There is a level of correctness that is deeper than the grammar

```
fie(a,b,c,d)
   int a, b, c, d;
   { ... }
fee()
   { int f[3], g[4], h, i, j, k;
      char *p;
      call fie(h, i, “ab”, j, k);
      k = f * i + j;
      h = g[17];
      printf(“%s,%s\n”,p,q);
      p = 10;
   }
```

What is wrong with this program?
- declared g[4], used g[17]
- wrong number of args to fie()
- “ab” is not an int
- wrong dimension on use of f
- undeclared variable q
- 10 is not a character string

All of these are “deeper than syntax”
Beyond Syntax

To generate code, the compiler needs to answer many questions

• Is the identifier “x” a scalar, an array, or a function? Is “x” declared?
• Are there names that are not declared? Declared but not used?
• Which declaration of “x” does each use reference?
• Is the expression “x * y + z” type-consistent?
• In “a[i,j,k]”, does “a” have three dimensions?
• Where can variable “z” be stored? (register, local, global, heap, static)
• How many arguments does “fie()” take?
• Does “*p” reference the result of a “malloc()”?  
• Do “p” and “q” refer to the same memory location?
• Is “x” defined before it is used?

These questions are beyond what can be specified using a Context Free Grammar

Beyond Syntax

These questions are part of context-sensitive analysis

• Answers depend on values, not just tokens
  – answers depend on attributes of tokens
• Questions and answers involve non-local information
  – variable declarations, procedures
• Answers may involve computation

How can we answer these questions?

• Use formal methods
  – Context-sensitive grammars
  – Attribute grammars (semantic rules do not have side effects)
• Use ad-hoc techniques
  – Symbol tables
  – Ad-hoc code (use semantic rules that can have side effects)
Context-Sensitive Analysis

- Properties we want to check involve non-local information
  - type-checking, variable declarations, procedure declarations
- Can be implemented as a traversal on the abstract syntax tree
- Can be integrated to parsing phase using ad-hoc translation schemes
- Formalisms such as attribute grammars can be used
- Ad-hoc techniques are more common than the formal techniques

Context Sensitive Analysis

- One way of implementing context sensitive analysis is to first construct an intermediate representation of the program
  - While constructing the intermediate representation we remove the syntax related details since we do not need them once the parsing is done
- After we construct the intermediate representation, we can check context sensitive properties by traversing the intermediate representation
Intermediate Representations: Overview

- There is more than one way to represent code as it is being generated, analyzed, and optimized.

**Graphical IRs**
- Abstract Syntax Trees (AST)
- Directed Acyclic Graphs (DAG)
- Control Flow Graphs (CFG)
- Static Single Assignment Form (SSA)

**Linear IRs**
- Stack Machine Code
- Three Address Code

Why Use an Intermediate Representation?

- It might be possible to write an entire compiler that fits within the semantic action phrases of a Yacc / Bison parser (i.e., syntax-driven translation, e.g., like in a simple calculator interpreter). However, such a compiler is difficult to read and maintain and forces the compiler to analyze the program in exactly the order it is parsed.

- To improve modularity, it is better to separate issues of syntax (parsing) from issues of semantics (type-checking and translation to target code).

- So, we generate an intermediate representation during parsing (an abstract syntax tree) and then do the context sensitive analyses by traversing this intermediate representation.
Abstract Syntax Trees: Overview

- A compiler must do more than to recognize whether a sentence belongs to the language of a grammar (this corresponds to checking the syntax and this is what we have discussed so far).
- A compiler is also responsible for context-sensitive analysis (semantic analysis such as type checking), intermediate code generation, optimization, and target code generation.
- Typically the first thing a compiler does after checking the syntax is construct an intermediate representation (such as an abstract syntax tree).
- The notion of abstract syntax is due to John McCarthy, 1963, who designed the abstract syntax for Lisp (no need to deal with all those parenthesis in the abstract syntax).

Difference between Parse Trees and ASTs

- A parse tree (aka the concrete syntax tree) represents the concrete syntax of the source languages. It contains many punctuation tokens that are redundant, may have extra non-terminal symbols and extra productions for factoring, elimination of left recursion, elimination of ambiguity. In the parse tree each leaf corresponds to each token of the input, and for each grammar production rule used during the parse there is one internal node that is labeled with the nonterminal symbol on the left hand side of the production.
- An abstract syntax tree eliminates a lot of this redundant (but needed by the parser) information from the parse tree (it removes punctuation tokens for example). An abstract syntax makes a clean interface between the parser and later phases of a compiler, and takes the form of the abstract syntax tree. In the abstract syntax tree, the nodes of the tree do not have to have a one-to-one correspondence with the symbols in the concrete syntax of the language.
Example: Abstract versus Concrete Syntax

- **Example**: Abstract syntax tree - the parser uses the concrete syntax to generate a simplified abstract tree corresponding to the ambiguous context free grammar (ambiguous grammar is not good for a parser), but good enough for further phases of compilation.

### Example concrete syntax

```
S → Expr
Expr → Term Expr'
Expr' → + Term Expr'
  | − Term Expr'
  | ε
Term → Factor Term'
Term' → * Factor Term'
  | / Factor Term'
  | ε
Factor → num
  | id
  | ( Expr )
```

### Example abstract syntax

```
S → Expr
Expr → Expr + Expr
  | Expr − Expr
  | Expr * Expr
  | Expr / Expr
  | id
  | num
```

Much simpler, no punctuation is required, just enough to describe the structure of a program

---

Abstract Syntax Trees (ASTs)

What exactly is an Abstract Syntax Tree (AST) in practice?
- It is basically a data structure that is used to represent the input program
  - All the information that we need to analyze the program and to translate the program to the target language is available in the AST
  - The syntactic details about the input are not kept in the AST since we do not need them after the parsing is over

- AST is a tree shaped data structure corresponding to the recursive nature of the abstract syntax of the program
Abstract Syntax Trees (ASTs)

\[
\text{if} \ (x < y) \\
\quad x = 5*y + 5*y/3; \\
\text{else} \\
\quad y = 5; \\
\quad x = x+y;
\]

How to Build ASTs?

- Instead of using a formal approach like attribute grammars (see next lecture)
  - We will use ad-hoc techniques (semantic actions that can have side effects)

**Ad-hoc syntax-directed translation**

- Associate a snippet of code (could be any C/C++ code for yacc/bison) with each production
- Evaluation method: At each reduction for that production, the corresponding snippet of code is executed
- Evaluation method fits nicely into LR(1) parsing algorithm
Building ASTs with Syntax-Directed Translation

- Allowing arbitrary code provides complete flexibility
  - But gives ability to do tasteless and bad things too!

To make this approach work
- Need names for attributes of each symbol on lhs & rhs
  - Typically, one attribute passed through parser + arbitrary code (structures, globals, statics, …)
  - Yacc uses $$, $1, $2, … $n, to denote the values of the symbols in a production (from left to right)

\[
\text{Expr} := \text{Expr} + \text{Term} \{ $$ = $1 + $3; \}
\]

$$ denotes the value for the lhs,
$1 denotes the value for the 1st symbol on the rhs,
$2 denotes the value for the 2nd symbol on the rhs, etc.

Building ASTs with Syntax Directed Translation

- Assume constructors for each node
- Assume stack holds pointers to nodes
- Assume Yacc syntax

\[
\begin{array}{lcl}
S & \rightarrow & \text{Expr} & \quad $$ = $1; \\
\text{Expr} & \rightarrow & \text{Expr} + \text{Term} & \quad $$ = \text{MakeAddNode}($1,$3); \\
& | & \text{Expr} - \text{Term} & \quad $$ = \text{MakeSubNode}($1,$3) \\
& | & \text{Term} & \quad $$ = $1; \\
\text{Term} & \rightarrow & \text{Term} * \text{Factor} & \quad $$ = \text{MakeMulNode}($1,$3); \\
& | & \text{Term} / \text{Factor} & \quad $$ = \text{MakeDivNode}($1,$3); \\
& | & \text{Factor} & \quad $$ = $1; \\
\text{Factor} & \rightarrow & ( \text{Expr} ) & \quad $$ = $2; \\
& | & \text{num} & \quad $$ = \text{MakeNumNode}(\text{token}); \\
& | & \text{id} & \quad $$ = \text{MakeIdNode}(\text{token});
\end{array}
\]
Syntax Directed Translation

Most compilers use *ad-hoc* style of context-sensitive analysis using syntax directed translation (instead of formalisms like attribute grammars).

Advantages
- Addresses the shortcomings of the attribute grammars (see next lecture)
- Efficient, flexible

Disadvantages
- Must write the code with little assistance
- Programmer deals directly with the details

Typical uses
- Building the AST and the symbol table
- Simple error and type checking can also be done

AST Design

- We decided that we will use syntax directed translation to build the AST
- Now we need to decide how our AST is going to look like
  - We need to design the data structure to use for the AST
- There is a convenient way of structuring ASTs based on object oriented design principles.
Object Oriented ASTs

- Object oriented AST:
  - An AST is described by one or more abstract classes, corresponding to a symbol in the grammar
  - Each abstract class is extended by one or more sub-classes, one for each grammar rule
  - For each non-trivial symbol on the right hand side, there is a field in the corresponding class
  - Each class has a constructor that initializes all fields and is thereafter immutable

- This design style corresponds to an object oriented design pattern called Composite Pattern
- There is also another pattern that is used in conjunction with the Composite Pattern called Visitor Pattern
  - Using composite and visitor patterns together we get a nice representation for ASTs and an elegant way of implementing analyses (such as type checking)

Digression: Design Patterns

- Design patterns provide a mechanism for expressing common object-oriented design structures
- Design patterns identify, name and abstract common themes in object-oriented design
- Design patterns can be considered micro architectures that contribute to overall system architecture
- Design patterns are helpful
  - In developing a design
  - In communicating the design
  - In understanding a design
Composite Design Pattern Class Diagram

An Object Diagram Showing an Instance
Composite Pattern

Intent
Composite lets clients treat individual objects and compositions of objects uniformly.

Applicability
Use Composite pattern when
• you want to represent part-whole hierarchies of objects
• you want client to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Composite Pattern

Participants
• Component
  - declares the interface for objects in the composition.
  - implements default behavior for the interface common to all classes
  - declares an interface for accessing and managing child components
• Leaf
  - represent leaf objects, does not have any children
  - defines behavior of primitive objects in the composition
• Composite
  - defines behavior for components having children
  - stores child components
• Client
  - manipulates objects in the composition through Component interface
Representing Expressions

- Assume that we would like to implement a set of classes for representing and manipulating expressions

- These classes can be used in a compiler implementation

- We need to store the expressions in some form (i.e., abstract syntax tree)

- We need to perform operations on the expressions such as
  - type checking

Problem 1: How To Represent Expressions?

- There are different types of expressions such as:
  - boolean literal, integer literal, identifier, binary expression, unary expression, etc.

- Different types of expressions have different attributes so it would make sense to have a different class for each expression type

- However, we should be able to treat expressions uniformly
  - For example, children of binary expressions or unary expressions could be any type of expression
An Expression:  
\[-x + 2 \times y + 1\]

Corresponding Object Diagram:

- \(\text{:Plus}\)
  - \(\text{:Plus}\)
  - \(\text{:Plus}\)
  - \(\text{:IntLit}\) \(\text{value} = 1\)
  - \(\text{:IntLit}\) \(\text{value} = 2\)
  - \(\text{:IntLit}\) \(\text{value} = 1\)
  - \(\text{:Id}\) \(\text{name} = \text{“x”}\)
  - \(\text{:Id}\) \(\text{name} = \text{“y”}\)
Using the Composite Pattern

- Using composite pattern enables us to treat all the expressions uniformly using the Expression interface

```java
Expression e1 = new Id(...);
Expression e2 = new BoolLit(...);
Expression e3 = new Plus(e1, e2);
...
printer.printSomeExpression(e3);
...

public void printSomeExpression(Expression e) {
    e.print();
}
```

client code may not need to know what type of expression e is

We Used the Composite Pattern

```
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)

Leaf
Operation()

Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
```
Problem 2: Analyzing Expressions

- There are many analyses that we may need to do on programs.
  - Can we have a general pattern to write all the analyses

- For each analysis we need to traverse the abstract syntax tree and do some analysis for each node

- Encapsulate the traversal with the analysis as follows:
  - Visit each item
  - Perform a type-specific action on each item
    - For example, type check

- We can abstract recursive traversal in a class
  - Create a visit operation for each element that performs the analysis
  - The visitor can call the operations of the element while performing the analysis

Encapsulating Analysis: Visitor Pattern
Type Checking Expressions

- Assume that we need to do type checking on expressions
  - For example, arguments of an addition operation should be integers; types of left and right children of an equality expression should match, etc.

- We may need to add other checks later on
  - For example, are all the identifiers used in the expression have been declared
Checking Expressions with Visitor Pattern

Expression
- Accept(Visitor v)
- getType()
- setType(type)

Client

Visitor
- VisitPlus(Plus)
- VisitUMinus(UMinus)

Plus
- Accept(Visitor v)
- getType()
- setType(type)
- v.VisitPlus(this);

UMinus
- Accept(Visitor v)
- getType()
- setType(type)

TypeChecker
- VisitPlus(Plus)
- VisitUMinus(UMinus)

CheckDeclared
- VisitPlus(Plus)
- VisitUMinus(IMinus)
- VisitId(Id)

Checking Expressions

- The visitPlus method first calls the Accept methods of the left and right children and passes itself (type-checker) as the argument
  - This will type check all subexpressions recursively
- If the children have type errors or if the types of children do not match it sets its own type to type error

VisitPlus(Plus e) {
  e.left.Accept(this);
  e.right.Accept(this);
  if (e.left.getType() == type_error
      || e.right.getType() == type_error
      || e.left.getType() != number
      || e.left.getType() != number)
    e.setType(type_error);
  else
    e.setType(number); // the argument here will depend on the type of the operator
}
Project’s AST Implementation in C++

- We generate the AST classes using a script. This script takes a grammar like input that represents the abstract syntax.
- We generate a set of abstract classes, corresponding to symbols in the grammar.
- Each abstract class is extended by one or more sub-classes, one for each grammar rule.
- For each non-trivial symbol on the right hand side, there is a field in the corresponding class.
- Each class has a constructor that initializes all fields and is thereafter immutable.

```c++
// an example CDEF file for expressions
Program => *Expr
Expr:Plus => Expr Expr
Expr:Minus => Expr Expr
Expr:Div => Expr Expr
Expr:Times => Expr Expr
Expr:Variable => Identifier
Expr:IntLit => Integer
```

Implementation (ast.hpp)

```c++
// Define abstract base class for all Visitors
class Visitor {
public:
  // Declare all virtual visitor functions (all must be implemented in visitors)
  virtual void visitProgramNode(ProgramNode* node) = 0;
  virtual void visitPlusNode(PlusNode* node) = 0;
  virtual void visitMinusNode(MinusNode* node) = 0;
  virtual void visitDivNode(DivNode* node) = 0;
  virtual void visitTimesNode(TimesNode* node) = 0;
  virtual void visitVariableNode(IdentNode* node) = 0;
  virtual void visitIntegerLiteralNode(IntLitNode* node) = 0;
  virtual void visitIntegerNode(IntegerNode* node) = 0;
};

// Define abstract base class for all AST Nodes
// (this also serves to define the visitable objects)
class ASTNode {
public:
  // All AST nodes have a member which stores their basetype (int, bool, none, object)
  // Further type information will come from the symbol table.
  BaseType basetype;

  // All AST nodes provide visit children and accept methods
  virtual void visit_children(Visitor* v) = 0;
  virtual void accept(Visitor* v) = 0;
};
```
// Define all abstract AST node classes
class ExprNode : public ASTNode { public:
    std::string name;
    virtual void visit_children(Visitor* v) { /* No Children */ }
    virtual void accept(Visitor* v) { v->visitExprNode(this); } 
};
// Define leaf AST nodes for ids and ints (also used for bools)
// Identifiers have a member name, which is a string
Class IdentifierNode : public ASTNode { public:
    std::string name;
    virtual void visit_children(Visitor* v) { /* No Children */ }
    virtual void accept(Visitor* v) { v->visitIdentifierNode(this); } 
    IdentifierNode(std::string name) { this->name = name; }
};
// Integers have a member value, which is a C int
class IntegerNode : public ASTNode { public:
    int value;
    virtual void visit_children(Visitor* v) { /* No Children */ }
    virtual void accept(Visitor* v) { v->visitIntegerNode(this); } 
    IntegerNode(int value) { this->value = value; }
};

// AST Node for Program
class ProgramNode : public ASTNode { public:
    virtual void visit_children(Visitor* v) { ProgramNode(std::list<ExprNode*>* expr_list); }
    virtual void accept(Visitor* v) { v->visitProgramNode(this); }
    std::list<ExprNode*>* expr_list; 
};

// AST Node for Plus
class PlusNode : public ExprNode { public:
    virtual void visit_children(Visitor* v) { ExprNode* expr_1;
    ExprNode* expr_2;
    PlusNode(ExprNode* expr_1, ExprNode* expr_2); }
    virtual void accept(Visitor* v) { v->visitPlusNode(this); } 
};

// AST Node for Variable
class VariableNode : public ExprNode { public:
    virtual void visit_children(Visitor* v) { VariableNode(IdentifierNode* identifier); }
    virtual void accept(Visitor* v) { v->visitVariableNode(this); } 
    IdentifierNode* identifier;
};

// AST Node for IntegerLiteral
class IntegerLiteralNode : public ExprNode { public:
    virtual void visit_children(Visitor* v) { IntegerLiteralNode(IntegerNode* integer); }
    virtual void accept(Visitor* v) { v->visitIntegerLiteralNode(this); } 
    IntegerNode* integer;
};
Implementation (ast.cpp)

// Constructor for Program AST node
ProgramNode::ProgramNode(std::list<ExprNode*>* expr_list) {
  this->expr_list = expr_list;
}

// Constructor for Plus AST node
PlusNode::PlusNode(ExprNode* expr_1, ExprNode* expr_2) {
  this->expr_1 = expr_1;
  this->expr_2 = expr_2;
}

// Constructor for Variable AST node
VariableNode::VariableNode(IdentifierNode* identifier) {
  this->identifier = identifier;
}

// Constructor for IntegerLiteral AST node
IntegerLiteralNode::IntegerLiteralNode(IntegerNode* integer) {
  this->integer = integer;
}

// Visit Children method for Program AST node
void ProgramNode::visitChildren(Visitor* v) {
  if (this->expr_list) {
    for (std::list<ExprNode*>::iterator iter = this->expr_list->begin();
      iter != this->expr_list->end(); iter++) {
      (*iter)->accept(v);
    }
  }
}

// Visit Children method for Plus AST node
void PlusNode::visitChildren(Visitor* v) {
  expr_1->accept(v);
  expr_2->accept(v);
}

// Visit Children method for Variable AST node
void VariableNode::visitChildren(Visitor* v) {
  identifier->accept(v);
}

// Visit Children method for IntegerLiteral AST node
void IntegerLiteralNode::visitChildren(Visitor* v) {
  integer->accept(v);
}
Implementation (ast.cpp)

// Define concrete implementations of all abstract visitor functions
// Concrete visit function for Program nodes
void Print::visitProgramNode(ProgramNode* node) {
    this->pushLevel("Program", true);
    node->visit_children(this);
    this->popLevel(true, true);
}

// Concrete visit function for Plus nodes
void Print::visitPlusNode(PlusNode* node) {
    this->pushLevel("Plus");
    node->visit_children(this);
    this->popLevel();
}

// Concrete visit function for Variable nodes
void Print::visitVariableNode(VariableNode* node) {
    this->pushLevel("Variable");
    node->visit_children(this);
    this->popLevel();
}

// Concrete visit function for IntegerLiteral nodes
void Print::visitIntegerLiteralNode(IntegerLiteralNode* node) {
    this->pushLevel("IntegerLiteral");
    node->visit_children(this);
    this->popLevel();
}

// Concrete visit function for Identifiers (leaf nodes)
void Print::visitIdentifierNode(IdentifierNode* node) {
    stringstream ss;
    // Print the name of the identifier in quotes
    ss << "" << node->name << "";
    this->addElement(ss.str());
    node->visit_children(this);
}

void Print::visitIntegerNode(IntegerNode* node) {
    stringstream ss;
    // Print the value of the integer
    ss << node->value;
    this->addElement(ss.str());
    node->visit_children(this);
}

Example AST for: 2 + 5 - x