOLLOW(A)

Grammar is LL(k<=1) When

- No left recursion
- For all nonterminals with multiple productions, the *FIRST* sets of the RHSs are disjoint (excluding ϵ)

- In addition, if any RHS is ϵ or nullable, the *FOLLOW* of that nonterminal must be disjoint from all *FIRST* sets for the nonterminal

Example

Rewrite the last expression grammar until proven LL(1).

Removing left recursion gives

E -> TQ		k=0
Q -> +TQ | -TQ | ϵ	FIRST(+TQ)={+} FIRST(-TQ)={-} FOLLOW(Q)={EOFtk)}	k=1
T -> FR		k=0
R -> *FR | /FR | ϵ	FIRST(*FR)={*} FIRST(/FR)={/} FOLLOW(R)={+ -) EOFtk}	k=1
F -> (E) | id	FIRST(id)={id} FIRST((E))={(}	k=1

Entire grammar is LL(1)

Predictions in LL(1)

- Ensure grammar is LL(k<=1)
- In the top-down algorithm
  - When terminal is on top of the stack, just match as before
    - remove from the stack and get next token from the scanner
  - When nonterminal is on top of the stack
    - if the is single production for this nonterminal, just apply it
    - otherwise look at the next token from the scanner and
      - if it is in one of the *FIRST* sets for the RHSs, apply that RHS
      - otherwise if the is empty production, apply the empty production
      - otherwise there is an error