## 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:

  \[
  \text{Integer} \rightarrow \text{Digit} \mid \text{Integer Digit}
  \]

  \[
  \text{Digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 10
  \]

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

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

  **Example:** Integer Digit, Digit Digit, 3 Digit and 32 are sentential forms.

### 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
  /      \   \nDigit  5    2
   \     \   \n   \     3
```

### 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:

\[
\text{Integer} \rightarrow \text{Digit} \mid \text{Integer} \text{Digit} \\
\text{Digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 10
\]

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

![Parse tree diagram]

• A grammar is said ambigious 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.

\[
\text{Expression} \rightarrow 0 \mid 1 \mid \text{Expression} \rightarrow \text{Expression}
\]

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

• Example 2: Dangling-else ambiguity

The following grammar is ambiguous.

\[
\text{Statement} \rightarrow \text{if} \ \text{Expression} \ \text{then} \ \text{Statement} \mid \text{if} \ \text{Expression} \ \text{then} \ \text{Statement} \ \text{else} \ \text{Statement} \\
\text{Expression} \rightarrow ...
\]

if \( E_1 \) then if \( E_2 \) then \( S_1 \) else \( S_2 \) can be parsed in 2 parse trees.

• Example 3: The following grammar is ambiguous.

\[
\text{Assignment} \rightarrow \text{Variable} = \text{Expression} \\
\text{Expression} \rightarrow \text{variable} \mid \text{Expression} + \text{Expression} \\
\text{Variable} \rightarrow x \mid y \mid z \\
x = x + y + z \text{ can be parsed in 2 parse trees.}
\]

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:

\[
\text{Expression} \rightarrow 0 \mid 1 \mid \text{Expression} - \text{Expression}
\]

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:

\[
\text{ifThenStatement} \rightarrow \text{if (Expression) Statement} \\
\text{ifThenStatementStatement} \rightarrow \text{if (Expression) StatementNoShortIf else Statement}
\]

– JAY has the following BNF grammar for conditionals:

\[
\text{Statement} \rightarrow : \mid \text{Block} \mid \text{Assignment} \mid \text{IfStatement} \mid \text{WhileStatement} \\
\text{Block} \rightarrow \{ \text{Statement} \}
\]

\[
\text{IfStatement} \rightarrow \text{if (Expression) Statement} \mid \text{if (Expression) Statement else Statement}
\]
• Example 3:
  Assignment → Variable = Expression
  Expression → Variable | Expression + Expression
  Variable → x | y | z
  To solve ambiguity we use the fact that + is left-associative.

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-Z-]*/(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 $a^nb^n$ cannot be generated by a regular expression. Can it be generated by a BNF grammar?

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>

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 recursion.
Consider the following EBNF grammar:

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

- EBNF-Based Parse tree for the expression \(x + 2 \times y\)

**Syntax diagrams** represent another alternative for specifying a language.

**Example:**

- Syntax diagrams was famous because of PAS-CAL.

**Example:**

**Compiler**

- 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).
- **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.
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 \( \ell \) 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 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.

- See java.sun.com/docs/books/jls/second_edition/
  html/j.title.doc.html

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 JAVA program.

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

The following stream of tokens is associated to the
JAVA 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

Note: Letter, Digit... can be described by regular expressions.
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;
    }
}
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)
```

## Syntactic Analysis

### Concrete Syntax for JAY

Appendix B on pages 351-352

- 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.

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;
end
```

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.

**Example:**
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:**

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

Recursion naturally occurs among the definitions in the Abstract syntax (mutual recursion).
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 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:

Assignment = 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 → ω:
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 ω,
   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 ω contains a series of symbols that is repeated (indicated by *), insert an appropri-
   ate while loop that accommodates any number of repetitions of that series.
5. If there is more than one rule of the form A → ω, insert appropriate if...else
   statements that distinguishes the alternatives.
6. Return x.

ConcreteSyntax.java

import java.util.Scanner;

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
• Consider the following grammar.

\[
\begin{align*}
\text{sentence} & \rightarrow \text{noun} \ \text{verb} \\
\text{noun} & \rightarrow \text{dog} \ | \ \text{man} \\
\text{verb} & \rightarrow \text{bit}
\end{align*}
\]

"Man bit" is syntactically correct but it makes no sense!

• In JAVA:

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

\( \text{sum} = a + b \) is syntactically correct but: What does it mean to add a character to a real number? Is this accepted or not? Are \( \text{sum}, a \) 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

• 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?

```assembly
INCREMENT X
INCREMENT X
INCREMENT X
```

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

```assembly
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.