Regular Expressions

- In ***formal languages***, *regular expressions* are used to define ***strings* of *symbols***. The set of all possible strings that a regular expression defines is called the formal language of the regular expression.
- When defining the syntax of programming languages the symbols of interest are characters, e.g. ASCII or UNICODE characters.
- In programming languages, regular expressions and grammars are used to define the syntax of the programming language. ***Regular expressions*** are used to define the basic syntactical elements – identifiers, keywords, numbers, strings[1], operators, brackets etc. ***Grammars*** are used to define how these basic elements can be composed into complex syntactical elements – expressions, statements, declarations, functions, programs etc.
- In a compiler, the ***lexical analyser***[2] transforms a stream of characters (i.e. symbols) into a stream of tokens[3] (i.e. objects).
- Although lexical analysers can be written by hand, a more approach professional approach is to use a tool[4] that reads a list of regular expressions that define the basic syntactical elements and generates the lexical analyser which can be linked into the rest of the compiler[5].
- Lexical analyser generators are implemented by converting regular expressions into very fast ***finite automata***[6], which are formal way for describing certain algorithms, e.g. pattern matching.

Symbols	
Is A a symbol	Yes
Is B a symbol	Yes
Is C a symbol	Yes
Is £ a symbol	Yes
Is ▼ a symbol	Yes
Is ⚖ a symbol	Yes
Strings	
Is AA a symbol	No
Is AA a string	Yes
Is A a string	Yes
Is ABBA a string	Yes
Is ε the *empty string*	Yes, ε is a string with no symbols
Regular Expressions - Symbol	
Is **A** a symbol	No
Is **A** a string	No
Is **A** a regular expression	Yes
Is A a regular expression	No
Will A be recognised by **A**	Yes
Will a be recognised by **A**	No

[1] Not to be confused with strings of symbols, which are a formal concept.
[2] A lexical analyser is also known as a ***scanner***.
[3] When we study grammars, we will refer to tokens as terminals or terminal symbols.
[4] The classic example is **LEX** and its modern version FLEX that outputs lexical analysers in C.
[5] Modern tools like **ANTLR** often handle all syntax generating both the lexical analyser and the parser from the regular expressions and the grammar.
[6] Students are highly encouraged to read more on this in textbooks and also last year's slides.

Will B be recognised by `A`	No
Can `A` generate A	Yes
Can `A` generate B	No
Is ε a regular expression	Yes
Will ε be recognised by `ε`	Yes
Regular Expressions - Concatenation	
Is `AA` a regular expression	Yes
Will A be recognised by `AA`	No
Will AA be recognised by `AA`	Yes
Will AAA be recognised by `AA`	No
Can `AA` generate A	No
Can `AA` generate AA	Yes
Can `AA` generate AAA	No
Is `WHILE` a regular expression	Yes, will match WHILE
Is `ABBA` = `A B B A`	Yes, spaces are ignored unless escaped (see later)
Is `23` a regular expression	Yes, will match 23
Is `<>` a regular expression	Yes, will match <>
Will A be recognised by `AB`	No
Will B be recognised by `AB`	No
Will AB be recognised by `AB`	Yes
Will BA be recognised by `AB`	No
Is `AB` = `BA`	No
Will A be recognised by `Aε`	Yes
Will A be recognised by `εA`	Yes
Can `Aε` generate A	Yes
Can `εA` generate A	Yes
Regular Expressions - Repetition	
Is `A*` a regular expression	Yes, will match zero or more A's
Will A be recognised by `A*`	Yes
Will AA be recognised by `A*`	Yes
Will AAA be recognised by `A*`	Yes
Can `A*` generate A	Yes
Can `A*` generate AA	Yes
Can `A*` generate AAA	Yes
Can `A*` generate ε	Yes
Can `A*` generate A*	No
Is `*` a regular expression	No
Will * be recognised by `*`	No, `*` is illegal by itself
Will * be recognised by `'*'`	Yes, quotes are often used to escape special regex characters
Will * be recognised by `\*`	Yes, some authors/systems use backslash to escape

	1 character
Will ** be recognised by `'**'`	Yes, matches two asterisks or `\*\*`
Will ** be recognised by `'*'*`	Yes, matches zero or more asterisks or by `\**`
Will \ be recognised by `'\'`	Yes, or by `\\`
Will A be recognised by `AA*`	Yes
Will A be recognised by `A*A`	Yes
Can `AA*` generate A	Yes
Can `AA*` generate ε	No
Can `A*A` generate ε	No
Will AABBB be recognised by `A*B*`	Yes
Will BBB be recognised by `A*B*`	Yes
Will A*B* be recognised by `A*B*`	No
Will A*B* be recognised by `'A*B*'`	Yes, or by `A\*B\*`
Regular Expressions – Alternation	
Will A be recognised by `A|B`	Yes
Will B be recognised by `A|B`	Yes
Can `A|B` generate A	Yes
Can `A|B` generate B	Yes
Can `A|B` generate AB	No
Is `A|B` = `B|A`	Yes
Will A be recognised by `A|ε`	Yes
Will B be recognised by `ε|A`	No
Can `A|ε` generate A	Yes
Can `A|ε` generate ε	Yes
Regular Expressions – Grouping	
Will AB be recognised by `A|B*`	No
Will AB be recognised by `(A|B)*`	Yes
Will ABBA be recognised by `(A|B)*`	Yes
Will AABBA be recognised by `(A|B)*`	Yes
Will (A|B)* b.r.b. `'(A|B)*'`	Yes, or by `\(A|B\)\*`
Regular Expressions – Precedence (lowest)	Repetition (highest), concatenation, alternation
Is `AB*` = `(AB)*`	No
Is `AB*` = `A(B*)`	Yes
Is `A*B*` = `(A*B)*`	No
Is `A*B*` = `(A*)(B*)`	Yes
Is `A**` = `(A*)*` = `A*`	Yes
Is `A|B*` = `(A|B)*`	No
Is `A|B*` = `A|(B*)`	Yes
Is `AB|BA` = `BA|AB`	Yes

Is `AB*|BA*` = `(AB|BA)*`	No
Is `AB*|BA*` = `A(B*)|B(A*)`	Yes
Is `AB**` = `(AB*)*`	No
Is `AB**` = `A((B*)*)`	Yes
Regular Expressions – Shortcuts	
Is `A?` = `A|ε`	Yes, 0 or 1
Is `A+` = `AA*`	Yes, 1 or more
Is `[ABC]` = `A|B|C`	Yes, any symbols in the set
Is `[A-C]` = `A|B|C`	Yes, range of symbols
Is `[0-9]` = `0|1|2|3|4|5|6|7|8|9`	Yes, range of symbols
Is `[0-2A-C]` = `0|1|2|A|B|C`	Yes, two ranges of symbols
`.` denotes any permissible symbol	Any symbol[7] in the alphabet (see below)
`[^A]` denotes …	Any permissible symbol except A
`[^AB]` denotes …	Any permissible symbol except A and B

Formal Languages

Formal Language	
The set of all possible strings that can be generated by a regular expression `regex` is called the *formal language* of the regular expression and is written L(`regex`).	Formal languages (sets of strings) defined by regular expressions are called *regular languages*.
Is L(`A`)={A}	Yes
L(`ε`)={ε}	Yes
L{`∅`}={}	Yes
L(`A|B`) ={A,B}	Yes
L(`AB`)={AB}	Yes
L`((A|B)C`}={AC, BC}	Yes
L(`A*`)={ε, A, AA, AAA,…}	Yes
Alphabet	
Is {A,B,C} an alphabet	Yes, the alphabet is the set of all permitted symbols
Is A a symbol of the alphabet {A,B,C}	Yes
Is Z a symbol of the alphabet {A,B,C}	No
Alphabets are typically denoted by ∑	Yes
Can ε be part of an alphabet	No
For ∑={A,B}, is ∑*= {ε,A,B,AA,AB,BA,BB,AAA,…}	Yes, ∑* is the *closure* of the alphabet ∑ and is the set of all possible strings of symbols from ∑ including ε
Is ∑* a formal language	Yes, ∑* is the formal language of the alphabet ∑
For the alphabet ∑={A,B}, how many strings in ∑* have length 2, 3 and n	2^2, 2^3, 2^n

[7] In most tools, . (dot) stands for any character other an end-of-line character.

Exercises	
What does `(0|1)+` recognise	Binary numbers
What does `(0|1)*0` recognise	Even binary numbers
What does `[0-9]+` recognise	Unsigned decimal numbers
What does `(+|-)[0-9]+` recognise	Signed decimal numbers
What does `(+|-)?[0-9]+` recognise	Optionally signed decimal numbers
Give a regular expression for:	
(i) a letter	`[a-zA-Z]`
(ii) an identifier	`[a-zA-Z] [a-zA-Z0-9_]*`
(iii) the keywords `if`, `while`, and `do`	`if	while	do`
(iv) comment of the form `{ … }`	`{ [^}]* }`
(v) comment of the form `/* … */`	`'/*' [^*]* ('*' ([^*/]*)?)* '*/'`
For the alphabet ∑={A,B,C} give a regular expression for all strings that:	
(i) contain exactly one B	`(A|C)* B (A|C)*`
(ii) contain at most one B	`(A|C)* B? (A|C)*`
(iii) do not contain 2 consecutive B's	`(A|C|BA|BC)* B?`
For the alphabet ∑={A,B} give a regular expression for all strings:	
(i) beginning with a B followed by zero or more A's	`BA*`
(ii) in which every A is immediately followed by at least two B's	`(B|ABB)*`
(iii) containing exactly two B's	`A* B A* B A*`
(iv) containing two consecutive A's or two consecutive B's	`(A|B)* (AA|BB) (A|B)`
(v) an even number of A's and an even number of B's	`(AA	BB	(AB|BA) (AA|BB)* (AB|BA))*`

Tokens

- Lexical analysers typically provide a function that returns a token (an object) that corresponds to the next basic syntactic element. There are several common types of token.

- ***Identifier tokens*** are usually classified into keyword identifiers and non-keyword identifiers.
- ***Keyword identifiers*** e.g. `class`, `package`, `while` have special meaning in the programming language and are represented by their own token e.g. CLASS token, PACKAGE token, WHILE token. There are small number in most languages, e.g. <50 keywords.
- ***Non-keyword identifiers*** e.g. `year`, `quad09`. These are defined by the programmer and normally represented by a general identifier token, e.g. IDENT plus a string attribute holding the actual name. For example, `year` could be mapped to the token IDENT("`year`").
- Lexical analysers need to be able to quickly determine if an identifier is a keyword. A fast technique for this is to use a ***perfect hash function***. More generally, lexical analysis is one of the simplest parts of a compiler, a bad-implementation can lead to very poor performance. Profiling a compiler often reveals the compiler can spend more than 50% of its time in the lexical analyser.

- ***Literal tokens*** (i.e. *constants*) are usually classified into unsigned integers, reals and strings.
- ***Unsigned integers*** e.g. `123` are represented by an INTEGER token, plus an integer attribute holding the actual value e.g. INTEGER(`123`). What about negative integers?
- ***Unsigned reals*** e.g. `123.45` are represented by a REAL token for reals, plus a floating-point attribute holding the actual value e.g. REAL(`123.45`).
- ***Strings*** e.g. "`hello world`" are represented by a STRING token for strings, plus a string attribute holding the actual value e.g. STRING("`hello world`")
- We need to take care that the programming language that the compiler is implemented in correctly represents values of the compiled language, e.g. Unicode strings using a compiler that only supports ASCII strings, big integers using a compiler that only supports short integers.

- In additional to identifiers and literals, there are several other tokens types:
- ***Operators and other punctuation*** e.g. `+ : <= (` are normally represented by their own token e.g. PLUS, COLON, LESSEQUAL, LPAREN
- ***Whitespace*** e.g. spaces, tabs, newlines are *not* normally returned as tokens (unless they occur in a string literal). Whitespace is normally needed to separate adjacent identifiers, literals e.g. in `public static void`. They can be represented by regular expressions however.
- ***Comments*** are also not normally returned as tokens.

- What extra information would be useful for the lexical analyser to record in returned tokens? Hint: think about for reporting errors later in the compilation or at runtime.

Disambiguation Rules

- If a string can be matched by more than one regular expression we need rules to resolve the ambiguity. A workable set of disambiguation rules is:
  (1) match the longest matching string, otherwise
  (2) assume that the list of regular expressions defined for the programming language have textual precedence such that the earliest written regular expression takes precedence.
- For example, if we would write the regular expression for keywords before the regular expression for identifiers, the string `doughnut` would return an IDENTIFIER token, while the string `do` would return a DO token.

Context Free Grammars

- Regular expressions are not sufficiently expressive to define the more complex syntactic structures of programming languages, for example, nested constructs. For this, we need grammars.

- There are many classes of grammars. An important classification is due to Chomsky, who identified 4 classes: **regular grammars**, **context free grammars**, **context-sensitive grammars** and **unrestricted** grammars - succeeding grammar is a superset of the preceding grammar. We'll consider the second of these, since the syntax of many programming languages can be described by a context-free grammar.

- In formal languages, a ***context-free grammar*** G consists of:
  (1) a set of ***terminal symbols***. When defining programming languages, terminal symbols correspond to the tokens returned by the lexical analyser, that is, they correspond to basic syntactical elements, like identifiers, keywords, numbers, operators and other punctuation characters.
  (2) a set of ***non-terminal symbols*** (disjoint from the set of terminal symbols). Non-terminal symbols correspond to grammar rules[8]. This set includes a ***start symbol*** for the grammar, which is either explicitly identified or corresponds to the non-terminal symbol on the left-hand side of the first rule. We use the latter approach for the start symbol.
  (3) a set of **rules** of the form: N → α where N is a non-terminal symbol, and α is a string of terminal and non-terminal symbols.
- The formal language corresponding to a grammar is the set of all strings[9] of terminals that can be produced from the start symbol using the rules of the grammar.

- In these notes, we will write terminal symbols either in (1) lowercase italics, e.g. *id*, *while*, *num* or (2) or in quotes for operators and other punctuation characters/character sequences, e.g. `'+'`, `'<='`, or (3) ε for the empty string.
- Non-terminal symbols will written as 1st-letter uppercase identifiers, e.g. `Program`, `Expression`

```
Program     → Block
Block       → Statement Block
Block       → ε
Statement   → id '=' Expression
Statement   → read id
Statement   → write Expression
Expression  → Expression Operator Expression
Expression  → '(' Expression ')'
Expression  → num
Operator    → '+'
Operator    → '-'
Operator    → '*'
Operator    → '/'
```

- In the example above, the terminals are *id*, *num*, *read*, *write*, `+`, `*` Non-terminals are `Program`, `Block`, `Statement`, `Expression` and `Op`. The start symbol is `Program`.
- For conciseness, vertical bars can be used to group alternative rules[10] for the same non-term terminal, e.g.

```
Program     → Block
Block       → Statement Block | ε
Statement   → id '=' Expression | read id | write Expression
Expression  → Expression Operator Expression | '(' Expression ')' | num
Operator    → '+' | '-' | '*' | '/'
```

[8] Rules are also known as ***productions***.
[9] Some textbooks use the term ***sentence*** instead of string.
[10] Note: a rule with N alternatives are really N rules not one.

- The rule `Expression` → `Expression Operator` *num* is *left-recursive*.
- The rule `Expression` → *num* `Operator Expression` is *right-recursive*.

Derivations

- A ***derivation*** is a sequence of substitutions of rules occurring on the right hand side of grammar. One substitution in each derivation step (⇒). Example:

```
Expression ⇒ Expression Operator Expression[11]
           ⇒ Expression '+' Expression
           ⇒ num '+' Expression
           ⇒ num '+' num
```

- Note: each *num* token in the derivation would include the actual value of the number in the input, e.g. if the input was 45+35, the first num token would be *num(45)* while the second num token would be *num(35)*. We will sometimes just use 45 or 35 to emphasise concrete terminals.
- The **formal language** corresponding to a grammar is the set of all strings of terminals that can be derived from the start symbol using derivations i.e. { *string* | *startSymbol* ⇒* *string* } where ⇒* is a derivation.
- For the grammar corresponding to the start symbol `Program` above, each string in the formal language corresponds to a **syntactically legal program**.

Grammar	**Formal Language**
`E` → `E 'a' | 'a'`	{ a^n, n>=1 }
`E` → `'(' E ')' | 'a'`	{ $(^n$ a $)^n$, n>=0 }
`E` → `'(' E ')`	{ } - no strings can be derived, no base case

- There can be more than one derivation for the same terminal string. For example for *num* `'+'` *num* there are 5 other derivations including the following two:
- If at each step, the leftmost rule is substituted, the derivation is called a ***leftmost-derivation***, e.g.:

```
Expression ⇒ Expression Operator Expression
           ⇒ num Operator Expression
           ⇒ num '+' Expression
           ⇒ num '+' num
```

- If at each step, the rightmost rule is substituted, the derivation is called a ***rightmost-derivation***, e.g.:

```
Expression ⇒ Expression Operator Expression
           ⇒ Expression Operator num
           ⇒ Expression '+' num
```

Derivations	*Quotes around 1 and 9 omitted for clarity*

For `E` → `E '9' | '1'` give derivations for:
1
19
199
91
	
	`E` ⇒ `1`
	`E` ⇒ `E 9` ⇒ `1 9`
	`E` ⇒ `E 9` ⇒ `E 9 9` ⇒ `E 1 9 9`
	not possible

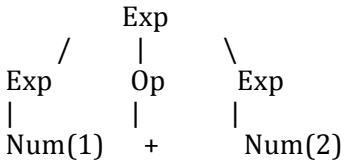
For `E` → `'9' E | '1'` give derivations for:
1
19
199
91
	
	`E` ⇒ `1`
	not possible
	not possible
	`E` ⇒ `9 E` ⇒ `9 1`

[11] Each right-hand-side is called a *sentential form*. The final sentential form of terminals is called the *yield* of the derivation.

For E → '1' E | '9' give derivations for:	
1	not possible
19	E ⇒ 1 E ⇒ 1 9
199	not possible
91	not possible
For E → E '1' | '9' give derivations for:	
1	not possible
19	not possible
199	not possible
91	E ⇒ E 1 ⇒ 9 1

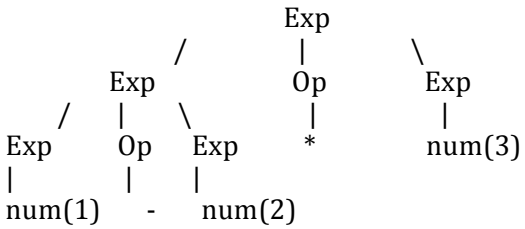
Parse Trees and Ambiguous Grammars

- The ***parse tree*** for a derivation is constructed by connecting each symbol (child node) in a derivation to the non-terminal symbol (parent node) from which it was derived. For the 1+2 above, we have:

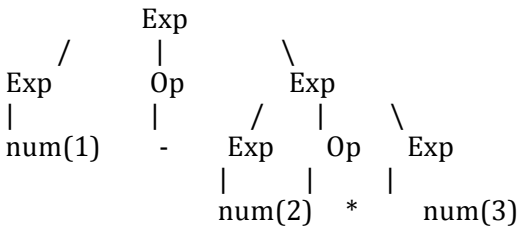


- A context free grammar is ***ambiguous*** if for at least one string in the language there are two possible derivations that correspond to distinct parse trees. For example, for the string 1-2*3, has two different parse trees:

Parse tree 1



Parse tree 2



- An alternative way of stating this, is that a grammar is ambiguous if there is exists a string in the language that has multiple left-most derivations (or multiple right-most derivations).
- Ambiguity is a property of grammars not formal languages.

Ambiguous Grammars	
For each of the following show that the grammar is ambiguous by giving two leftmost derivations for the same string: Quotes around terminal symbols are omitted for clarity.	
E → '9' F → '9'	E ⇒ 9 F ⇒ 9
E → E | ε	E ⇒ ε

	E ⇒ E ⇒ ε
E → E '-' E | '9'	E ⇒ E-E ⇒ 9-E ⇒ 9-E-E ⇒ 9-9-E ⇒ 9-9-9 E ⇒ E-E ⇒ E-E-E ⇒ 9-E-E ⇒ 9-9-E ⇒ 9-9-9
E → E E | '<' E '>' | ε for the string <<><>> <<>>	E ⇒EE⇒<EE>E⇒<ε E>E⇒<ε ε>E⇒<ε ε>ε E ⇒<EE>⇒<E>⇒<>

- ***Why is ambiguity undesirable*?**
- Although the language for an ambiguous grammar is completely determined, the different parse trees/derivations might have different semantics.
- For example, should 1-2*3 mean (1-2)*3=-3 or 1-(2*3)=-5? Similarly, do we want 1-2-3 to mean (1-2)-3=-4 or 1-(2-3)=0?
- A nice practical solution, when it's available, is to have disambiguating rules. For example, we could handle the first ambiguity with a disambiguating rule that defines the *precedence* of operators, while the second ambiguity could be handled by a disambiguating rule that defines the *associativity* of operators.

- An alternative approach is to rewrite the grammar to remove the ambiguity. For example, by writing extra rules, `Expression` can be rewritten as:

```
Expression  →  Expression AddOp Term | Term
AddOp       →  '+' | '-'
Term        →  Term  MulOp Factor | Factor
MultOp      →  '*' | '/'
Factor      →  '(' Expression ')' | num
```

- This formulation is unambiguous.
- Addition and subtraction are grouped in Expression and closer to the root of the parse tree so will be of lower precedence than multiplication and division.
- Using `Expression AddOp Term` makes addition and subtraction *left-associative* (`Term AddOp Expression` would make them *right-associative*). Similarly for multiplication and division.
- For expressions, which approach is better? Although the second approach is purer in the sense that no disambiguating rules are required, it leads to grammars that are harder to understand (for humans) and more complex parse trees (for the compiler writer to deal with). So *a system that supports and resolves some ambiguous grammars is desirable*.
- Another classic example of an ambiguous grammar is the ***dangling-else*** found in many programming languages:

```
IfStatement →  if  Expression  then  Statement
IfStatement →  if  Expression  then  Statement  else  Statement
```

- For `if A then if B then X else Y` this can generate 2 distinct derivations where Y binds either with A or with B. The different derivations have different meaning and would generate different code..
- We can remove this ambiguity in favour of Y associating with B by re-writing the grammar as:

```
IfStatement →  if  Expression  then  Statement
IfStatement →  if  Expression  then  WithElse  else  Statement
WithElse    →  if  Expression  then  WithElse  else  WithElse
```

- Now each *else* binds with the nearest preceding unmatched *if*.
- An alternative, quite nice approach is to change the syntax to the ***if-then-elseif-endif*** form.
- **Note1**: if ambiguity doesn't affect the semantics then there is no need to remove the ambiguity. For example for `Seq → Seq ';' Seq | Statement` and `Statement → id` it doesn't matter what the parse tree is.
- **Note2**: there is no algorithm for removing ambiguity in context-free grammars. In fact there is no algorithm for determining whether a grammar is ambiguous!
- **Note3**: certain context-free formal languages are ***inherently ambiguous*** - every grammar for the language is ambiguous. For example the formal language $\{0^a1^b2^c$, where a=b or b=c$\}$ is inherently ambiguous.

Extended Backus-Naur Form (EBNF)

- As we have seen, the rules for a context free grammar take the following general form: `N` → α where `A` is a non-terminal and α is a possibly empty sequence of terminals and non-terminals. Note: N can be defined more than once.
- In the computer science community this basic form is often referred to as **Backus-Naur**[12] Form (BNF) after two of the early pioneers of programming language design.
- Because Repetitive and Optional constructs are very common in programming languages many authors/systems use an Extended form of BNF for writing context free grammars - EBNF. EBNF doesn't make the grammar any more powerful however.

- Repetition: {α} 0 or more occurrences of α §
  Optional: [α] 0 or 1 occurrence of α
  Choice: α|β α or β.
  Grouping: (α) α. Useful for grouping alternatives within a rule e.g. (α|β)

- Examples:

BNF:	`Seq → Stmt ';' Seq`	EBNF:	`Seq → Stmt { ';' Stmt }`	left recursive or
	`Seq → Stmt`		`Seq → { Stmt ';' } Stmt`	right-recursive
BNF:	`Exp → Exp AddOp Term`	EBNF:	`Exp → Term { AddOp Term }`	left associative or
	`Exp → Term`		`Exp → Term [ AddOp Exp ]`	right-associative

- For the expression example above, an EBNF version is:

```
Expression  →  Term {('+'|'-') Term}
Term        →  Factor {('*'|'/') Factor }
Factor      →  '(' Expression ')' | num
```

Efficiency of Parsers for Context Free Grammars

- Given that most programming languages can be described by a context-free grammar, why are they not implemented in practice?
- The primary reason is that the **worst-case time complexity** to parse using a context-free grammar is $O(n^3)$. This is unacceptable for all but the smallest of programs.
- Fortunately, there are subsets of context-free grammars that can parsed in O(n) time and are able to describe the syntax of most programming languages. The two subsets that we will consider are **LL(1)** grammars and **LR(1)**[13] grammars. LR(1) grammars are a superset of LL(1) grammars.
- The main issue to resolve is to ensure that the context-free grammar is amenable to LL(1) or LR(1) parsing

LL(1) Grammars

- A context-free grammar where it is possible to determine which rule to use in a leftmost-derivation by scanning the input left-to-right (beginning-to-end) and by looking at the next token only is called LL(1). LL(*k*) parsers have *k*-token lookahead.
- LL parsers construct a parse-tree from the root (from the start symbol) down to the leave nodes (the terminal symbols), and know at each step what to do next.
- There are two main types of LL parser: recursive descent, and LL parse tables / pushdown automaton. We'll only consider recursive descent parsers.

---

[12] John Backus and Peter Naur who used BNF to define the syntax of Algol 60. John Backus also designed Fortran.
[13] and its more common variant: **LALR(1).**

- Recursive descent parsers can be written by hand or generated by a parser-generator tool like **ANTLR**.
- Like ambiguity removal for a context-free grammar, there is no automatic method of ensuring that a context-free grammar is an LL(1) grammar. We need to rewrite the grammar if any LL(1) issues arise.

Recursive Descent Parser

- We'll first outline a method to translate a set of EBNF rules into code.
  - Our recursive descent parsers will consist of:
  - a set of *parse functions,* one for each non-terminal `A` e.g. `A()` Note: the parse function could also return an AST for the rule.
  - the *current input token* e.g. global variable `token`
  - a *token match and advance to next token* function, e.g.

```
def match(expected):
     if token==expected: token=lex.get_token()
     else error("unexpected token encountered...")
```

- A a non-terminal `N` will be translated to `N()`
- A terminal *T* will be translated to `match(T)`
- Concatenation `A B` becomes

```
A(); B();
```

- Alternation `A | B` becomes

```
if token in FIRST(A):       A()
elif token in FIRST(B):     B()
```

- Repetition `{A}` becomes

```
while token in FIRST(A):    A()
```

- Optional `[A]` becomes

```
if token in FIRST(A): A()
```

- For the translations above, use `match(A)` instead of `A()` if *A* is a terminal.
- **FIRST(A)** corresponds to the set of all tokens that could possibly start an `A`. If A is empty, then include tokens that could follow A also.
- **FOLLOW(A)** corresponds to the set of all tokens that could possibly follow an `A` anywhere in the grammar.
- For alternation, if the FIRST sets of the alternatives are not disjoint, e.g. if there is a token in both FIRST(A) and FIRST(B) then the grammar is not LL(1).

- In our code, token types will be given in uppercase, for example IF for the *if* token.

Example:

```
Statement      → IfStatement | BeginStatement | PrintStatement
IfStatement    → if Expression then Statement [else Statement]
BeginStatement → begin Statement { ';' Statement } end
PrintStatement → print Expression
```

```
def Statement():
  if token == IF:     IfStatement()
  elif token == BEGIN:       BeginStatement()
  elif token == PRINT:       PrintStatement()

def IfStatement():
  match(IF)
  Expression()
  match(THEN)
  Statement()
  if token == ELSE:
       match(ELSE)
       Statement()
  else error()

def BeginStatement():
  match(BEGIN)
  Statement()
  while token == SEMICOLON:
     match(SEMICOLON)
     Statement()
  match(END)

def PrintStatement():
  match(PRINT)
  Expression()
```

-

- **Parsing and Building an Abstract Syntax Tree**

- The example above will recognise a legal statement but will not produce a parse tree nor an abstract syntax tree that captures the syntax of the program for use by the semantic analysis or code generation phases of the compiler.
- We can adapt our approach by returning an AST node from each parse function.
- We do this by defining the following classes:

```
class StatementAST extends AST:
  Position p  # useful for error handling

class IfAST extends StatementAST:
  ExprAST e
  StatementAST thenS, elseS

class BeginAST extends StatementAST:
  StatementAST statlist[]    # list

class PrintAST extends StatementAST:
  ExprAST e
```

- We can now implement the parser for our example as follows:

```
def Statement():
```

```
  if   token == IF:    return IfStatement()
  elif token == BEGIN: return BeginStatement()
  elif token == PRINT: return PrintStatement()
  else error()

def IfStatement():
  match(IF)
  e = Expression()
  match(THEN)
  thenS = Statement()
  if token == ELSE:
     match(ELSE)
     elseS = Statement()
  else:
     elseS = None
  return IfAST(e,thenS,elseS)

def BeginStatement():
  match(BEGIN)
  statlist = []
  statlist.append(Statement())
  while token == SEMICOLON:
     match (SEMICOLON)
     statlist.append(Statement())
  match(END)
  return BeginAST(statlist)

def PrintStatement():
  match(PRINT)
  return PrintAST(Expression())
```

Expression calculator

- The approach to translating EBNF into parsing functions can also be used for other tasks. For example for calculating expressions with correct precedence and associativity we could use:

```
def Expression:
   result = Term()
   while token in {PLUS, MINUS}:
        op=token; match(op); right=Term()
        if op==PLUS: result += right else result -= right
  match(token);
  return result

def Term:
   result = Factor()
   while token in {MULT, DIVIDE}:
        op=token; match(op); right=Factor()
        if op==MULT:  result *= right
        else if right !=0:  result /= right
        else error("Divide by zero")
  match(token);
  return result

def Factor:
   if token == LPAREN:
       match(LPAREN);  result=Expression(); match(RPAREN)
   elif token === NUM: result=token.value; match(token);
   else error("(" or number expected.")
   return result
```

CFG to LL(1)

- In order to employ top-down parsing we need to ensure that our grammar is LL(1). Unfortunately transforming a non-LL(1) context-free grammar to LL(1) cannot be fully automated and often requires care and a little ingenuity. In particular we need to ensure that the transformed grammar has the same semantics as the original grammar and if possible retains the readability of the original.

- The following 3 transformations are commonly tried: (1) Left Factorisation, (2) Substitution, (3) Left Recursion Removal. Left recursion removal and substitution are normally carried out first then left factorisation.

- (1) ***Left Factorisation***. If two or more alternatives of a rule share a common prefix we can factor out the common prefix using Left Factorisation.
  In EBNF: `A → B C | B D` can be changed to `A → B (C | B)`
  `A → B C | B` can be changed to `A → B [C]`
  In BNF: `A → B C | B D` can be changed to `A → B X` and `X → C | D`
  `A → B C | B` can be changed to `A → B X` and `X → C | ε`
  If we don't do this the parser won't know whether to choose the first alternative or the second.

- (2) ***Substitution*** involves replacing a non-terminal `N` on the right-hand side of a rule by each of the alternatives of `N`. Substitution is useful when conflicts are "indirect" in order to make the conflict "direct" and hopefully amenable to left factoring. Substitution is also used to write clearer grammars.
- Consider the following example:

```
ForStatement → for ControlVar ':= ' Expr Direction Expr do Statement
ControlVar   → id
Direction    → to | downto
```

- Although this grammar is LL(1) it seems sensible to remove both `ControlVar` and `Direction` using substitution. The `ControlVar` rule seems to be defined purely as commentary - it serves no grammatical role otherwise. `Direction` is defined as a rule because the above grammar is given in BNF which doesn't support grouping of alternatives. By using substitution and EBNF we obtain a simpler rule:

```
ForStatement → for id ':=' Expr (to | downto) Expr do Statement
```

- Now consider the following example:

```
Statement  → Assignment | Call | pass
Assignment → id '=' Expr
Call       → id '(' Expr ')'
```

- The grammar is not LL(1) because *id* can start both `Assignment` and `Call` and these are alternatives of Statement. In addition the grammar isn't in a form suitable for left factorisation. We can however bypass these issues by substituting `Assignment` and `Call` in Statement and then left factorising `Statement`:

```
Statement → id ('=' Expr | '(' Expr ')' ) | pass
```

- Although this form of the grammar is LL(1) it merges/obscures the clean separation in the original grammar. An LL(1) parser will need to handle this by re-introducing ASTnodes and parse functions for the alternatives and passing id to them in order to preserve the original form in later passes.

- (3) ***Left Recursion Removal***. For example `Expr` → `Expr Op Expr` cannot be handled by an LL parser because we would recurse forever as soon as we enter the parse function. We need to eliminate left-recursion.
- We'll cover the common case of direct left recursion removal, which leaves the grammar largely intact. To eliminate a direct left recursive rule we re-write the rule using right-repetition (in BNF we can do the rewrite using a new auxiliary rule and right-recursion) – ANTLR 4 does this for you! Handling of other indirect recursive cases is more involved and often results in incomprehensible LL(1) grammars.
- In EBNF: `A` → `X` | `A Y` can be changed to `A` → `X {Y}`
  More generally we have:
  `A` → `X1` | `X2` | … | `XN` | `A Y1` | `A Y2` | … | `A YM` can be changed to
  `A` → (`X1` | `X2` | … | `XN`) { `Y1` | `Y2` | … | `YM` }

- ***LL(2) Parser***. When developing an LL(1) grammar and a hand-written parser, a practical solution to developing clean abstract syntax trees and nice code is to just to peek-ahead and look at another token. This is equivalent to an LL(2) parser. **ANTLR4** is capable of emitting LL(k) parsers.

A word about handling compilation errors

- The response of a compiler to errors is an important factor in its usefulness.
- Some compilers stop after producing a single error message.
- Most attempt some form of recovery and are capable of detecting further errors.
- Some compilers even attempt to correct syntactic errors in order to continue with semantic analysis and even code generation.
- The following are good guidelines:
  - Produce informative messages for the user.
  - Report the position of an error as accurately as possible.
  - If the compiler is part of an Integrated Development Environment (IDE), highlight the region of each error in an editor window.
  - If error recovery is attempted, skip as little as possible in order to parse as much of the remaining code as possible, avoid infinite looping in the recovery process, and avoid spurious error messages being generated as a result of the recovery.
  - If error correction is attempted, ensure that the corrected program has the same syntax tree and semantics as the original (assuming we know what that is!).

LR(1) Grammars

- The second class of practical (i.e. expressive and fast) parsers are those for LR(1) grammars.
- LR(1) grammars are a superset of LL(1) grammars and allow us to write clearer grammars, for example, they allow left recursion in rules.
- Unlike LL(1) parsers, LR(1) parsers are too complex to write by hand. We need to use a LR parser-generator tool like **bison** (GNU version of yacc), Sable CC, CUP, Beaver etc
- Most LR parser generators support a subset of LR(1), called **LALR(1)** which is more memory efficient, but very close in expressiveness to LR(1). We will assume that LR tools support LR(1) grammars even though many actually support LALR(1) grammars. We won't look at how LR parser generators are implemented [14].

- LR(1) builds the parse tree bottom up, from the leaves (non-terminals) to the root (start symbol).
- They achieve this by ***reducing*** child nodes corresponding to the RHS (***handle***) of a rule to a parent node corresponding to the left-hand side of the rule. A stack is used to maintain the current state of the parse.
- For example, given the input 9+4 and the grammar

```
Expression → Expression '+' Term |Term
Term       → Term  '*' Factor | Factor
Factor     → '(' Expression ')' | num
```

- an LR parser will proceed to build the parse tree as follows

	Stack Action	Stack	Tree Action	Parse Tree
1	**Shift**[15] 9 onto the stack.	9	Create a leaf node for 9	9
2	Reduce 9 i.e. recognise that 9 is a *handle* (corresponds to a RHS for a rule) and replace 9 by Factor on stack.	Factor	Create a Factor parent node for 9 node	F | 9
3	Reduce Factor, i.e. recognise that `Factor` is a handle and replace Factor by Term on stack	Term	Create a parent `Term` node for Factor	T | F | 9
4	Shift[16] + onto the stack and a	Term+	Create a leaf node for +	T + | F | 9
5	Shift 4 onto the stack	Term+4	Create a leaf node for 4	T + 4 | F | 9
6	Reduce 4 to Factor	Term+Factor	Create a parent Factor	T + F | |

[14] Interested students are highly encouraged to study the implementation of LR parsers in textbooks and also last year's slides.
[15] Traditional LR parsing term for *push*.
[16] Why not reduce Term to Factor?

			node for 4.	F 4 | 9
7	Reduce[17] Term+Factor to Term	Term	Create a parent Term node for Term, +, Factor child nodes	T / | \ T + F | | F 4 | 9
8	Reduce Term to Expression	Expression	Create a parent Expression node for Term child node	E | T / | \ T + F | | F 4 | 9

•

[17] Again why not reduce Factor to Term?