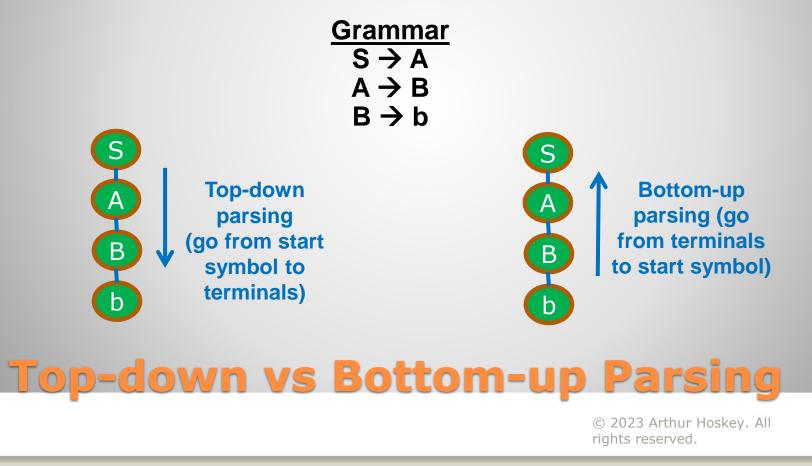
Compilers

Arthur Hoskey, Ph.D. Farmingdale State College Computer Systems Department

Recursive Descent Parsers



- **Top-down parsing**. Begin with the start symbol and keep doing substitutions until only terminals are left.
- Bottom-up parsing. A bottom-up parser starts with terminals and does substitutions in reverse until only the start symbol is left.



- Recursive descent parsers work on LL(1) grammars.
- LL(1)
 - The first L means scan input from left to right.
 - The second L means do a left-most derivation.
 - The 1 means there is one character of lookahead.
- A recursive descent parser is a <u>top-down parser</u>.

Recursive Descent Parsers and LL(1)

Recursive Descent Parsing Overview

Setup a Recursive Descent Parser

- Nonterminals. Write methods for each nonterminal on the LHS.
- Terminals. Use an enum to define tokens that correspond to the terminals.

Parsing

- Parsing Nonterminals. To process or substitute for a nonterminal call its corresponding method.
- Parsing terminals. Use if statements to check for a given terminal. If the next token is the one you are checking for then read the next token from the input stream (this is matching the terminal).

Recursive Descent Parsing Overview

- Sample grammar:
- $\mathsf{S} \not \to \mathsf{AB}$
- $A \rightarrow a$
- $B \rightarrow b$
- Each nonterminal has a corresponding method. This grammar has three nonterminals so we will need three methods.
- To process or substitute for a nonterminal in a RHS call the method that corresponds to the nonterminal.

The S() method corresponds to the $S \rightarrow AB$ production

A() \leftarrow Call A() to process the A nonterminal of S \rightarrow AB

B() \leftarrow Call B() to process the B nonterminal of S \rightarrow AB

<u>A()</u>

S()

// Code to recognize terminal a goes here

B() // Code to recognize terminal b goes here The A() method corresponds to the A→a production

The B() method corresponds to the B→b production

Processing Nonterminals

 Sample grammar: S → AB A → a B → b Check if the next token matches what In the example below, if A() is called S() A() B() 	
AU	od A() is called, then terminal a should the next token in the input stream the next token
B() If (nextToken != TOKEN.b) ← If method B() is called, then terminal b should be the next token in the input stream nextToken = getNextToken() Processing Terminals	
	© 2023 Arthur Hoskey. All rights reserved.

- Here are the basic rules for creating recursive descent parsers:
- Write Methods for Nonterminals. Each nonterminal corresponds to a method. For example, if there is a production A→r then there will need to define a method A().
- **Call Nonterminal Methods.** When doing a substitution for a nonterminal call the method that corresponds to that nonterminal. For example, A().
- Check Next Token for Expected Terminals. Use if statements to check if the next terminal is what is expected. If the next token is what was expected, then consume it.
- **Read Next Token to Consume Terminals.** When consuming a terminal read the next token in the input stream.

Basic Rules for a Recursive Descent Parser

 The following slides will show different grammars and recursive descent parsers for those grammars.

Recursive Descent Parsers

Parser Class Member Variables

- Assume that a Scanner class has been defined.
 - The Scanner class has the TOKEN enum defined inside of it.
 - The Scanner class has a scan() method that returns a TOKEN (this is the next token in the input stream).

Class Parser { Declare Scanner scanner Declare Scanner.TOKEN nextToken

}

Stores the next token. This will be populated by calling the scan() method on the Scanner class. The token enum is defined in the Scanner class.

Scanner will be used to get

Recursive Descent Parsers – Parser Class Member Variables

Parser Helper methods

- getNextToken() Reads the next token from the input stream (use the scanner to do this).
- error() Should print an error message. You can also stop parsing at this point. A real compiler would likely keep going even with the error though.

getNextToken() returns Scanner.TOKEN

nextToken = Scanner.scan()

error(String message)

Print message Exit program

Recursive Descent Parsers – Parser Helper Methods

- Tokens: ID, EOF
- Here is a grammar that only allows one id (an identifier).
- $\mathsf{S} \not \to \mathsf{id}$
- There is a nonterminal so S in this grammar so there needs to be a method for it.
- Inside the method S it expects to find an id in the input stream.
- The only valid token for this grammar is ID.



- Tokens: ID, EQUALS, INTLITERAL, EOF
- Assume the following grammar:
- $S \rightarrow id = intliteral$

An INTLITERAL is just an integer constant

Write pseudocode for a recursive descent parser of the above grammar. Start with a method named parse.

Recursive Descent Parsers

- Tokens: ID, EQUALS, INTLITERAL, EOF
- Assume the following grammar:
- $S \rightarrow id = intliteral$

parse() getNextToken() S() ← Match EOF after S If (nextToken == EOF) print "Success" Else print "Unmatched EOF" Match the terminal id

<u>S()</u>

If (nextToken == ID) getNextToken() If (nextToken == EQUALS) getNextToken() If (nextToken == INTLITERAL) getNextToken() Return (NextToken()

error("S() failed")
Recursive Descent Parsers

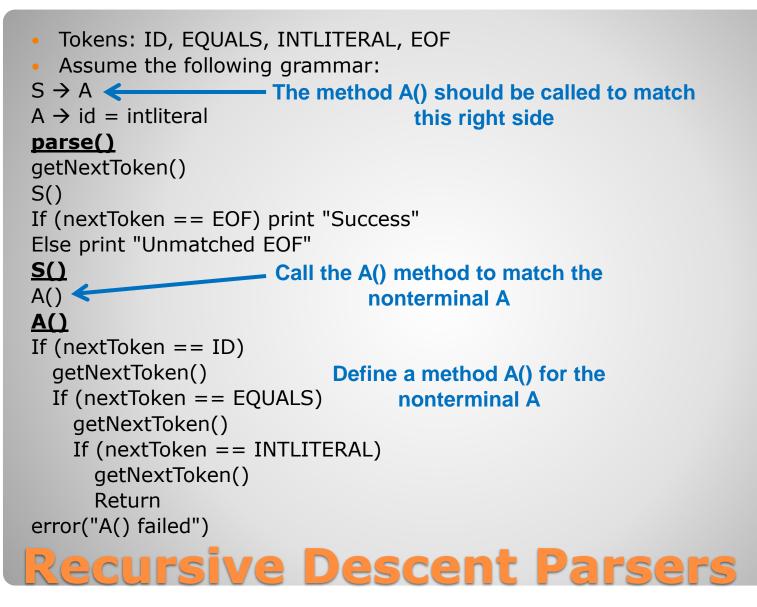
- Tokens: ID, EQUALS, INTLITERAL, EOF
- Assume the following grammar:

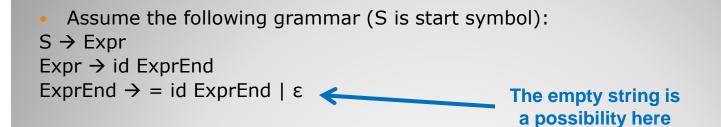
 $\mathsf{S} \to \mathsf{A}$

 $A \rightarrow id = intliteral$

Write pseudocode for a recursive descent parser of the above grammar. Start with a method named parse.

Recursive Descent Parsers





• What are the first, follow, and first+ sets?

Recursive Descent Parsers

Assume the following grammar (S is start symbol):
 S → Expr
 Expr → id ExprEnd
 ExprEnd → = id ExprEnd | ε

• What are the first, follow, and first+ sets?

First(ExprEnd) = { =, ɛ }
First(Expr) = { id }
First(S) = First(Expr) = { id }

Follow(S) = { eof }
Follow(Expr) = Follow(S) = { eof }
Follow(ExprEnd) = Follow(Expr) = { eof }

First+(**ExprEnd** \rightarrow = id **ExprEnd**) = { = }

First+(Expr \rightarrow id ExprEnd) = { id }

First+(S \rightarrow Expr) = { id }

First+(ExprEnd $\rightarrow \epsilon$) = Follow(ExprEnd) = { eof }

Two productions have ExprEnd as the lhs. The first+ sets of these productions do NOT intersect!

This means the choice of which production to use when processing the nonterminal ExprEnd is unambiguous.

> If nextToken is = then use: ExprEnd → = id ExprEnd

If nextToken is eof then use: ExprEnd $\rightarrow \epsilon$

Recursive Descent Parsers

- Tokens: ID, EQUALS, INTLITERAL, EOF
- Assume the following grammar and write the parser:

This grammar recognizes an id followed by "= id" an arbitrary number of times

Expr \rightarrow id ExprEnd

 $S \rightarrow Expr$

ExprEnd \rightarrow = id ExprEnd | ϵ <

The empty string is a possibility here

Recursive Descent Parsers

Tokens: ID, EQUALS, INTLITERAL, EOF Assume the following grammar and write the parser: Expr \rightarrow id ExprEnd ExprEnd \rightarrow = id ExprEnd | ϵ parse() getNextToken() Expr() If (nextToken == EOF) print "Success" Else print "Unmatched EOF" Expr() If (nextToken == ID) getNextToken() ExprEnd() ExprEnd() If (nextToken == EQUALS) getNextToken() If (nextToken == ID) getNextToken() ExprEnd() Else error("Expected ID") If (nextToken == EOF) Return error("ExprEnd() failed")

This grammar recognizes an id followed by "= id" an arbitrary number of times

ExprEnd → = id ExprEnd Match EQUALS then match ID. If it matched them both then it recursively calls ExprId(). The recursive call allows it to keep matching "= id" an arbitrary number of times. First+ set is { = }.

<u>ExprEnd $\rightarrow \epsilon$ </u>

Checking for EOF here determines if we should use the empty string. EOF is in the Follow set of ExprEnd (for this grammar). If EOF is there, then we should use ExprEnd $\rightarrow \epsilon$. First+ set is { eof }.

Recursive Descent Parsers

Additional Parser Helper Method

 match() – Checks if a target token was matched and also reads the next token from the input stream.

match(Scanner.TOKEN expectedToken) returns boolean

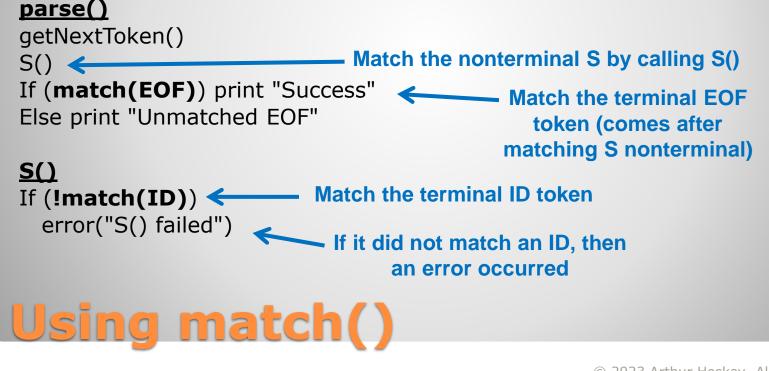
If (nextToken == expectedToken) getNextToken() return true Checks if the next token is the same as what is expected

Print "Token mismatch" Return false If the expected token was matched, then get the next token and return true

If we get here, then the expected token was NOT matched

Recursive Descent Parsers – Additional Parser Helper Method

- Tokens: ID, EOF
- Here is a grammar that only allows one id (an identifier).
- $\mathsf{S} \not \to \mathsf{id}$
- There is a nonterminal so S in this grammar so there needs to be a method for it.
- Inside the method S it expects to find an id in the input stream.
- The only valid token for this grammar is ID.



- Tokens: ID, EQUALS, INTLITERAL, EOF
- Assume the following grammar:
- $S \rightarrow id = intliteral$

Write the parse() and S() methods using the match() helper method.



- Tokens: ID, EQUALS, INTLITERAL, EOF
- Assume the following grammar:
- $S \rightarrow id = intliteral$

Write the parse() and S() methods using the match() helper method.

parse()

getNextToken()
S()
If (match(EOF)) print "Success"
Else print "Unmatched EOF"

<u>S()</u>

If (match(ID)) If (match(EQUALS)) If (match(INTLITERAL)) Return error("S() failed") When S() is called it must match an ID followed by EQUALS followed by INTLITERAL. If it does not, then there is an error.

Note: Match will test for the given token and then get the next token if the test was successful

Using match – Example 1

- Tokens: COLON, ID, PLUS, INTLITERAL, EOF
- Assume the following grammar:

 $\mathsf{S} \not \to \mathsf{A}$

- $\mathsf{A} \not \rightarrow : \mathsf{id}$
- $A \rightarrow + intliteral$

parse() getNextToken() S() If (match(EOF)) print "Success" Else print "Unmatched EOF"

Write the S() and A() methods using the match() helper method (assume parse is the same as the previous example).

Using match – Example 2

- Tokens: COLON, ID, PLUS, INTLITERAL, EOF
- Assume the following grammar:

 $S \rightarrow A$

- $\mathsf{A} \not \rightarrow : \mathsf{id}$
- $A \rightarrow + intliteral$

parse() getNextToken() S() If (match(EOF)) print "Success" Else print "Unmatched EOF"

Write the S() and A() methods using the match() helper method (assume parse is the same as the previous example).

<u>S()</u> A()

<u>A()</u>

If (nextToken == COLON)
 If (match(COLON))
 If (match(ID))
 Return
If (nextToken == PLUS)
 If (match(PLUS))
 If (match(INTLITERAL))
 Return
error("S() failed")

A() checks for COLON and PLUS. If it finds either one, then it matches tokens accordingly. If it does not find either one, then an error has occurred (the RHSs of each A production respectively start with COLON and PLUS).

Using match – Example 2



