Transforming Grammars

CF Grammar Terms

- Parse trees.
  - Graphical representations of derivations.
  - The leaves of a parse tree for a fully filled out tree is a sentence.

- Regular language v.s. Context Free Languages
  - how do CFL compare to regular expressions?
  - Nesting (matched ()'s) requires CFG,'s RE's are not powerful enough.

- Ambiguity
  - A string has two derivations
  - E -> E + E | E * E | id
    - x + x * y

- Left-recursion
  - E -> E + E | E * E | id
  - Makes certain top-down parsers loop

Grammar Transformations

- Backtracking and Factoring
- Removing ambiguity.
  - Simple grammars are often easy to write, but might be ambiguous.
- Removing Left Recursion
- Removing Λ-productions

Removing Left Recursion

- Top down recursive descent parsers require non-left recursive grammars
- Technique: Left Factoring

```
E -> E + E  |  E * E |  id

E   ->  id E'
E'  ->  +  E  E'
      | *  E  E'
      | Λ
```

General Technique to remove direct left recursion

- Every Non terminal with productions

  T -> T n | T m (left recursive productions)

  | a | b (non-left recursive productions)

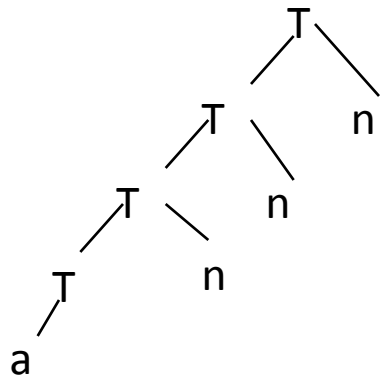
- Make a new non-terminal T’
- Remove the old productions
- Add the following productions

  T -> a T’ | b T’

  T’ -> n T’ | m T’ | Λ

“a” and “b” because they are the rhs of the non-left recurive productions.

(a | b) (n | m) *



Backtracking and Factoring

- Backtracking may be necessary:

```
S -> ee | bAc | bAe
A -> d | cA
```

- try on string "bcde"

```
S -> bAc      (by S -> bAc)
  -> bcAc     (by A -> cA)
  -> bcdc     (by A -> d)
```

- But now we are stuck, we need to backtrack to
  - S -> bAc
  - And then apply the production (S -> bAe)

- Factoring a grammar
  - Factor common prefixes and make the different postfixes into a new non-terminal

```
S -> ee | bAQ
Q -> c | e
A -> d | cA
```

Removing ambiguity.

- Adding levels to a grammar

E -> E + E | E * E | id | (E)

Transform to an equivalent grammar

E -> E + T | T

T -> T * F | F

F -> id | (E)

Levels make formal the notion of precedence. Operators that bind “tightly” are on the lowest levels

The dangling else grammar.

- st -> if exp then st else st
  | if exp then st
  | id := exp

- Note that the following has two possible parses

  if x=2 then if x=3 then y:=2 else y := 4

  if x=2 then (if x=3 then y:=2) else y := 4

  if x=2 then (if x=3 then y:=2 else y := 4)

Adding levels (cont)

- Original grammar

```
st ::=  if exp  then st  else st
     |  if exp then st
     |  id := exp
```

- Assume that every `st` between then and else must be matched, i.e. it must have both a then and an else.
- New Grammar with addtional levels

```
st        ->  match  |  unmatch
match     ->  if exp  then match  else  match
          |   id := exp
unmatch   ->  if  exp  then  st
          |   if  exp  then  match  else  unmatch
```

Removing Λ-productions

- It is possible to write every CFL that does not contain Λ (the empty string) without any Λ in any RHS

- S -> aDaE
- D -> bD | E
- E -> cE | Λ

Rules

1. Find all non-terminal, N, such that N derives Λ

2. For each production, A -> w, create new productions, A -> w', where w' is derived from w by removing non-terminals that derive Λ (found in rule 1 above)

3. Create a new grammar from the original productions, together with the productions formed in step 2, removing any productions of the form, A -> Λ .

S -> aDaE
D -> bD | E
E -> cE | Λ

1. Find all non-terminal, N, such that N derives Λ

E -> Λ

2. For each production, A -> w, create new productions, A -> w', where w' is derived from w by removing non-terminals that derive Λ (found in rule 1 above)

S -> aDa
D -> Λ
E -> c

3. Create a new grammar from the original productions, together with the productions formed in step 2, removing any productions of the form, A -> Λ .

S -> aDaE
D -> bD | E
E -> cE

S -> aDa
E -> c

Top to bottom example

- Start with an easy (to understand) grammar
- Transform it to one that is easier to parse
- Apply some of the transformation rules

```
RE -> RE bar RE
RE -> RE RE
RE -> RE *
RE -> id
RE -> ^
RE -> ( RE )
```

A datatype suitable for representing Regular Expresions

- Build an instance of the datatype:

```
data RegExp a
  = Lambda                          -- the empty string ""
  | Empty                           -- the empty set
  | One a                           -- a singleton set {a}
  | Union (RegExp a) (RegExp a)     -- union of two RegExp
  | Cat (RegExp a) (RegExp a)       -- Concatenation
  | Star (RegExp a)                 -- Kleene closure
```

Ambiguous grammar

```
RE -> RE bar RE
RE -> RE RE
RE -> RE *
RE -> id
RE -> ^
RE -> ( RE )
```

- Transform grammar by layering
- Tightest binding operators (*) at the lowest layer
- Layers are Alt, then Concat, then Closure, then Simple.

```
Alt -> Alt bar Concat
Alt -> Concat
Concat -> Concat  Closure
Concat -> Closure
Closure -> simple star
Closure -> simple
simple -> id  | ( Alt ) | ^
```

Left Recursive Grammar

```
Alt -> Alt bar Concat
Alt -> Concat
Concat -> Concat  Closure
Concat -> Closure
Closure -> simple star
Closure -> simple
simple -> id  |  (Alt )  | ^
```

For every Non terminal with productions

```
T ::= T n | T m    (left recursive prods)
    |  a  |  b     (non-left recursive prods)
```

Make a new non-terminal T'

Remove the old productions

Add the following productions

```
T  ::=  a T'  |  b T'
T' ::=  n  T' |  m  T' | Λ
```

```
Alt         ->  Concat moreAlt
moreAlt     -> Bar Concat moreAlt
             | Λ
Concat      -> Closure moreConcat
moreConcat  -> Closure moreConcat
             | Λ
Closure     ->  Simple Star
             |  Simple
Simple      -> Id
             | ( Alt )
             | ^
```