Recursive Descent Parsing

Introduction to Recursive Descent Parsing

1. Finished the first part of the course on automaton and languages
2. Next is recursive descent parsers
    - Parsers that expand start symbol into program which is being parsed

Parser Classification

1. LR: Bottom Up Parser
    - L: Scan left to right
    - R: Traces rightmost derivation of input string
2. LL: Top Down Parser
    - L: Scan left to right
    - L: Traces leftmost derivation of input string
3. Parser must be deterministic
    - Looking at leftmost token allows parser to uniquely choose a rule
    - Also must look at end of sentence to choose the correct, unique rule
    - This is why LR parsers are popular
4. Maximum amount of lookup needed for a parser
    - LL(0), LL(1), LR(1), LR(k)
        - 0: No lookahead required
        - 1: 1 token lookahead is required
        - Lower is better!
    - Number (k) refers to maximum look ahead
    - Most parsers don't require more than 2 tokens of lookahead
    - Deterministic parsing is known as non-backtracking parsing
    - Non-deterministic parsing means it can't uniquely choose a rule and will have to backtrack and try another path
        - Not time efficient for very large programs

Recursive Descent Parsing

1. Writing a function to parse each of the non-terminal variables
    - Non-terminal variable -> convert -> select rule for expansion
    - Select correct rule for expansion
        - matchToken(token)
        - Matching: Consumes token
        - Non-matching: Error
    - How do we select the correct rule?
        - peekToken
    - Output is an abstract syntax tree
2. Familiar example:
    - expr ::= expr addop term | term
    - term ::= term '*' factor | factor
    - factor ::= '(' expr ')' | num | id
    - addop ::= '+' | '-'

Backus Naur Form (BNF)

1. Backus Naur Form
    - expr ::= expr addop term | term
    - term ::= term '*' factor | factor
    - factor ::= '(' expr ')' | num | id

- addop ::= ‘+’ | ‘-’
- expr and term have left recursion
  - How do we handle this?

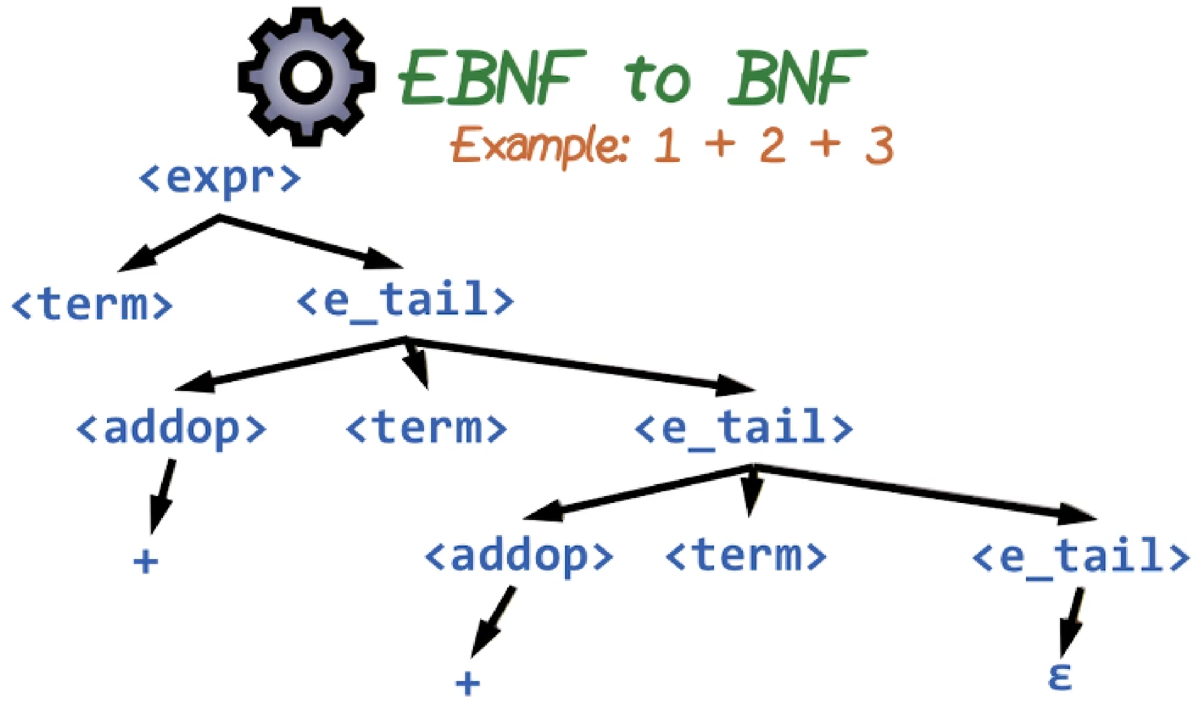
2. Extended Backus Naur Form (EBNF)
   - Uses {} notation to indicate 0 or more
     - Concept is similar to ‘*’ operator of regexp
     - Num ::= [0-9][0-9]*
   - expr ::= term addop term
     - Head: term
     - Tail: addop term
   - This can be rewritten as:
     - expr ::= term {addop term}

EBNF Quiz

1. Match the operator with its symbol. Put the corresponding letter in the text box.
   - [bnf] {1,3}: Between one and three repetitions
   - [bnf]+: One or more repetitions
   - [bnf]*: Zero or more repetitions

EBNF to BNF

1. term ::= term ‘*’ factor | factor
   - EBNF
     - term ::= factor {‘*’ factor}
   - BNF
     - term ::= factor t_tail
     - t_tail ::= ‘*’ factor t_tail | ’ ’
2. Expressions
   - expr ::= term e_tail
   - e_tail ::= addop term e_tail | ’ ’



EBNF to BNF

Revised Grammar Rules

1. EBNF to BNF
   - expr ::= term e_tail
   - e_tail ::= addop term e_tail | ’ ’
   - term ::= factor t_tail
   - t_tail ::= ‘*’ factor t_tail | ’ ’
   - factor ::= ‘(’ expr ‘)’ | num | id
   - addop ::= ‘+’ | ‘-’

Revised Grammar Rules Quiz

1. Consider the following revised grammar which has removed left recursion and converted it to right. State whether the following statement is true or false.
   - Conversion of left recursion to right recursion makes the grammar ambiguous.
     - False

Solutions Parts 1 and 2

1. EBNF Nonrecursive version

```
// expr ::= term {addop term}
// addop ::= '+' | '-'
void expr()
{
    term();
    int token;
    while ((token = peekToken()) == PLUS || token == MINUS) {
```

```
        matchToken(token);
        term();
    }
}
```

2. BNF Recursive Version

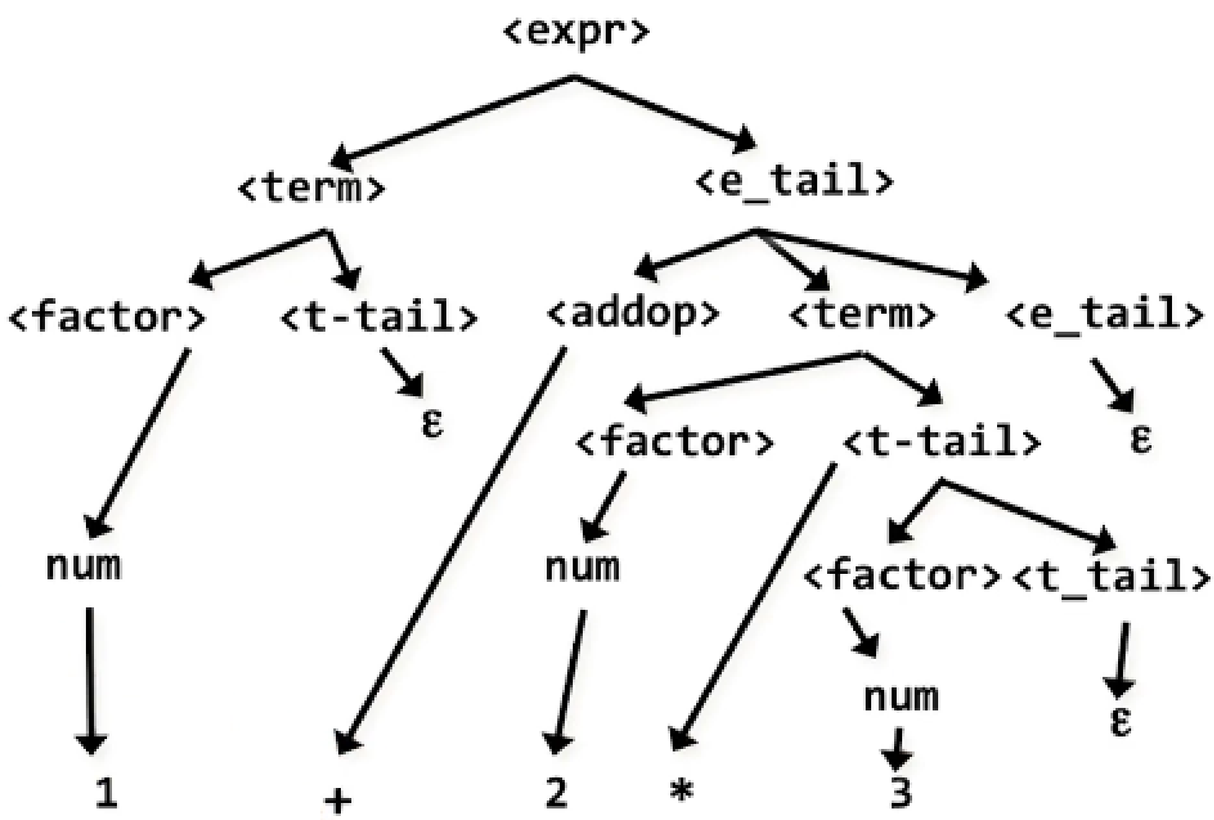
```
// expr ::= term e_tail
// e_tail ::= addop term e_tail | ''
// addop ::= '+' | '-;

enum {PLUS, MINUS, MULT, LPAREN, RPAREN, NUM, ID};
void expr()
{
    term();
    e_tail();
}

void e_tail()
{
    int token;
    if ((token = peekToken()) == PLUS || token == MINUS) {
        matchToken(token);
        term();
        e_tail();
    } else {
        return;
    }
}

void term()
{
    factor();
    t_tail();
}

// term ::= factor t_tail
// t_tail ::= '*' factor t_tail | ''
void t_tail()
{
    if (peekToken() == MULTI) {
        matchToken(MULTI);
        factor();
        t_tail();
    } else {
        return;
    }
}

// factor ::= '(' expr ')' | num | id
void factor()
{
    if (peekToken() == LPAREN) {
        matchToken(LPAREN);
        expr();
```

```
        matchToken(RPAREN);
    } else if (peekToken() == NUM) {
        matchToken(NUM);
    } else if (peekToken() == ID) {
        matchToken(ID);
    }
}
```

Regex Grammar Quiz

1. Write a grammar in BNF to generate regular expression:
   - a* b | c+
   - Use e for epsilon and appropriate non-terminals and S as start symbol
   - a, b, and c are tokens
   - Use ':' for the arrows
2. Solution:
   - S -> A B | C
   - A -> e
   - A -> a A
   - B -> b
   - C -> c D
   - D -> c D
   - D -> e

Solutions Example Part 1 and 2



Recursive Descent Parsing

More of Left Recursion

1. Remove left recursion by converting from BNF to EBNF
2. If a grammar is left recursive we must first rewrite it to make it right recursive
   - Simple immediate left recursion
   - A-> A u | v where v does not start with A
     - Change to A -> v A’
       - A’ -> u A’ | ’ ’
   - Change exp -> exp addop term
     - exp -> term exp’
     - exp’ => addop term exp’ | ’ ’
3. General Immediate Left Recursion
   - A -> Au1 | Au2 | . . . | Aun | v1 | v2 | . . . | vm
     - Where vi does not start with A
   - Solution:
     - A -> v1A’ | v2A’ | . . . | vmA’
     - A’ -> u1A’ | u2A’ | . . . | unA’ | ’ ’
   - exp -> exp + term | exp - term | term
     - exp -> term exp’
     - exp’ -> +term exp’ | -term exp’ | ’ ’

Left Recursion Quiz 1

1. Given the following grammar, determine if it is left recursive.
    - E -> E + T | T
    - T -> T * F | F
    - F -> (E) | id
2. Solution: Yes

Left Recursion Quiz 2

1. Apply the transformation to E and rewrite the grammar.
    - Use 'e' for epsilon.
    - E -> E+T|T
    - Solution:
        - E -> TE'
        - E' -> +TE'|e

Left Recursion Quiz 3

1. Apply the transformation to T and rewrite the grammar.
    - Use 'e' for epsilon.
    - T -> T*F|F
    - Solution:
        - T -> FT'
        - T' -> *FT'|e

Left Recursion Quiz 4

1. Rewrite the grammar in 5 lines.
    - E -> TE'
    - E' -> +TE'|e
    - T -> FT'
    - T' -> *FT'|e
    - F -> (E)|id

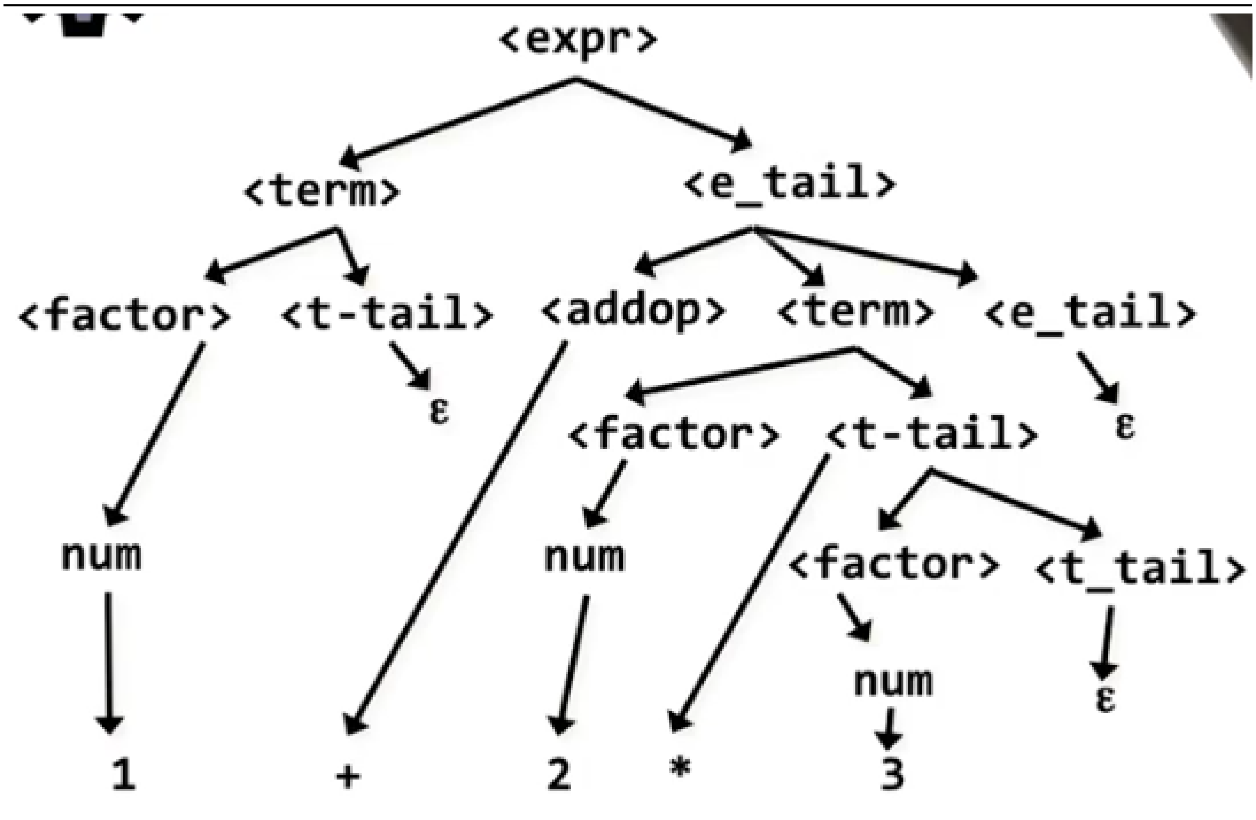
Left Factoring

1. Left factoring is required if two or more grammar rule choices share a common prefix string
    - A -> uv | uw
2. Would cause difficulties if we look ahead only one token
    - Solution:
        - A -> uA'
        - A' -> v | w

Construct Syntax Tree Part 1

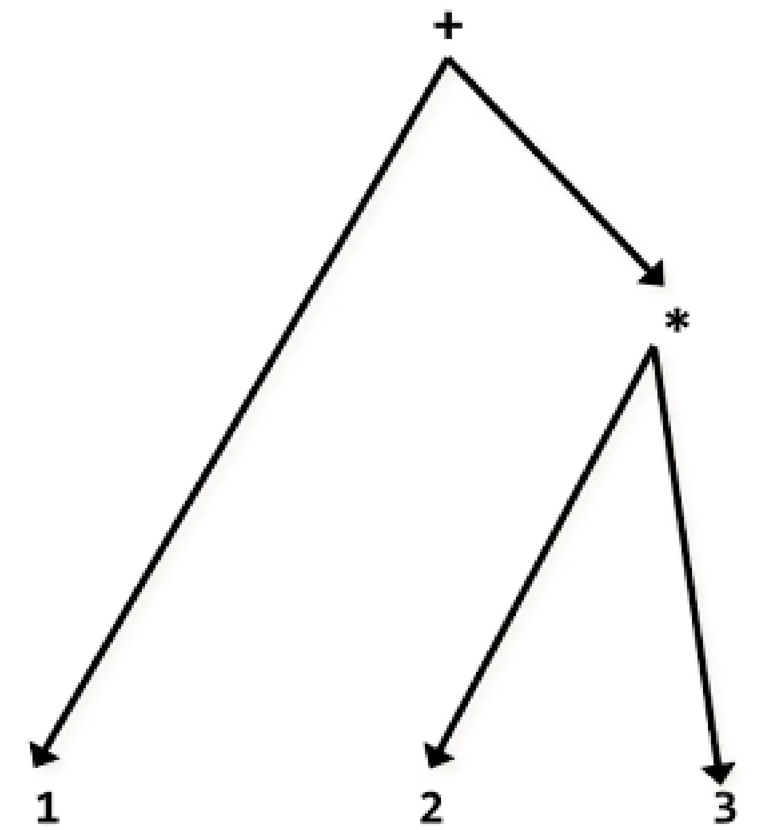
1. What is the desired output? How do we convert a parse tree to an abstract syntax tree?
    - Goal: Remove intermediate non-terminal nodes to connect the token to the sub-tree root node for derivation
    - Epsilon expansions are meant to terminate recursion
        - Remove all e edges
        - Recursively remove their predecessors with outdegree 1
    - Remove intermediate non-terminal nodes to connect the token to the sub-tree root node for derivation
        - Recursively apply this process

- Replace expr with operator



Parse Tree

Construct Syntax Tree Part 2



Abstract Syntax Tree

1. Syntax-driven Directed Construction

```
SyntaxTree* expr()
{
    SyntaxTree* temp = term();
    int token;
    while ((token = peekToken()) == PLUS || token == MINUS) {
        matchToken(token);

        SyntaxTree* tree = makeOpNode(token);
        tree->leftChild = temp;
        tree->rightChild = term();
        temp = tree;
    }
    return temp;
}
```

2. Can either construct a parse tree and convert it to an AST or generate the AST directly using the above approach