Compilers

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

Today's Lecture

- 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.

Grammar

S → A
A → B
B → b

Top-down vs Bottom-up Parsing

Recursive Descent Parsers and LL(1)

- 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 top-down parser.

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

Processing Nonterminals

- Sample grammar:

S → AB

A → a

B → 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.

S() ← The S() method corresponds to the S→AB production

A() ← Call A() to process the A nonterminal of S→AB

B() ← Call B() to process the B nonterminal of S→AB

A()

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

B()

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

Processing Terminals

- Sample grammar:

S → AB
A → a
B → b

- Check if the next token matches what is supposed to be there.
- In the example below, if A() is called then it must match the terminal a.

S()
A()
B()

A()
If (nextToken != TOKEN.a)
    Display error message
nextToken = getNextToken()

If method A() is called, then terminal a should be the next token in the input stream

Get the next token

B()
If (nextToken != TOKEN.b)
    Display error message
nextToken = getNextToken()

If method B() is called, then terminal b should be the next token in the input stream

Basic Rules for a Recursive Descent Parser

- 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.

- 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

}
```

Scanner will be used to get tokens from the input stream

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.

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

Recursive Descent Parsers

- Tokens: ID, EOF
- Here is a grammar that only allows one id (an identifier).

S → 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.

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

S()
If (nextToken == ID)
    getNextToken()
    Return
error("S() failed")

Get the next token and store it in the Parser's nextToken member variable

Match the nonterminal S by calling S()

Match the terminal EOF token (comes after matching S nonterminal)

Match the terminal ID token

The ID token was matched so get the nextToken to match (stores the next token in the Parser nextToken member variable)

Recursive Descent Parsers

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

S → 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 → id = intliteral

parse()

```
getNextToken()
S()
If (nextToken == EOF) print "Success"
Else print "Unmatched EOF"
```

S()

```
If (nextToken == ID)
   getNextToken()
   If (nextToken == EQUALS)
      getNextToken()
      If (nextToken == INTLITERAL)
         getNextToken()
         Return
error("S() failed")
```

Match EOF after S

Match the terminal id

Match the terminal =

Match the terminal for an int literal (a constant)

Recursive Descent Parsers

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

S → A

A → id = intliteral

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 → A

A → id = intliteral

The method A() should be called to match this right side

parse()

```
getNextToken()
S()
If (nextToken == EOF) print "Success"
Else print "Unmatched EOF"
```

S()

```
A()
```

Call the A() method to match the nonterminal A

A()

```
If (nextToken == ID)
   getNextToken()
   If (nextToken == EQUALS)
      getNextToken()
      If (nextToken == INTLITERAL)
         getNextToken()
         Return
error("A() failed")
```

Define a method A() for the nonterminal A

- Assume the following grammar (S is start symbol):

S → Expr
Expr → id ExprEnd
ExprEnd → = id ExprEnd | ε

The empty string is a possibility here

- 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) = { =, ε }
First(Expr) = { id }
First(S) = First(Expr) = { id }

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

First+(ExprEnd → ε) = Follow(ExprEnd) = { eof }
First+(ExprEnd → = id ExprEnd) = { = }
First+(Expr → id ExprEnd) = { id }
First+(S → Expr) = { id }

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 → ε

Recursive Descent Parsers

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

S → Expr

Expr → id ExprEnd

ExprEnd → = id ExprEnd | ε

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

The empty string is a possibility here

Recursive Descent Parsers

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

Expr → id ExprEnd
ExprEnd → = 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 { = }.

ExprEnd → ε
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→ ε. 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

Print "Token mismatch"
Return false
```

Checks if the next token is the same as what is expected

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

Using match()

- Tokens: ID, EOF
- Here is a grammar that only allows one id (an identifier).

S → 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.

parse()

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

Match the nonterminal S by calling S()

Match the terminal EOF token (comes after matching S nonterminal)

S()

```
If (!match(ID))
   error("S() failed")
```

Match the terminal ID token

If it did not match an ID, then an error occurred

Using match – Example 1

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

S → id = intliteral

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

Using match – Example 1

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

S → 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"

S()

```
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 2

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

S → A

A → : id

A → + intliteral

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

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

Using match – Example 2

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

S → A
A → : id
A → + 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).

S()
A()

A()
```
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).

- **End of Slides**

End of Slides