Overview of Lecture 3

- Mededelingen
- Ch 4 – Syntactic Analysis
  4.1 Subphrases of syntactic analysis
  4.2 Grammars revisited
  4.3 Parsing
  4.4 Abstract Syntax Trees
- 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 z.s.m. beschikbaar op de Vertalerbouw-website
  - deadline: woensdag 18 mei 2011 om 18.00 uur
  - wees precies: slordigheden zijn meestal fout

Ch 4 – Syntactic Analysis

- 4.1 Subphrases of syntactic analysis
- 4.2 Grammars revisited
- 4.3 Parsing
- 4.4 Abstract Syntax Trees
- 4.5 Scanning
- 4.6 Case study: Triangle compiler
**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

**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`:

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

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

**Abstract Syntax Trees (2)**

Program ::= Command
Command ::= Command ; Command
         | V-name := Expression
         | Identifier ( Expression )
         | if Expression then Command else Command
         | while Expression do Command
         | let Declaration in Command
Expression ::= Integer-Literal
         | V-name
         | Operator Expression
         | Expression Operator Expression
V-name ::= Identifier
Declaration ::= Declaration ; Declaration
           | const Identifier ~ Expression
           | var Identifier : Type-denoter
Type-denoter ::= Identifier

Grammar for Mini-Triangle’s abstract syntax.
• We need to define Java classes to capture the structure of Mini-Triangle ASTs. We introduce the abstract class `AST`.

```java
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.

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

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

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

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

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

```java
protected ASTN parseN() {
    ASTN itsAST;
    parse X, at the same time constructing itsAST
    return itsAST;
}
```
protected Command parseCommand() {
    Command c1AST = parseSingleCommand();
    while (currentToken.kind == Token.SEMICOLON) {
        acceptIt();
        Command c2AST = parseSingleCommand();
        c1AST = new SequentialCmd(c1AST, c2AST);
    }
    return c1AST;
}
**Contextual Analysis**

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

**Syntax Analysis**

- Program
  - LetCmd
    - SequentialCmd
      - VarDecl
      - VarDecl
      - CharExpr
        - SimpleVar
          - SimpleT
            - Ident.
      - AssignCmd
        - IntExpr
          - VnameExpr
            - BinaryExpr
              - IntLit
  - AssignCmd
  - IntExpr
  - Char-Lit

**Things to check**

- An **applied occurrence** of an identifier must have a matching defining occurrence.
  
  *The relation between applied and defining occurrence might be added to the AST.*

- Function calls must refer to defined functions.

- In an **assignment**, the identifier on the left-hand side should refer to a variable.

- 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

- **Symbol table** (called identification table in W&B)
  - Dictionary-style data structure in which identifiers are stored together with their attributes.
  - **Attributes**
    - type: int, char, boolean, record, array, pointer, etc.
    - kind: constant, variable, procedure, function, value-parameter, reference-parameter, etc.
    - visibility: public, private, protected
    - other important characteristics
- **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.

See exercise 1.3 of the laboratory session of week 1.

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.

```java
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

- **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 unless 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).

Flat block structure

- **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 unless declared (globally or locally)
- **Symbol table**
  - Symbol table should contain entries for both global and local declarations.
  - After analysis of a block has completed, its local declarations can be discarded.
**Nested block structure (1)**

- **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).

**Scope structure**

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

### Symbol Table (2)

<table>
<thead>
<tr>
<th>level</th>
<th>id</th>
<th>Attr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>(1)</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>(2)</td>
</tr>
<tr>
<td>1</td>
<td>c</td>
<td>(3)</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>(4)</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>(5)</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>(6)</td>
</tr>
<tr>
<td>2</td>
<td>d</td>
<td>(7)</td>
</tr>
</tbody>
</table>

**Example**
Symbol Table (3)

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

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

Attributes (1)

- 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 occurrence.
  - 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.
  - As the AST node of the binding occurrence should have access to all necessary information, we could also store references (pointers) to these nodes in the symbol table.
Attributes (2)

- Imperative approach for storing attributes explicitly.

```java
public class Attribute {
    public static final byte constKind = 0,
    varKind = 1,
    procKind = 2,
    ... ;
    public static final byte boolType = 0,
    charType = 1,
    intType = 2,
    arrayType = 3,
    ... ;
    public byte kind;
    public byte type;
}
```

suitable for “little” languages

Attributes (3)

- OO approach for storing attributes explicitly.

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

Works, ... but for a realistic language this can become quite tedious and complex.

Attributes (4)

Using references to the AST.

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

Types

- What is a type?
  - “A restriction on the possible interpretations of a segment of memory or other program construct”.
  - A set of 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 type checking.
  But object-oriented programming languages (e.g. Java) require some runtime type checking.

Type Checking (2)

- Type checking involves (i) calculating or inferring the types of expressions (by using information about the types of their components) and (ii) 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 for statically typed programming languages:
  - The types of expression AST leaves are known:
    - literals: denotation (true/false, 2, 3, 'a')
    - variables: retrieve from symbol table
    - constants: retrieve from symbol table
  - Types of internal nodes are inferred from the type of the children and the type rule for that kind of expression.

Type Checking (3)

- Identification and type checking could be done by two separate passes over the AST.
  However, this is not needed. Both passes can be interleaved, as long as the declaration of an identifier is before its use (and hence its type is available for type checking to proceed).

- Possible algorithm
  - One depth-first left-to-right traversal of the AST, doing both identification and type checking.
  - Results of the analysis are recorded in the AST by decorating it.
Abstract Syntax Trees

Grammar for Mini-Triangle's abstract syntax.

Program ::= Command
Command ::= Command ; Command
V-name ::= Expression
Identifier ( Expression )
if Expression
then Command
else Command
let Declaration in Command
Expression ::= Integer-Literal
V-name
Operator Expression
V-name
Expression Operator Expression
Declaration ::= Declaration ; Declaration
const Identifier ~ Expression
var Identifier : Type-denoter
Type-denoter ::= Identifier
AST nodes of Mini-Triangle

AST Class Hierarchy

AST

Declaration

Command

SeqDecl

ConstDecl

VarDecl

Expression

IntegerExpr

VnameExpr

UnaryExpr

BinaryExpr

Expression ::= Integer-Literal
V-name
Operator Expression
V-name
Expression Operator Expression

Contextual Analysis

Two sub phases:
• scope rules are checked in the identification phase
• type rules are checked in the type checking phase

Abstract Syntax Tree

Decorated Abstract Syntax Tree

Errors

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>Integer</td>
<td>c</td>
<td>Char</td>
<td>c</td>
<td>'6'</td>
</tr>
<tr>
<td>IntExpr</td>
<td>Int</td>
<td>int</td>
<td>Int</td>
<td>Int</td>
<td>Int</td>
</tr>
<tr>
<td>VarDecl</td>
<td>var</td>
<td>var</td>
<td>var</td>
<td>var</td>
<td>var</td>
</tr>
<tr>
<td>SimpleVar</td>
<td>SimpleVar</td>
<td>SimpleVar</td>
<td>SimpleVar</td>
<td>SimpleVar</td>
<td>SimpleVar</td>
</tr>
<tr>
<td>SimpleT</td>
<td>SimpleT</td>
<td>SimpleT</td>
<td>SimpleT</td>
<td>SimpleT</td>
<td>SimpleT</td>
</tr>
<tr>
<td>Int</td>
<td>Int</td>
<td>Int</td>
<td>Int</td>
<td>Int</td>
<td>Int</td>
</tr>
<tr>
<td>Int-Lit</td>
<td>Int-Lit</td>
<td>Int-Lit</td>
<td>Int-Lit</td>
<td>Int-Lit</td>
<td>Int-Lit</td>
</tr>
</tbody>
</table>

Program ::= SequentialCmd
LetCmd
AssignCmd
CharExpr
IntExpr

Contextual Analysis in Java

public class BinaryExpr extends Expression {
  public Expression E1, E2;
  public Operator O;
}

public class UnaryExpr extends Expression {
  public Expression E;
  public Operator O;
}

AST Hierarchy in Java

Expression ::= Integer-Literal
V-name
Operator Expression
V-name
Expression Operator Expression

Decoration

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

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

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

---

Straightforward OO-approach (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.

```java
public abstract AST() {
    public abstract Object check(Object arg);
    public abstract Object encode(Object arg);
    public abstract Object prettyPrint(Object arg);
}
```

```java
Program program;
program.check(null);
```

- advantage: OO-idea is easy to understand and implement
- disadvantage: checking (and encoding) methods are spread over all AST classes: not very modular

---

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.

- This pattern is particular 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:
  - Visitors makes adding new operations easy.
  - A visitor gathers related operations and separates unrelated ones.
  - Visitor pattern breaks encapsulation.
Using a Visitor (1)

- 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.

```java
public abstract class AST {
    public abstract Object visit(Visitor v, Object arg);
}
```

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

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

Using a Visitor (2)

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

Checker as a Visitor (1)

- Any implementation of Visitor can traverse the AST.

```java
public class Checker implements Visitor {
    private SymbolTable symtab;
    public void check(Program prog) {
        symtab = new SymbolTable();
        prog.visit(this, null);
    }
    ...
    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;
    }
    ...
}
```

Checker as a Visitor (2)

- 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!
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;
}

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

public Object visitBinaryExpr(BinaryExpr expr, Object arg) {
    Type e1Type = (Type) expr.E1.visit(this, null);
    Type e2Type = (Type) expr.E2.visit(this, null);
    OperatorDeclaration opdecl = (OperatorDeclaration) expr.O.visit(this, null);
    if (opdecl == null) {
        error: no such operator
        expr.type = Type.error;
    } else if (opdecl instanceof BinaryOperatorDeclaration) {
        BinaryOperatorDeclaration bopdecl = (BinaryOperatorDeclaration) 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.

Summary of visiting methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>visitProgram</td>
<td>return null</td>
</tr>
<tr>
<td>visit..Cmd</td>
<td>return null</td>
</tr>
<tr>
<td>visit..Expr</td>
<td>decorate it with its type, return that type</td>
</tr>
<tr>
<td>visitSimpleVname</td>
<td>decorate it with its type, set a flag indicating if it is variable, return the type</td>
</tr>
<tr>
<td>visit..Decl</td>
<td>enter all declared identifiers into symbol table, return null</td>
</tr>
<tr>
<td>visit..TypeDenoter</td>
<td>decorate it with its type, return that type</td>
</tr>
<tr>
<td>visitIdentifier</td>
<td>check that the identifier is declared, set a reference to its binding declaration, return that declaration</td>
</tr>
<tr>
<td>visitOperator</td>
<td>check that the operator is declared, set a reference to its binding declaration, return that declaration</td>
</tr>
</tbody>
</table>

Not much is gained by this renaming, though.

- The interface Visitor declares for each AST class XYZ the method visitXYZ(XYZ x, Object arg). It is possible to rename all visitXYZ methods to plain visit and rely on Java's overloading mechanism to select the correct visitor method.
- Still all AST classes should have an visit method, calling another overloaded visit method with the this argument (otherwise the overloading will not work).
- As in general the visit methods for the AST classes are all different, you will not profit from 'inheriting' visit methods of superclasses.
Visitor pattern

- The Visitor pattern has some drawbacks.
  - Arguments and return types of the visiting methods have to be known in advance.
    - For new type of visiting methods, these methods have to be added to each AST class.
  - Visitor pattern requires (substantial) preparation:
    - Visitor interface with an abstract method for each AST node;
    - Each AST class should have a visit method;
    - Code itself is tedious to write.
  - Visitor pattern should be there from the start.
  - Visitor code within the visit methods in the AST classes look obscure: they are meant for visiting, not for checking.

Furthermore, the visitor pattern should not be used when the object structure (i.e. AST hierarchy) on which it works is still changing.

Nice

- Nice = Object-Oriented Programming Language
  - research project: adds features from functional languages to Java
  - implemented on top of Java, generates bytecode
  - additional features:
    - parametric/generic functions (like templates in C++)
    - anonymous functions (instead of the heavy anonymous classes of Java)
    - multi-methods (see below)
    - tuples (to return several values from function; type-safe)
    - optional parameters to methods
  - Nice compiler is written in Java and in Nice.

- Multi-methods (or "multiple dispatch")
  - Allow to define methods outside classes.
  - Instead of using (only) the receiver class to dispatch the method (using overriding), one can use the arguments of the method to dispatch the method.

```
Object check(AST node) {
    Object check(node@BinaryExpr) {
        ...
    }
    Object check(node@IfCmd) {
        ...
    }
    Object check(node@AssignCmd) {
        ...
    }
}
```

Within this method you can access node as being an AssignCmd.

With Nice the standard OO-approach and the Visitor-pattern can be combined quite nicely using multi-methods.

The Nice project is not very active (understatement!).

The MultiJava project is another "multiple dispatch" attempt for Java.