Full Playlist: <https://www.youtube.com/playlist?list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS>

# Lecture 1 - Phases of a Compiler

<https://youtu.be/Qkwj65l_96I>

Not all compilers have the same exact phases, but the following is common:

|  |  |
| --- | --- |
| **High-Level Language** |  |
| **↓** |  |
| **Lexical Analysis** |  |
| **↓** | *Stream of Lexical Tokens* |
| **Syntax Analysis** |  |
| **↓** | *Parse Tree* |
| **Semantic Analysis** |  |
| **↓** | *Semantically-Verified Parse Tree* |
| **Code Generation** |  |
| **↓** | *Assembly Code* |
| **Object Code** |  |

**Lexical Tokens**:

A lexical token can take the pattern , i.e. a letter followed by a letter or digit. The pattern is known to the **lexical analyser**.

**Syntax Analyser**:

The syntax analyser takes a stream of lexical tokens one at a time, and a grammar (context-free grammar).

**Grammars** define the rules of the programming language with its production rules e.g.:

|  |
| --- |
| *A statement can be an identifier equal to an expression then a semicolon.*  *An expression can be an expression plus a term, or a term.*  *A term can be a term into a factor, or a factor.*  *A factor can be an identifier.* |

The syntax analyser constructs a **parse tree**, which begins with the start symbol.

**Semantic Analyser**:

The semantic analyser verifies that the parse tree is semantically correct. e.g. It verifies that the LHS identifier in an assignment is in fact a variable, and not a function.

**Code Generator**:

The code generator has multiple stages:

1) **Intermediate Code Generator**

In the ICG, each statement is three-address code, i.e. each statement only has three addresses:

|  |
| --- |
| becomes |

These statements can easily be translated into machine code.

**Everything so far is common to all platforms for the compiler. The following phases will be different based on the platform**:

2) **Code Optimiser**

The CO reduces the number of lines of code.

3) **Target Code Generator**

The TCG produces machine code for the target platform e.g. ARM / Intel.

# Lecture 2 - Lexical Analysers & Grammars

<https://youtu.be/WccZQSERfCM>

### Lexical Analysers

Lexical analysis is the only phase which reads the program as text, character by character. It is the only phase that knows the line number and position of each lexeme.

A lexical analyser:

1. Converts lexemes into tokens.
2. Removes comments.
3. Removes whitespace.

### Grammars

A grammar is where:

* is the grammar.
* is the variables.
* is the terminals.
* is the productions.
* is the start symbol.

For the following grammar :

|  |
| --- |
|  |

If we parse the left-most symbol first, it is called **left-most derivation**.

For the string :

**or**

Parsing from right-to-left is called **right-most derivation**.

Parse tree:

|  |
| --- |
| E  /|\  E + E  / /|\  id E \* E  | |  id id |

If we can produce the parse tree or derivation in multiple ways, the grammar is said to be **ambiguous**. This sample grammar is ambiguous.

In an ambiguous grammar, the parser does not know which parse tree to use. Ambiguous grammars are **not** allowed by most parsers.

# Lecture 03 - Ambiguous Grammars

<https://youtu.be/9vmhcBpZDcE>

Some ambiguous grammars can be made unambiguous:

|  |
| --- |
|  |

When there are two operators on either side of the operand, we must choose the associativity.

Correct

Incorrect

We want **left-associativity** for addition, so we define the grammar to be **left-recursive** to remove ambiguity.

Correct

Incorrect

**Operator precedence** must be followed. Multiplication has a higher precedence than addition.

This grammar can be **rewritten** to be left-recursive and unambiguous:

|  |
| --- |
| *\* has a higher precedence than +* |

+ and \* are left-associative, but ^ is right-associative.

# Lecture 04 - Elimination of Left-Recursion

<https://youtu.be/3_VCoBfrt9c>

A production is **left-recursive** if the symbol on the left of the RHS is equal to the symbol on the LHS. e.g.

The issue with left-recursion is that left-recursion calls itself before doing any work. It must reach a base case first.

**Top-down parsers cannot work with left-recursive grammars**.

Left-recursive grammar:

|  |
| --- |
|  |

This produces strings of the form .

Rewriting this grammar to be right-recursive:

|  |
| --- |
|  |

### Types of Grammars

Ambiguous / Unambiguous

Left-Recursive / Right-Recursive

Deterministic / Non-Deterministic

A **non-deterministic** grammar has many options for a single symbol.

We need to backtrack and try a new production every time the string does not match the current rule. We are making a decision of which production to choose without enough information.

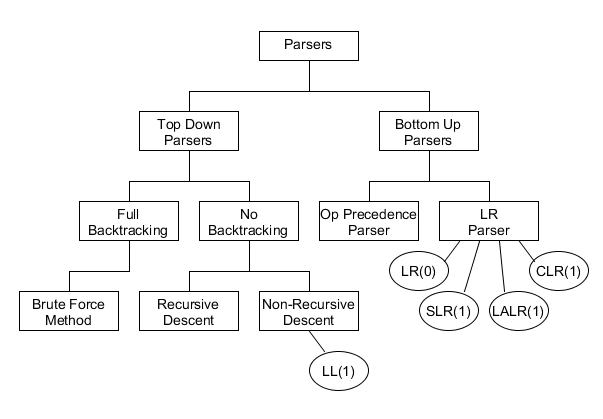
**Left-factoring** a non-deterministic grammar can make it deterministic:

|  |
| --- |
|  |
|  |

Lecture 5 - Parsers & LL(1) Parsing

<https://youtu.be/N9UuAPU6DAg>

### Types of Parsers



In order to parse with LL(1), the grammar must **not** be:

1. Ambiguous (non-deterministic)
2. Left-recursive

### Top-Down Parsers (TDPs) vs Bottom-Up Parsers (BUPs)

TDPs use **left-most derivation**.

BUPs use right-most derivation (reverse order because we begin with the terminals).

Using the following grammar as an example:

|  |
| --- |
|  |

Two ways to construct a parse tree for a grammar:

1. TDP - At each point, choose a production. What to use?
2. BUP - Take the terminals, and go to the root. When to reduce?

Left-most derivation - always expand the left-most production.

|  |  |  |  |
| --- | --- | --- | --- |
| Top Down Parsing:   |  | | --- | | S  //\\  a A B e  /|\ \  A b c d  |  b | | Bottom-Up Parsing:   |  | | --- | | S  \_\_\_|\_\_\_\_  / | \ \  / A B \  / / | \ \ \  | A | | | |  | | | | | |  a b b c d e | |

### LL(1) Parsers

L: Scan the input from left to right.

L: Use left-most derivation.

(1): Number of lookaheads. How many symbols can you see when you make a decision?

The input string uses ⊣ as an endmarker.

Stack:

|  |
| --- |
|  |
|
|
| ⊣ |

In order to construct the LL(1) **parsing table**, we need two functions: First() and Follow().

### First() and Follow()

**First()**:

If I start with any symbol and generate all its strings, what is the starting symbol?

|  |  |
| --- | --- |
|  |  |

**Follow()**:

What are the terminals that could follow a variable during derivation?

|  |  |
| --- | --- |
|  |  |

Lecture 6 - First() & Follow() Examples

<https://youtu.be/_uSlP91jmTM>

Example 1:

|  |  |  |
| --- | --- | --- |
|  |  |  |

Example 2:

|  |  |  |
| --- | --- | --- |
|  |  |  |

Example 3:

|  |  |  |
| --- | --- | --- |
|  |  |  |

Example 4:

|  |  |  |
| --- | --- | --- |
|  |  |  |

Lecture 7 - LL(1) Parsing Table

<https://youtu.be/R1ZlWEZWMKk>

|  |  |  |
| --- | --- | --- |
|  |  |  |

We are constructing a **top-down parser**.

Construct an **LL(1) parsing table** with the variables on the LHS, and the terminals on top:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

*If I am at a rule in the derivation process and I see a terminal, what production should I use?*

If the of a production contains the terminal, we use it.

If the of a production contains the terminal, we use the production that produces .

*Note*: We will never see in the input.

For the grammar:

|  |  |  |
| --- | --- | --- |
|  |  |  |

The parsing table is:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |

For the string :

Stack:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Input** |  |  |  |  |  |  |  |
| **Action**  **Taken** |  | *Replace*  *Retain* | *Pop stack*  *Advance* | *Replace*  *Retain* | *Pop stack*  *Advance* | *Pop stack*  *Retain* | *Pop stack*  *Advance* |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| **Stack** |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

Continued...

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Input** |  |  |  |  |  |  |  |
| **Action** | *Pop stack*  *Advance* | *Accept* |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| **Stack** |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

For the grammar:

|  |  |  |
| --- | --- | --- |
|  |  |  |

The parsing table is:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Lecture 8 - Recursive Descent Parsers

<https://youtu.be/SH5F-rwWEog>

|  |
| --- |
|  |

|  |
| --- |
| char l = getChar(); // Current char in input.  E() {  if (l == 'i') {  match('i');  E\_Dash();  } }  E\_Dash() {  if (l == '+') {  match('+');  match('i');  E\_Dash();  }  else return; // Match with nothing for the third production. }  match(char t) {  if (l == t) {  l = getChar(); // Increment pointer.  }  else printf("error"); }  main() {  E();  if (l == '⊣') {  printf("parsing success");  } } |

Lecture 19 - S-Attributed & L-Attributed

<https://youtu.be/rdnAJBoFKOw>

Types of SDT (Syntax-Directed Translation):

|  |  |
| --- | --- |
| **S-Attributed**: | **L-Attributed**: |
| Uses only synthesized attributes. | Uses both **inherited** and **synthesized** attributes.  Each inherited attribute is restricted to inherit either from the **parent** or a **left sibling** only. |
| Semantic actions are placed at the right end of the production. | Semantic actions are placed **anywhere** on the RHS. |
| Attributes are evaluated during BUP. | Attributes are evaluated by traversing **depth-first**, **left to right**. |

**Synthesized**: Node takes a value from its **children**.

**Inherited**: Node takes a value from a **parent** or **sibling**.