Parsing

- There are two general approaches to Parsing, corresponding to leftmost and rightmost derivations
  - What is a leftmost derivation?

- If we trace out the steps of a leftmost derivation, adding nodes to the AST as we do each step, we will perform top-down parsing of the code
- If we do a rightmost derivation of the grammar, we end up building the tree from the bottom, using bottom-up parsing

- Bottom-up parsing is the more powerful, and easier to automate approach
  - However, our project will be to build a top-down parser because it is much easier to hand-code
- We will do top-down in this chapter, and save bottom-up for Chapter 5

Top-Down Parsing

- There are 2 major approaches to top-down parsing
  - Backtracking parsers
  - Non-backtracking or predictive parsers

- Backtracking parsers are nice theoretically, but are too time-consuming for a practical compiler
  - We will not cover in CS-3510

- There are 2 major predictive algorithms we will cover:
  - Recursive-descent parsing
    - Will be used on term project
  - LL(1) parsing
    - 1st L means left-to-right scan, 2nd L means leftmost derivation, 1 means it looks ahead at 1 token to do prediction

Recursive-Descent Parsing

- In Recursive-Descent Parsing, we (as in: you) essentially do a direct implementation of the grammar rules
- For every non-terminal in the language, we write a method that knows how to scan for it
- Consider the arithmetic grammar below
  - We would write 5 methods, one for each non-terminal
    - E.g., parseExpr(), parseTerm(), etc.
    - But we will NOT make 5 classes in our parser!

```
expr -> expr addop term  | term
term -> term mulop factor | factor
factor -> ( expr ) | IDENT | NUM
 addop -> + | -
 mulop ->  * | /
```

Recursive-Descent Parsing

- Let's look at parseFactor

```
expr -> expr addop term  | term
term -> term mulop factor | factor
factor -> ( expr ) | IDENT | NUM
 addop -> + | -
 mulop ->  * | /
```

```
private Expression parseFactor () {
    switch (currentToken.tokenType) {
        case Token.LPAREN_TOKEN:
            advanceToken();
            Expression returnExpr = parseExpression ();
            matchToken(Token.RPAREN_TOKEN);
            return returnExpr;
            break;
        case Token.IDENT_TOKEN:
            Token oldToken = advanceToken();
            return createIdentExpr(oldToken);
            break;
        case Token.NUM_TOKEN:
           Token oldToken = advanceToken();
            return createNumExpr(oldToken);
            break;
        default:
            logParseError();
            return null;
    }
}
```

Recursive-Descent Parsing

- How about parseExpression
  - The technique we just looked at doesn't really work with choice real well
  - Any ideas of how to attack this?
- Remember EBNF? How would you express the expr production in EBNF?
- expr -> term {addop term}

```
expr -> expr addop term  | term
term -> term mulop factor | factor
factor -> ( expr ) | IDENT | NUM
 addop -> + | -
 mulop ->  * | /
```

Recursive-Descent Parsing

- The EBNF form suggests the solution
  – expr -> term {addop term}
  – Look for a term, and then do a while looking for addop term

```
private Expression parseExpression () {

    Expression lhs = parseTerm();

    while (isAddop (currentToken.tokenType)) {
        Token oldToken = advanceToken();
        Expression rhs = parseTerm();
            // make lhs the result, so set up for next iter
        lhs = createBinopExpr (oldToken.tokenType, lhs, rhs);
    }

    return lhs;
}
```

Recursive-Descent Parsing

- Let's look at an IfStatement: if_stmt -> if (expr) stmt [else stmt]

```
private Statement parseIfStmt () {
    matchToken (Token.IF_TOKEN);
    matchToken (Token.LPAREN_TOKEN);
    Expression ifExpr = parseExpression();
    matchToken(Token.RPAREN_TOKEN);
    Statement thenStmt = parseStatement();
    Statement elseStmt = null;

    if (currentToken.tokenType == Token.ELSE_TOKEN) {
        AdvanceToken();
        elseStmt = parseStatement();
    }

    Statement returnStmt = new IfStatement(ifExpr, thenStmt, elseStmt);
    return returnStmt;
}
```

Recursive-Descent Parsing

- What would the code for parseWhileStmt look like?
  - First, what does a while production look like?
  - Second, what does the WhileStmt class look like?
  - Third, what does the actual parseWhileStmt routine look like?

```
private Statement parseWhileStmt () {



}
```

Recursive-Descent Parsing

- Piece of Cake … project #2 is gonna take about 45 minutes to code up

  … well not so fast

- There are a few minor details we've overlooked

- We looked at parseFactor, where: factor -> (expr) | IDENT | NUM
  - There are 3 separate productions for factor
    - We combined into 1 procedure
    - How did we choose which of the 3 productions we should use?
  - What do we do if the first token of the right-hand side is a non-terminal?

Recursive-Descent Parsing

- Example: Look at item #26 on page 492
  - factor -> (expr) | var | call | **NUM**
  - We have two terminals, and two non-terminals
  - How would we write this routine?

- We can only differentiate between the 4 choices based on the input token
- Look at var and call
  - Both must start with ID
- So, if nextToken == (we should use factor -> (expr)
- But if nextToken == ID, we still have a problem
  - Could be either a call or var
- Need to look at token after nextToken
  - What do we do if it is a [?

Recursive-Descent Parsing

- So, what we need to do for non-terminals on rhs of a production is to find the first terminal in their productions
  - If any of these non-terminals' productions have a non-terminal as the first item on their rhs, then we recursively follow this 2nd non-terminal and find what tokens are legal for it
    - This is called finding the **first set** for a non-terminal
  - If we can resolve all ambiguity, using the first sets for all non-terminals on rhs, then we are able to parse the grammar

- However, there is another possible item (other than terminals and non-terminals) on the rhs - an ε
  - If we have an ε on the rhs, then the nextToken may not be part of this production

Recursive-Descent Parsing

- Example: Look at item #12 on page 492
  - stmt_list -> stmt_list stmt | ε
  - To differentiate between the 2 possible productions, we need to use nextToken
    - What can legally follow a stmt_list in this grammar?
- If the nextToken is a } (see item #10), we know we have completed a stmt_list and are looking at the enclosing compound_stmt
  - We should use the stmt_list -> ε production
- Anytime we have ε productions, we must look at tokens which can follow the current non-terminal
  - This is called finding the **follow set** of the non-terminal

Recursive-Descent Parsing

- So, to properly build a recursive-descent parser, we are going to have to be able to generate first sets and follow sets for our grammar
  - We will put off formally looking at the algorithms for this until after we talk about LL(1) parsing
  - Once you understand these algorithms, you should have all the tools you need to build a recursive-descent parser
  - For now, just understand there is some added complexity we have to worry about in some cases

- Note: EBNF may simplify this a bit
  - We could write   stmt_list -> { stmt }
  - Then if the nextToken is in the first set of stmt, and not in the follow set of stmt_list, we know to recurse

LL(1) Parsing

- We will return to first/follow sets, but time for LL(1) parsing
  - **LL(1) parsing** is another top-down, leftmost derivation parsing technique
  - Left-to-right scan, leftmost derivation, 1 token look-ahead
  - Not great for hand coding
  - Good for automation; however, the LR parsing techniques in Chapter 5 are more powerful
    - Thus, LL(1) parsing is only occasionally used in practice
    - However, it provides a good introduction to the type of parsers we will see next chapter
- An LL(1) parser does not use recursive calls like the recursive-descent parser
  - It uses a stack, a parse table, and a simple algorithm which iterates till the stack and input stream are empty

LL(1) Parsing

- Easiest way to understand an LL(1) parser is to see an example
  - We will see what the parser does, and later explain how it knew to make the proper choices

- We start with a stack with just the start symbol S on it
- We terminate the input stream of tokens with a $
- We can show the operation of the parse with a table, showing the state of the stack, the current state of the input string, and the action we take at each state
  - 2 possible actions
    1. If a non-terminal is on the top of stack, expand it, pushing rhs onto stack (called a generate)
    2. If a terminal is on the top of stack, it had better match the first token of input string
       - If so, remove token from both stack and string (called a match)

LL(1) Parsing

- Example: Consider grammar S -> (S) S | ε
  - Input string: ( ) ( )

- We put S on stack
- Since we have a non-terminal on top of stack, we expand it
  - S -> (S) S
    - We will see later how we chose which production to use (parse table created from first/follow sets)
  - We push the tokens on rhs of production onto stack, so the first symbol on rhs is now on top of stack
    - Note this means a leftmost derivation
- A (is on top of stack, we match it to input stream, removing both
- Now, an S is on top of stack – we choose S -> ε
  - Continue till stack and input string both empty

LL(1) Parsing

- Before we look further into the details, back to the big picture
  - We are building an AST
  - When we do a generate step, we may create a new node in the tree
    - This node's children are associated with the non-terminals being pushed back on the stack
    - The items on the stack must then be referenced back to their parent node in some way, so that when they generate their node can be connected as a child of the parent

LL(1) Parsing

- When the top of stack is a token, it must match first token in the input string, or an error has occurred
  - No choice for parser to make
- When the top of stack is a non-terminal, the parser will do a generate
  - The parser may have to make a choice of which production to use for the generate
  - We express the choices the parser should make in the LL(1) Parse Table
    - This is simply a 2D array of non-terminals versus possible look-ahead tokens
    - For a given non-terminal and a given look-ahead, the LL(1) parse table contains the proper production to use

LL(1) Parsing

- For the grammar S -> (S) S | ε the table would be as follows:

	(	)	$
S	S->(S)S	S-> ε	S-> ε

- When we started the parse, the (was the look-ahead
  - Thus, we chose S -> (S) S

- So, when we encounter a non-terminal on top of stack, we simply look this symbol up in the parse table using the look-ahead, and this tells us what to do
  - We just need to know how to generate the table, and we should be able to make a parser
  - You may have guessed that first/follow sets will be used to generate the table

LL(1) Parsing

- For the following grammar and parse table, what would be the steps in an LL(1) parse (show the stack/input states)?

E -> TE'
E' -> +TE' | ε
T -> FT'
T' -> *FT' | ε
F -> (E) | **id**

Input = id + id * id

	id	+	*	(	)	$
E	E->TE'			E->TE'		
E'		E'->+TE'			E'->ε	E'->ε
T	T->FT'			T->FT'		
T'		T'-> ε	T'->*FT'		T'-> ε	T'-> ε
F	F->id			F->(E)		

LL(1) Parsing

- Before we look at how to generate the first/follow sets, need to look at a few issues
- For the parse table to be effective at directing a parse, it can have at most one possible production in each entry
  - It may not be possible to create a parse table with only one valid entry, and thus LL(1) parsing cannot be done
  - If a valid LL(1) parse table can be generated for a particular grammar, we say the grammar is an **LL(1) grammar**
    - If not, the grammar in its current form is not an LL(1) grammar
      - An LL(1) parser cannot be used to parse it (unless you build in a special case)
      - However, more powerful parsers may be successful
        - E.g., LR parsers in next chapter

LL(1) Parsing

- An example of a special case in the parse table would be for nested if-statements
  - We've already seen that they can be ambiguous
  - Specifically, the productions else_part -> **else** stmt | ε create a conflict in an LL(1) parse table for the ELSE_TOKEN look-ahead
  - However, we know that if the look-ahead is an ELSE_TOKEN, we want to associate it with the closest if, so we can just remove the else_part -> ε production from the parse table at the conflicting location

LL(1) Parsing

- Although we can hard-wire special cases into the parser (as seen in the if-else example), we prefer not to do this too often
  - However, you will find that the C- grammar on pg 492 is far from being LL(1)
  - We need techniques for converting a non-LL(1) grammar to be LL(1)
- We will look at 2 techniques for possibly converting a grammar to be LL(1)
  - Left recursion removal
  - Left factoring

- However, there is no guarantee they will be successful in making an LL(1) grammar
  - For grammars common to HLLs, they typically are fairly successful

Left Recursion Removal

- If we want to make an operation left associative, we often make the production involving it left recursive
  - expr -> expr addop term | term
  - For recursive-descent, we would write this as EBNF
    - expr -> term { addop term }
    - But this doesn't help if we are trying to build an automated LL(1) parser
  - Instead, we can rewrite this to eliminate the left recursion
    - expr -> term expr'
      expr' -> addop term expr' | ε
  - This simple conversion will frequently make a grammar LL(1)
  - Note: the same technique can be applied to multiple cases
    - We can convert expr -> expr + term | expr – term | term

Left Recursion Removal

- A more general form is A -> A B | A F | D | E
  - A -> D A' | E A'

    A' -> B A' | F A' | ε

- That was relatively straightforward – however, there is a more complicated case (which thankfully doesn't occur very frequently in common languages)
  - A -> B a

    B -> A a

- A technique exists for this case, but it will not always be successful
  - Approach is to arbitrarily order the non-terminals (A comes before B), and then systematically remove recursion

Left Recursion Removal

- Example:
  - A -> B a | A a | c
  - B -> B b | A b | d
- We will assume that A comes before B
  - First eliminate left recursion in A
    - A -> B a A' | c A'
      A' -> a A' | ε
      B -> B b | A b | d
  - Next, since A is before B, we must eliminate any A as first token on rhs of a B production (I.e., fix B -> A b above)
    - Direct substitute the values of A
    - A -> B a A' | c A'
      A' -> a A' | ε
      B -> B b | B a A' b | c A' b | d

Left Recursion Removal

- Current state of the grammar (from prev page)
  - A -> B a A' | c A'
    A' -> a A' | ε
    B -> B b | B a A' b | c A' b | d
- Next we must eliminate the immediate recursion in B
  - A -> B a A' | c A'
    A' -> a A' | ε
    B -> c A' b B' | d B'
    B' -> b B' | a A' b B' | ε
- Algorithm is in text – basically just nested loops
  - When outer is pointing at A, fix immediate A recursion and any references to A within subsequent productions
- Well, that's real ugly – remember, this technique is used as part of automatic parser generators, not for hand coding
  - Also makes creation of the AST more complicated

Left Recursion Removal

- Consider the original grammar: expr -> expr addop term | term
  - We changed it to: expr -> term expr'
    expr' -> addop term expr' | ε
  - Did this alter the grammar and the parse tree we created?
- Unfortunately, there was a reason we chose left recursion to capture the associativity properly
  - The new parse tree (if built straight from grammar) doesn't capture the associativity correctly
  - Therefore, we must be sure that the routines that actually build the AST build it correctly
  - Adds further complexity to the LL(1) parser

Left Recursion Removal

- New grammar: expr -> term expr'
  expr' -> addop term expr' | ε

- When we expand first production, will push expr' on stack, followed by term
  - Eventually, we will have parsed the entire term (resulting in an Expression), and the expr' will be on top of the stack
  - Expr' needs to add the Expression created by term to the 2nd Expression created by the term it parses
  - Thus, the original term and expr' must be interconnected in some way, so that the 1st Expression gets passed to expr'
  - We're not going to talk about how that would be done, but just be aware that left recursion removal adds problems

Left Recursion Removal

- Examples
  - Look at items 1-5 on page 492.
    - What transformations need to be made to eliminate left recursion?
  - Look at item 12
    - What transformations need to be made to eliminate left recursion?
      - Does this answer make sense?
      - Would this change alter the parse tree or AST?
      - Does it change the syntax of the language?

Left Factoring

- Eliminating left recursion may be effective in converting many grammars
  - However, problems other than left recursion cause conflicts in the parse table
  - Look at item #20 on pg 492
    - simple_expr -> add_expr relop add_expr | add_expr
    - Creates a conflict in parse table
    - Left factoring can fix this
- Left factoring is used where two or more grammar choices have a common prefix string (starting with either terminal or non-terminal)
  - A -> B a | B b becomes
    - A –> B A’
    - A’ -> a | b

Left Factoring

- If we were working with regular expressions, we are essentially doing: A -> B a | B b becomes A -> B (a | b) and then making up a new non-terminal for (a | b)

- How would you left factor item #20, pg 492 ?

- How would you left factor item # 19, pg 492 ?

- How would you left factor
  - if_stmt -> if (expr) stmt | if (expr) stmt else stmt

- Text gives algorithm for this (remember we want to automate)
  - Iterative – outer while iterates while (changesMade)

Left Factoring

- Example:
  - expr -> ident := expr | ident (expr_list) | other
  - What does this look like left-factored?
  - Note that we have combined an assignment statement and a call statement into one production
    - Typically, when we parse the expr -> production we would want to create the expression node in the AST
      - But we don’t know it’s type yet
      - AST generation using the LL(1) parser will get a bit tricky, and require us delay creating the statement
        - But requires us to pass the ident along until the statement gets created

First/Follow Sets

- Hopefully you see how an LL(1) parser works, and how to convert a grammar to be LL(1)
  - But we haven't talked about how to create the parse table
- We will use the idea of first sets and follow sets to create the table
  - Recall we also need them for recursive-descent parsing
- First sets
  - Consider expr -> var = expr | simple_expr (item #18)
  - Your recursive-descent (or LL(1)) parser needs to decide which production to choose, based only on nextToken
    - But both var and simple_expr are non-terminals, which gives us no clue
    - If we knew the legal 1st tokens of both var and simple_expr, perhaps we could make the correct choice of productions
    - We develop first sets for non-terminals to guide us

First/Follow Sets

- First sets (cont)
  - Definition – If X is a non-terminal, first(X) is the set of all terminals that begin the strings which can be derived from X
    - If ε is a legal derivation of X, then ε is in first(X)
  - first(X) contains
    - For each production X -> X_1 X_2 X_3 …
      - If X_1 is a terminal, add X_1 to first(X)
      - If X_1 is a non-terminal, add first(X_1) – ε to first(X)
        - If first(X_1) contains ε, then add first(X_2)- ε to first(X)
        - Continue down chain as long as first(X_i) contains ε
        - If first(X_n) contains ε, then first(X) contains ε

First/Follow Sets

```
for (nt = 0; nt < numNonTerminals; nt++)
    nonTerm[nt].first = null;

while (changesMade) {
    changesMade = false;
    for (prod = 0; prod < numProd; prod++) {
        rhsIndex = 0;
        foundEpsilon = true;
        currProd = production [prod];

        while (foundEpsilon && (rhsIndex < currProd.maxIndex) ) {
            changesMade |= addFirstMinusEpsilon (currProd.rhs[rhsIndex], currProd.lhs);
            foundEpsilon = isEpsilonInFirst(currProd.rhs[rhsIndex]);
            rhsIndex++;
        }
        if (foundEpsilon) {
            changesMade |= AddEpsilonToFirst (currProd.lhs);
        }
    }
}
```

First/Follow Sets

- Before we do examples, want to do one definition we will need later
  - We say that a non-terminal A is **nullable** if there exists a derivation A =>* ε
  - Same as saying nullable if first(A) contains ε

- Example: find the first sets for the following grammar

```
expr -> expr addop term | term
addop -> + | -
term -> term mulop factor | factor
mulop -> *
factor ->( expr ) | NUM
```

First/Follow Sets

- We will do more examples in conjunction with follow sets

- Recall we were trying to find the first sets in order to decide between 2 or more productions, given a next Token
  - But we just saw that sometimes a non-terminal can be nullable, I.e., its first set contains ε
  - Consider items 27,28 on page 493
    - Certainly args is nullable
    - If nextToken is a), then perhaps this would guide us to choose the args -> ε production
  - This is the reason we get interested in follow sets
    - If a non-terminal is nullable, its follow set becomes a player in deciding what production the top-down parser should use

First/Follow Sets

- Definition of follow(A): For a non-terminal A, follow(A) is the set of terminals which can appear immediately to the right of A in some sentential form, i.e., a derivation of the form

  S =>* B A a C exists, where B, C are strings and a is a terminal
- follow(A) will contain:
  - If A is the start symbol, follow(A) contains $ (the input string terminator)
  - If a production B -> C A D exists, then follow(A) contains first(D)- ε
  - If there is a production B -> C A D, and first(D) contains ε, then follow(A) contains follow(B)

First/Follow Sets

- Algorithm for computing follow sets

```
follow (S) = { $ }
follow (all other non-terminals) = { }

while (changesMade) {
    for (each production A -> X1 X2 X3 …) {
        for (each non-terminal Xi) {
            add first(Xi+1) to follow(Xi)
            if ε is in first(Xi+1 …)
                add first (Xi+2) to follow(Xi)
            if ε is in first(Xi+1 … Xn)
                add follow(A) to follow(Xi)
        }
    }
}
```

First/Follow Sets

- Let's do the earlier example and compute follow sets

expr -> expr addop term | term
addop -> + | -
term -> term mulop factor | factor
mulop -> *
factor ->(expr) | NUM

first(expr) = { (, NUM }
first(addop) = { +, - }
first(term) = { (, NUM }
first(mulop) = { * }
first(factor) = { (, NUM }

First/Follow Sets

- Example: compute first and follow sets for the following grammar

```
stmt -> if_stmt | other
if_stmt -> if ( expr ) stmt else_part
else_part -> else stmt | ε
expr -> 0 | 1
```

- Example: compute first and follow sets for the following grammar

```
stmt_seq -> stmt stmt_seq'
stmt_seq' -> ; stmt_seq | ε
stmt -> s
```

First/Follow Sets

- OK, we were computing these sets for use in creating an LL(1) parsing table, or for use in a recursive-descent parser

- Building a parse table
  - For each production A -> B, for each terminal a in first(B) add this production to the parse table at location M[A, a]
    - If first(B) contains ε, for each terminal a in follow(A), add this production to M[A, a]
- Using this same definition of building a parse table, we can define an LL(1) grammar in terms of first and follow sets
  - A grammar is LL(1) if
    - For every production A -> B | C, first(B) and first(C) contain no common elements
    - For every non-terminal A such that first(A) contains ε, first(A) intersect follow(A) is empty

First/Follow Sets

- Construct parse table for the following

```
expr -> expr addop term | term
addop -> + | -
term -> term mulop factor | factor
mulop -> *
factor ->( expr ) | NUM
```

```
first(expr) = { (, NUM }
first(addop) = { +, - }
first(term) = { (, NUM }
first(mulop) = { * }
first(factor) = { (, NUM }
```

```
follow(expr) = { ($, +, -, ) }
follow(addop) = { (, NUM }
follow(term) = { $, +, -, *, ) }
follow(mulop) = {(, NUM }
follow(factor) = {$, +, -, *, ) }
```

First/Follow Sets

- Construct parse table for the following

```
expr -> term expr2
expr2 -> addop term expr2 | ε
term -> factor  term2
term2 -> mulop factor term2 | ε
factor -> ( expr ) | NUM
addop -> + | -
mulop ->  *
```

first(expr) = { (, NUM }
first(expr2) = { +, -, ε }
first(term) = { (, NUM }
first(term2) = { *, ε }
first(factor) = { (, NUM }
first(addop) = { +, - }
first(mulop) = { * }

follow(expr) = { $,) }
follow(expr2) = { $,) }
follow(term) = { $, +, -,) }
follow(term2) = { $, +, -,) }
follow(factor) = {$, *, +, -,) }
follow(addop) = { (, NUM }
follow(mulop) = {(, NUM }

Error Recovery

- For your project, you will need to develop first and follow sets by hand, and use them to guide your development of the recursive-descent routines

- We didn't say a lot about Error Recovery when we looked at Scanners
  - One option would be to raise an Exception when you find a series of characters which doesn't make a legal token
    - Early compilers halted on the 1st error
  - A better approach is to "log" the error in some manner, and then initiate recovery
    - In the Scanner, when you see a character which isn't legal, you simply log the error, return to state 0, essentially throw away any previous characters, and then press on
      - Logging error may mean passing on to parser as an ERROR_TOKEN and letting the parser handle it

Error Recovery

- Error Recovery within the parser is a bit more tricky
- Wide range of possible solutions
  1. Print the word "ERROR" and then stop compiling
     - That's all that's required for Project #2
     - Not all that helpful
  2. When encounter an error, log it, and try to recover somehow so you can keep looking for other errors (error recovery)
  3. When encounter an error, attempt to correct the error and continue compiling (error correction)
     - Must attempt to determine the simplest change (adding, deleting, or changing a token) which will produce correct code
     - Very complicated, and not frequently used on non-academic compilers

Error Recovery

- Probably you will want an error recovery plan which finds errors, logs their occurrence and location (line #), stores a description of the compiler's best guess for what is wrong, and then presses on
  - Pressing on is non-trivial, because pressing on can result in secondary errors being reported which were caused by the original error
    - Remember, we said to always debug errors starting with the 1st one – the rest are suspect!
    - Even if you recover well, if you have a syntax error in something like a variable declaration, you will have many secondary errors

Error Recovery

- There are not well-established practices for error recovery as there are for many other compiler areas
  - Most techniques tend to be specific to the language and the parsing technique
- General principles
  - Discover error as close to where it occurred as possible
    - Want to be able to identify to user
  - Should recover as quickly as possible – I.e., skip over as little code as possible in order to get to a “sync” point from which to start parsing again
  - Should minimize secondary errors
  - Must avoid infinite loop on errors

- Note these aren’t independent – e.g., trying to skip too few characters may result in an infinite loop

Error Recovery

- One technique which works for recursive-descent parsers is **panic mode recovery**
  - It is actually more intelligent and effective than name implies
- For each procedure (parseExpression, parseIfStmt, etc.), we define what the text calls synchronizing tokens
  - If an error is found, we start to consume (throw away) tokens until a synchronizing token is found
- The set of synchronizing tokens contains the follow set to the present structure being parsed
  - It may also contain other tokens too important to be ignored, like { or ELSE, from which we can figure out how to recover
- The parse routines pass the synchronizing set along from routine to routine
  - E.g., if doing parseParenExpr (expr), when call parseExpr we can tell it that it's follow set will be)

Error Recovery

- We aren't going to go into recursive-descent error recovery any deeper in this course
  - Error recovery for LL(1) parsers is similar; you can store the synchronizing set in the parse table
  - We will briefly visit error recovery regarding bottom-up parsers in Chap 5

- That's it for top-down parsing
  - Recursive-descent is the best choice for hand coding
  - Most real-world compilers use bottom-up parsing (Chap 5)