Languages and Compilers
(SProg og Oversættere)

Lecture 3

Bent Thomsen
Department of Computer Science
Aalborg University

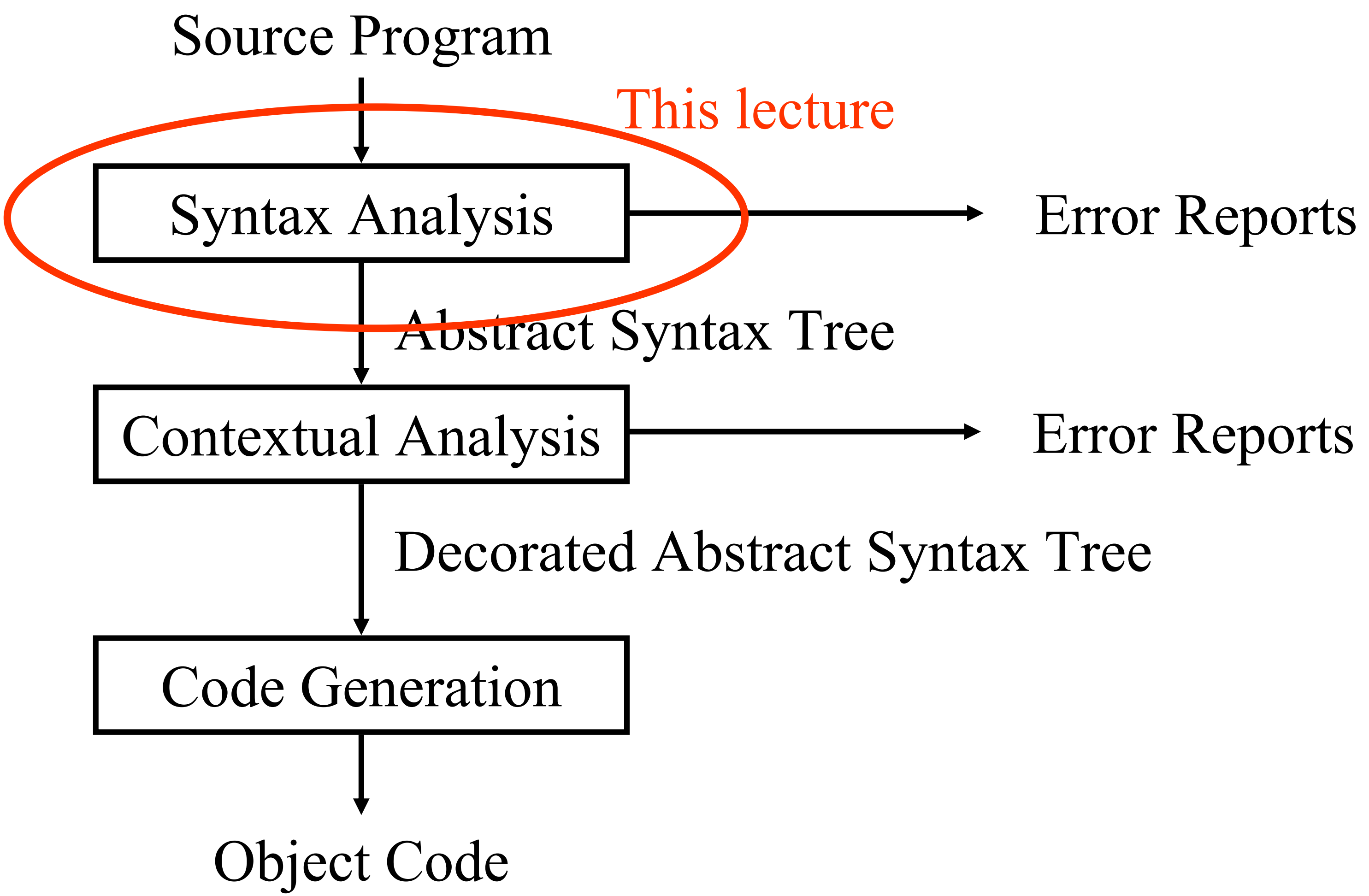
With acknowledgement to Norm Hutchinson whose slides this lecture is based on.

In This Lecture

- Syntax Analysis
  - (Scanning: recognize “words” or “tokens” in the input)
  - Parsing: recognize phrase structure
    - Different parsing strategies
    - How to construct a recursive descent parser
  - AST Construction
- Theoretical “Tools”:
  - Regular Expressions
  - Grammars
  - Extended BNF notation

Beware this lecture is a tour de force of the front-end, but should help you get started with your projects.

The “Phases” of a Compiler



Syntax Analysis

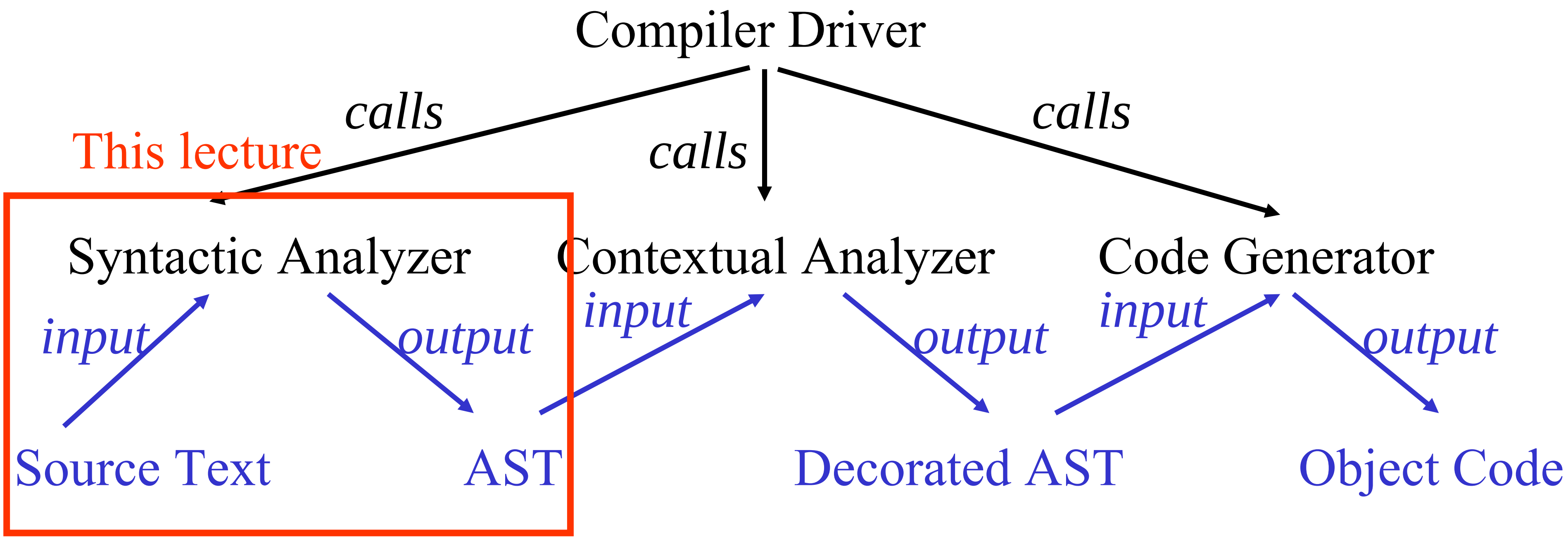
- The "job" of syntax analysis is to read the source text and determine its phrase structure.
- Subphases
  - Scanning
  - Parsing
  - Construct an internal representation of the source text that reifies the phrase structure (usually an AST)

Note: A single-pass compiler usually does not construct an AST.

Multi Pass Compiler

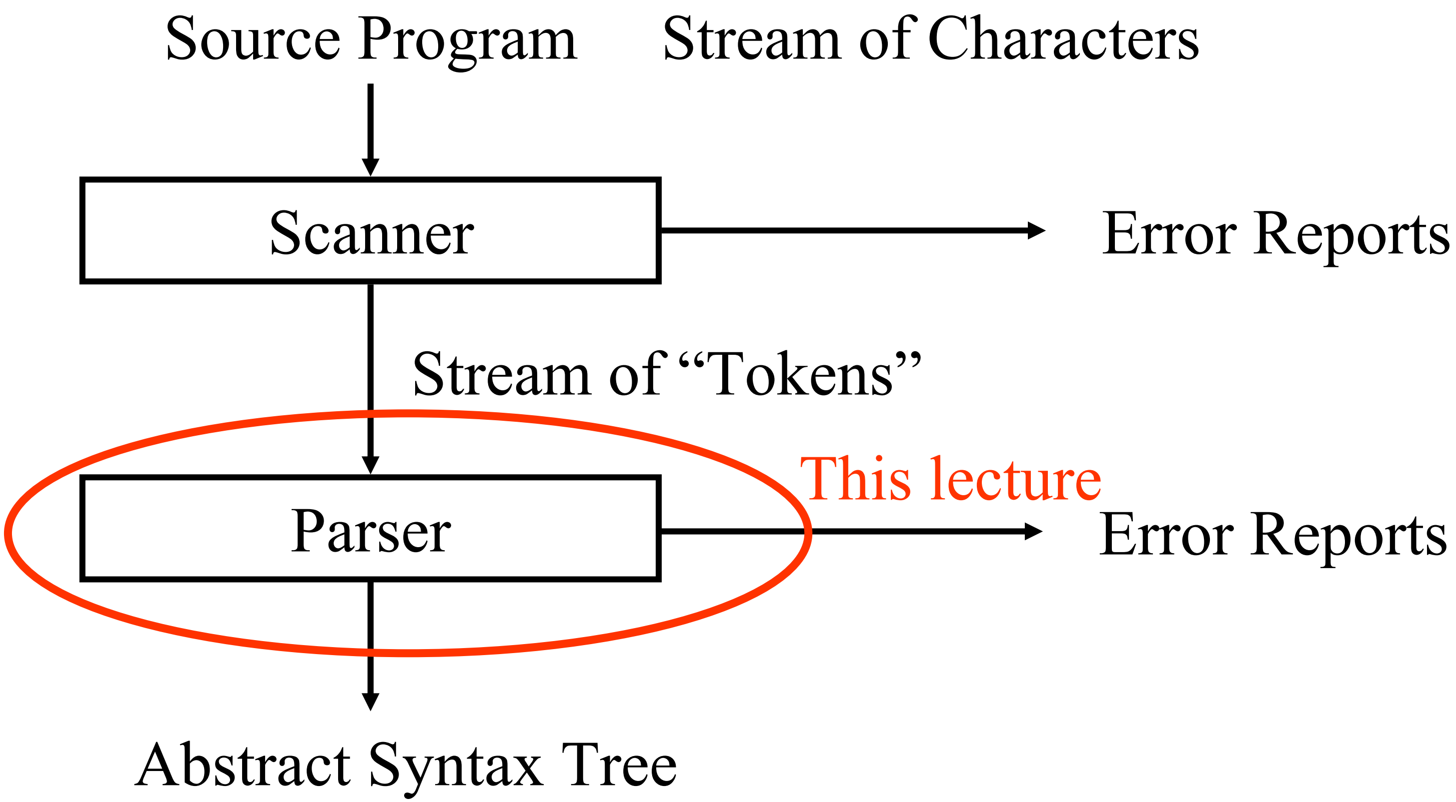
A multi pass compiler makes several passes over the program. The output of a preceding phase is stored in a data structure and used by subsequent phases.

Dependency diagram of a typical Multi Pass Compiler:



Syntax Analysis

Dataflow chart



1) Scan: Divide Input into Tokens

An example Mini Triangle source program:

```
let var y: Integer
in  !new year
   y := y+1
```

Tokens are “words” in the input, for example keywords, operators, identifiers, literals, etc.

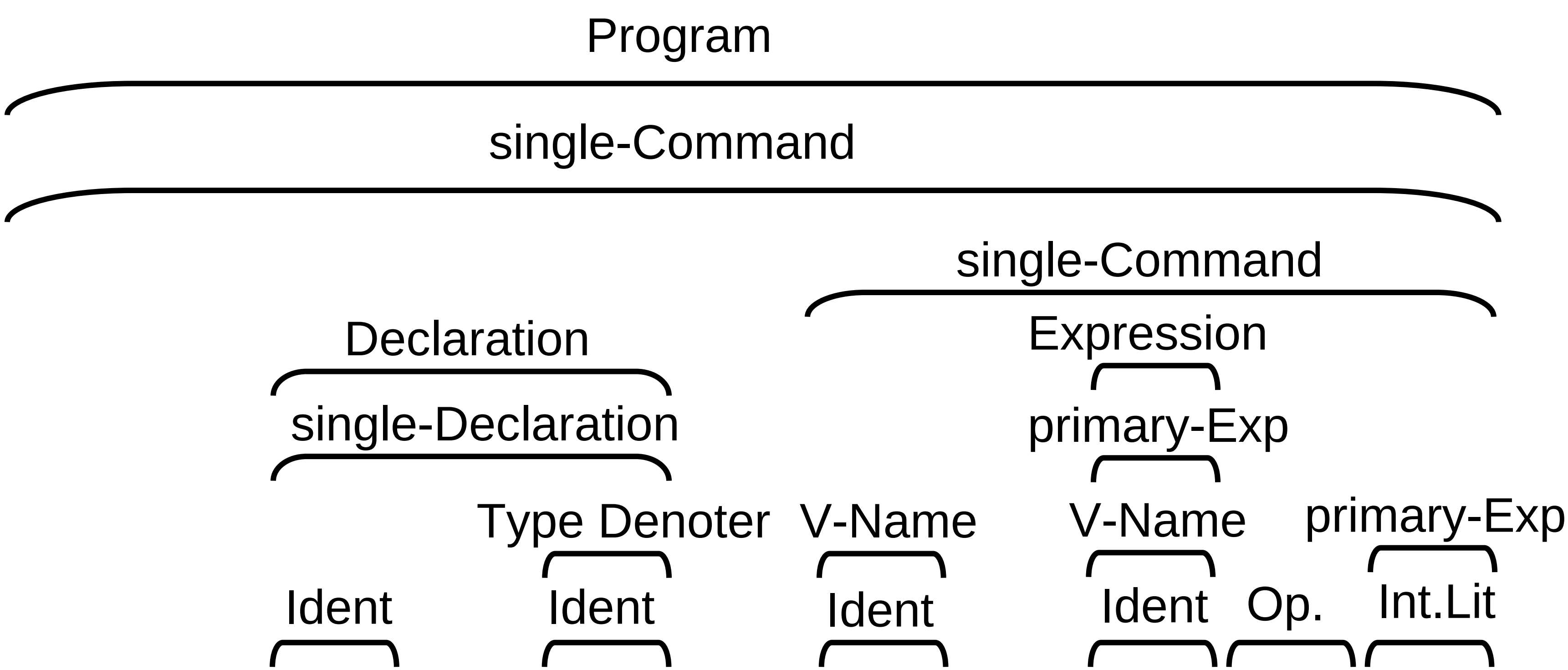
scanner

let	*var*	*ident.*	*colon*	*ident.*	*in*	
`let`	`var`	`y`	`:`	`Integer`	`in`	...

	ident.	*becomes*	*ident.*	*op.*	*intlit*	*eot*
...	`y`	`:=`	`y`	`+`	`1`	

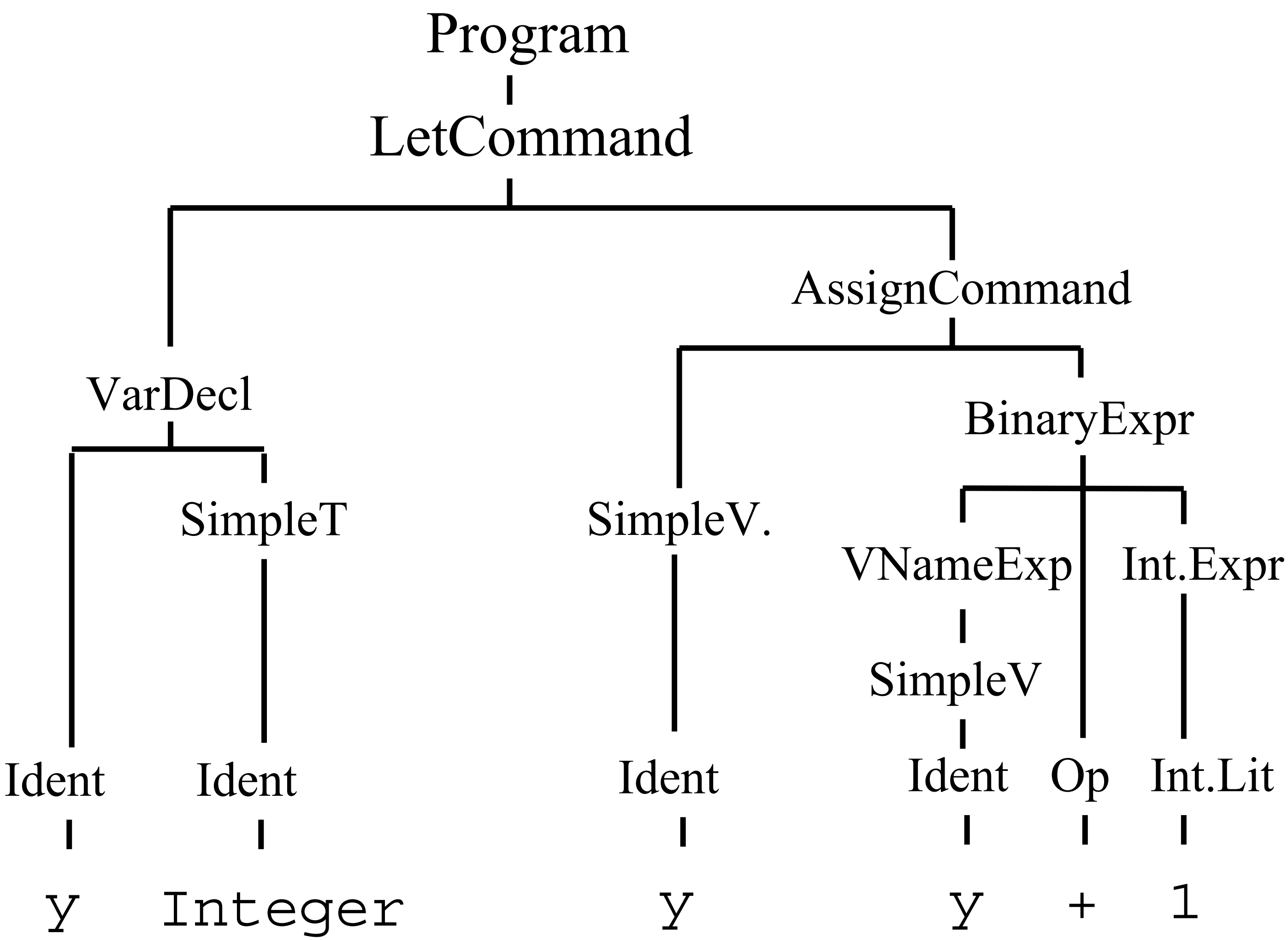
2) Parse: Determine "phrase structure"

Parser analyzes the phrase structure of the token stream with respect to the grammar of the language.



let	*var*	*id.*	*col.*	*id.*	*in*	*id.*	*bec.*	*id.*	*op*	*intlit*	*eot*
`let`	`var`	`y`	`:`	`Int`	`in`	`y`	`:=`	`y`	`+`	`1`	

3) AST Construction



Grammars

RECAP:

- The Syntax of a Language can be specified by means of a CFG (Context Free Grammar).
- CFG can be expressed in BNF (Backus-Naur Formalism)

Example: Mini Triangle grammar in BNF

```
Program ::= single-Command
Command ::= single-Command
          | Command ; single-Command
single-Command
      ::= V-name := Expression
        | begin Command end
        | ...
```

Grammars (ctd.)

For our convenience, we will use EBNF or “Extended BNF” rather than simple BNF.

EBNF = BNF + **regular expressions**

* means 0 or more occurrences of

Example: Mini Triangle in EBNF

```
Program ::= single-Command
Command ::= ( Command ; )* single-Command
single-Command
        ::= V-name := Expression
          | begin Command end
          | ...
```

Regular Expressions

- RE are a notation for expressing a set of strings of terminal symbols.

Different kinds of RE:

ε	The empty string
t	Generates only the string *t*
X Y	Generates any string *xy* such that *x* is generated by *x* and *y* is generated by *Y*
X | *Y*	Generates any string which is generated either by *X* or by *Y*
*X**	The concatenation of zero or more strings generated by *X*
(*X*)	For grouping,

Regular Expressions

- The "languages" that can be defined by RE and CFG have been extensively studied by theoretical computer scientists. These are some important conclusions / terminology
  - RE is a "weaker" formalism than CFG: Any language expressible by a RE can be expressed by CFG **but not the other way around!**
  - The languages expressible as RE are called regular languages
  - Generally: a language that exhibits "self embedding" cannot be expressed by RE.
  - Programming languages exhibit self embedding. (Example: an expression can contain an (other) expression).

Extended BNF

- Extended BNF combines BNF with RE
- A production in EBNF looks like

  `LHS ::= RHS`

  where LHS is a non terminal symbol and RHS is an **extended regular expression**
- An extended RE is just like a regular expression except it is composed of terminals and non terminals of the grammar.
- Simply put... EBNF adds to BNF the notation of
  - "(...)" for the purpose of grouping and
  - "*" for denoting "0 or more repetitions of … "
  - ("+" for denoting "1 or more repetitions of … ")
  - ("[…]" for denoting "(ε | …)")

Extended BNF: an Example

Example: a simple expression language

```
Expression ::=
  PrimaryExp (Operator PrimaryExp)*
PrimaryExp ::=
  Literal | Identifier | ( Expression )
Identifier ::= Letter (Letter|Digit)*
Literal ::= Digit Digit*
Letter ::= a | b | c | ... |z
Digit ::= 0 | 1 | 2 | 3 | 4 | ... | 9
```

A little bit of useful theory

- We will now look at a few useful bits of theory. These will be necessary later when we implement parsers.
  - Grammar transformations
    - A grammar can be transformed in a number of ways without changing the meaning (i.e. the set of strings that it defines)
  - The definition and computation of "starter sets"

1) Grammar Transformations

Left factorization

$$X\ Y \mid X\ Z \Rightarrow X\,(\,Y \mid Z\,)$$

X Y= ε Z

Example:

```
single-Command
   ::= V-name := Expression
     | if Expression then single-Command
     | if Expression then single-Command
                          else single-Command
```

```
single-Command
   ::= V-name := Expression
     | if Expression then single-Command
                ( ε | else single-Command)
```

1) Grammar Transformations (ctd)

Elimination of Left Recursion

N ::= X | N Y ⇒ N ::= X Y*

Example:

```
Identifier ::= Letter
             | Identifier Letter
             | Identifier Digit
```

⤷

```
Identifier ::= Letter
             | Identifier (Letter|Digit)
```

⤷

```
Identifier ::= Letter (Letter|Digit)*
```

1) Grammar Transformations (ctd)

Substitution of non-terminal symbols

$$N ::= X \qquad M ::= \alpha\ N\ \beta \quad \Rightarrow \quad N ::= X \qquad M ::= \alpha\ X\ \beta$$

Example:

```
single-Command
   ::= for contrVar := Expression
       to-or-dt Expression do single-Command
to-or-dt ::= to | downto
```

```
single-Command ::=
    for contrVar := Expression
    (to|downto) Expression do single-Command
```

2) Starter Sets

Informal Definition:

The starter set of a RE *X* is the set of terminal symbols that can occur as the start of any string generated by *X*

Example :

starters[(+|-|ε)(0|1|...|9)*] = {+,-, 0,1,…,9}

Formal Definition:

starters[ε] ={}

starters[*t*] ={*t*} (where *t* is a terminal symbol)

starters[*X Y*] = *starters*[*X*] ∪ *starters*[*Y*] (if *X* generates ε)

starters[*X Y*] = *starters*[*X*] (if not *X* generates ε)

starters[*X* | *Y*] = *starters*[*X*] ∪ *starters*[*Y*]

starters[*X**] = *starters*[*X*]

2) Starter Sets (ctd)

Informal Definition:

The starter set of RE can be generalized to extended BNF

Formal Definition:

starters[*N*] = *starters*[*X*] (for production rules N ::= X)

Example :

starters[Expression] = *starters*[PrimaryExp (Operator PrimaryExp)*]

= *starters*[PrimaryExp]

= *starters*[Identifiers] ∪ *starters*[(Expression)]

= *starters*[**a** | **b** | **c** | ... |**z**] ∪ {**(**}

= {**a**, **b**, **c**,…, **z**, **(**}

Parsing

Topics:

- Some terminology
- Different types of parsing strategies
  - bottom up
  - top down
- Recursive descent parsing
  - What is it
  - How to implement one given an EBNF specification
  - (How to generate one using tools – in Lecture 4)
- (Bottom up parsing algorithms – in Lecture 5)

Parsing: Some Terminology

- Recognition

  To answer the question "does the input conform to the syntax of the language"

- Parsing

  Recognition + determine phrase structure (for example by generating AST data structures)

- (Un)ambiguous grammar:

  A grammar is unambiguous if there is at most one way to parse any input. (i.e. for a syntactically correct program there is precisely one parse tree)

Different kinds of Parsing Algorithms

- Two big groups of algorithms can be distinguished:
  - bottom up strategies
  - top down strategies
- Example parsing of "Micro-English"

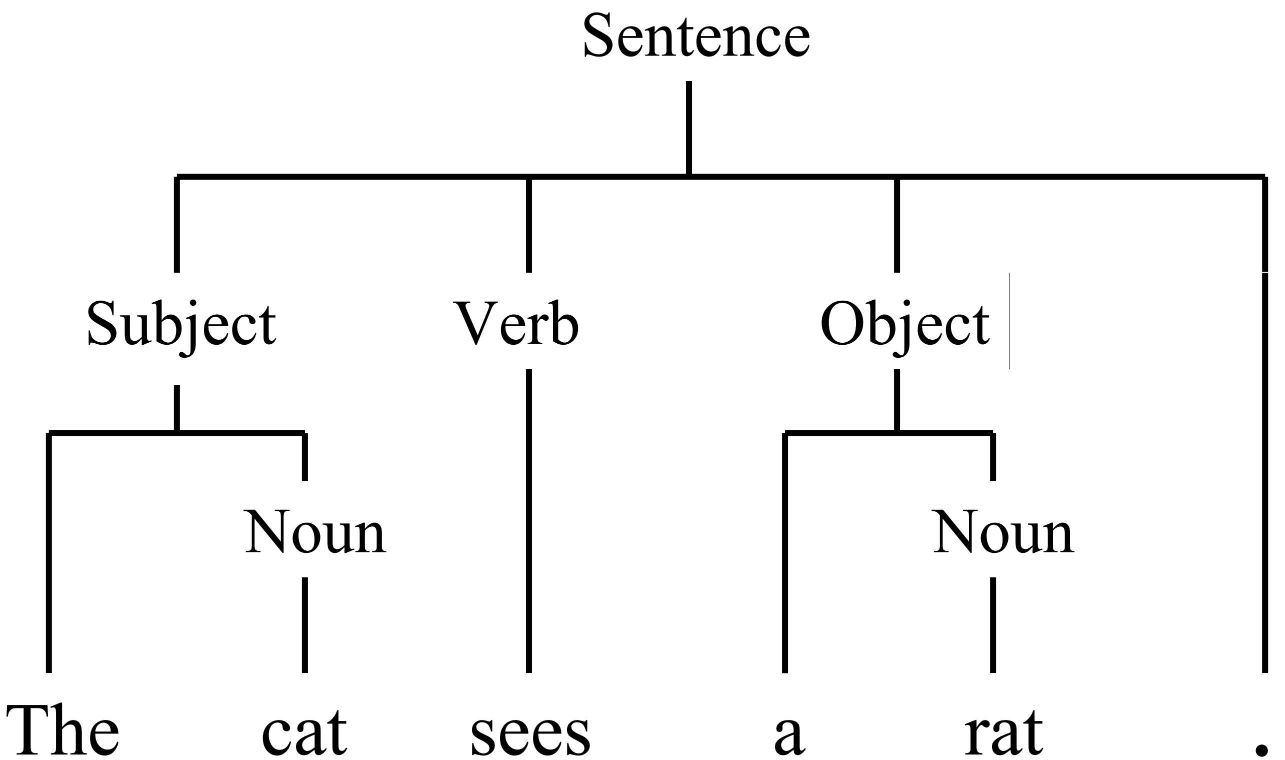
```
Sentence  ::= Subject Verb Object .
Subject   ::= I | a Noun | the Noun
Object    ::= me | a Noun | the Noun
Noun      ::= cat | mat | rat
Verb      ::= like | is | see | sees
```

The cat sees the rat.
The rat sees me.
I like a cat

The rat like me.
I see the rat.
I sees a rat.

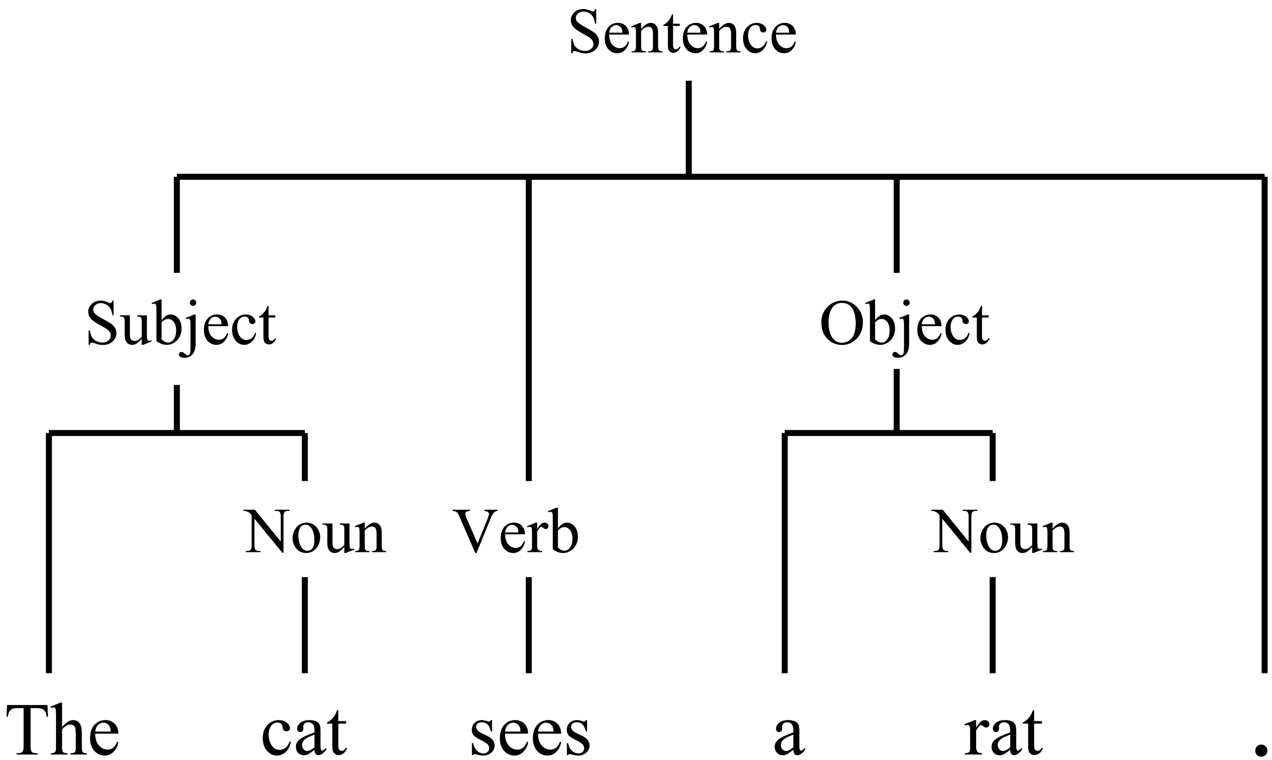
Top-down parsing

The parse tree is constructed starting at the top (root).



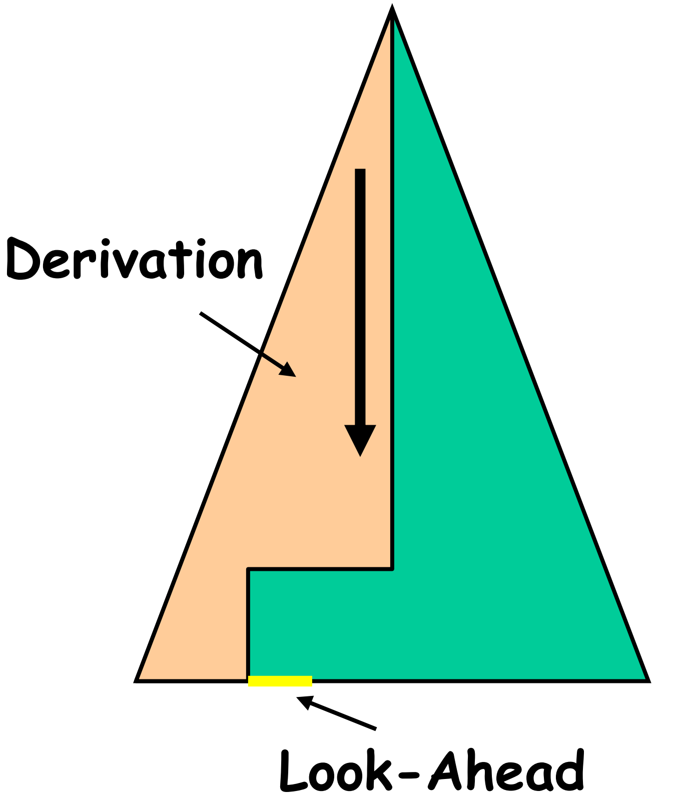
Bottom up parsing

The parse tree "grows" from the bottom (leaves) up to the top (root).

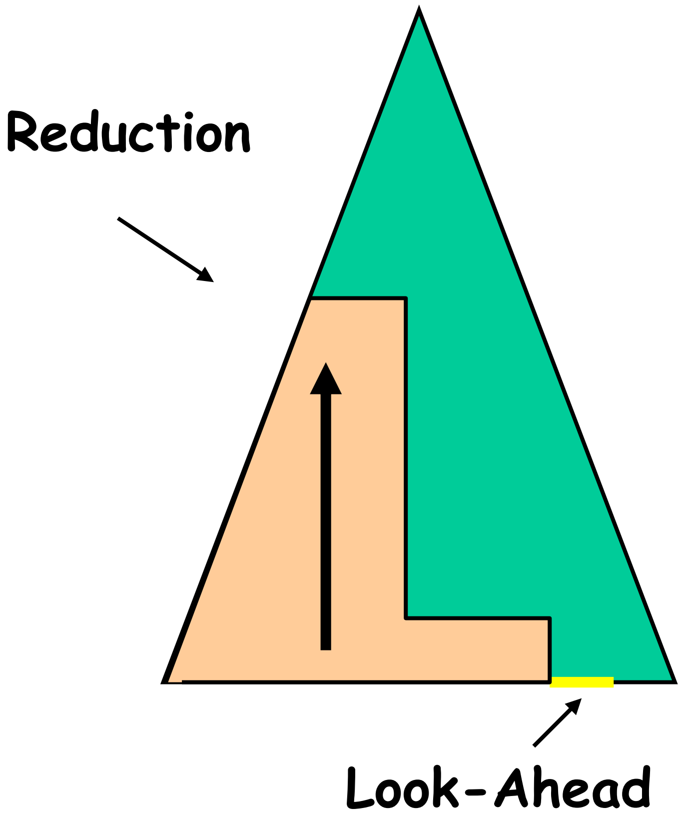


Top-Down vs. Bottom-Up parsing

LL-Analyse (Top-Down)
Left-to-Right Left Derivative



LR-Analyse (Bottom-Up)
Left-to-Right Right Derivative



Recursive Descent Parsing

- Recursive descent parsing is a straightforward top-down parsing algorithm.
- We will now look at how to develop a recursive descent parser from an EBNF specification.
- Idea: the parse tree structure corresponds to the "call graph" structure of parsing procedures that call each other recursively.

Recursive Descent Parsing

```
Sentence  ::= Subject Verb Object .
Subject   ::= I | a Noun | the Noun
Object    ::= me | a Noun | the Noun
Noun      ::= cat | mat | rat
Verb      ::= like | is | see | sees
```

Define a procedure parseN for each non-terminal N

```
private void parseSentence() ;
private void parseSubject();
private void parseObject();
private void parseNoun();
private void parseVerb();
```

Recursive Descent Parsing

```
public class MicroEnglishParser {

   private TerminalSymbol currentTerminal;

   //Auxiliary methods will go here
   ...

   //Parsing methods will go here
   ...
}
```

Recursive Descent Parsing: Auxiliary Methods

```
public class MicroEnglishParser {

   private TerminalSymbol currentTerminal

   private void accept(TerminalSymbol expected) {
      if (currentTerminal matches expected)
               currentTerminal = next input terminal ;
      else
               report a syntax error
   }

   ...
}
```

Recursive Descent Parsing: Parsing Methods

```
Sentence  ::= Subject Verb Object .
```

```
private void parseSentence() {
   parseSubject();
   parseVerb();
   parseObject();
   accept('.');
}
```

Recursive Descent Parsing: Parsing Methods

```
Subject     ::= I | a Noun | the Noun
```

```
private void parseSubject() {
   if (currentTerminal matches ‘I’)
      accept(‘I’);
   else if (currentTerminal matches ‘a’) {
      accept(‘a’);
      parseNoun();
   }
   else if (currentTerminal matches ‘the’) {
      accept(‘the’);
      parseNoun();
   }
   else
     report a syntax error
}
```

Recursive Descent Parsing: Parsing Methods

```
Noun        ::= cat | mat | rat
```

```
private void parseNoun() {
   if (currentTerminal matches ‘cat’)
      accept(‘cat’);
   else if (currentTerminal matches ‘mat’)
      accept(‘mat’);
   else if (currentTerminal matches ‘rat’)
      accept(‘rat’);
   else
     report a syntax error
}
```

Developing RD Parser for Mini Triangle

Before we begin:

- The following non-terminals are recognized by the scanner
- They will be returned as tokens by the scanner

```
Identifier := Letter (Letter|Digit)*
Integer-Literal ::= Digit Digit*
Operator ::= + | - | * | / | < | > | =
Comment ::= ! Graphic* eol
```

Assume scanner produces instances of:

```
public class Token {
   byte kind; String spelling;
   final static byte
       IDENTIFIER = 0,
       INTLITERAL = 1;
       ...
```

(1+2) Express grammar in EBNF and factorize...

```
Program ::= single-Command
Command ::= single-Command
          | Command ; single-Command
single-Command
      ::= V-name := Expression
        | Identifier ( Expression )
        | if Expression then single-Command
                         else single-Command
        | while Expression do single-Command
        | let Declaration in single-Command
        | begin Command end
V-name ::= Identifier
...
```

Left recursion elimination needed

Left factorization needed

(1+2)Express grammar in EBNF and factorize...

After factorization etc. we get:

```
Program ::= single-Command
Command ::= single-Command (;single-Command)*
single-Command
      ::= Identifier ( := Expression
                     | ( Expression ) )
        | if Expression then single-Command
                        else single-Command
        | while Expression do single-Command
        | let Declaration in single-Command
        | begin Command end
V-name ::= Identifier
...
```

Developing RD Parser for Mini Triangle

```
Expression
  ::= primary-Expression
    | Expression Operator primary-Expression
primary-Expression
  ::= Integer-Literal
    | V-name
    | Operator primary-Expression
    | ( Expression )
Declaration
  ::= single-Declaration
    | Declaration ; single-Declaration
single-Declaration
  ::= const Identifier ~ Expression
    | var Identifier : Type-denoter
Type-denoter ::= Identifier
```

Left recursion elimination needed

Left recursion elimination needed

(1+2) Express grammar in EBNF and factorize...

After factorization and recursion elimination :

```
Expression
  ::= primary-Expression
           ( Operator primary-Expression )*
primary-Expression
  ::= Integer-Literal
    | Identifier
    | Operator primary-Expression
    | ( Expression )
Declaration
::= single-Declaration (;single-Declaration)*
single-Declaration
  ::= const Identifier ~ Expression
    | var Identifier : Type-denoter
Type-denoter ::= Identifier
```

(3) Create a parser class with ...

```
public class Parser {
   private Token currentToken;
   private void accept(byte expectedKind) {
      if (currentToken.kind == expectedKind)
          currentToken = scanner.scan();
      else
          report syntax error
   }
   private void acceptIt() {
      currentToken = scanner.scan();
   }
   public void parse() {
      acceptIt(); //Get the first token
      parseProgram();
      if (currentToken.kind != Token.EOT)
          report syntax error
   }
   ...
```

(4) Implement private parsing methods:

```
Program ::= single-Command
```

```
private void parseProgram() {
   parseSingleCommand();
}
```

(4) Implement private parsing methods:

```
single-Command
      ::= Identifier ( := Expression
                     | ( Expression ) )
        | if Expression then single-Command
                        else single-Command
        | ... more alternatives ...
```

```
private void parseSingleCommand() {
   switch (currentToken.kind) {
     case Token.IDENTIFIER : ...
     case Token.IF : ...
     ... more cases ...
     default: report a syntax error
   }
}
```

(4) Implement private parsing methods:

```
single-Command
      ::= Identifier ( := Expression
                     | ( Expression ) )
        | if Expression then single-Command
                         else single-Command
        | while Expression do single-Command
        | let Declaration in single-Command
        | begin Command end
```

From the above we can straightforwardly derive the entire implementation of parseSingleCommand (much as we did in the microEnglish example)

Algorithm to convert EBNF into a RD parser

- The conversion of an EBNF specification into a Java implementation for a recursive descent parser is so "mechanical" that it can easily be automated!

=> JavaCC "Java Compiler Compiler"

- We can describe the algorithm by a set of mechanical rewrite rules

N ::= *X*

```
private void parseN() {
   parse X
}
```

Algorithm to convert EBNF into a RD parser

parse t where *t* is a terminal

```
accept(t);
```

parse N where *N* is a non-terminal

```
parseN();
```

parse ε

```
// a dummy statement
```

parse XY

parse X
parse Y

Algorithm to convert EBNF into a RD parser

*parse X**

```
while (currentToken.kind is in starters[X]) {
   parse X
}
```

parse X|Y

```
switch (currentToken.kind) {
   cases in starters[X]:
      parse X
      break;
   cases in starters[Y]:
      parse Y
      break;
   default: report syntax error
}
```

Example: “**Generation” of parseCommand**

Command ::= single-Command (; single-Command)*

```
private void parseCommand() {
  parseSingleCommand();
  while (currentToken.kind==Token.SEMICOLON) {
     acceptIt();
     parseSingleCommand();
  }
}
```

Example: **Generation of parseSingleDeclaration**

single-Declaration
  ::= **const** Identifier **~** Type-denoter
   | **var** Identifier **:** Expression

```
private void parseSingleDeclaration() {
  switch (currentToken.kind) {
     case Token.CONST:
       acceptIt();
       parseIdentifier();
       acceptIt(Token.IS);
       parseTypeDenoter();
     case Token.VAR:
       acceptIt();
       parseIdentifier();
       acceptIt(Token.COLON);
       parseExpression();
     default: report syntax error
  }
}
```

LL(1) Grammars

- The presented algorithm to convert EBNF into a parser does not work for all possible grammars.
- It only works for so called LL(1) grammars.
- What grammars are LL(1)?
- Basically, an **LL(1) grammar** is a grammar which can be parsed with a **top-down parser** with a **lookahead** (in the input stream of tokens) of **one token**.

How can we recognize that a grammar is (or is not) LL(1)?

⇒There is a formal definition which we will skip for now

⇒We can deduce the necessary conditions from the parser generation algorithm.

LL(1) Grammars

*parse X**

```
while (currentToken.kind is in starters[X]) {
   parse X
}
```

Condition: *starters*[*X*] must be disjoint from the set of tokens that can immediately follow *X* *

parse X|Y

```
switch (currentToken.kind) {
   cases in starters[X]:
      parse X
      break;
   cases in starters[Y]:
      parse Y
      break;
   default: report syntax error
}
```

Condition: *starters*[*X*] and *starters*[*Y*] must be disjoint sets.

LL(1) grammars and left factorisation

The original Mini Triangle grammar is **not** LL(1):

For example:

```
single-Command
      ::= V-name := Expression
        | Identifier ( Expression )
        | ...
V-name ::= Identifier
```

Starters[V-name := Expression]
= *Starters*[V-name] = *Starters*[Identifier]

Starters[Identifier (Expression)]
= *Starters*[Identifier]

NOT DISJOINT!

LL(1) grammars: left factorisation

What happens when we generate a RD parser from a non LL(1) grammar?

```
single-Command
        ::= V-name := Expression
          | Identifier ( Expression )
          | ...
```

```
private void parseSingleCommand() {
 switch (currentToken.kind) {
   case Token.IDENTIFIER:
    parse V-name := Expression
   case Token.IDENTIFIER:
    parse Identifier ( Expression )
      ...other cases...
   default: report syntax error
 }
}
```

wrong: overlapping cases

LL(1) grammars: left factorisation

```
single-Command
       ::= V-name := Expression
         | Identifier ( Expression )
         | ...
```

Left factorisation (and substitution of V-name)

```
single-Command
       ::= Identifier ( := Expression
                      | ( Expression ) )
          | ...
```

LL1 Grammars: left recursion elimination

```
Command ::= single-Command
          | Command ; single-Command
```

What happens if we don't perform left-recursion elimination?

```
public void parseCommand() {
 switch (currentToken.kind) {
    case in starters[single-Command]
      parseSingleCommand();
    case in starters[Command]
      parseCommand();
      accept(Token.SEMICOLON);
      parseSingleCommand();
    default: report syntax error
  }
}
```

wrong: overlapping cases

LL1 Grammars: left recursion elimination

```
Command ::= single-Command
          | Command ; single-Command
```

Left recursion elimination

```
Command
    ::= single-Command (; single-Command)*
```

Systematic Development of RD Parser

(1) Express grammar in EBNF

(2) Grammar Transformations:

Left factorization and Left recursion elimination

(3) Create a parser class with

- private variable `currentToken`
- methods to call the scanner: `accept` and `acceptIt`

(4) Implement private parsing methods:

- add private `parse`***N*** method for each non terminal ***N***
- public `parse` method that
  - gets the first token form the scanner
  - calls `parse`***S*** (***S*** is the start symbol of the grammar)

Abstract Syntax Trees

- So far we have talked about how to build a recursive descent parser which **recognizes** a given language described by an LL(1) EBNF grammar.
- Now we will look at
  - how to represent AST as data structures.
  - how to refine a recognizer to construct an AST data structure.

AST Representation: Possible Tree Shapes

The possible form of AST structures is completely determined by an AST grammar (as described before in lecture 1-2)

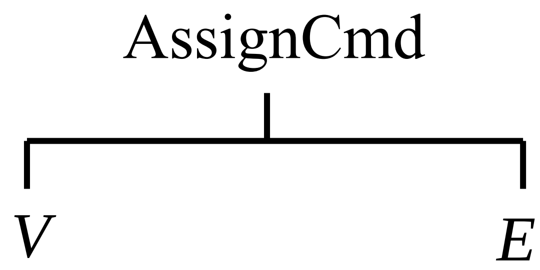
Example: remember the Mini Triangle abstract syntax

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

AST Representation: Possible Tree Shapes

Example: remember the Mini Triangle AST (excerpt below)

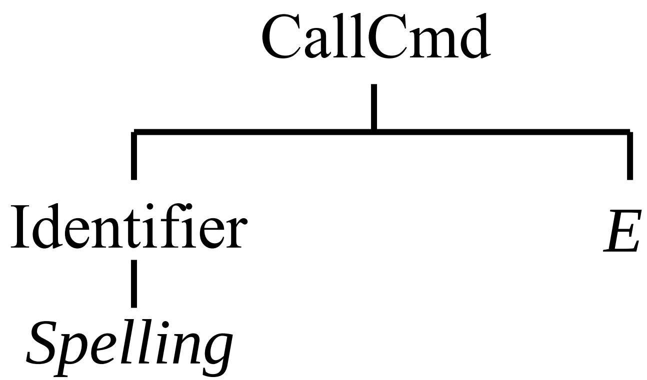
```
Command ::= VName := Expression        AssignCmd
          | ...
```



AST Representation: Possible Tree Shapes

Example: remember the Mini Triangle AST (excerpt below)

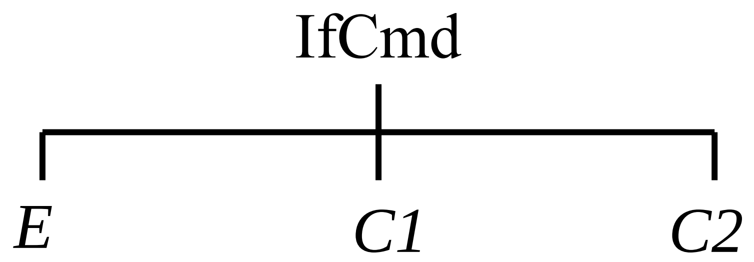
```
Command ::=
      ...
      | Identifier ( Expression )          CallCmd
      ...
```



AST Representation: Possible Tree Shapes

Example: remember the Mini Triangle AST (excerpt below)

```
Command ::=
      ...
      | if Expression then Command
                        else Command        IfCmd
      ...
```



AST Representation: Java Data Structures

Example: Java classes to represent Mini Triangle AST's

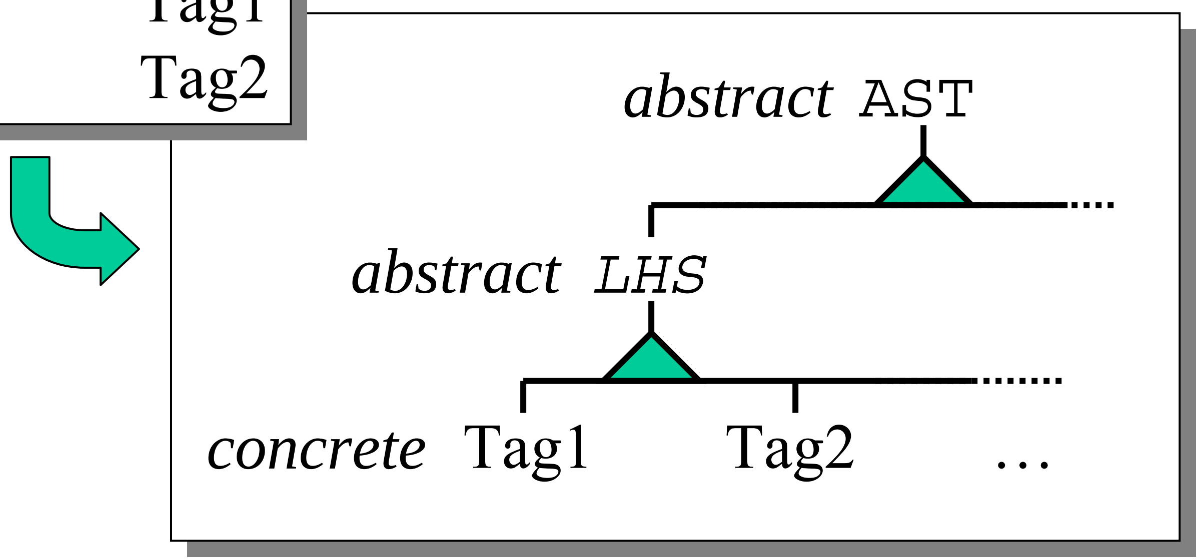
1) A common (abstract) super class for all AST nodes

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

2) A Java class for each "type" of node.

- abstract as well as concrete node types

```
LHS ::= ...        Tag1
      | ...        Tag2
```



Example: Mini Triangle Commands ASTs

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

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

public class AssignCommand extends Command { ... }
public class CallCommand extends Command { ... }
public class IfCommand extends Command { ... }
etc.
```

Example: Mini Triangle Command ASTs

```
Command ::= V-name := Expression          AssignCmd
    | Identifier ( Expression )           CallCmd
    | ...
```

```
public class AssignCommand extends Command {
   public Vname V;          // assign to what variable?
   public Expression E;     // what to assign?
   ...
}

public class CallCommand extends Command {
   public Identifier I;     //procedure name
   public Expression E;     //actual parameter
   ...
}
...
```

AST Terminal Nodes

```
public abstract class Terminal extends AST {
   public String spelling;
   ...
}

public class Identifier extends Terminal { ... }

public class IntegerLiteral extends Terminal { ... }

public class Operator extends Terminal { ... }
```

AST Construction

First, every concrete AST class needs a constructor.

Examples:

```
public class AssignCommand extends Command {
   public Vname V;         // Left side variable
   public Expression E;    // right side expression
   public AssignCommand(Vname V; Expression E) {
      this.V = V; this.E=E;
   }
   ...
}

public class Identifier extends Terminal {
   public class Identifier(String spelling) {
      this.spelling = spelling;
   }
   ...
}
```

AST Construction

We will now show how to refine our recursive descent parser to actually construct an AST.

```
N ::= X
```

```
private N parseN() {
  N itsAST;
  parse X at the same time constructing itsAST
  return itsAST;
}
```

Example: Construction of Mini Triangle ASTs

Command ::= single-Command (**;** single-Command)*

```
// AST-generating version
private Command parseCommand() {
  Command itsAST;
  itsAST = parseSingleCommand();
  while (currentToken.kind==Token.SEMICOLON) {
    acceptIt();
    Command extraCmd = parseSingleCommand();
    itsAST = new SequentialCommand(itsAST,extraCmd);
  }
  return itsAST;
}
```

Example: Construction of Mini Triangle ASTs

```
single-Command
      ::= Identifier ( := Expression
                     | ( Expression ) )
        | if Expression then single-Command
                        else single-Command
        | while Expression do single-Command
        | let Declaration in single-Command
        | begin Command end
```

```
private Command parseSingleCommand() {
  Command comAST;
  parse it and construct AST
  return comAST;
}
```

Example: Construction of Mini Triangle ASTs

```
private Command parseSingleCommand() {
  Command comAST;
  switch (currentToken.kind) {
    case Token.IDENTIFIER:
      parse Identifier ( := Expression
                       | ( Expression ) )
    case Token.IF:
      parse if Expression then single-Command
                       else single-Command
    case Token.WHILE:
      parse while Expression do single-Command
    case Token.LET:
      parse let Declaration in single-Command
    case Token.BEGIN:
      parse begin Command end
  }
  return comAST;
}
```

```
...
case Token.IDENTIFIER:
  //parse Identifier ( := Expression
  //                 | ( Expression ) )
  Identifier iAST = parseIdentifier();
  switch (currentToken.kind) {
    case Token.BECOMES:
       acceptIt();
       Expression eAST = parseExpression();
       comAST = new AssignmentCommand(iAST,eAST);
       break;
    case Token.LPAREN:
       acceptIt();
       Expression eAST = parseExpression();
       comAST = new CallCommand(iAST,eAST);
       accept(Token.RPAREN);
       break;
  }
  break;
...
```

Example: Construction of Mini Triangle ASTs

```
    ...
    break;
case Token.IF:
    //parse if Expression then single-Command
    //                   else single-Command
    acceptIt();
    Expression eAST = parseExpression();
    accept(Token.THEN);
    Command thnAST = parseSingleCommand();
    accept(Token.ELSE);
    Command elsAST = parseSingleCommand();
    comAST = new IfCommand(eAST,thnAST,elsAST);
    break;
case Token.WHILE:
    ...
```

Example: Construction of Mini Triangle ASTs

```
    ...
    break;
  case Token.BEGIN:
    //parse begin Command end
    acceptIt();
    comAST = parseCommand();
    accept(Token.END);
    break;
  default:
    report a syntax error;
 }
 return comAST;
}
```

Syntax Error Handling

- **Example:**

```
let
var x:Integer;
var y:Integer;
func max(i:Integer  ;   j:Integer) : Integer;
! return maximum of integers I and j
begin
 if I > j then max := I   ;
 else max := j
end;
in
getint (x);getint(y);
      puttint (max(x,y))
end.
```

Common Punctuation Errors

- Using a semicolon instead of a comma in the argument list of a function declaration (line 4) and ending the line with semicolon
- Leaving out a mandatory tilde (~) at the end of a line (line 4)
- Undeclared identifier I (should have been i) (line 7)
- Using an extraneous semicolon before an else (line 7)
- Common Operator Error : Using = instead of := (line 7 or 8)
- Misspelling keywords : puttint instead of putint (line 12)
- Missing begin or end (line 9 missing), usually difficult to repair.

Error Reporting

- A common technique is to print the offending line with a pointer to the position of the error.
- The parser might add a diagnostic message like "semicolon missing at this position" if it knows what the likely error is.
- The way the parser is written may influence error reporting is:

```
private void parseSingleDeclaration () {
        switch (currentToken.kind) {
        case Token.CONST: {
                acceptIT();
                …
        }
        break;
        case Token.VAR: {
                acceptIT();
                …
        }
        break;
        default:
                report a syntax error
        }
}
```

Error Reporting

```
private void parseSingleDeclaration () {
        if (currentToken.kind == Token.CONST) {
                acceptIT();
                …
        } else {
                acceptIT();
                …
        }
}
```

Ex: d ~ 7 above would report missing **var** token

How to handle Syntax errors

- Error Recovery : The parser should try to recover from an error quickly so subsequent errors can be reported. If the parser doesn't recover correctly it may report spurious errors.

- Possible strategies:
  - Panic-mode Recovery
  - Phase-level Recovery
  - Error Productions

Panic-mode Recovery

- Discard input tokens until a synchronizing token (like; or end) is found.
- Simple but may skip a considerable amount of input before checking for errors again.
- Will not generate an infinite loop.

Phase-level Recovery

- Perform local corrections
- Replace the prefix of the remaining input with some string to allow the parser to continue.
  - Examples: replace a comma with a semicolon, delete an extraneous semicolon or insert a missing semicolon. Must be careful not to get into an infinite loop.

Recovery with Error Productions

- Augment the grammar with productions to handle common errors

- Example:

```
param_list
::= identifier_list : type
|   param_list, identifier_list : type
|   param_list; error identifier_list : type
   ("comma should be a semicolon")
```

Quick review

- Syntactic analysis
  - Prepare the grammar
    - Grammar transformations
      - Left-factoring
      - Left-recursion removal
      - Substitution
  - (Lexical analysis)
    - Next lecture
  - Parsing - Phrase structure analysis
    - Group words into sentences, paragraphs and complete programs
    - Top-Down and Bottom-Up
    - Recursive Decent Parser
    - Construction of AST

Note: You will need (at least) two grammars

- One for Humans to read and understand
- (may be ambiguous, left recursive, have more productions than necessary, …)
- One for constructing the parser