

Syntax Analysis (a.k.a. Parsing)

Grammar analysis and recursive descent parsing.

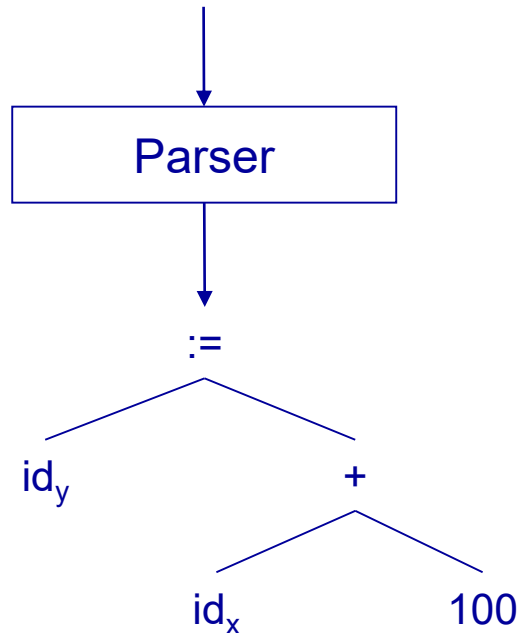
Parser

- Verifies that the grammatical rules of the language are satisfied
- Overall parser structure is based on context-free grammars (a.k.a. BNF/EBNF grammars)
- Input: stream of tokens from the scanner
From the perspective of the parser, each token is treated as a terminal symbol.
- Output: intermediate representation of the source code (e.g., abstract syntax trees)

Parser (continued)

Sequence of tokens returned by the scanner

identifier ["y", (1, 1)] := [(1, 3)] identifier ["x", (1, 6)] + [(1, 8)] intLiteral [("100", (1, 10))]



Functions of the Parser

- Language recognition based on syntax, as defined by a context-free grammar
- Error Handling/Recovery
- Generation of intermediate representation
(We will generate abstract syntax trees.)

The primary focus of this section is language recognition. Subsequent sections will cover error recovery and generation of abstract syntax trees.

Recursive Descent Parsing

- Parsing technique used in this course: recursive descent with single symbol lookahead.
(Briefly discuss other options.)
- Uses recursive methods to “descend” through the parse tree (top-down parsing) as it parses a program.
- The parser is constructed systematically from the grammar using a set of programming refinements.

Initial Grammar Transformations

- Start with an unambiguous grammar.
- Separate lexical grammar rules from structural rules.
 - Let the scanner handle simple rules (operators, identifiers, etc.).
 - Symbols from the scanner become terminal symbols in the grammar for the parser.
- Use a single rule for each nonterminal; i.e., each nonterminal appears on the left side of only one rule.
- Eliminate left recursion.
- Left factor wherever possible.
- Certain grammar restrictions will be discussed in subsequent slides.

Recursive Descent Parsing Refinement 1

- For every rule in the grammar

`N =`

we define a parsing method with the name

`parseN()`

- Example: For the rule

`assignmentStmt = variable "[:=" expression "];" .`

we define a parsing method named

`parseAssignmentStmt()`

- Grammar transformations can be used to simplify the grammar before applying this refinement; e.g., substitution of nonterminals.

Recursive Descent Parsing Methods

The `parseN()` methods of the parser function as follows:

- The scanner method `getSymbol()` provides one symbol “lookahead” for the parsing methods.
- On entry into the method `parseN()`, the symbol returned from the scanner should contain a symbol that could start on the right side of the rule $N = \dots$.
- On exit from the method `parseN()`, the symbol returned from the scanner should contain the first symbol that could follow a syntactic phrase corresponding to N .
- If the production rules contain recursive references, the parsing methods will also contain recursive calls.

Parsing the “Right” Side of a Rule

- We now turn our attention to refinement of the method `parseN()` associated with a production rule “N =” by examining the form of the grammatical expression on the right side of the rule.

- As an example, for the rule

`assignmentStmt = variable "[:=" expression "];" .`

we have defined a parsing method named

`parseAssignmentStmt()`

We focus on systematic implementation of this method by examining the right side of the rule.

Recursive Descent Parsing Refinement 2

- A sequence of syntax factors $F_1 F_2 F_3 \dots$ is recognized by parsing the individual factors one at a time in order.
- In other words, the algorithm for parsing $F_1 F_2 F_3 \dots$ is simply
 - the algorithm for parsing F_1 followed by
 - the algorithm for parsing F_2 followed by
 - the algorithm for parsing F_3 followed by
 - ...

Example: Recursive Descent Parsing Refinement 2

The algorithm used to parse

variable " := " expression ";"

is simply

- the algorithm used to parse variable followed by
- the algorithm used to parse " := " followed by
- the algorithm used to parse expression followed by
- the algorithm used to parse ";"

Recursive Descent Parsing Refinement 3

- A single terminal symbol `t` on the right side of a rule is recognized by calling the “helper” parsing method `match(t)` defined as

```
private void match(Symbol expectedSymbol)
    throws IOException, ParseException
{
    if (scanner.getSymbol() == expectedSymbol)
        scanner.advance();
    else
        ...    // throw ParseException
}
```

- Example: The algorithm for recognizing the assignment operator `":="` is simply the method call
`match(Symbol.assign)`

Recursive Descent Parsing

Refinement 4

- A nonterminal symbol `N` on the right side of a rule is recognized by calling the method corresponding to the rule for `N`; i.e., the algorithm for recognizing nonterminal `N` is simply a call to the method `parseN()`.
- Example: The algorithm for recognizing the nonterminal symbol `expression` on the right side of a rule is simply a call to the method `parseExpression()`.

Example: Application of the Recursive Descent Parsing Refinements

- Consider the rule for an assignment statement:
assignmentStmt = variable "[:=" expression ";" .
- The complete parsing method for recognizing an assignment statement is as follows:

```
public void parseAssignmentStmt()  
{  
    parseVariable();  
    match(Symbol.assign);  
    parseExpression();  
    match(Symbol.semicolon);  
}
```

First Sets

- The set of all **terminal** symbols that can appear at the start of a syntax expression E is denoted $\text{First}(E)$.
- First sets provide important information that can be used to guide decisions during parser development.

First Set Examples from CPRL

- `constDecl = "const" constId "!=" literal ";" .`
`First(constDecl) = { "const" }`
- `varDecl = "var" identifiers ":" typeName ";" .`
`First(varDecl) = { "var" }`
- `arrayTypeDecl = "type" typeId "=" "array" ... ";" .`
`First(arrayTypeDecl) = { "type" }`
- `initialDecl = constDecl | arrayTypeDecl | varDecl .`
`First(initialDecl) = { "const", "var", "type" }`
- `statementPart = "begin" statements "end" .`
`First(statementPart) = { "begin" }`
- `loopStmt = ("while" booleanExpr)? "loop" ... ";" .`
`First(loopStmt) = { "while", "loop" }`

Rules for Computing First Sets

- If t is a terminal symbol, $\text{First}(t) = \{ t \}$
- If all strings derived from E are nonempty, then $\text{First}(E F) = \text{First}(E)$
- If some strings derived from E can be empty, then $\text{First}(E F) = \text{First}(E) \cup \text{First}(F)$
- $\text{First}(E \mid F) = \text{First}(E) \cup \text{First}(F)$

Computing First Sets: Special Cases

The following rules can be derived as special cases of the previous rules

- $\text{First}((E)^*) = \text{First}(E)$
- $\text{First}((E)^+) = \text{First}(E)$
- $\text{First}((E)?) = \text{First}(E)$
- $\text{First}((E)^* F) = \text{First}(E) \cup \text{First}(F)$
- $\text{First}((E)^+ F) = \text{First}(E)$ if all strings derived from E are nonempty
- $\text{First}((E)^+ F) = \text{First}(E) \cup \text{First}(F)$ if some strings derived from E are empty
- $\text{First}((E)? F) = \text{First}(E) \cup \text{First}(F)$

Strategy for Computing First Sets

- Use a bottom-up approach
- Start with simplest rules and work toward more complicated (composite) rules.

Follow Sets

- The set of all **terminal** symbols that can follow immediately after a syntax expression E is denoted $\text{Follow}(E)$.
- Understanding Follow sets is important not only for parser development but also for error recovery.
- If N is a nonterminal, we will use $\text{Follow}(N)$ during error recovery when trying to parse N . To compute $\text{Follow}(N)$ for a nonterminal N , you must analyze all rules that reference N .
- Computation of follow sets can be a bit more involved than computation of first sets.

Follow Set Examples from CPRL

- What can follow an initialDecl?

- From the rule

`initialDecls = (initialDecl)* .`

we know that any initialDecl can follow an initialDecl, so the follow set for initialDecl includes the first set of initialDecl; i.e., “const”, “var”, and “type”.

- From the rules

`declarativePart = initialDecls subprogramDecls .`

`subprogramDecls = (subprogramDecl)* .`

`subprogramDecl = procedureDecl | functionDecl .`

we know that a procedureDecl or functionDecl can follow an initialDecl, so the follow set for initialDecl includes “procedure” and “function”.

Follow Set Examples from CPRL

(continued)

- From the rules

`program = declarativePart statementPart "." .`

`statementPart = "begin" statements "end" .`

we know that `statementPart` can follow an `initialDecl`, so the follow set for `initialDecl` includes “begin”.

- Conclusion:

`Follow(initialDecl) =`

`{ “const”, “var”, “type”, “procedure”, “function”, “begin” }`

- What can follow a `loopStmt`?

- ... (left as an exercise)

- Conclusion:

`Follow(loopStmt) =`

`{ identifier, “return”, “end”, “if”, “elsif”, “else”, “while”, “loop”,
“exit”, “read”, “write”, “writeln” }`

Rules for Computing Follow Sets

Computing Follow(T)

- Consider all production rules similar to the following:
 $N = S T U . \quad N = S (T)^* U . \quad N = S (T)? U .$
 - Follow(T) includes First(U).
 - If U can be empty, then Follow(T) also includes Follow(N).
- Consider all production rules similar to the following:
 $N = S T . \quad N = S (T)^* . \quad N = S (T)? .$

In all these cases, Follow(T) includes Follow(N).
- If T occurs in the form $(T)^*$ or $(T)^+$, then Follow(T) includes First(T).

Strategy for Computing Follow Sets

- Use a top-down approach.
- Start with first rule (the one containing the start symbol) and work toward the simpler rules.

Recursive Descent Parsing

Refinement 5

- A syntax factor of the form $(E)^*$ is recognized by the following algorithm:
while current symbol is in $\text{First}(E)$ loop
apply the algorithm for recognizing E
end loop
- **Grammar Restriction 1:** $\text{First}(E)$ and $\text{Follow}((E)^*)$ must be disjoint in this context; i.e.,
$$\text{First}(E) \cap \text{Follow}((E)^*) = \emptyset$$

(Why?)

Example: Recursive Descent Parsing Refinement 5

- Consider the rule for `initialDecls`:

`initialDecls = (initialDecl)* .`

- The CPRL method for parsing `initialDecls` is

```
public void parseInitialDecls()
{
    while (scanner.getSymbol() == Symbol.constRW ||
           scanner.getSymbol() == Symbol.varRW ||
           scanner.getSymbol() == Symbol.typeRW)
    {
        parseInitialDecl();
    }
}
```

- In CPRL, the symbols “const”, “var”, and “type” cannot follow `initialDecls`. (Which symbols can follow?)

Helper Methods in Class Symbol

Class Symbol provides several helper methods for testing properties of symbols.

```
public boolean isReservedWord()  
public boolean isInitialDeclStarter()  
public boolean isSubprogramDeclStarter()  
public boolean isStmtStarter()  
public boolean isLogicalOperator()  
public boolean isRelationalOperator()  
public boolean isAddingOperator()  
public boolean isMultiplyingOperator()  
public boolean isLiteral()  
public boolean isExprStarter()
```

Method isStmtStarter()

```
/**
 * Returns true if this symbol can start a statement.
 */
public boolean isStmtStarter()
{
    return this == Symbol.exitRW
        || this == Symbol.identifier
        || this == Symbol.ifRW
        || this == Symbol.loopRW
        || this == Symbol.whileRW
        || this == Symbol.readRW
        || this == Symbol.writeRW
        || this == Symbol.writelnRW
        || this == Symbol.returnRW;
}
```

Method isInitialDeclStarter()

```
/**
 * Returns true if this symbol can start an initial declaration.
 */
public boolean isInitialDeclStarter()
{
    return this == Symbol.constRW
        || this == Symbol.varRW
        || this == Symbol.typeRW;
}
```

Using Helper Methods in Class Symbol

Using the helper methods in class `Symbol`, we can rewrite the code for `parseInitialDecls()` as follows:

```
public void parseInitialDecls()
{
    while (scanner.getSymbol().isInitialDeclStarter())
        parseInitialDecl();
}
```

Recursive Descent Parsing

Refinement 6

- Since a syntax factor of the form $(E)^+$ is equivalent to $E (E)^*$, a syntax factor of the form $(E)^+$ is recognized by the following algorithm:

apply the algorithm for recognizing E
while current symbol is in First(E) loop
apply the algorithm for recognizing E
end loop

- Equivalently, the algorithm for recognizing $(E)^+$ can be written using a loop that tests at the bottom.

loop
apply the algorithm for recognizing E
*exit when current symbol is **not** in First(E)*
end loop

Recursive Descent Parsing

Refinement 6 (continued)

- In Java, the loop structure that tests at the bottom is called a do-while loop, so the algorithm implemented in Java would more closely resemble the following:
do
apply the algorithm for recognizing E
while current symbol is in First(E)
- **Grammar Restriction 2:** If E can generate the empty string, then $\text{First}(E)$ and $\text{Follow}((E)^+)$ must be disjoint in this context; i.e., $\text{First}(E) \cap \text{Follow}((E)^+) = \emptyset$ (Why?)

Recursive Descent Parsing Refinement 7

- A syntax factor of the form $(E)?$ is recognized by the following algorithm:
if current symbol is in $\text{First}(E)$ then
apply the algorithm for recognizing E
end if
- **Grammar Restriction 3:** $\text{First}(E)$ and $\text{Follow}((E)?)$ must be disjoint in this context; i.e.,
$$\text{First}(E) \cap \text{Follow}((E)?) = \emptyset$$

(Same reason as before.)

Helper Method matchCurrentSymbol()

- Method `matchCurrentSymbol()` is similar to method `match()` except that it takes no parameters and doesn't throw an exception. It simply advances the scanner.
- Method `matchCurrentSymbol` is used when we already know that the next symbol in the input stream is the one we want. We could use `match()` for this purpose, but `matchCurrentSymbol()` is slightly more efficient.
- Method `matchCurrentSymbol()`

```
private void matchCurrentSymbol() throws IOException
{
    scanner.advance();
}
```

Example: Recursive Descent Parsing Refinement 7

- Consider the rule for an exit statement:

`exitStmt = "exit" ("when" booleanExpr)? ";" .`

- The method for parsing an exit statement is

```
public void parseExitStmt()
```

```
{
```

```
    match(Symbol.exitRW);
```

```
    if (scanner.getSymbol() == Symbol.whenRW)
```

```
    {
```

```
        matchCurrentSymbol();
```

```
        parseExpression();
```

```
    }
```

```
    match(Symbol.semicolon);
```

```
}
```

first check for
optional when
clause

slightly more efficient than
`match(Symbol.whenRW)`

Example: Recursive Descent Parsing Refinement 7 (continued)

- The first set for the optional when clause is simply { “when” }, so we use the reserved word when to tell us whether or not to parse a when clause.
- Questions: What is the follow set for the optional when clause? What problem would we have if it contained the reserved word “when”?

Recursive Descent Parsing

Refinement 8

- A syntax factor of the form $E \mid F$ is recognized by the following algorithm:

if current symbol is in $\text{First}(E)$ then
 apply the algorithm for recognizing E
elseif current symbol is in $\text{First}(F)$ then
 apply the algorithm for recognizing F
else
 parsing error
end if

- **Grammar Restriction 4:** $\text{First}(E)$ and $\text{First}(F)$ must be disjoint in this context; i.e., $\text{First}(E) \cap \text{First}(F) = \emptyset$

Example: Recursive Descent Parsing Refinement 8

- Consider the rule in CPRL for `initialDecl`:
`initialDecl = constDecl | varDecl | arrayTypeDecl .`

- The CPRL method for parsing `initialDecl` is

```
public void parseInitialDecl()
{
    if (scanner.getSymbol() == Symbol.constRW)
        parseConstDecl();
    else if (scanner.getSymbol() == Symbol.varRW)
        parseVarDecl();
    else if (scanner.getSymbol() == Symbol.typeRW)
        parseArrayTypeDecl();
    else
        ...    // throw an InternalErrorException
}
```

(This logic could also be implemented using a switch statement.)

LL(1) Grammars

- If a grammar satisfies the restrictions imposed by the previous parsing rules, then the grammar is called an ***LL(1) grammar***.
- Recursive descent parsing using one symbol lookahead can be used only if the grammar is LL(1).
 - First 'L': read the source file from left to right
 - Second 'L': descend into the parse tree from left to right
 - Number '1': one token lookahead
- Not all grammars are LL(1).
 - e.g., any grammar has left recursion is not LL(1)

LL(1) Grammars

(continued)

- In practice, the syntax of most programming languages can be defined, or at least closely approximated, by an LL(1) grammar.
 - e.g., by using grammar transformations such as eliminating left recursion
- The phrase “recursive descent” refers to the fact that we descend (top-down) the parse tree using recursive method/function calls.

Recursive Decent Parsing

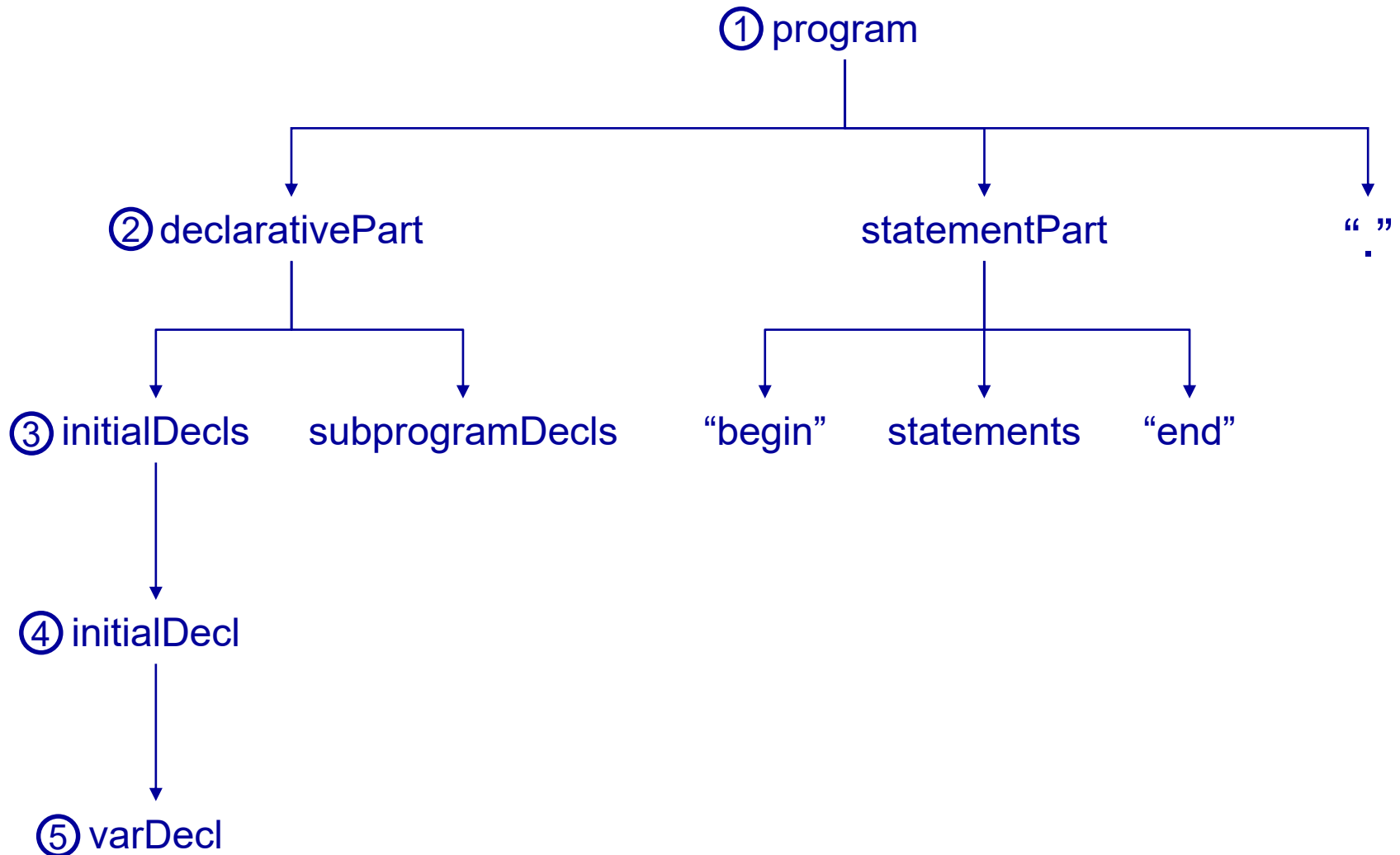
- The “recursive” part of the phrase “recursive descent” comes from the use of recursive method calls in the parser; e.g., to parse nested loop statements.

```
parseLoop()          // called when parsing the outer loop
...
parseStatements()
...
    parseLoop()      // called when parsing the inner loop
```

- For the “descent” part of “recursive descent”, consider a portion of the parse tree for a simple CPRL program.

```
var x : Integer;
begin
    ...
end.
```

Recursive Decent Parsing (continued)



Recursive Decent Parsing

(continued)

- The numbers on the left side of the parse tree correspond to the order of calls to parsing methods; i.e., these are the first five parsing methods called when parsing the program.

```
parseProgram()  
parseDeclarativePart()  
parseInitialDecls()  
parseInitialDecl()  
parseVarDecl()
```

Developing a Parser

Three major versions of the parser for the compiler project:

- Version 1: Language recognition based on a context-free grammar (with minor checking of language constraints)
- Version 2: Add error-recovery
- Version 3: Add generation of abstract syntax trees

Developing a Parser for CPRL

Version 1: Language Recognition

- Use the parsing refinements discussed earlier.
- Verify that the grammar restrictions (in terms of first and follow sets) are satisfied by the grammar for CPRL.
- Use the grammar to develop version 1 of the parser.
 - requires grammar analysis
 - computation of first and follow sets

Variables versus Named Values

- From the perspective of the grammar, there is no real distinction between a variable and a named value.

```
variable = ( varId | paramId ) ( "[" expression "]" )* .  
namedValue = variable .
```

- Both are parsed similarly, but we make a distinction based on the context where the identifier appears.
- For example, consider the assignment statement
 `x := y;`
The identifier “x” represents a variable, and the identifier “y” represents a named value.

Variables versus Named Values

(continued)

- Loosely speaking, it's a variable if it appears on the left side of an assignment statement, and it's a named value if it is used as an expression.
- The distinction between a variable and a named value will become important later when we consider the topics of error recovery and code generation – the error recovery and code generation are different for a variable than for a named value.

Handling Grammar Limitations

- As given, the grammar for CPRL is “not quite” LL(1)
- Example: Parsing a statement.

```
statement = assignmentStmt | ifStmt | loopStmt | exitStmt  
          | readStmt | writeStmt | writelnStmt  
          | procedureCallStmt | returnStmt .
```
- Use the lookahead symbol to select the parsing method.
 - “if” → parse an “if” statement
 - “while” → parse a loop statement
 - “loop” → parse a loop statement
 - identifier → parse either an assignment statement or
procedure call statement (which one?)
- An identifier is in the first set of both an assignment statement and a procedure call statement.

Handling Grammar Limitations (continued)

- A similar problem exists when parsing a factor.

`factor = "not" factor | constValue | namedValue
| functionCall | "(" expression ")" .`

- An identifier is in the first set of `constValue`, `namedValue`, and `functionCall`.

Possible Solutions

1. Use additional token lookahead – LL(2)
 - If the symbol following the identifier is “[” or “:=”, parse an assignment statement.
 - If it is “(” or “;”, parse a procedure call statement.
2. Redesign/factor the grammar; e.g., replace
“s = i x | i y .” with “s = i (x | y) .”
3. Use an identifier table to store information about how the identifier was declared, and then later use the declaration information to determine if the identifier is a constant, variable, procedure name, etc.

We will use the third approach.

Class IdTable

(Version 1)

- We will create a preliminary version class IdTable to help track identifiers that have been declared and to assist with basic constraint analysis of scope rules.
- Types of Identifiers

```
enum IdType
{
    constantId,  variableId,  arrayTypeId,
    procedureId,  functionId;
}
```

Class IdTable will be extended in subsequent assignments to perform a more complete analysis of CPRL scope rules.

Procedure Example – Scope

```
var x : Integer;
```

```
var y : Integer;
```

```
procedure p is
```

```
    var x : Integer;
```

```
    var n : Integer;
```

```
begin
```

```
    x := 5;      // which x?
```

```
    n := y;      // which y?
```

```
end p;
```

```
begin
```

```
    x := 8;      // which x?
```

```
end.
```

Handling Scopes within Class IdTable

- Variables and constants can be declared at the program level or at the subprogram level, introducing the concept of scope.
- Class IdTable will need to search for names both within the current scope and possibly in enclosing scopes.
- Class IdTable is implemented as a stack of maps from identifier strings to their IdType.
 - When a new scope is opened, a new map is pushed onto the stack.
 - Searching for a declaration involves searching within the current level (top map in the stack) and then within enclosing scopes (maps under the top).

Selected Methods in IdTable

```
/**
 * Opens a new scope for identifiers.
 */
public void openScope()

/**
 * Closes the outermost scope.
 */
public void closeScope()

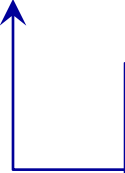
/**
 * Add a token and its type at the current scope level.
 * @throws ParseException if the identifier token is already
 *                        defined in the current scope.
 */
public void add(Token idToken, IdType idType)
    throws ParseException
```

Selected Methods in IdTable (continued)

```
/**  
 * Returns the id type associated with the identifier.  
 * Returns null if the identifier is not found. Searches  
 * enclosing scopes if necessary.  
 */  
public IdType get(Token idToken)
```

Adding Declarations to IdTable

- When an identifier is declared, the parser will attempt to add its token and IdType to the table within the current scope.
 - throws an exception if an identifier with the same name (same token text) has been previously declared in the current scope.
- Example from method `parseConstDecl()`
`idTable.add(constId, IdType.constantId);`



Throws a `ParserException` if the identifier is already defined in the current scope

Using IdTable to Check Applied Occurrences of Identifiers

When an identifier is encountered in the statement part of the program or a subprogram, the parser will

- check that the identifier has been declared
- use the information about how the identifier was declared to facilitate correct parsing (e.g., you can't assign a value to an identifier that was declared as a constant.)

Example Using IdTable to Check Applied Occurrences of Identifiers

```
// in method parseFactor()
else if (scanner.getSymbol() == Symbol.identifier)
{
    // Handle identifiers based on whether they are
    // declared as variables, constants, or functions.
    Token idToken = scanner.getToken();
    IdType idType = idTable.get(idToken);

    if (idType != null)
    {
        if (idType == IdType.constantId)
            parseConstValue();
        else if (idType == IdType.variableId)
            parseNamedValue();
        else if (idType == IdType.functionId)
            parseFunctionCall();
    }
}
```

Example Using IdTable to Check Applied Occurrences of Identifiers (continued)

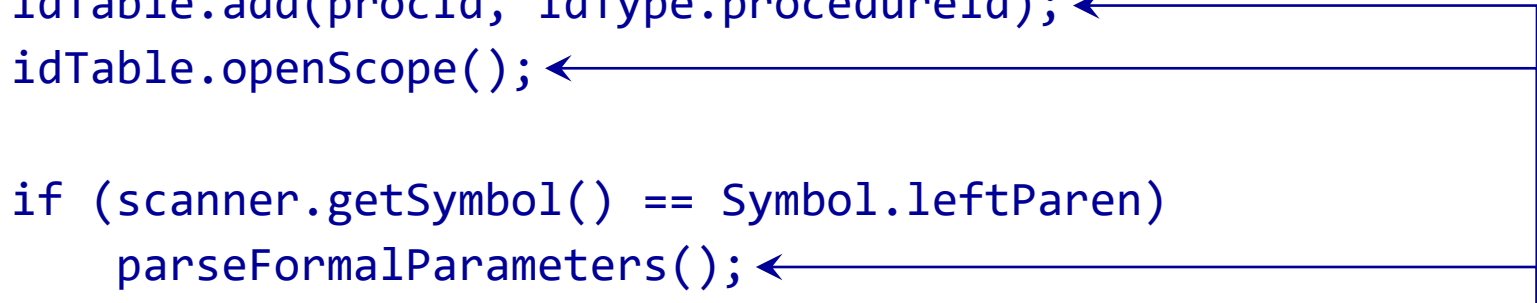
```
        else
            throw error("Identifier \"" + scanner.getToken()
                        + "\" cannot be part of an expression.");
    }
    else
        throw error("Identifier \"" + scanner.getToken()
                    + "\" has not been declared.");
}
```

Example: Parsing a Procedure Declaration

```
// procedureDecl = "procedure" procId ( formalParameters )?
//           "is" initialDecls statementPart procId ";" .
match(Symbol.procedureRW);
Token procId = scanner.getToken();
match(Symbol.identifier);
idTable.add(procId, IdType.procedureId); ←
idTable.openScope(); ←

if (scanner.getSymbol() == Symbol.leftParen)
    parseFormalParameters(); ←

match(Symbol.isRW);
parseInitialDecls();
parseStatementPart();
idTable.closeScope();
```



Note that the procedure name is defined in the outer (program) scope, but its parameters and initial declarations are defined within the scope of the procedure.

Example: Parsing a Procedure Declaration (continued)

```
Token procId2 = scanner.getToken();  
match(Symbol.identifier);
```

```
if (!procId.getText().equals(procId2.getText()))  
    throw error(procId2.getPosition(),  
                "Procedure name mismatch.");
```

```
match(Symbol.semicolon);
```

Note the check that the procedure names (procId and procId2) match. Technically, ensuring that the procedure names match goes beyond simple syntax analysis and represents more of a constraint check. As far as the context-free grammar is concerned, they are both just identifiers.

Example: Parsing a Statement

```
Symbol symbol = scanner.getSymbol();
if (symbol == Symbol.identifier)
{
    IdType idType = idTable.get(scanner.getToken());
    if (idType != null)
    {
        if (idType == IdType.variableId)
            parseAssignmentStmt();
        else if (idType == IdType.procedureId)
            parseProcedureCallStmt();
        else
            throw error(...);
    }
    else
        throw error(...);
}
```

(continued on next slide)

Example: Parsing a Statement (continued)

```
else if (symbol == Symbol.ifRW)
    parseIfStmt();
else if (symbol == Symbol.loopRW || symbol == Symbol.whileRW)
    parseLoopStmt();
else if (symbol == Symbol.exitRW)
    parseExitStmt();
...
```

Class ErrorHandler

- Used for consistency in error reporting.
- Implements the singleton pattern (only one instance)
- Obtain an instance of ErrorHandler by calling `ErrorHandler.getInstance()`

Key Methods Class ErrorHandler

```
/**
 * Reports the error. Stops compilation if the maximum
 * number of errors have been reported.
 */
public void reportError(CompilerException e)

/**
 * Reports the error and exits compilation.
 */
public void reportFatalError(Exception e)

/**
 * Reports a warning and continues compilation.
 */
public void reportWarning(String warningMessage)
```

Using ErrorHandler for Parser Version 1

- Version 1 of the parser does not implement error recovery. When an error is encountered, the parser will print an error message and then exit.
- In order to ease the transition to error recovery in the next version of the parser, most parsing methods will wrap the basic parsing logic in a try/catch block.
- Any parsing method that calls `match()` or the `add()` method of `IdTable` will need to have a try/catch block.
- Error reporting will be implemented within the catch clause of the try/catch block.

Using ErrorHandler for Parser Version 1 (continued)

```
public void parseAssignmentStmt() throws IOException
{
    try
    {
        parseVariable();
        match(Symbol.assign);
        parseExpression();
        match(Symbol.semicolon);
    }
    catch (ParserException e)
    {
        ErrorHandler.getInstance().reportError(e);
        exit();
    }
}
```

} wrap the parsing statements in a try/catch block

This approach provides the framework that we will use for error recovery in the next section.

Implementing methods `parseVariable()` and `parseNamedValue()`

- To implement methods `parseVariable()` and `parseNamedValue()` we use a helper method to provide common logic for both methods.
- The helper method does not handle any parser exceptions but instead throws them back to the calling method where they can be handled appropriately.
- An outline of the helper method, `parseVariableExpr()`, is shown on the next couple of slides.
- Both `parseVariable()` and `parseNamedValue()` call the helper method to parse the grammar rule for variable.

Method parseVariableExpr()

```
// parses the following grammar rule:
// variable = ( varId | paramId ) ( "[" expression "]" )* .
public void parseVariableExpr()
    throws IOException, ParseException
{
    Token idToken = scanner.getToken();
    match(Symbol.identifier);
    IdType idType = idTable.get(idToken);

    if (idType == null)
        throw error("...");    // has not been declared
    else if (idType != IdType.variableId)
        throw error(" ... ");  // not a variable
```

(continued on next slide)

Method parseVariableExpr() (continued)

```
while (scanner.getSymbol() == Symbol.leftBracket)
{
    matchCurrentSymbol();
    parseExpression();
    match(Symbol.rightBracket);
}
}
```

Method parseVariable()

- Method parseVariable() simply calls the helper method to parse its grammar rule.

```
public void parseVariable() throws IOException
{
    try
    {
        parseVariableExpr();
    }
    catch (ParserException e)
    {
        ErrorHandler.getInstance().reportError(e);
        exit();
    }
}
```

- Method parseNamedValue() is implemented similarly.