CSE431 – Translation of Computer Languages

- Context Free Grammars

Doug Shook

Quick Review

- What is a language?
- What is a grammar?
  - What are the parts of a grammar?
- So far we have only seen right-linear grammars and regular languages
  - Not enough!

Context Free Grammars

- Right linear grammars only allow non-terminals on the right
  - And only one of them
- With context-free grammars, anything is fair game
- Example:

S -> AB
A -> aA
  | λ
B -> bB
  | λ

Derivations

- The process of applying productions produces a derivation
- ->*
  - Apply zero or more productions
- ->+
  - Apply one or more productions
- Therefore:

  A string w is in L(G) iff S ->* w

Derivation Types

- Leftmost: rewrite the leftmost nonterminal each time

  S -> AB -> aAB -> aB -> abB -> ab

  – Referred to as *left sentential* form

- Rightmost: rewrite the rightmost nonterminal each time

  S -> AB -> AbB -> Ab -> aAb -> ab

  – Referred to as *canonical* form

Parse Trees

- Let's create a parse tree for the derivations on the previous slide

- Does the order of the productions matter?

Ambiguity

- Consider the following CFG:
  - E -> E + E
    | E * E
    | E
    | x
- Let's construct a parse tree for the string

x + x * x

Ambiguity

- Ambiguity occurs when multiple parse trees exist for some string
  - Or if there are multiple leftmost derivations
- This happens in English too!
- Why is this undesirable?
  - What can we do about it?

Removing Ambiguity

- If an ambiguous grammar is found there may be a non-ambiguous grammar for that same language
- Try the following:
  - Regroup symbols
  - Add productions / non-terminals
  - Enforce precedence
- How can we rewrite our grammar to be non-ambiguous?

Reducing Grammars

- Sometimes non-terminals will be useless:

```
S -> A
A -> aS
   |  λ
B -> b

S -> A
   |  λ
A -> aA
   |  B
   |  S
B -> bB
```

Practice

- Given the following grammar:

```
S -> AA
A -> AAA
   |  bA
   |  Ab
   |  a
```

give a leftmost and rightmost derivation for the string "aabaa"

- Construct parse trees for each derivation
- What is L(G)?

Practice

- Is the grammar on the preceding slide ambiguous?
  - Provide a string that proves ambiguity
  - Fix the ambiguity, if necessary

- Is the following grammar ambiguous?

```
S -> if E then S
   |  if E then S else S
   |  λ
```

- If so, come up with a string and parse trees to prove ambiguity
  - Fix the ambiguity

Parsing

- Does the stream of tokens conform to this language's grammar?
  - This is the task of the parser
- There are two approaches to parsing:
  - Top Down
  - Bottom Up

Top Down Parsers

- A top down parser will start at the root and work downward
  - The parser must predict which production to take

- Example:

```
P -> ( P )
   |  a
```

If we are given the string "((a))", which productions would our parser predict?
  - What will the parse tree look like?

Predict Sets

- In order for this to work, we have to know what predictions to make
  - Given some non-terminal *N* which production should I take?
- Predict sets are used for this purpose:
  - Derives-λ
  - FIRST
  - FOLLOW

Derives - λ

- Used to determine which non-terminals can derive the empty string
  - Why is this important?
- A non-terminal *A* derives λ if there exists some production for *A* that derives λ
- A production derives λ if every symbol on the right hand side of the production derives λ

Derives - λ

S -> Ba
B -> CD
| b
C -> c
| λ
D -> d
| λ

- Which non-terminals derive lambda?
  - Which productions derive lambda?

Derives - λ

- Algorithm is in the text

- Short version:

```
Initialize all productions to length of RHS
   While (more work to do)
      If the length of a productions RHS = 0
         Remove LHS nonterminal from all other
         productions, update lengths
```

FIRST(A)

- Given a non-terminal *A*, which terminals can begin the RHS?

- Algorithm (short version):

```
Initialize FIRST(A) to be empty
For each production p from A
    If RHS of p starts with terminal a, add a to FIRST
    Else RHS starts with non-terminal X
       Add FIRST(X) to FIRST(A)
       If Derives-λ(X), continue to next symbol
```

FIRST(A)

S -> Ba
B -> CD
| b
C -> c
| λ
D -> d
| λ

FOLLOW(A)

- Given a nonterminal *A*, which terminals can follow *A*?
  - Augment grammar with an end-of-input token ($)
    - Ensure every non-terminal (except S) must be followed by a terminal

- Algorithm (short version):

```
For each non-term, A
   Initialize FOLLOW(A) to be empty
   For each RHS containing A
      Let tail(a) be all symbols after A
      Add FIRST(tail(a)) to FOLLOW(A)
      If Derives-λ(tail(a))
         add FOLLOW(LHS) to FOLLOW(A)
```

FOLLOW(A)

S -> Ba$
B -> CD
| b
C -> c
| λ
D -> d
| λ

PREDICT(P)

- Given a production P, which tokens will trigger the application of P?

- Algorithm:

```
For each production, P
Initialize PREDICT(P) to be empty
   add FIRST(RHS) to PREDICT(P)
   if Derives-λ(RHS)
     Add FOLLOW(LHS) to PREDICT(P)
```

PREDICT(P)

S -> Ba$
B -> CD
| b
C -> c
| λ
D -> d
| λ

Exercises

- Generate Derives-λ, FIRST, FOLLOW, and PREDICT for the following:

```
S -> AC$
C -> c
   | λ
A -> aBCd
   | B
B -> bB
   | λ
```

Exercises

- Generate Derives-λ, FIRST, FOLLOW, and PREDICT for the following:

S -> A$
A -> BC
  | DEFG
  | G
B -> b
C -> c
  | λ
D -> d
  | λ
E -> CD
F -> f
G ->g
  | λ