Abstract Syntax Trees

 Slide 1

1

Abstract Syntax Trees

- We will modify our parser one more time so that, as it parses the source code, it will also generate an intermediate representation of the program known as abstract syntax trees.
- An abstract syntax tree is similar to a parse tree but without extraneous nonterminal and terminal symbols.
- Abstract syntax trees provide an explicit representation of the structure of the source code that can be used for
  - additional constraint analysis (e.g., for type constraints)
  - some optimization (tree transformations)
  - code generation

 Slide 2

2

Representing Abstract Syntax Trees

- We will use different classes to represent different node types in our abstract syntax trees. Examples include
  - `Program`
  - `ProcedureDecl`
  - `AssignmentStmt`
  - `LoopStmt`
  - `Variable`
  - `Expression`
- Each AST class has named instance variables (fields) to reference its children. These instance variables provide the “tree” structure.
- Occasionally we also include additional fields to support error handling (e.g., position) and code generation.

Terence Parr refers to this type of AST structure as an irregular (named child fields) heterogeneous (different node types) AST.

 Slide 3

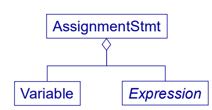
3

Abstract Syntax Trees: Example 1

- Consider the grammar for an assignment statement.

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

- The important parts of an assignment statement are
  - variable (the left side of the assignment)
  - expression (the right side of the assignment)
- We create an AST node for an assignment statement with the following structure:



 Slide 4

4

Class `AssignmentStmt`

```
class AssignmentStmt(private val variable : Variable,
                     private val expr : Expression,
                     private val assignPosition : Position)
    : Statement()
  {
    ...
  }
```

position of assignment operator (for error reporting)

 Slide 5

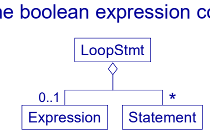
5

Abstract Syntax Trees: Example 2

- Consider the following grammar for a loop statement:

```
loopStmt = ( "while" booleanExpr )?
           "loop" statements "end" "loop" ";" .
```

- Once a loop statement has been parsed, we don’t need to retain the nonterminal symbols. The AST for a loop statement would contain only the statements in the body of the loop and the optional boolean expression (e.g., the reference to the boolean expression could be null).



 Slide 6

6

Class `LoopStmt`

```
class LoopStmt : Statement()
  {
    var whileExpr  : Expression? = null
    var statements : List<Statement> = emptyList()

    ...
  }
```

Note that `whileExpr` can be null.

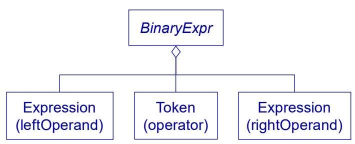
7

Abstract Syntax Trees: Example 3

- For binary expressions, part of the grammar exists simply to define operator precedence.
- Once an expression has been parsed, we do not need to preserve additional information about nonterminals that were introduced to define precedence (`relation`, `simpleExpr`, `term`, `factor`, etc.).
- A binary expression AST would contain only the operator and the left and right operands. The parsing algorithm would build the AST so as to preserve operator precedence.

8

Abstract Syntax Trees: Example 3
(continued)



9

Class `BinaryExpr`

```
abstract class BinaryExpr(val leftOperand  : Expression,
                          val operator     : Token,
                          val rightOperand : Expression)
    : Expression(operator.position)
```


10

Structure of Abstract Syntax Trees

- There is an abstract class `AST` that serves as the superclass for all other abstract syntax tree classes.
- Class `AST` contains implementations of methods common to all subclasses plus declarations of abstract methods required by all concrete subclasses.
- All AST classes will be defined in an "…`ast`" subpackage.

Note the use of `AST` (in monospaced font) for the specific class and AST (in normal font) as an abbreviation for "abstract syntax tree".

11

Outline of Class `AST`

```
abstract class AST
  {
    ...

    /** Check semantic/contextual constraints. */
    abstract fun checkConstraints()

    /** Emit the object code for the AST. */
    abstract fun emit()
  }
```

Methods `checkConstraints()` and `emit()` provide a mechanism to "walk" the tree structure using recursive calls to subordinate tree nodes.

12

Subclasses of AST

- We will create a hierarchy of classes, some of which are abstract, that are all direct or indirect subclasses of `AST`.
- Each node in the abstract syntax tree constructed by the parser will be an object of a class in the `AST` hierarchy.
- Most classes in the hierarchy will correspond to and have names similar to the nonterminal symbols in the grammar, but not all abstract syntax trees have this property. See, for example, the earlier discussion about binary expressions. We do not need abstract syntax tree classes corresponding to nonterminals `simpleExpr`, `term`, `factor`, etc.

 Slide 13

13

Using Collection Classes

- Some parsing methods simply return lists of AST objects.
- Examples

```
fun parseInitialDecls()      : List<InitialDecl>
fun parseSubprogramDecls()   : List<SubprogramDecl>
fun parseIdentifiers()       : List<Token>
fun parseStatements()        : List<Statement>
fun parseFormalParameters() : List<ParameterDecl>
fun parseExpressions()       : List<Expression>
fun parseActualParameters() : List<Expression>
```

 Slide 14

14

Naming Conventions for AST

- Most AST classes have names similar to nonterminals in the grammar.
  - `Program`
  - `AssignmentStmt`
  - `FunctionDecl`
  - `LoopStmt`
- The parsing method for that nonterminal will create the corresponding AST object.
  - `parseProgram` returns a `Program` object
  - `parseLoopStmt` returns a `LoopStmt` object
  - etc.
- Parsing methods with plural names will return lists of AST objects.
  - the grammar was written to have this property.

 Slide 15

15

Naming Conventions for AST (continued)

- Example

```
abstract class Statement : AST() ...
class LoopStmt : Statement() ...
```

- The parsing method `parseLoopStmt()` would be responsible for creating the AST node for `LoopStmt`. Instead of returning `void`, method `parseLoopStmt()` will return an object of class `LoopStmt`.
- Similarly, the parsing method `parseStatements()` will return a list of `Statement` objects, where each `Statement` object is either an `AssignmentStmt`, a `LoopStmt`, an `IfStmt`, etc.

 Slide 16

16

Method `parseLiteral()`

- Method `parseLiteral()` is a special case.
- Since literals are tokens returned from the scanner, method `parseLiteral()` simply returns a `Token`. There is no AST class named `Literal`.
- Relevant Grammar Rules

```
literal = intLiteral | charLiteral | stringLiteral
        | booleanLiteral .
booleanLiteral = "true" | "false" .
```

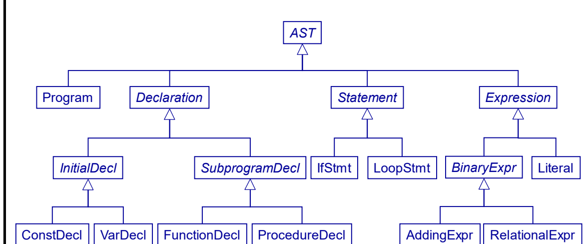
- Method

```
fun parseLiteral() : Token?
// returns null if parsing fails
```

 Slide 17

17

Partial AST Inheritance Diagram for the Language CPRL



(names for abstract classes are shown in italics)

 Slide 18

18

Language Constraints Associated With Identifiers

- A parser built using only the set of parsing rules will not reject programs that violate certain language constraints such as "an identifier must be declared exactly once".
- Examples: Valid syntax but not valid with respect to contextual constraints

```
var x : Integer;
begin
   y := 5;
end.
```

```
var c : Char;
begin
   c := -3;
end.
```


19

Class `IdTable`

- We will extend class `IdTable` to help track not only of the types of identifiers that have been declared, but also of their declarations.
- Class `Declaration` is part of the AST hierarchy. A declaration object contains a reference to the identifier token and information about its type. We will use different subclasses of `Declaration` for kinds of declarations; e.g., `ConstDecl`, `VarDecl`, `ProcedureDecl`, etc.

20

A Property and Selected Methods in the Modified Version of `IdTable`

```
/**
 * The current scope level.
 */
val scopeLevel : ScopeLevel

/**
 * Opens a new scope for identifiers.
 */
fun openScope()

/**
 * Closes the outermost scope.
 */
fun closeScope()
```

`ScopeLevel` is an enum class with only two values, `PROGRAM` and `SUBPROGRAM`.

21

A Property and Selected Methods in the Modified Version of `IdTable` (continued)

```
/**
 * Add a declaration at the current scope level.
 * @throws ParserException if the identifier token associated
 *                         with the declaration is already
 *                         defined in the current scope.
 */
fun add(decl : Declaration)

/**
 * Returns the Declaration associated with the identifier
 * token's text.  Returns null if the identifier is not found.
 * Searches enclosing scopes if necessary.
 */
operator fun get(idToken : Token) : Declaration?
```


22

Adding Declarations to `IdTable`

- When an identifier is declared, the parser will attempt to add the declaration to the table within the current scope. (The declaration already contains the identifier token.)
  - throws an exception if a declaration with the same name (same token text) has been previously declared in the current scope.
- Example (in method parseConstDecl())

```
val constId = scanner.token
...
val constDecl = ConstDecl(constId, constType, literal)
idTable.add(constDecl)
```

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

23

Interface `NamedDecl`

- Identifiers declared using `VarDecl` (which we convert to a list of `SingleVarDecl` as described later) or `ParameterDecl` have similar uses within CPRL; e.g.,
  `x := y;`
- Variable x could have been declared in a variable declaration or a parameter declaration.
  - similarly for the named value y
- There is a need to treat both types of declarations uniformly at several points during parsing, which we achieve by creating interface `NamedDecl` and specifying that `SingleVarDecl` and `ParameterDecl` implement this interface.

24

Interface `NamedDecl` (continued)

- Four properties in interface `NamedDecl`

```
val type : Type
val size : Int
val scopeLevel : ScopeLevel
var relAddr : Int
```


25

Example: Using Interface `NamedDecl`

```
// excerpt from parseStatement()

when (scanner.symbol)
  {
    Symbol.identifier ->
      {
        val decl = idTable[scanner.token]

        if (decl != null)
          {
            if (decl is NamedDecl)
                stmt = parseAssignmentStmt()
            ...
          }
      }
    ...
```


26

Using `IdTable` to Check Applied Occurrences of Identifiers

- When an identifier is encountered in the statement part of the program or a subprogram (e.g., as part of an expression or subprogram call), 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.)

27

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

- Example (in method `parseVariableExpr()`)

```
val idToken = scanner.token
match(Symbol.identifier)
val decl = idTable[idToken]

if (decl == null)
    throw error("Identifier \"$idToken\" has not "
              + "been declared.")
else if (decl !is NamedDecl)
    throw error("Identifier \"$idToken\" is not "
              + "a variable.")
```


28

Types in CPRL

- The compiler uses two classes to provide support for CPRL types.
- Class `Type` encapsulates the language types and their sizes.
  - Predefined types are declared as constants in the companion object.
  - Class `Type` also contains a method that returns the type of a literal symbol.

```
fun getTypeOf(literal : Symbol): Type
```

- Class `ArrayType` extends `Type` to provide additional support for arrays.

29

Class Type

- Class Type encapsulates the language types and sizes (number of bytes) for the programming language CPRL.
- Type sizes are initialized to values appropriate for the CPRL virtual machine.
  - 4 for Integer
  - 2 for Character
  - 1 for Boolean
  - etc.
- Predefined types are declared in the companion object.

```
val Boolean = Type("Boolean", Constants.BYTES_PER_BOOLEAN)
val Integer = Type("Integer", Constants.BYTES_PER_INTEGER)
val Char    = Type("Char",    Constants.BYTES_PER_CHAR)
val String  = Type("String")
val Address = Type("Address", Constants.BYTES_PER_ADDRESS)
val UNKNOWN = Type("UNKNOWN")
```


30

Class `ArrayType`

- Class `ArrayType` extends class `Type`.
  - therefore array types are also types
- In addition to the total size of the array, class `ArrayType` also keeps track of the number and type of elements.

```
class ArrayType(typeName : String, val numElements : Int,
                val elementType :  Type)
     : Type(typeName, numElements*elementType.size)
```

- When the parser parses an array type declaration, the constructor for AST class `ArrayTypeDecl` creates an `ArrayType` object.

31

Example: Parsing a `ConstDecl`

```
/**
 * Parse the following grammar rule:
 * `constDecl = "const" constId ":=" literal ";" .`
 *
 * @return the parsed constant declaration.
 *         Returns null if parsing fails.
 */
fun parseConstDecl() : ConstDecl?
  {
    try
      {
        match(Symbol.constRW)
        val constId = scanner.token
        match(Symbol.identifier)
```

(continued on next slide)

32

Example: Parsing a `ConstDecl` (continued)

```
        match(Symbol.assign)
        val literal = parseLiteral()
        match(Symbol.semicolon)

        var constType = Type.UNKNOWN
        if (literal != null)
            constType = Type.getTypeOf(literal.symbol)

        val constDecl = ConstDecl(constId, constType, literal)
        idTable.add(constDecl)
        return constDecl
      }
```

(continued on next slide)

33

Example: Parsing a `ConstDecl` (continued)

```
    catch (e : ParserException)
      {
        ErrorHandler.reportError(e)
        recover(initialDeclFollowers)
        return null
      }
  }
```


34

The Scope Level of a Variable Declaration

- During code generation, when a variable or named value is referenced in the statement part of a program or subprogram, we need to be able to determine where the variable was declared.
- Class `IdTable` contains a property named `scopeLevel` that returns the block nesting level for the current scope.
  - `PROGRAM` for objects declared at the outermost (program) scope.
  - `SUBPROGRAM` for objects declared within a subprogram.
- When a variable is **declared**, the declaration is initialized with the current level.

```
val varDecl = VarDecl(identifiers, varType, idTable.scopeLevel)
```


35

Example: Scope Levels

```
var x : Integer;   // scope level of declaration is PROGRAM
var y : Integer;   // scope level of declaration is PROGRAM

procedure p is       // scope level of declaration is PROGRAM
   var x : Integer;  // scope level of declaration is SUBPROGRAM
   var b : Integer;  // scope level of declaration is SUBPROGRAM
begin
   ... x ...    // x was declared at SUBPROGRAM scope
   ... b ...    // b was declared at SUBPROGRAM scope
   ... y ...    // y was declared at PROGRAM scope
end p;

begin
    ... x ...      // x was declared at PROGRAM scope
    ... y ...      // y was declared at PROGRAM scope
    ... p ...      // p was declared at PROGRAM scope
end.
```


36

`VarDecl` versus `SingleVarDecl`

- A variable declaration can declare several identifiers all with the same type, as in

```
var x, y, z : Integer;
```

- This declaration is logically equivalent to declaring each variable separately, as in

```
var x : Integer;
var y : Integer;
var z : Integer;
```

- To simplify constraint checking and code generation, within the AST we will view a variable declaration as a collection of single variable declarations.

37

Class `SingleVarDecl`

```
class SingleVarDecl(identifier : Token, varType : Type,
                    override val scopeLevel : ScopeLevel)
    : InitialDecl(identifier, varType), NamedDecl
  {
    ...
  }
```


38

Class `VarDecl`

```
class VarDecl(identifiers : List<Token>, varType : Type,
              scopeLevel : ScopeLevel)
    : InitialDecl(Token(), varType)
  {
    // the list of single var decls for the variable declaration
    val singleVarDecls =
                    ArrayList<SingleVarDecl>(identifiers.size)

    init
      {
        for (id in identifiers)
            singleVarDecls.add(SingleVarDecl(id, varType,
                                             scopeLevel))
      }
  }
```

A `VarDecl` is simply a list of `SingleVarDecls`.

39

Method `parseInitialDecls()`

- Method `parseInitialDecls()` constructs/returns a list of initial declarations.
- For constant and array type declarations, this method simply adds them to the list.
- For variable declarations (`VarDecls`), this method extracts the list of single variable declarations (`SingleVarDecls`) and adds them to the list. The original `VarDecl` is no longer used after this point.

40

Method `parseInitialDecls()` (continued)

```
...

val decl = parseInitialDecl()

if (decl is VarDecl)
  {
    // add the single variable declarations
    for (singleVarDecl in decl.singleVarDecls)
        initialDecls.add(singleVarDecl)
  }
else if (decl != null)
    initialDecls.add(decl)
```


41

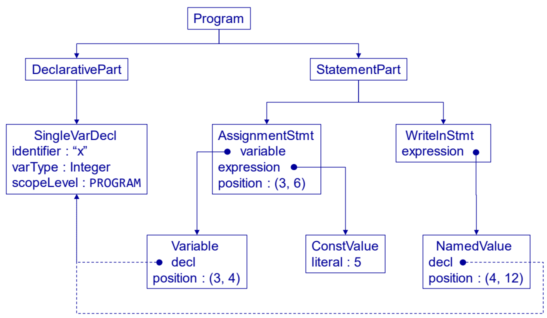
Example: Abstract Syntax Tree

```
var x : Integer;
begin
   x := 5;
   writeln x;
end.
```

(AST for this example is on the next slide.)

42

Example: Abstract Syntax Tree (continued)



43

Determining Types of Expressions

- Since CPRL is statically typed, it is possible to determine the type of every expression at compile time, and AST class Expression has a property for the expression type that is inherited by all expression subclasses.
- Where within the compiler should type determination take place? In general, we will determine the type of an expression in the constructor for the expression’s AST class.

44

Example: `RelationalExpr`

- A relational expression is a binary expression where the operator is a relational operator such as “<=” or “>”.
- Regardless of the types of the two operands, a relational expression always has type `Boolean`.

45

Example: `RelationalExpr` (continued)

- Constructor for `RelationalExpr`

```
class RelationalExpr(leftOperand  : Expression,
                     operator     : Token,
                     rightOperand : Expression)
    : BinaryExpr(leftOperand, operator, rightOperand)
  {
    /**
     * Initialize the type of the expression to Boolean.
     */
    init
      {
        type = Type.Boolean
      }
   ...
  }
```


46

Example: `AddingExpr`

- For most “real” programming languages, determining the type of an adding expression can be somewhat complicated.
  – C and Java have multiple numeric types with rules about automatic conversions (coercions) when an operator has different operand types.
- In CPRL, an adding expression always has type Integer. (Similarly for a multiplying expression in CPRL.)

47

Example: `AddingExpr` (continued)

- Constructor for `AddingExpr`

```
class AddingExpr(leftOperand  : Expression,
                 operator     : Token,
                 rightOperand : Expression)
    : BinaryExpr(leftOperand, operator, rightOperand)
  {
    /**
     * Initialize the type of the expression to Boolean.
     */
    init
      {
        type = Type.Boolean
      }
   ...
  }
```


48

Example: `Variable`

- The type for a variable (and therefore also for a named value) is initialized to the type specified in the variable's declaration.
- Constructor for Variable

```
open class Variable(val decl : NamedDecl,
                    position : Position,
                    var indexExprs : List<Expression>)
    : Expression(decl.type, position)
```


Example: `Variable` (continued)

- The initialized type for a variable is correct for predefined types such as `Integer` or `Char`, but additional work is required for arrays.
- Consider the following declarations:

```
type T1 is array(10) of Integer;
type T2 is array(10) of T1;
var a, b : T2;
```

- While the declared (initialized) type of both a and b is `T2`, we could have a variable or named value with zero, one, or two index expressions, as in the following:

```
a := b;             // type of var and named val is T2
a[0] := b[0];       // type of var and named val is T1
a[1][6] := b[5][7]; // type of var and named val is Integer
```


Example: `Variable` (continued)

- For arrays, we determine the actual type of a variable or named value in method `checkConstraints()`.

```
for (expr in indexExprs)
  {
    expr.checkConstraints()

    if (expr.type != Type.Integer)
        throw error(...)

    if (type is ArrayType)
      {
        val arrayType as ArrayType
        type = arrayType.elementType
      }
    else
        throw error(...);
  }
```


Maintaining Context During Parsing

- Certain CPRL statements need access to an enclosing context for constraint checking and code generation.
- Example: `exit when n > 10;`
  An exit statement has meaning only when nested inside a loop., and code generation for an exit statement requires knowledge of which loop encloses it.
- Similarly, a return statement needs to know which subprogram it is returning from.
- Classes `LoopContext` and `SubprogramContext` will be used to maintain contextual information in these cases.

Class `LoopContext`

```
/**
 * The loop statement currently being parsed;
 * null if not currently parsing a loop.
 */
val loopStmt : LoopStmt?

/**
 * Called when starting to parse a loop statement.
 */
fun beginLoop(stmt : LoopStmt)

/**
 * Called when finished parsing a loop statement.
 */
fun endLoop()
```


Class `SubprogramContext`

```
/**
 * The subprogram declaration currently being parsed;
 * null if not currently parsing a subprogram.
 */
var subprogramDecl : SubprogramDecl? = null

/**
 * Called when starting to parse a subprogram declaration.
 */
fun beginSubprogramDecl(subprogDecl : SubprogramDecl)

/**
 * Called when finished parsing a subprogram declaration.
 */
fun endSubprogramDecl()
```

Example: Using Context During Parsing

- When parsing a loop statement:

```
val stmt = LoopStmt()
...
loopContext.beginLoop(stmt)
stmt.statements = parseStatements()
loopContext.endLoop()
```

- When parsing an exit statement:

```
val loopStmt = loopContext.loopStmt ?: throw
        error(exitPosition,
              "Exit statement is not nested within a loop.")
match(Symbol.semicolon)
return ExitStmt(whenExpr, loopStmt)
```

 Slide 55

55

Version 3 of the Parser
(Abstract Syntax Trees)

- Create AST classes in package "…`ast`"
- Add generation of AST structure; i.e., parsing methods should return AST objects or lists of AST objects.
- Use empty bodies when overriding abstract methods `checkConstraints()` and `emit()`.
- Use complete version of `IdTable` to check for scope errors.
- Use class `Context` to check exit and return statements.

At this point your compiler should accept all legal programs and reject most illegal programs. Some programs with type or other miscellaneous errors will not yet be rejected.

 Slide 56

56