Derivation

- Given a BNF grammar called $G$ and a grammatical category called $C$.

- A derivation w.r.t $G$ is a sequence:

  $C \Rightarrow C_1 \Rightarrow C_2 \Rightarrow \ldots$

  where each instance of $\Rightarrow$ denotes the application of a *single* rule of $G$.

  - One wants a derivation to **terminate** and the **last right hand** of a derivation to be composed of **terminals only**.

- **Example:** Consider the following BNF grammar:

  **Integer** $\rightarrow$ Digit $|$ Integer Digit

  Digit $\rightarrow$ 0 $|$ 1 $|$ 2 $|$ 3 $|$ 4 $|$ 5 $|$ 6 $|$ 7 $|$ 8 $|$ 9 $|$ 10

  Integer $\Rightarrow$ Integer Digit $\Rightarrow$ Digit Digit $\Rightarrow$ 3 Digit $\Rightarrow$ 32 is a derivation.

- Each string on the right of a $\Rightarrow$ is called a **sentential form**.

  **Example:** Integer Digit, Digit Digit, 3 Digit and 32 are sentential forms.
• A **left-most derivation** is a derivation in which the left-most nonterminal in the sentential form is replaced at each step.

  **Example:** Integer $\Rightarrow$ Integer Digit $\Rightarrow$ Digit Digit $\Rightarrow$ 3 Digit $\Rightarrow$ 32 is a left-most derivation.

• A **right-most derivation** is a derivation in which the right-most nonterminal in the sentential form is replaced at each step.

  **Example:** Integer $\Rightarrow$ Integer Digit $\Rightarrow$ Integer 2 $\Rightarrow$ Digit 2 $\Rightarrow$ 32 is a right-most derivation.
Parse tree

- A parse tree is a graphical representation of a derivation.
  - The root node of a parse tree is the particular grammatical category of interest (here $C'$).
  - The internal nodes of a parse tree are grammatical categories (left hand sides of rules of $G'$).
  - The leaves of a parse tree are terminals.

- The following tree is a parse tree:

```
   Integer
  /      \
 Integer   Digit
 /  \      /  \
 Digit  5  Digit  2
        /    /  \
  3     5    2
```

```
Language

- Given a BNF grammar called $G$ with start symbol called $S$.

- **Parsing** a string $s$ to check if $s$ is an instance of the grammatical category called $C$ from $G$ can be done:
  
  - Using a **derivation** (Is there a derivation $C \Rightarrow \ldots \Rightarrow s$?)
  
  - Using a **Parse tree** (Is there a parse tree with root $C$, such that reading the leaves from left to right reconstructs the string $s$?)

- The **Language** defined by a BNF grammar is that set of *all* strings that can be parsed or derived using the rules of the grammar (starting from $S$).

- **Property:** If $s$ is a string of the language $L$ described by $G$, there is a derivation:

  $S \Rightarrow \ldots \Rightarrow s$

  and a parse tree with root $S$ and reading the leaves from left to right reconstructs $s$.

  The number of internal node of the parse tree is equal to the number steps needed to derived $s$ from $S$. 

• BNF grammar:
  \textbf{Integer} \rightarrow \text{Digit} | \text{Integer Digit}
  \text{Digit} \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

• Justify that 352 is an \textit{Integer}.
  – First method (left-most derivation): \text{Integer} \Rightarrow \text{Integer Digit} \Rightarrow \text{Integer Digit Digit} \Rightarrow \text{Digit Digit Digit} \Rightarrow 3 \text{Digit Digit} \Rightarrow 35 \text{Digit} \Rightarrow 352
  – Second method (right-most derivation): \text{Integer} \Rightarrow \text{Integer Digit} \Rightarrow \text{Integer 2} \Rightarrow \text{Integer Digit 2} \Rightarrow \text{Integer 5 2} \Rightarrow \text{Digit 5 2} \Rightarrow 352
  – Third method (Parse tree):

\[
\begin{array}{c}
\text{Integer} \\
\text{Integer} \quad \text{Digit} \\
\text{Digit} \quad 5 \\
\text{Digit} \quad 3
\end{array}
\]
A grammar is said **ambiguous** if it permits a string in its language to be parsed into two or more parse trees.

- Ambiguous grammars should be avoided.

**Example 1:** The following grammar is ambiguous.

Expression → 0 | 1 | Expression − Expression

because 1 − 0 − 1 can be parsed into 2 parse trees.

**Example 2: Dangling-else ambiguity**

The following grammar is ambiguous.

Statement → if Expression then Statement | if Expression then Statement else Statement

Expression → ...

*if E1 then if E2 then S1 else S2* can be parsed in 2 parse trees.

**Example 3:** The following grammar is ambiguous.

Assignment → Variable = Expression

Expression → variable | Expression + Expression

Variable → x | y | z

\(x = x + y + z\) can be parsed in 2 parse trees.
Solving Ambiguity

- To solve the ambiguity:
  - Use an explicit and formal specification outside the BNF grammar considering the properties of some symbols of the grammar.
  
  Example: left-associativity, right-associativity, precedence (priority) on symbols ...
  - Use an explicit and non formal specification outside the BNF syntax.
  
  Example: The language designer can stipulate the extra-grammatical rule that every *else* clause will be associated with the textually closest preceding *if* statement. If a different attachment is desired, the programmer can always make it clear (by inserting braces for example).
  - Redesign the BNF grammar.

- Example 1:
  
  Expression → 0 | 1 | Expression − Expression
  
  To solve ambiguity we use the fact that − is left-associative.
• **Example 2: Dangling Else**

How is it solve in programming languages?

- C and C++ stipulate the extra-grammatical rule that every `else` clause will be associated with the textually closest preceding `if` statement.
- SR and ADA provide a special keyword `fi` (end if).
- JAVA expands the BNF grammar with the following rules:

  - `IfThenStatement → if (Expression) Statement`
  - `IfThenStatementStatement → if (Expression) StatementNoShortIf else Statement`

- JAY has the following BNF grammar for conditionals:

  - `Statement → ; | Block | Assignment | IfStatement | WhileStatement`
  - `Block → {Statement}`
  - `IfStatement → if (Expression) Statement | if (Expression) Statement else Statement`
• **Example 3:**
  Assignment → Variable = Expression
  Expression → Variable | Expression + Expression
  Variable → x | y | z
  To solve ambiguity we use the fact that + is left-associative.
Regular Expressions

- An alternative to BNF for specifying a language is the use of regular expressions.

- Conventions for Writing Regular Expressions:

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>A character (stands for itself)</td>
</tr>
<tr>
<td>&quot;xyz&quot;</td>
<td>A literal string (stands for itself)</td>
</tr>
<tr>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>M N</td>
<td>M followed by N (concatenation)</td>
</tr>
<tr>
<td>M*</td>
<td>Zero or more occurrences of M</td>
</tr>
<tr>
<td>M+</td>
<td>One or more occurrences of M</td>
</tr>
<tr>
<td>M?</td>
<td>Zero or one occurrence of M</td>
</tr>
<tr>
<td>[a-zA-Z]</td>
<td>Any alphabetic character</td>
</tr>
<tr>
<td>[0-9]</td>
<td>Any digit</td>
</tr>
<tr>
<td>.</td>
<td>Any single character</td>
</tr>
</tbody>
</table>
Examples

• “true” | “false” is a regular expression to describe Boolean values.

• \([a-zA-Z][a-zA-Z0-9]*\) is a regular expression to describe Identifiers composed of letters and digits only. An Identifier begins with a letter.

• “//\([a-zA-Z0-9]*\)(return)” is a regular expression to describe Comments as a series of characters introduced by // and followed by a return.

• The language containing strings of the form \(a^n b^n\) cannot be generated by a regular expression. Can it be generated by a BNF grammar?
EBNF

- Extended BNF

- EBNF was introduced to simplify the specification of recursion in grammar rules and to introduce the idea of an optional part in a rule’s right-hand side.

- EBNF uses Regular Expressions.

**Example:** The following BNF rules:

Expression → Term | Expression + Term | Expression − Term

Term → Factor | Term * Factor | Term / Factor

Factor → Identifier | Literal | (Expression)

can be written equivalently using EBNF rules the following way:

Expression → Term { [+ | -] Term }*

Term → Factor {[ ’*’ | / ] Factor }*

Factor → Identifier | Literal | (Expression)

- EBNF definitions tend to be slightly clearer and briefer than BNF definitions.

- The star notation (*) in EBNF definitions suggests a loop rather than recursion.
Consider the following EBNF grammar:

- **Expression** → **Term** \{[\+ | -] **Term**\}*
- **Term** → **Factor**\{'\*\' | /\}Factor\}*
- **Factor** → **Identifier** | **Literal** | (**Expression**)

• EBNF-Based Parse tree for the expression \( x + 2 \ast y \)
Syntax diagrams

- **Syntax Diagrams** represent another alternative for specifying a language.

- This representation was famous because of PASCAL.

- Example:

```
Expression ———> Term ———> + ———> Term ———> Expression
```
Compilation Process
• High-level languages must be translated to machine language prior to execution.

• This is done using a software called a **compiler**.

• **Compilers** are complex software to design and implement.

• Why?
  To one high-level language statement correspond many machine language or assembly language statements.
  High-level languages are **one-to-many**

  *Whereas:*
  To one assembly language statement correspond one machine language statement.
  Assembly languages are **one-to-one**.

• Translation must be **correct**.
  The machine language program is a **correct** translation of the high-level language program. (They do the same thing).

• The translated code must be **efficient** and **concise** (speed and size of the compiled program).
Major Stages in the Compiling Process
Major Stages in the Compiling Process

- **Lexical Analysis** translates the program text into a stream of **Tokens**, passing the individual tokens one-by-one to the syntactic analysis stage.

- **Syntactic Analysis** develops an abstract representation or **Parse** for the program, detecting syntactic errors along the way.

- Absent syntactic errors, the **semantic analysis and optimization** stages analyze the parse for semantic consistency and transform the parse so that it can efficiently utilize the architecture where the program will run.

- The **code generation** stage uses the resultant abstract representation as a basis for generating executable machine code and optimizes it.
Major Stages in the Interpreting Process

1. **Program**
2. **Analysis**
3. **Abstract program**
4. **Interpreter**

**Input**

**Output**
Lexical Analysis
Lexical Syntax

- The **Lexicon** of a programming language is the set of all grammatical categories that define strings of nonblank characters, called **Tokens**, from which programs are written.
  - **Identifiers, Literals** (Example: integer numbers), Operators, Separators (;,.,{,},...) and Keywords (int, main...) are the tokens of most programming languages.

- A **Token** is described by a **type** (Identifier, Literal,...) and by a **value** (the string it represents).
  Example: The token \( t \) is an Identifier (its type) and its value is \( x \).

- The **Lexical Syntax** of a programming language is defined by the lexicon of the programming language.
  - The lexical syntax of a programming language may be defined by a BNF grammar.

*Token class or Token category*
Lexical Syntax of JAY

Appendix B on page 351

InputElement → WhiteSpace | Comment | Token
WhiteSpace → space | \t | \r | \n | \f | \r\n
Comment → // any sequence of characters followed by \r, \n, or \r\n
Token → Identifier | Keyword | Literal | Separator | Operator
Identifier → Letter | Identifier Letter | Identifier Digit
Letter → a | b | . . . | z | A | B | . . . | Z
Digit → 0 | 1 | 2 | . . . | 9
Keyword → boolean | else | if | int |
          | main | void | while
Literal → Boolean | Integer
Boolean → true | false
Integer → Digit | Integer Digit
Separator → ( | ) | { | } | ; | .
Operator → = | + | - | * | / |
          < | <= | > | >= | == | != | && | || | !

Note: Letter, Digit... can be described by regular expressions.
Lexical Syntax of JAVA

- The 5 lexical classes of JAVA form the basis for JAVA’s lexical syntax.

- JAVA identifiers are made up of JavaLetters, which include A-Z, a-z, _ and $.

- The JAVA keywords are 47 in all.

- JAVA literals falls into the following classes: Integer, Boolean, FloatingPoint, Character, String, and Null.

Lexical Analysis

- **Lexical analysis** transforms a program into a stream of tokens. Non essential strings (blanks, new lines, comments...) are discarded.

- **Example:**
  Consider the following JAY program.

```java
// First JAY Program
void main(){
    int n;
    n=8;
}
```

The following stream of tokens is associated to the JAY program above:

- Type of the token 1: Keyword
- Value of the token 1: void
- Type of the token 2: Keyword
- Value of the token 2: main
- Type of the token 3: Separator
- Value of the token 3: (  
- Type of the token 4: Separator
- Value of the token 4: )
- Type of the token 5: Separator
Value of the token 5: {
Type of the token 6: Keyword
Value of the token 6: int
Type of the token 7: Identifier
Value of the token 7: n
Type of the token 8: Separator
Value of the token 8: ;
Type of the token 9: Identifier
Value of the token 9: n
Type of the token 10: Operator
Value of the token 10: =
Type of the token 11: Literal
Value of the token 11: 8
Type of the token 12: Separator
Value of the token 12: ;
Type of the token 13: Separator
Value of the token 13: }
Lexical Analysis of JAY in JAVA

Token.java class:

class Token {
    public String type; // Token type: Identifier Keyword...
    public String value; // Token value
}

TokenStream.java:

public class TokenStream {

    private boolean isEof = false;
    // next character in input stream
    private char nextChar = ' ';
    private BufferedReader input;

    // Pass a filename for the program text as a source for
    // the TokenStream
    public TokenStream (String fileName) {
        try {
            input = new BufferedReader
                (new FileReader(fileName));
        }
        catch (FileNotFoundException e) {
            System.out.println("File not found: " + fileName);
            System.exit(1);
        }
    }
}
// Return next token type and value
public Token nextToken() {
    Token t = new Token();
    t.type = "Other";
    t.value = "";

    // first check for whitespace and bypass it
    while (isWhiteSpace(nextChar)) {
        nextChar = readChar();
    }

    // Then check for a comment, and bypass it
    // but remember that / is also a division operator
    if (nextChar=='/') {
        ...
    }

    // Then check for an Operator; recover 2-character
    // operators as well as 1-character ones
    if (isOperator(nextChar)) {
        t.type = "Operator";
        ...
    }

    // Then check for a Separator
    if (isSeparator(nextChar)) {
        t.type = "Separator";
        ...
    }
}
// Then check for an Identifier, Keyword, or Literal
if (isLetter(nextChar)) { // get an Identifier
  t.type = "Identifier";
  ...
}

// check for integers
if (isDigit(nextChar)) {
  t.type = "Literal";
  ...
}
...
return t;
}
Syntactic Analysis
**Concrete Syntax**

- The **Concrete Syntax** of a language defines the structure of all the parts of a program that occur above the lexical level, such as arithmetic expressions, assignments, loops, functions definitions... and programs themselves. It tells the programmer concretely what to write in order to have a valid program.

- A language’s concrete syntax uses BNF as a primary tool to provide a precise definition. The definition of this BNF grammar is based on the use of the token classes of the lexical syntax.

- BNF concrete grammars should not be ambiguous.

- **Example:**
  
  Assignment → Identifier = Expression ;
  Expression → Term | Expression + Term | Expression − Term
  Term → Factor | Term * Factor | Term / Factor
  Factor → Identifier | Literal | (Expression)
Concrete Syntax for JAY

Appendix B on pages 351-352
Syntactic Analysis

- The output of the lexical analysis (tokens) is used as a basis for defining the structure of all the different parts of a program.

- **Syntactic analysis** develops an abstract representation or **parse** for the program, detecting syntactic errors along the way.

- A program is **syntactically correct** if it can be parsed into a tree whose root is the start symbol of the concrete syntax.
This JAY program has 2 main parts: a Declaration part and a Statement part.
Linking Syntax and Semantics
Abstract Syntax

- The **Abstract Syntax** of a program describes the actual information that is carried by a program stripping away syntactic sugar.

- **Example:**
  Consider following loop written in PASCAL:
  
  ```pascal
  while i<n do begin 
  i:= i+1;
  }
  ```
  
  If we think about a loop abstractly the only essential elements are a *test* expression for continuing a loop and the *body* of the loop to be repeated.

- The **Abstract Syntax** of a programming language can be defined using a set of rules of the form, \( Lhs = Rhso \) where:
  - \( Lhs \) is the name of an abstract syntactic class
  - \( Rhso \) is a list of essential components that define a member of that class. The components are separated by 

- Recursion naturally occurs among the definitions in the Abstract syntax (mutual recursion).
• **Example:**

Loop = Expression test; Statement body

The abstract class Loop has two components, a *test* which is a member of the abstract class Expression and a *body* which is a member of an abstract class Statement.

• **Example:**

Statement = Assignment | Loop
Assignment = Variable target; Expression source
Loop = Expression test; Statement body
Expression = Variable | Value | Binary
Binary = Operator op; Expression term1, term2

• One immediate by-product of an abstract syntax is that it provides a basis for defining the abstract structure of a language as a set of classes.
• Examples:

class Loop extends Statement{
    Expression test;
    Statement body;
}

class Assignment extends Statement {
    // Assignment = Variable target; Expression source
    Variable target;
    Expression source;
}

(See the AbstractSyntax.java file)
Abstract Syntax for JAY

Appendix B on page 353
Abstract Syntax Tree

- An Abstract syntax tree can be built from the Abstract syntax.

- Consider the following abstract syntax:

  \[ \text{Expression} = \text{Variable} \mid \text{Value} \mid \text{Binary} \]
  \[ \text{Binary} = \text{Operator} \; \text{op}; \; \text{Expression} \; \text{term1}, \; \text{term2} \]

- The abstract syntax tree for the expression \( x + 2 * y \)
  is the following:

![Diagram of the abstract syntax tree for the expression \( x + 2 * y \).]
Example in JAY

Example of program

```c
// compute result = the nth Fibonacci number
void main () {
    int n, fib0, fib1, temp, result;
    n = 8;
    fib0 = 0;
    fib1 = 1;
    while (n > 0) {
        temp = fib0;
        fib0 = fib1;
        fib1 = fib0 + temp;
        n = n - 1;
    }
    result = fib0;
}
```
Abstract syntax tree for the previous program
Concrete and Abstract Syntax

- The concrete syntax tells the programmer concretely what to write in order to have a valid program in a language $X$.

- The abstract syntax allows valid programs in language $X$ and language $Y$ to share common abstract representations.

- The concrete syntax provides a link between syntax and semantics.

- Concrete and Abstract syntax definitions are necessary.
Recursive Descent Parser

- Based on the lexical, concrete and abstract syntax and on the concrete and abstract parse tree representations.

- Algorithm that translates the input stream of tokens, which is the program, into an abstract syntax tree, which is the parse.

- The tree is generated top-down.

- Algorithm based on the use of an EBNF concrete syntax.
Overview

Concrete syntax:

Assignment -> Identifier = Expression ;
Expression -> Term \{[ + | - ] Term\}*
Term -> Factor \{[* | /] Factor\}*
Factor -> Identifier | Literal | (Expression)

Abstract syntax:

Assignement = Variable target; Expression source
Expression = Variable | Value | Binary
Recursive Descent Parser Algorithm

For each nonterminal symbol $A$ and set of rules of the form $A \rightarrow \omega$:

1. Add a new method definition with $A$ as its return type.
2. Create a new object of class $A$, say $x$.
3. For each member $y$ of the sentential form $\omega$,
   a. if $y$ is a nonterminal, call the method associated with $y$ and assign the result to an appropriate field within $x$.
   b. if $y$ is a terminal, check that the value of that token is identical with $y$ and, if so, call the `nextToken` method. Otherwise the token is in error.
4. If $\omega$ contains a series of symbols that is repeated (indicated by *), insert an appropriate while loop that accommodates any number of repetitions of that series.
5. If there is more than one rule of the form $A \rightarrow \omega$, insert appropriate `if...else` statements that distinguishes the alternatives.
6. Return $x$. 
In JAVA

ConcreteSyntax.java

public class ConcreteSyntax {
    Token token; // current token
    TokenStream input;

    private void match (String s) {
        if (token.value.equals(s))
            token = input.nextToken();
        else
            SyntaxError(s);
    }

    private Assignment assignment () {
        // Assignment --> Identifier = Expression;
        Assignment a = new Assignment();
        if (token.type.equals("Identifier")) {
            a.target = new Variable();
            a.target.id = token.value;
            token = input.nextToken();
            match("=");
            a.source = expression();
            match(";");
        }
        else SyntaxError("Identifier");
        return a;
    }
}
private Expression expression () {
    // Expression --> Conjunction { || Conjunction }*
    Binary b; Expression e;
    e = conjunction();
    while (token.value.equals("||")) {
        b = new Binary();
        b.term1 = e;
        b.op = new Operator(token.value);
        token = input.nextToken();
        b.term2 = conjunction();
        e = b;
    }
    return e;
}
...

Semantics and code generation
Semantics Errors

- Consider the following grammar.

  sentence -> noun verb
  noun -> dog | man
  verb -> bit

  “Man bit” is syntactically correct but it makes no sense!

- In JAVA:

  char a = 'c';
  double b = 1.2;
  int sum = 0;

  sum = a+b;

  *sum = a + b* is syntactically correct but: What does it mean to add a character to a real number? Is this accepted or not?

  Are *sum, a* and *b* declared before being used?
Semantics and code generation

- Use of the Abstract Syntax tree.

- During **Semantics Analysis** the compiler:
  - analyzes the **meaning** of the program and
  - tries to understand the **actions** it performs.
  At this point **Semantics Errors** are detected.

- The compiler also generates the proper sequence of machine language instructions to carry out these actions.
  This is **Code Generation**. The code is then optimized.
Code optimization
Code optimization

- The (machine language) generated code must be efficient in space and time.

- Assume that one instruction is executed in 1 microsecond except ADD and SUBTRACT takes 2 microseconds and MULTIPLY and DIV takes 3 microseconds.

- What is the time needed to execute the following program?

```
INCREMENT X
INCREMENT X
INCREMENT X
```

- What is the time needed to execute the following program?

```
LOAD X, R
ADD THREE, R
STORE R, X
THREE: .DATA 3
```

- Are these programs equivalent? Which one is the most efficient?
Different optimizations

- **Constant evaluation**: Evaluation of expression during compilation instead of execution.

- **Strength reduction**: A slow operation is replaced by a faster one.

- **Eliminating unnecessary operations**.
Constant evaluation

- Consider $x = 1 + 1$.

- LOAD ONE, R
  ADD ONE, R
  STORE R, X
  ONE: .DATA 1

  is equivalent to

  LOAD TWO, R
  STORE R, X
  TWO: .DATA 2
Strength reduction

- Consider $x = 2 \times x$.

- LOAD X, R
  MULTIPLY TWO, R
  STORE R, X
  TWO: .DATA 2

  is equivalent to

  LOAD X, R
  ADD X, R
  STORE R, X
Elimination of unnecessary operations

- Consider the following code:

  \[
  \begin{align*}
  &x = y \\
  &z = y
  \end{align*}
  \]

- LOAD Y, R  
  STORE X, R  
  LOAD Y, R  
  STORE R, Z  

is equivalent to

- LOAD Y, R  
  STORE R, X  
  STORE R, Z
Conclusion

- Topics we studied:
  - Syntax - lexical, concrete, abstract
  - Grammars - BNF, EBNF, Syntactic diagrams
  - Parse Tree

- We surveyed the compilation process.