Contextual Analysis

Vertalerbouw **HC3**

VB HC3

Arend Rensink
kamer: Zilverling 5090
telefoon: 4862
email: rensink@cs.utwente.nl

Original design: Theo Ruys
University of Twente
Department of Computer Science
Formal Methods & Tools

What have you seen last time?

- More tombstones
- Compiler phases and passes
  - Syntax analysis: Scanning and parsing
  - Contextual analysis
  - Code generation
- Parsing
  - Top-down parsing: LL(k)
  - Recursive descent for LL(1)
  - Bottom-up parsing: LR(k), PDA's
- Scanning (lexical analysis)
  - Requires only regular expressions

Overview of Lecture 3

- Mededelingen
- Ch 4 – Syntactic Analysis
  - 4.1-3 ...
  - 4.4 Abstract Syntax Trees
  - 4.5-6 ...
- Ch 5 – Contextual Analysis
  - 5.1 Identification
  - 5.2 Type Checking
  - 5.3 Contextual Analysis algorithm
  - 5.4 Case study: Contextual Analysis for Triangle

Mededelingen

- Opgavenserie 1 komt vandaag beschikbaar op Blackboard
  - deadline: maandag 21 mei 2012 om 18.00 uur
  - wees precies: slordigheden zijn meestal fout

Ch 4 – Syntactic Analysis

HC2

Again: Recursive-Descent Parsing

Systematic development of a recursive-descent parser:

1. Express the grammar in EBNF.
2. Grammar transformations:
   - eliminate left recursion
   - left-factorization
3. Create a Java Parser class with
   - `protected` variable `currentToken`
   - methods to call the scanner: `accept` and `acceptIt`
   - `public` method `parse` which
     - gets the first token from the scanner, and
     - calls the parse method of the root non-terminal of the grammar
4. Implement `protected` parsing methods
   - add `protected` methods `parseN` for each non-terminal N

Again: Recursive-Descent Parsing (2)

```
public class MicroEnglishParser {
  protected Token currentToken;

  public void parse() {
     currentToken = first token;
     parseSentence();
     check that no token follows the sentence
  }

  protected void accept(Token expected) { ... }
  protected void parseSentence() { ... }
  protected void parseSubject() { ... }
  protected void parseObject() { ... }
  protected void parseNoun() { ... }
  protected void parseVerb() { ... }

  ...
}
```

connection to the scanner which provides the tokens

Allows customization of the parser through inheritance.

Abstract Syntax Trees (1)

- A recursive-descent parser builds the syntax tree implicitly by the call graph of the parse methods.
  - In a *one-pass* compiler this is OK.
  - In a *multi-pass compiler* we need an explicit representation of the (abstract) syntax tree.
- Remember that each nonterminal XYZ is converted to a parse method `parseXYZ`:

```
protected void parseXYZ( ) { ... }
```

Instead of returning nothing, the method could return something interesting. What about an AST node?

Furthermore, other parse methods that call this method could pass useful information using parameters.

Abstract Syntax Trees (2)

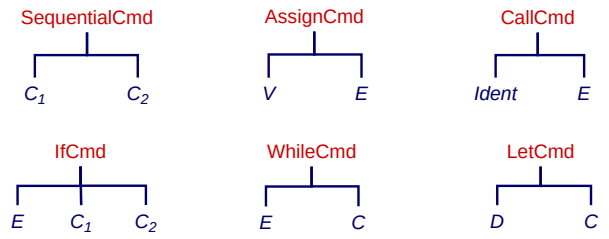
Grammar for Mini-Triangle's abstract syntax.

Program	::=	Command	Program
Command	::=	Command ; Command	SequentialCmd
	|	V-name := Expression	AssignCmd
	|	Identifier (Expression)	CallCmd
	|	if Expression then Command else Command	IfCmd
	|	while Expression do Command	WhileCmd
	|	let Declaration in Command	LetCmd
Expression	::=	Integer-Literal	IntegerExpr
	|	V-name	VnameExpr
	|	Operator Expression	UnaryExpr
	|	Expression Operator Expression	BinaryExpr
V-name	::=	Identifier	SimpleVname
Declaration	::=	Declaration ; Declaration	SeqDecl
	|	const Identifier ~ Expression	ConstDecl
	|	var Identifier : Type-denoter	VarDecl
Type-denoter	::=	Identifier	SimpleTypeDen

AST nodes of Mini-Triangle

Abstract Syntax Trees (3)

Command	::=	Command ; Command	SequentialCmd
	|	V-name := Expression	AssignCmd
	|	Identifier (Expression)	CallCmd
	|	if Expression then single-Command else single-Command	IfCmd
	|	while Expression do single-Command	WhileCmd
	|	let Declaration in single-Command	LetCmd



Abstract Syntax Trees (4)

- We need to define Java classes to capture the structure of Mini-Triangle ASTs. We introduce the abstract class **AST**.

```
public abstract class AST { ... }
```

- Every node in the AST will be an object of a subclass of AST. Each subclass has instance variables for the children nodes.

```
public class Program extends AST {
   public Command C;
   ...
}
```

Command is the abstract base class for all *Command* AST nodes.

```
public abstract class Command extends AST { ... }
```

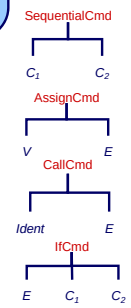

ASTs (5)

```
abstract class Command
  extends AST { ... }
```

Command			
	::=	Command ; Command	SequentialCmd
	|	V-name := Expression	AssignCmd
	|	Identifier (Expression)	CallCmd
	|	if Expression then single-Command else single-Command	IfCmd
	|	while Expression do single-Command	WhileCmd
	|	let Declaration in single-Command	LetCmd

```
public class SequentialCmd extends Command {
   public Command C1, C2;
   ...
}
public class AssignCmd extends Command {
   public Vname       V;
   public Expression  E;
   ...
}
public class CallCmd extends Command {
   public Identifier  I;
   public Expression  E;
   ...
}
public class IfCmd extends Command {
   public Expression  E;
   public Command     C1, C2;
   ...
}
```

etc.



The AST subclasses should have constructors to build an object of these classes.

Abstract Syntax Trees (6)

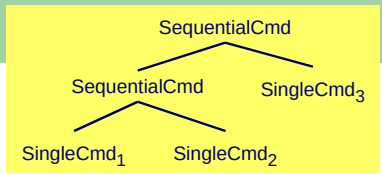
- It is straightforward to make a recursive-descent parser construct an AST to represent the phrase structure:
  - Make each method **parseN** (as well as parsing a N-phrase), return the N-phrase's AST.
  - Let the body of a method **parseN** construct the N-phrase AST by combining the ASTs of any subphrases.
- Thus, for production rule N::=X

```
protected ASTN parseN() {
   ASTN itsAST;
   parse X, at the same time constructing itsAST
   return itsAST;
}
```


Abstract Syntax Trees: Example

EBNF Command ::= SingleCommand (; SingleCommand)*

```
protected Command parseCommand() {
   Command c1AST = parseSingleCmd();
   while (currentToken.kind == Token.SEMICOLON) {
      acceptIt();
      Command c2AST = parseSingleCmd();
      c1AST = new SequentialCmd(c1AST, c2AST);
   }
   return c1AST;
}
```



AST Command ::= Command ; Command SequentialCmd

Ch 5 – Contextual Analysis

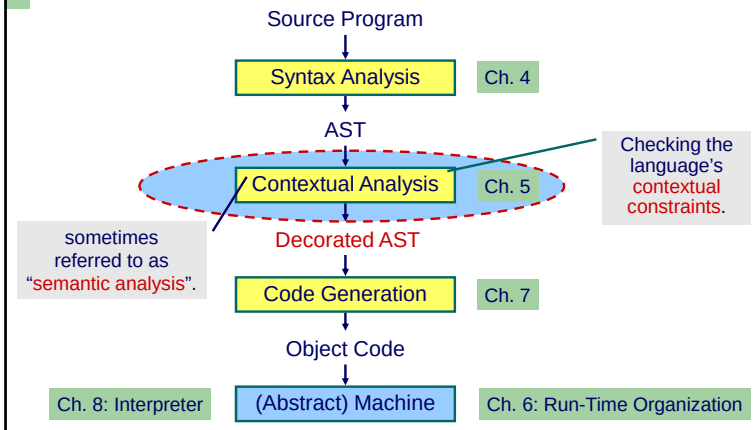
5.1 Identification

5.2 Type checking

5.3 Contextual Analysis algorithm

5.4 Case study: Triangle compiler

Compiler Phases



HC1

Context Constraints – scope rules

- Scope rule - examples

```
let
   const m ~ 2;
   var n: Integer
in begin
   ...
   n := m*2;
   ...
end
```

declaration of n: binding occurrence

use of n: applied occurrence

??

```
let
   var n: Integer;
in begin
   ...
   n := m*2;
end
```

applied occurrence

If there is no enclosing scope (i.e. LetCmd) where m is declared, the binding occurence of m is missing: scope error!

HC1

Context Constraints – type rules

- Type rule - example

```
let
   var n: Integer
in begin
   ...
   while n > 0 do
      n := n-1;
   ...
end
```

Type rule for $E_1 > E_2$ (GreaterOp): If E_1 and E_2 are both of type int, then the result is of type bool.

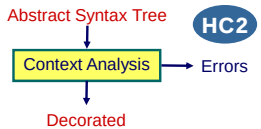
Type rule for **while** E **do** C (WhileCmd): E must be a boolean.

Type rule for V:=E (AssignCmd): The types of V and E must be equivalent.

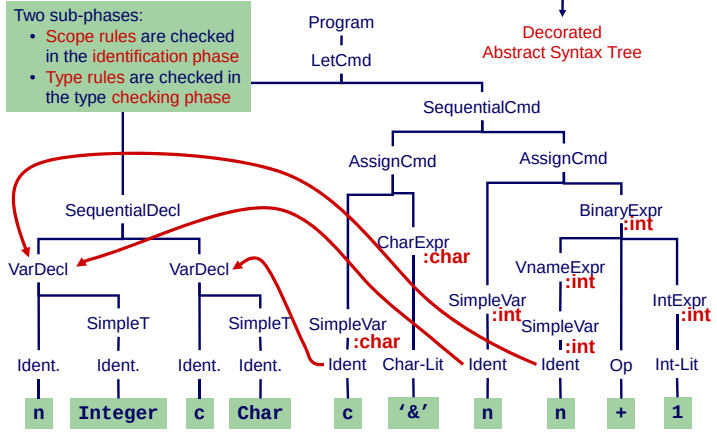
Type rule for $E_1 - E_2$ (SubOp): If E_1 and E_2 are both of type int, then the result is of type int.

HC2

Contextual Analysis



Two sub-phases:
- Scope rules are checked in the identification phase
- Type rules are checked in the type checking phase



Things to check

- An applied occurrence of an identifier must have a matching defining occurrence.

  A pointer to the defining occurrence can be added to the AST.
- In an assignment, the identifier on the left-hand side should refer to a variable.
- Function calls must refer to defined functions.
- The expression of an **if** or **while** should be **boolean**.
- When calling procedures, the number and types of the actual parameters (arguments) should match the number and types of the formal parameters.
- etc.

All follow from the context constraints defined for the (programming) language.

Symbol Table (1)

See exercise 1.3 of the laboratory session of week 1.

- Symbol table (called identification table in W&B)
  - Dictionary-style data structure in which identifiers are stored together with their attributes.
  - Typical attributes
    - type: int, char, boolean, record, array, pointer, etc.
    - kind: constant, variable, procedure, function, value-parameter, reference-parameter, etc.
    - visibility: public, private, protected
  - Typical operations
    - enter an identifier and its attributes into symbol table
    - retrieve the attributes for an identifier
  - other operations depend on the block structure of the language.

Block

- Scope of a declaration: area of the program over which the declaration takes effect.
- Block: area of program text that delimits the scope of declarations within it.
  - Triangle's block commands

    `let Declarations in Commands`

    `proc P(`*formal-parameters*`) ~ Commands`
  - In Java:

    A block is a sequence of statements, local class declarations and local variable declaration statements within braces.

```
for (int i=0; i<a.length; i++) {
  String s = a[i];
  System.out.println(s);
}
```

Both `i` and `s` cannot be used outside the `for`-loop.

Monolithic block structure

```
program
  Declarations

begin
  Commands
end
```

sequence of declarations

sequence of commands

- Characteristics
  - Only one block: entire program
  - All declarations are global in scope.
- Scope rules
  - *No* identifier may be declared more than once.
  - *No* identifier may be used before declared.
- Symbol table
  - For every identifier there is a single entry in the symbol table.
  - Retrieval should be fast (e.g. binary search tree or hash table).

BASIC, COBOL, …

Flat block structure

```
program
  D

  procedure P
    D
  begin
    C
  end

  procedure Q
    D
  begin
    C
  end

  begin
    C
  end
```

- Characteristics
  - program has several disjoint blocks
  - two scope levels: global and local
- Scope rules
  - *No* globally declared identifier may be redeclared globally.
  - *No* locally declared identifier may be redeclared in the same block.
  - *No* identifier may be used before declared (globally or locally)
- Symbol table
  - Both global and local declarations.
  - After analysis of a block has completed, its local declarations can be discarded.

Fortran, C

Nested block structure (1)

```
program
  D
  procedure P
    D
    procedure PP
      D
      proc PPP
    begin C end
  begin
    C
  end
  procedure Q
    D
  begin
    C
  end
  begin
    C
  end
```

- Characteristics
  - blocks can be nested within each other
  - many scope levels
- Scope rules
  - *No* identifier may be declared more than once in the same block.
  - *No* identifier may be used unless declared (in local or enclosed blocks).
- Symbol table
  - Several entries for each identifier.
  - But, at most one entry for each scope level+identifier combination.
  - Highest-level entry of an identifier should be retrieved (fast).

Pascal, Modula, Ada, Java, etc.

Nested block structure (2)

```
let !level 1
   var a, b, c ;
in begin
   let !level 2
      var a, b ;
   in begin
      let !level 3
         var a, c ;
      in begin
         a := b + c ;
      end;
      a := b + c ;
   end;
   a := b + c ;
end
```

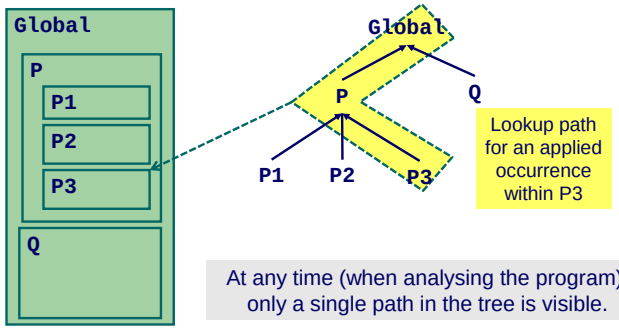
Scope and visibility

a and **b** of level 1 get redefined and are not visible on level 2

a of level 2 and **c** of level 1 get redefined and are not visible on level 3

Scope structure

- For a statically scoped language with nested block structure, the structure of the scopes can be seen as a tree.



At any time (when analysing the program), only a single path in the tree is visible.

Symbol Table (2)

Example

```
let !level 1
(1) var a: Integer;
(2) var b: Boolean
in begin
   ...
   let !level 2
   (3) var b: Integer;
   (4) var c: Boolean
   in begin
      let !level 3
      (5) const x ~ 3
      in ...
   end
   let !level 2
   (6) var d: Boolean;
   (7) var e: Integer
   in begin
end
```

level	id	Attr.
1	a	(1)
1	b	(2)
2	b	(3)
2	c	(4)
		(5)

level	id	Attr.
1	a	(1)
1	b	(2)
2	d	(6)
2	e	(7)

Symbol Table Operations

- Symbol table – additional operations
  - open a new scope level
  - close the highest scope level

Attribute: holds all important information on a defined occurrence (type, kind, level, visibility, etc.)

```
public class SymbolTable {
    /** Open a new scope. */
    public void openScope()

    /** Closest the highest (current) scope. */
    public void closeScope()

    /** Enters an id together with its Attribute. */
    public void enter(String id, Attribute attr);

    /** Returns the Attribute of id, defined on the
     * highest level. Return null if not in table. */
    public Attribute retrieve(String id)

    /** Returns the current scope level. */
    public int currentLevel()
}
```


Symbol Table Implementation

- Possible implementation:

```
public class SymbolTable {
    private Map<String,Stack<Attribute>> symtab;
    private Stack<List<String>>          scopeStack;
    ...
}
```

Only used to optimize the closing of a scope.

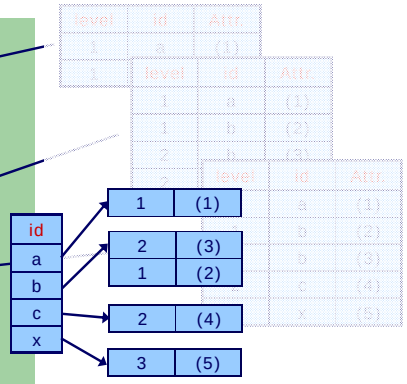
The **symtab** is a Map from Strings to Stacks of Attributes.
- Keys: the String-representations (names) of the identifiers.
- Values: Stack of Attributes; the Attributes of the identifier declared on the highest scope level is always on top.

The **scopeStack** is a Stack of Lists of Strings.
- **openScope:** An empty List is pushed on the **scopeStack**; the String-representation of each identifier found in the newly opened scope will be added to this list.
- **closeScope:** The identifiers of this "old" scope (which are all in the top List of **scopestack**) are removed from **symtab**. The List of the old scope is popped from **scopeStack**.

Symbol Table (5)

Previous example using a map of **<String, Stack<Attribute>>**.

```
let !level 1
(1) var a: Integer;
(2) var b: Boolean
in begin
   ...
   let !level 2
   (3) var b: Integer;
   (4) var c: Boolean;
   in begin
      let !level 3
      (5) const x ~ 3
      in ...
   end
   let !level 2
   (6) var d: Boolean
   (7) var e: Integer
   in begin
end
```



Attributes

- Attributes should (at least) contain information for
  - checking the scope rules
    - *Successful retrieval of an applied occurrence in the symbol table is enough: it means that there is a binding occurence.*
  - checking the type rules
    - *The type of an identifier has to be stored.*
  - (code generation)
    - *address of the variable*
- Possible approaches
  - Store the attribute completely into the symbol table.
  - Alternative: store a pointer to the AST node of the binding occurrence

Explicit Attributes: Imperative

- Imperative approach for storing attributes explicitly.

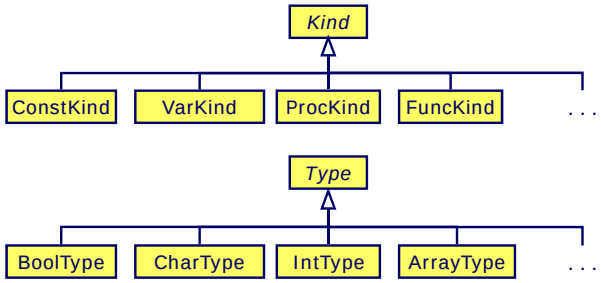
```
public class Attribute {
    public static enum Kind {
        CONST,
        VAR,
        PROC,
        ... ;
    public static enum Type {
        BOOL,
        CHAR,
        INT,
        ARRAY,
        ... ;

    public Kind kind;
    public Type type;
}
```


Explicit Attributes: OO-approach

- OO approach for storing attributes explicitly.

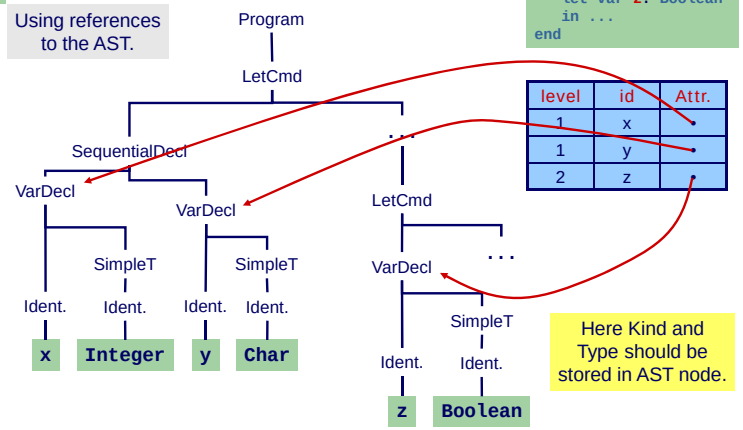
```
public class Attribute {
    public Kind kind;
    public Type type;
}
```



Attributes in AST

```
let var x: Integer;
    var y: Char
in begin
   ...
   let var z: Boolean
   in ...
end
```

Using references to the AST.



level	id	Attr.
1	x	•
1	y	•
2	z	•

Here Kind and Type should be stored in AST node.

Types

- What is a type?
  - Low-level view: “A restriction on the possible interpretations of a segment of memory or other program construct”.
  - High-level view: A set of allowed values.
- Why use types?
  - Error avoidance: prevent programmer from making type errors (e.g. round peg in square hole).
  - Runtime optimization: earlier binding leads to fewer runtime decisions (e.g. C).
- Are types really needed?
  - No, many languages can operate (fine) without them
    - assembly languages, script languages (e.g. Python, Tcl)

Type Checking (1)

- In a statically typed language every expression E is either (i) ill-typed, or (ii) has a static type that can be computed without actually evaluating E.

  When an expression E has static type T this means that when E is evaluated then the returned value will always have type T.

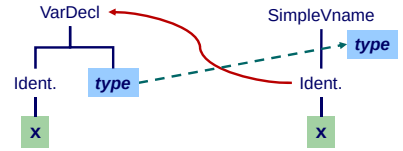
- Most modern languages have a large emphasis on static typechecking.

  But object-oriented programming languages (e.g. Java) require some runtime type checking.

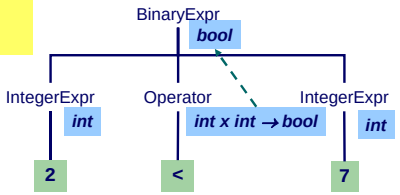
Type Checking (2)

- Type checking involves
  1. calculating or inferring the types of expressions (by using information about the types of their components)
  2. checking that these types are what they should be (e.g. the condition of **`if`**-statement must have type Boolean).
- Bottom-up type checking algorithm:
  - Types for expression AST leaves are known:
    - literals: denotation (true/false, 2, 3, 'a')
    - variables: retrieve from symbol table
    - constants: retrieve from symbol table
  - Types for internal nodes are inferred from the type of the children and the type rule for that kind of expression.

Type Checking (3)



Type rule for binary expression E_1 op E_2:
- If op is operation of type $T_1 x T_2 \rightarrow R$
- and E_1 and E_2 have types compatible with T_1 and T_2 respectively.
- then E_1 op E_2 is of type R



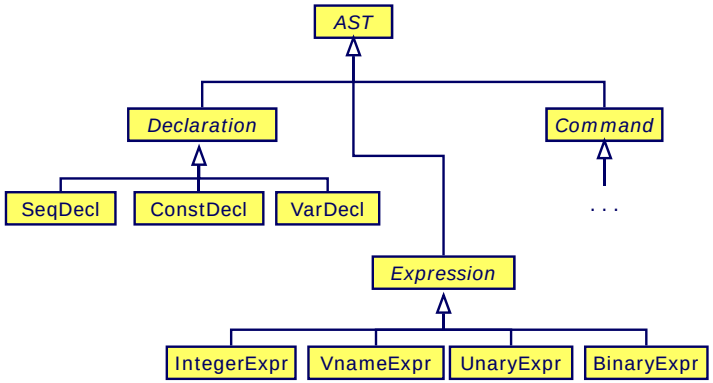
AST Hierarchy for Expressions

Expression	::=	Integer-Literal	IntegerExpr
	|	V-name	VnameExpr
	|	Operator Expression	UnaryExpr
	|	Expression Operator Expression	BinaryExpr

```
public class BinaryExpr extends Expression {
    public Expression E1, E2;
    public Operator   O;
}
public class UnaryExpr extends Expression {
    public Expression E;
    public Operator   O;
}
...
```

AST Class Hierarchy

See W&B D.2



Decoration

- Decoration is done by adding some instance variables to some of the AST classes.

```
public abstract class Expression extends AST {
    // Every expression has a type
    public Type type;
    ...
}
```

```
public class Identifier extends Token {
    // Binding occurrence of this identifier
    public Declaration decl;
    ...
}
```


AST Traversal (1)

- Add to each AST class methods for type checking (or code-generation, pretty printing, etc.).
  In each AST node class, the methods traverse their children.

```
public abstract AST() {
    public abstract Return1 check(Arg1 arg);
    public abstract Return2 encode(Arg2 arg);
    public abstract Return3 prettyPrint(Arg3 arg);
}
...
Program program;
program.check(null);
```

Return value can be used to pass information up the AST tree.

Extra **arg** can be used to pass information down the AST tree.

- Advantage: OO-idea is easy to understand and implement
- Disadvantage: Methods are spread over all AST classes: not very modular

AST Traversal (2)

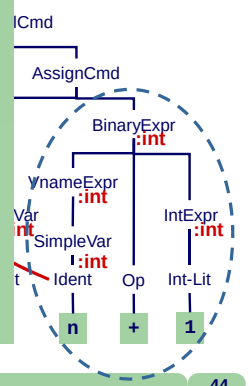
Example

```
public abstract class Expression extends AST {
    public Type type;
    ...
}
public class BinaryExpr extends Expression {
    public Expression e1, e2;
    public Operator   o;

    public Type check(Object arg) {
        Type t1 = e1.check(null);
        Type t2 = e2.check(null);
        Type result = o.compatible(t1,t2);
        if (result == null)
            report type error
        return result;
    }
    ...
}
```

or

```
Type[] tmp = new Type { t1, t2 };
Type result = O.check(tmp);
```



Visitor pattern (1)

- The Visitor pattern – from the famous "Design Patterns" book by Gamma et. al. (1994) – lets you define a new operation on the elements of an object (e.g. the nodes in an AST) without changing the classes of the elements on which it operates.
  - Useful if many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations.
- Some characteristics:
  - Good: Visitors makes adding new operations easy.
  - Good: A visitor gathers related operations and separates unrelated ones.
  - Bad: Visitor pattern breaks encapsulation.

Using a Visitor (1)

In literature on software patterns the method `visit` is usually named `accept`.

- Idea: use an extra level of indirection
  - define a special `Visitor` class to *visit* the nodes in the tree.
  - add (only-one) `visit` method to the AST classes, which lets the visitor actually visit the AST node.

```
public abstract class AST {
    public abstract Object visit(Visitor v, Object arg);
}
public class AssignCmd extends Command {
    public Object visit(Visitor v, Object arg) {
        return v.visitAssignCmd(this, arg);
    }
}
```

So instead of several methods like `check`, `encode`, etc, only a single `visit` method.

(an implementation of) this method will do the type-checking (or code generation, printing, etc.).

```
public class XYZ extends ... {
    public Object visit(Visitor v, Object arg) {
        return v.visitXYZ(this, arg);
    }
}
```

General template for all AST node classes.

Using a Visitor (2)

```
public class XYZ extends ... {
    Object visit(Visitor v, Object arg) {
        return v.visitXYZ(this, arg);
    }
}
```

```
public interface Visitor {
    public Object visitProgram
            (Program prog, Object arg);
    ...
    public Object visitAssignCmd
            (AssignCmd cmd, Object arg);
    public Object visitSequentialCmd
            (SequentialCmd cmd, Object arg);
    ...
    public Object visitVnameExpression
            (VnameExpression e, Object arg);
    public Object visitBinaryExpression
            (BinaryExpression e, Object arg);
    ...
}
```

The Visitor interface defines `visitXYZ` methods for all AST classes!

```
    public Object visitXYZ
            (XYZ x, Object arg);
```


Checker as a Visitor (1)

- Any implementation of `Visitor` can traverse the AST.

```
public class Checker implements Visitor {

    private SymbolTable symtab;

    public void check(Program prog) {
        symtab = new SymbolTable();
        prog.visit(this, null);
    }

    ... + implementations of all methods of Visitor
}
```

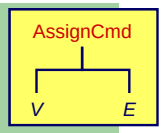
Root node of the AST.

All methods for a specific pass over the AST end up in the same class, i.e. the same file!

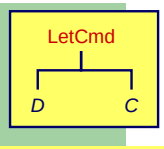
Checker as a Visitor (2)

```
public class XYZ extends ... {
  Object visit(Visitor v,
               Object arg) {
    return v.visitXYZ(this, arg);
  }
}
```

```
public Object visitAssignCmd
        (AssignCmd com, Object arg) {
   Type vType = (Type) com.V.visit(this, null);
   Type eType = (Type) com.E.visit(this, null);
   if (! com.V.isVariable())
       error: left side is not a variable
   if (! eType.equals(vType))
       error: types are not equivalent
   return null;
}
```




```
public Object visitLetCmd
        (LetCmd com, Object arg) {
   symtab.openScope();
   com.D.visit(this, null);
   com.C.visit(this, null);
   symtab.closeScope();
   return null;
}
```

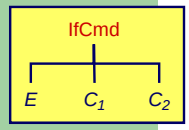


Note that the letCmd opens (and closes) the scope of the Symbol Table.

Checker as a Visitor (3)

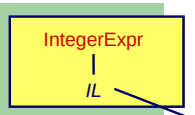
```
public class XYZ extends ... {
  Object visit(Visitor v,
               Object arg) {
    return v.visitXYZ(this, arg);
  }
}
```

```
public Object visitIfCmd
        (IfCmd com, Object arg) {
   Type eType = (Type)com.E.visit(this, null);
   if (! eType.equals(Type.bool))
       error: condition is not a boolean
   com.C1.visit(this, null);
   com.C2.visit(this, null);
   return null;
}
```




```
public Object visitIntegerExpr
        (IntegerExpr expr, Object arg) {
   expr.type = Type.int;
   return expr.type;
}
```

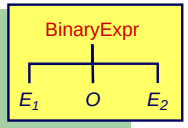
decorating the IntegerExpr node in the AST



no need to visit the Terminal leaf

Checker as a Visitor (4)

```
public Object visitBinaryExpr
        (BinaryExpr expr, Object arg) {
   Type e1Type = (Type) expr.E1.visit(this, null);
   Type e2Type = (Type) expr.E2.visit(this, null);
   OperatorDecl opdecl =
       (OperatorDecl) expr.O.visit(this, null);
   if (opdecl == null) {
       error: no such operator
       expr.type = Type.error;
   } else if (opdecl instanceof BinaryOperatorDecl) {
       BinaryOperatorDecl bopdecl =
           (BinaryOperatorDecl) opdecl;
       if (! e1Type.equals(bopdecl.operand1Type))
           error: left operand has the wrong type
       if (! e2Type.equals(bopdecl.operand2Type))
           error: right operand has the wrong type
       expr.type = bopdecl.resultType;
   } else {
       error: operator is not a binary operator
       expr.type = Type.error;
   }
   return expr.type;
}
```



See W&B for the other visitor methods.

Visiting scoping + type checking

Table 5.1

Program	`visitProgram`	• return `null`
Command	`visit..Cmd`	• return `null`
Expression	`visit..Expr`	• decorate it with its type • return that type
Vname	`visitSimpleVname`	• decorate it with its type • set a flag indicating if it is variable • return the type.
Declaration	`visit..Decl`	• enter all declared identifiers into symbol table • return `null`
TypeDenoter	`visit..TypeDenoter`	• decorate it with its type • return that type
Identifier	`visitIdentifier`	• check that the identifier is declared • set a reference to its binding declaration • return that declaration
Operator	`visitOperator`	• check that the operator is declared • set a reference to its binding declaration • return that declaration

All compound ASTs also check for well-formedness.

Visitor pattern: Drawbacks

- Visitor pattern requires (substantial) preparation:
  - `Visitor` interface with method for each AST node
  - Each AST class `X` needs a corresponding `visitX` method
- Visitor pattern should be there from the start
- Visit methods in the AST classes look obscure
  - Methods are meant for visiting, not for checking.

Type-Generic Visitors

- Visitor interface generic in parameter and return types:

```
public interface Visitor<P,R> {
    public R visitXYZ(XYZ object, P arg);
    …
}
public class TypeChecker implements Visitor<Type,Type> {
    public Type visitXYZ(XYZ object, Type arg) { … }
}
```

- Visited classes need to cope with this:

```
public abstract class VisitedClass {
    public abstract <P,R> visit(Visitor<P,R>, P arg);
}
public class XYZ extends VisitedClass {
    public <P,R> visit(Visitor<P,R> visitor, P arg) {
        visitor.visitXYZ(this, arg);
    }
}
```


What have you seen today?

- AST implementation
  - Construction during parsing
- Scoping
  - Blocks
  - Symbol tables
- Type checking
  - Type rules
  - Visitor pattern