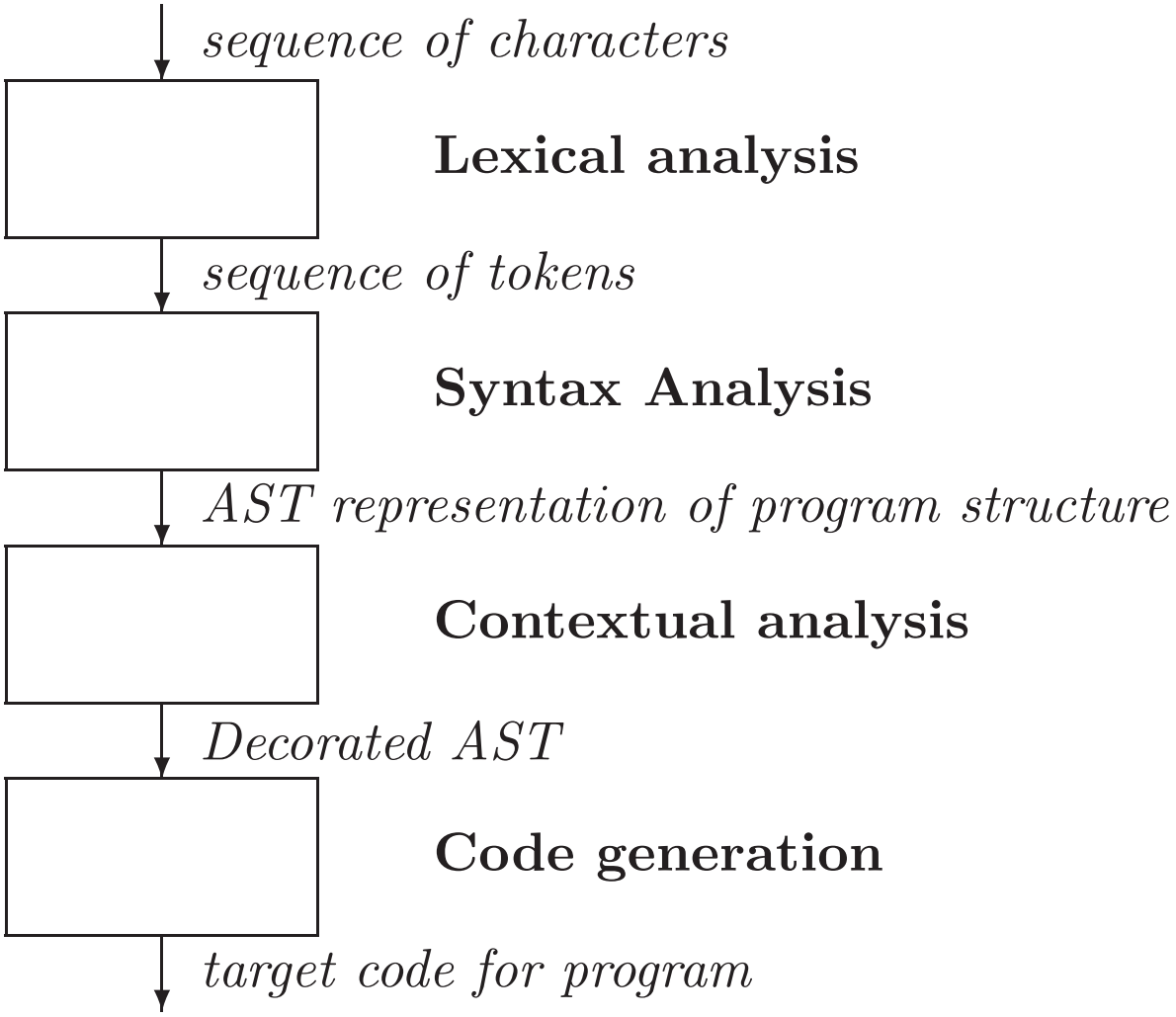
Abstract Syntax Trees and Contextual Analysis

Roland Backhouse

March 8, 2001

Phases of a Compiler



Phases of a Compiler

$$scanner \;::\; char^* \rightarrow token^*$$

$$parseProgram \;::\; token^* \rightarrow AST$$

$$checker \;::\; AST \rightarrow AST$$

$$codeGenerator \;::\; AST \rightarrow instruction^*$$

Phrase Structure Recognition

Specification For each nonterminal A the method *parse*A has the following functions:

- To determine whether a prefix u of the input string from the current input position onwards is in the language generated by A.
- To maintain the invariant property that a syntax error is flagged iff the input string up to the current input position is not a prefix of any sentence of the language being recognized.
- If no error is flagged, to return an object of (abstract syntax tree) class A representing the *longest* such prefix u and to advance the input pointer to the first position beyond u.
- If an error is flagged, to return a `null` object.

Contextual Analysis

Requirement

- Associate uses of identifiers with declarations (Identification)
- Check type correctness.

Specification Contextual analysis is a function from AST's to AST's with error reporting as a side effect.

Formally, we construct a function *checker* of type

$$\mathit{Program}\ \mathit{Identifier}\ \mathit{Operator}\ \mathit{Literal} \rightarrow \mathit{Program}\ \mathit{IdEntry}\ \mathit{Operator}\ \mathit{Literal}$$

that preserves the shape of the AST.

In other words, contextual analysis replaces identifiers and literals in the AST by their entries in the Identification Table.

Implementation Use *visitor pattern* to localise code for different traversals of the AST's.

Abstract Syntax Trees

```
public abstract class AST {

  public AST (SourcePosition thePosition) {
    position = thePosition;
   }

  public SourcePosition getPosition() {
    return position;
  }

}
```

Command

```
public abstract class Command extends AST {

  public Command (SourcePosition thePosition) {
    super (thePosition);
  }

}
```

AssignCommand

```
public class AssignCommand extends Command {

  public AssignCommand (Identifier iAST,
                        Expression eAST, SourcePosition thePosition)
    super (thePosition);
    I = iAST;
    E = eAST;
  }

  public Identifier I;
  public Expression E;

}
```

IfCommand

```
public class IfCommand extends Command {

  public IfCommand (Expression eAST, Command c1AST, Command c2AST,
                    SourcePosition thePosition) {
    super (thePosition);
    E = eAST;
    C1 = c1AST;
    C2 = c2AST;
  }

  public Expression E;
  public Command C1, C2;

}
```

Creating an IfCommand

```
Command parseSingleCommand() throws SyntaxError {
   Command commandAST = null; // in case there's a syntactic error
   SourcePosition commandPos = new SourcePosition();
   start(commandPos);
   switch (currentToken.kind) {
     ...
     case Token.IF:
     {
       acceptIt();
       Expression eAST = parseExpression();
       accept(Token.THEN);
       Command c1AST = parseSingleCommand();
       accept(Token.ELSE);
       Command c2AST = parseSingleCommand();
       finish(commandPos);
       commandAST = new IfCommand(eAST, c1AST, c2AST, commandPos);
     }
     break;
```

Identifiers

```
public class Identifier extends Terminal {

  public Identifier (String theSpelling, SourcePosition thePosition)
    super (theSpelling, thePosition);
    decl = null;
    type = null;
  }

    public String spelling;
    public AST decl;
    public TypeDenoter type;

}
```

Terminals

```
abstract public class Terminal extends AST {

  public Terminal (String theSpelling, SourcePosition thePosition) {
    super (thePosition);
    spelling = theSpelling;
  }

  public String spelling;
}
```

Parsing Identifiers

```
Identifier parseIdentifier() throws SyntaxError {
   Identifier I = null;

   if (currentToken.kind == Token.IDENTIFIER) {
     previousTokenPosition = currentToken.position;
     String spelling = currentToken.spelling;
     I = new Identifier(spelling, previousTokenPosition);
     currentToken = lexicalAnalyser.scan();
   } else {
     I = null;
     syntacticError("identifier expected here", "");
   }
   return I;
 }
```

Identification

Relate applied occurrences of identifiers to the corresponding binding occurrences (i.e. uses of identifiers to their declarations).

Class `IdentificationTable` associates identifiers with their attributes.

Terminology

Scope: Portion of program over which declaration takes effect.

Block: Program phrase delimiting scope of declarations.

Block structure: **Monolithic** (Basic, Cobol)

Flat (FORTRAN)

Nested Pascal, C, Java)