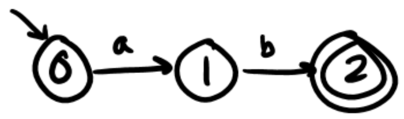
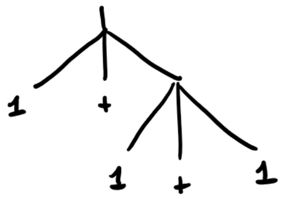
Discussion 8 (10/29)

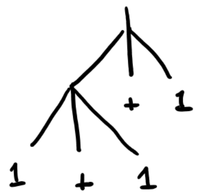
CFGs and Parsing

CFGs

- NFA and DFA states have no concept of memory or what happened prior to reaching that state
- 

- Above, state 1 has no knowledge of how state 0 got to it; all it knows is that now, to reach a final state, it has to take letter ‘b’
- CFGs are just another way to check if a string is accepted, but with some notion of memory
- CFGs have
  - Terminals: Something that terminates
  - Non-Terminals: Something that does not terminate
  - Production Rule: How to expand a nonterminal to get some combination of terminals and non-terminals
  - Start State: Where the CFG processing starts
- **Example 1:**
  - S -> aSa | B
    B -> bC | C
    C -> ε
  - Terminals: a and b
    - They do not lead to other matches, they terminate themselves
  - Non-Terminals: S, B, and C
    - S can be expanded into aSa, which could be expanded into aaSaa or aεa, and so forth for B and C too
  - Production Rules: S -> aSa | B (and B -> and C ->)
  - Start State: S
- Ambiguous Grammars: Grammars for which there are two leftmost derivations
- **Example 2:**
  - S -> S + S | 1 is ambiguous
  - Derive 1 + 1 + 1
  - Derivation 1: S -> S + S -> 1 + S -> 1 + S + S -> 1 + 1 + S -> 1 + 1 + 1
  - Derivation 2: S -> S + S -> S + S + S -> 1 + S + S -> 1 + 1 + S -> 1 + 1 + 1
- Ambiguity is bad because it becomes difficult to determine the string
- To avoid ambiguity, we have to refactor the CFG
- **Example 3:**
  - S -> S + S | 1
  - Convert into:
    - S -> 1 + S | 1

        - S -> S + 1 | 1
    - Now there is no choice for which S to replace - only one leftmost derivation
- S -> 1 + S | 1 is right recursive, meaning the self recursion only happens towards the right
    - This hold even if it is S -> T + S | 1, we only check right recursion for the non-terminal we are currently at, not T
    - It is also right associative, since the + operation happens on the right
- S -> S + 1 | 1 is, meanwhile, left recursive and left associative
- Left and right recursion matter in the context of parse trees
- Right recursive 1 + 1 + 1
    - S -> 1 + S -> 1 + (1 + S) -> 1 + (1 + 1)
    - Note the parentheses are only being written for clarity
    - 

- Left recursive 1 + 1 + 1
    - S -> S + 1 -> (S + 1) + 1 -> (1 + 1) + 1
    - 

- This is a key concept because for the way our computers implement top-down recursive descent parsers, we need the CFG to be right-recursive with right associative operators
- Left Recursive: S -> S + 1 | 1
    - let parse_S toklist =
      parse_S toklist
      ...
    - This leads to an infinite loop
- Right Recursive: S -> 1 + S | 1
    - let parse_S toklist =
      match toklist 1;
      parse_S
    - Will process correctly without infinite looping
- To convert a left-recursive CFG, we can often use simple substitution
- **Example 4:**
    - S -> S + S | M
    - Can rewrite as S -> M + S | M, since S will be substituted by M eventually anyways
- Sometimes, the CFG is more complex, in which case we use factoring

- **Example 5:**
  - S -> S + S | 1 | 2 | 3
  - 1 | 2 | 3 can be moved to a new non-terminal
  - S -> S + S | T
    T -> 1 | 2 | 3
  - Now we can apply substitution to get:
    S -> T + S | T
    T -> 1 | 2 | 3
- The lower we go in our CFG, the higher the precedence becomes, in that we evaluate the high precedence information first
- S -> M + S | M
  M -> N * M | N
  N -> n
  - Possible derivation: S -> M + S -> (N * M) + S -> … -> (n * n) + n
  - Even without the parentheses, which were added for clarity, we see that multiplication takes precedence over addition
  - Hence, multiplication occurs lower in the CFG than addition
- S -> M * S | M
  M -> N + M | N
  N -> n
  - Possible derivation: S -> M * S -> (N + M) * S -> … -> (n + n) * n
  - Now the addition takes precedence over multiplication
  - Thus we can reorder the CFG to set our own precedences

Converting to CFGs (No need to do all of them!! Do a few, then skip to lexing and parsing and come back - I just left them here if students wanted to try more examples)

- **Example 6:**
  - Strategy: write out some strings to see the pattern
  - Accepted Strings: Epsilon, ab, aabb, aaabbb, ...
  - Answer: S -> aSb | epsilon
- **Example 7:**
  - Accepted Strings: Epsilon, aab, aaaabb, …
  - a appears twice as many times as b does
  - We see the number or a’s and b’s are equality functions of each other, meaning a and b will appear in the same line of the CFG
  - Answer: S -> aaSb | epsilon
- **Example 8:**
  - x > y is equivalent to x - y > 0
  - I must have as many a’s as b’s and more
  - S -> aSb takes care of an equal number of a’s and b’s, but we also want strictly more a’s
  - Answer:
    S -> aSb | T
    T -> aT | a

    - S takes care of generating a’s and b’s, and once we’ve finished generating b’s, we need to add at least one a to meet the strict inequality condition
    - We don’t end with epsilon because there will always be at least 1 a, even if there are 0 b’s
- **Example 9:**
    - x and y are unrelated, so the number of a’s relative to b’s is arbitrary
    - However, z = x + y, so the CFG must contain c’s as a function of a’s and b’s
    - S -> aSc to take care of counting the number of a’s
    - S -> aSc | T
      T -> bTc to take care of counting the number of b’s
    - Answer:
      S -> aSc | T
      T -> bTc | epsilon
    - We could add additional constraints like x >= 1, in which case we would get
      S -> aSc | aTc
      T -> bTc | epsilon
      To account for at least one a and c
- **Example 10:**
    - To get an equal number of a’s on each side, we can do
      S -> aSa
    - To get an equal number of b’s on each side, we can do
      S -> aSa | bSb
    - We could have an even number of letters, in which case we end with epsilon, or an odd number of letters, in which case the middle will contain an a or a b
    - Answer:
      S -> aSa | bSb | T
      T -> a | b | epsilon
- **Example 11:**
    - Any regex can be expressed using a CFG
    - a*
        - S -> aS | epsilon
    - a+
        - S -> aS | a
    - a|b
        - S -> a|b
    - a?
        - S -> a | epsilon
    - ab
        - S -> ab
    - (wp)+g* can be broken up into A = (wp)+ and B = g*
      S -> AB
      A ->
      B ->
    - A = (wp)+ can have 1 or more wp

S -> AB
A -> TA | T
B ->
T -> wp

- B = g* can have 0 or more g
- Answer:
  S -> AB
  A -> TA | T
  B -> KB | epsilon
  T -> wp
  K -> g
- We could simplify this if we wanted to, but not required
  S -> AB
  A -> wpA | wp
  B -> gB | epsilon

- **Example 12:**
  - This is similar to z = x + y, but this time, the middle letter will be repeated
  - We may think a similar approach to z = x + y, but this is tricky because the middle letter is repeating this time
  - Think of this example as a concatenation: $a^x b^x b^z c^z$
  - Answer:
    S -> AB
    A -> aAb | epsilon
    B -> bBc | epsilon

Parsing

- Project 4a involves lexing and parsing
- Lexing involves converting some string into tokens to enforce uniformity
- Parsing uses recursive descent parsing
- We'll do a brief introduction to lexing and parsing
- **Example 13:**
  - We first want to check if the index (pos) we are at is greater than the length of the string, in which case we have reached the end of the string and need a [Tok_EOF]
  - Otherwise, we want to check the characters present at the current position using string_match
  - If we match a token, which we can check using regex, then we add it to our ongoing token list and make another recursive call
  - Increment position in the recursive call based on how many characters were matched
  - For integers, we'll need to keep track of the length of the int we find to increment pos correctly, so we'll use matched_string
  - Tok_Int also takes an integer, so we'll have to pass that in
  - Answer:

```
let rec lexer (input : string) : token list =
    let length = String.length input in
    let rec helper pos =
        if pos >= length then [Tok_EOF]
        else if (Str.string_match (Str.regexp "+") input pos) then
            Tok_Add :: (helper (pos + 1))
        else if (Str.string_match (Str.regexp "-?[0-9]+") input pos) then
            let matched_int = Str.matched_string input in
            Tok_Int (int_of_string matched_int) :: (helper (pos +
            (String.length matched_int))
```

- **Example 14:**
  - Match_token takes a token list and an expected token, and either returns the tail of the list if the token matched, or throws an error. This can be implementation dependent
  - Lookahead returns the first token in the list
  - Follow the CFG during implementation
  - Parser will just start the parsing, with start state S
  - If, after parsing S, there are still tokens left, we'll raise an error. We can check this using a comparison to Tok_EOF
  - Otherwise we can return the expression generated
  - For parse_S, either way we parse M first, so just get that out of the way
  - Then we want to lookahead. If we see a +, follow the first path and create a Plus expression. Else, follow the second path and just return what we got from parsing M
  - Similar logic for parse_M, except we will be looking for some integer n
  - We raise an error in parse_M because if we don't see an int, we have an invalid input that cannot be parsed
  - Answer:

```
let rec parser toks =
    let (rem_toks, expr) = parse_S toks in
    if rem_toks <> [Tok_EOF] then raise (InvalidInputException "no!")
    else expr
and parse_S toks =
    let (rem_toks, expr) = parse_M toks in
    match (lookahead rem_toks) with
    | Tok_Add -> let toks2 = match_token rem_toks Tok_Add in
        let (toks3, expr2) = parse_S toks2 in
        (toks3, Plus (expr, expr2))
    | _ -> (rem_toks, expr)
and parse_M toks =
    match lookahead toks with
    | Tok_Int i -> let toks2 = match_token toks (Tok_Int i) in (toks2, Int i)
    | _ -> raise (InvalidInputException "no!")
```