Notes on
Formal Language Theory
and Parsing

James Power

Department of Computer Science
National University of Ireland, Maynooth
Maynooth, Co. Kildare, Ireland.
James.Power@May.ie

This version prepared on November 29, 2002

Contents

1 REGULAR LANGUAGES **1**
1.1 Regular Expressions . . . 1
1.2 Non-Deterministic Finite-State Automata . . . 2
1.2.1 Converting a regular expression to a NFA - Thompson's Algorithm . . . 3
1.2.2 Kleene's Theorem . . . 4
1.2.3 Converting a NFA to a Regular Expression . . . 5
1.3 Deterministic Finite-State Automata . . . 5
1.3.1 Constructing a DFA from an NFA ("Subset Construction") . . . 5
1.4 DFA minimisation . . . 6
1.5 Further Properties of Regular Languages . . . 6
1.5.1 Total DFA . . . 7
1.5.2 Closure of Regular Languages . . . 7
1.5.3 Other Decision Problems for FSAs . . . 8
1.6 Grammars . . . 8
1.7 Regular Grammars . . . 9
1.8 Converting an FSA to a Regular Grammar . . . 9
1.9 Limitations of Regular Languages . . . 10
1.9.1 The Pumping Lemma: Examples . . . 10

2 CONTEXT-FREE LANGUAGES **11**
2.1 The Chomsky Hierarchy . . . 11
2.2 Grammars for Programming Languages . . . 12
2.2.1 BNF Notation . . . 12
2.3 Derivations and Parse Trees . . . 13
2.3.1 Leftmost and Rightmost Derivations . . . 14
2.4 Modifying Grammars . . . 15
2.4.1 Left Factoring . . . 15
2.4.2 Eliminating Ambiguity . . . 15
2.4.3 Eliminating Immediate Left Recursion . . . 17
2.4.4 Eliminating Indirect Left Recursion . . . 18
2.4.5 Eliminating ε-productions . . . 19
2.4.6 Eliminating Unit Productions . . . 19
2.5 PUSH-DOWN AUTOMATA . . . 19
2.5.1 Formal Definition . . . 19
2.5.2 Converting a CFG to a PDA . . . 20
2.5.3 Converting a PDA to a CFG . . . 20
2.5.4 Deterministic PDAs . . . 21
2.5.5 Non-Deterministic CFLs . . . 21
2.6 Proving that a Language is not Context-Free . . . 22
2.6.1 The Pumping Lemma for context-free languages . . . 22
2.6.2 The Pumping Lemma: Examples . . . 23
2.6.3 Limitations of the Pumping Lemma . . . 23
2.7 The Closure of Context-Free Languages . . . 23
2.8 Decision Problems for Context-Free Languages . . . 24

3 TOP-DOWN PARSING **26**
3.1 Recursive Descent Parsing . . . 26
3.2 Lookahead . . . 26
3.2.1 FIRST and FOLLOW . . . 27
3.3 Implementing a Parser . . . 27
3.4 The $LL(k)$ Parsers . . . 28
3.5 Early's Parser . . . 28
3.5.1 Early's Items . . . 29
3.5.2 Early's Algorithm . . . 29
3.6 Unger's Method . . . 30

4 BOTTOM-UP PARSING **32**
4.1 Bottom-Up (Shift-Reduce) Parsing . . . 32
4.2 $LR(0)$ items . . . 33
4.2.1 Kernel Items . . . 33
4.2.2 The closure and move operations on items . . . 33
4.3 Handle-recognising DFAs and LR(0) Parsers . . . 34
4.3.1 Construction of a handle-recognising DFA . . . 34
4.3.2 Operation of a handle-recognising DFA . . . 34
4.3.3 The $LR(k)$ parsers . . . 34
4.3.4 Inadequte States . . . 35
4.4 SLR(1) Parsers . . . 35
4.5 $LR(1)$ Parsers . . . 35
4.5.1 Constructing LR(1) Item sets . . . 36
4.6 $LALR(1)$ Parsers . . . 36
4.7 Tomita's Parser . . . 36
4.8 CYK Method . . . 37

REFERENCES

The Art of Compiler Design by T. Pittman & J. Peters (Prentice-Hall, 1992).

Lex and Yacc by T. Mason and D. Brown (Addison-Wesley, 1992).

Additional References

Introduction to Compiler Construction with UNIX by A Schreiner & H Friedman (Prentice Hall, 1985).

Introduction to Compiling Techniques by J. Bennett (McGraw-Hill, 1990).

Compilers: Principles, Techniques and Tools by A. Aho, R. Sethi & J.Ullman (Addison-Wesley, 1986).

Introduction to Languages and the Theory of Computation by J. Martin (McGraw-Hill, 1991).

Compiler Design in C by A. Holub (Prentice-Hall, 1990).

The theory and practice of compiler writing by J.Tremblay & P. Sorenson. (McGraw-Hill, 1985).

Elements of the Theory of Computation by H. Lewis & C. Papadimitriou. (Prentice-Hall, 1981).

Crafting a Compiler by C. Fischer & R. LeBlanc. (Benjamin/Cummings, 1988).

Syntax of Programming Languages by R. Backhouse. (Prentice-Hall, 1979).

1 REGULAR LANGUAGES

Lexical analysis, also called scanning, is the phase of the compilation process which deals with the actual program being compiled, character by character. The higher level parts of the compiler will call the lexical analyzer with the command "get the next word from the input", and it is the scanner's job to sort through the input characters and find this word.

The types of "words" commonly found in a program are:

- programming language keywords, such as `if, while, struct, int` etc.
- operator symbols like `=, +, -, &&, !, <=` etc.
- other special symbols like: `( ), { }, [ ], ;, &` etc.
- constants like `1, 2, 3, 'a', 'b', 'c', "any quoted string"` etc.
- variable and function names (called identifers) such as `x, i, t1` etc.

Some languages (such as C) are case sensitive, in that they differentiate between eg. `if` and `IF`; thus the former would be a keyword, the latter a variable name.

Also, most languages would insist that identifers cannot be any of the keywords, or contain operator symbols (versions of Fortran don't, making lexical analysis quite difficult).

In addition to the basic grouping process, lexical analysis usually performs the following tasks:

- Since there are only a finite number of types of words, instead of passing the actual word to the next phase we can save space by passing a suitable representation. This representation is known as a *token.*
- If the language isn't case sensitive, we can eliminate differences between case at this point by using just one token per keyword, irrespective of case; eg.

```
#define IF-TOKEN 1
#define WHILE-TOKEN 2
.....
.....
if we meet "IF", "If", "iF", "if"    then return IF_TOKEN
if we meet "WHILE, "While", "WHile", ...   then return WHILE-TOKEN
```

- We can pick out mistakes in the lexical syntax of the program such as using a character which is not valid in the language. (Note that we do not worry about the combination of patterns; eg. the pattern of characters `"+*"` would be returned as `PLUS-TOKEN, MULT-TOKEN`, and it would be up to the next phase to see that these should not follow in sequence.)
- We can eliminate pieces of the program that are no longer relevant, such as spaces, tabs, carriage-returns (in most languages), and comments.

In order to specify the lexical analysis process, what we need is some method of describing which patterns of characters correspond to which words.

1.1 Regular Expressions

Regular expressions are used to define patterns of characters; they are used in UNIX tools such as *awk, grep, vi* and, of course, *lex.*

A regular expression is just a form of notation, used for describing sets of words. For any given set of characters Σ, a **regular expression over** Σ is defined by:

- The **empty string**, ε, which denotes a string of length zero, and means "take nothing from the input". It is most commonly used in conjunction with other regular expressions eg. to denote optionality.
- Any character in Σ may be used in a regular expression. For instance, if we write `a` as a regular expression, this means "take the letter `a` from the input"; ie. it denotes the (singleton) set of words {"a"}
- The **union** operator, "`|`", which denotes the union of two sets of words. Thus the regular expression `a|b` denotes the set {"a", "b"}, and means "take either the letter `a` or the letter `b` from the input"

- Writing two regular expressions side-by-side is known as **concatenation**; thus the regular expression `ab` denotes the set {"ab"} and means "take the character `a` followed by the character `b` from the input".
- The **Kleene closure** of a regular expression, denoted by "*", indicates zero or more occurrences of that expression. Thus `a*` is the (infinite) set {ε, "a", "aa", "aaa", ...} and means "take zero or more `a`s from the input".

Brackets may be used in a regular expression to enforce precedence or increase clarity.

1.2 Non-Deterministic Finite-State Automata

In order to try and understand how *lex* builds a scanner, we will construct a formal model of the scanning process. This model should take characters one-by-one from the input, and identify them with particular patterns which match up with a regular expression.

The standard model used is the **finite-state automaton**. We can convert any definition involving regular expressions into an implementable finite automaton in two steps:

$$\text{Regular expression} \longleftrightarrow \text{NFA} \longleftrightarrow \text{DFA}$$

A **non-deterministic finite-state automaton** (NFA) consists of:

- An alphabet Σ of *input symbols*
- A set of *states*, of which we distinguish:
  - a unique *start state*, usually numbered "0"
  - a set of *final states*
- A transition relation which, for any given state and symbol gives the (possibly empty) set of next states

We can represent a NFA diagrammatically using a (labelled, directed) graph, where states are represented by nodes (circles) and transitions are represented by edges (arrows).

The purpose of an NFA is to model the process of reading in characters until we have formed one of the words that we are looking for.

How an NFA operates:

- We begin in the start state (usually labelled 0) and read the first character on the input.
- Each time we are in a state, reading a character from the input, we examine the outgoing transitions for this state, and look for one labelled with the current character. We then use this to move to a new state.
  - There may be more than one possible transition, in which case we choose one at random.
  - If at any stage there is an output transition labelled with the empty string, ε, we may take it without consuming any input.
- We keep going like this until we have no more input, or until we have reached one of the final states.
- If we are in a final state, with no input left, then we have succeeded in recognising a pattern.
- Otherwise we must *backtrack* to the last state in which we had to choose between two or more transitions, and try selecting a different one.

Basically, in order to match a pattern, we are trying to find a sequence of transitions that will take us from the start state to one of the finish states, consuming all of the input.

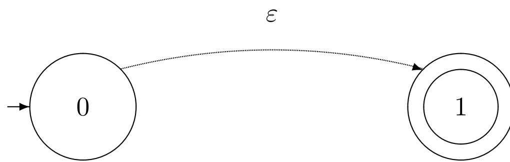
The key concept here is that: *every NFA corresponds to a regular expression*

Moreover, it is fairly easy to convert a regular expression to a corresponding NFA. To see how NFAs correspond to regular expressions, let us describe a conversion algorithm.

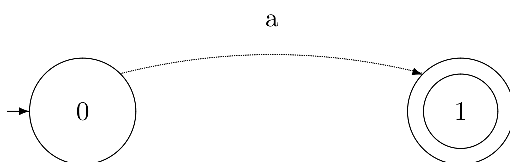
1.2.1 Converting a regular expression to a NFA - Thompson's Algorithm

We will use the rules which defined a regular expression as a basis for the construction:

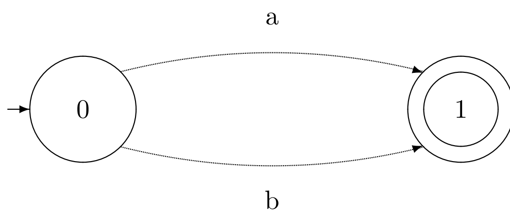
1. The NFA representing the empty string is:



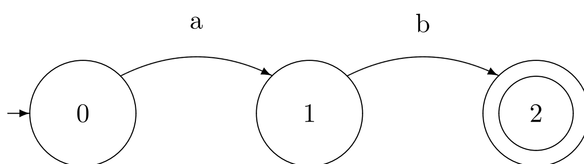
2. If the regular expression is just a character, eg. `a`, then the corresponding NFA is :



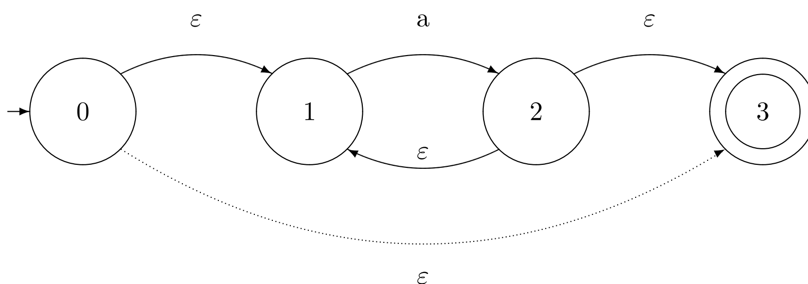
3. The union operator is represented by a choice of transitions from a node; thus `a|b` can be represented as:



4. Concatenation simply involves connecting one NFA to the other; eg. `ab` is:



5. The Kleene closure must allow for taking zero or more instances of the letter from the input; thus `a*` looks like:



1.2.2 Kleene's Theorem

We have shown how to convert a regular expression to an NFA; the proof that these represent the same language is straightforward (by induction). However, we have asserted that there is a correspondence in *both* directions; this is formalised in **Kleene's Theorem**:

Kleene's Theorem, part 1: To each regular expression there corresponds a NFA
Strategy: The corresponding NFA is the one constructed by Thompson's Algorithm; proof that it is *equivalent* is by induction over regular expressions.
Proof: Boring!

Kleene's Theorem, part 2: To each NFA there corresponds a regular expression
Strategy: Proof by induction over the states of the NFA (sort of).
The language represented by a NFA can be partitioned into the union of a number of smaller languages, each defined as follows:

> Let the states of the NFA be numbered from 1 to N.
> Let p and q be states, and let J be a number such that $0 \leq J \leq N$.
> Then:

$$L(p,q,J) = \left\{ \; x \in \Sigma^* \;\middle|\; \begin{array}{l} x \text{ corresponds to a path in the NFA between } p \text{ and } q \\ \text{that passes through no state numbered as high as } J \end{array} \right\}$$

Note that if S is the start state, then the union of all $L(S, F, N+1)$ for all finish states F is the language accepted by the NFA.

Equally, if we can show that $L(p, q, J)$ has a corresponding regular expression for all p, q and J, we will have proved the theorem. The proof of this will proceed by induction over J.

Proof: (Kleene's Theorem, part 2)

- Base Case: $J = 1$
  We must be able to get from p to q without passing through *any* states; therefore either p=q, or we go directly in one transition from p to q. The only strings that can be recognised in this case are ε and single characters from Σ. The rules R_1 and R_2 give us a regular expression corresponding to these situations.

- Inductive Step: $J = K + 1$
  The induction hypothesis is that for some K such that $1 \leq K \leq N$, the language $L(p, q, K)$ has a corresponding regular expression. Now we must prove that, based on this assumption, $L(p, q, K+1)$ has a corresponding regular expression.

  Suppose the machine $L(p, q, K + 1)$ consumes some string x; we must find a regular expression corresponding to x. Now suppose also that it passes through state K on its way (if it doesn't, then x is in $L(p, q, K)$, which has a corresponding regular expression by the induction hypothesis). Since the machine can "loop" on K arbitrarily many times, we can split x up into three substrings:

  a which moves the machine from state p to state K

  b which causes the machine to "loop" on state K

  c which moves the machine from state K to state q

  We note that while any of the above strings may cause us to start or finish at state K, none of them actually cause us to move *through* K. Thus we have:

  $$\begin{array}{lcl} a & \in & L(p, K, K) \\ b & \in & L(K, K, K) \\ c & \in & L(K, q, K) \end{array}$$

  By the induction hypothesis each of these have corresponding regular expressions, say `A`, `B` and `C` respectively, and thus the regular expression for x is `A(B*)C`.

  Since our choice of x was arbitrary, we have proved the theorem.

QED.

1.2.3 Converting a NFA to a Regular Expression

The last theorem actually yields an algorithm for converting a NFA to a regular expression. The proof was by induction; we can transform this into a recursive function where the recursion is based on an integer argument.

We note that the induction step tells us that to calculate the language $L(p, q, K+1)$ for any $K > 1$, it will either be:

- the language $L(p, q, K)$, or
- the language $(L(p, K, K).L(K, K, K) * .L(K, q, K))$.

If we represent this "or" by the union operation, we then have a recursive definition of $L(p, q, K+1)$ in terms of languages of the form $L(_, _, K)$.

The base case is where $K = 1$. As we have seen in the proof of Kleene's theorem, these machine components correspond to one-step transitions in the FSA, and the equivalent regular expressions are thus just the symbols which cause those transitions.

Based on this we can define function `L`, which takes three numbers as arguments and returns a regular expression, as follows:

```
regular-expression L(int I, int J, int K)
{
if (K==1)
  return {a | there is an 'a' transition between states I and J}
else
  return  L(p,q,K-1) | (L(p,K-1,K-1).L(K-1,K-1,K-1)*.L(K-1,q,K-1));
}
```

Given any FSA with N states altogether, where S is the start state and F is the final state, we convert it to a regular expression by calling the function `L(S,F,N+1)`

1.3 Deterministic Finite-State Automata

NFAs are useful, in that they are easily constructed from regular expressions, and give us some kind of computational idea of how a scanner might work. However, since they involve making decisions, and backtracking if that decision was wrong, they are not really suitable for implementation using conventional programming languages.

Instead, we use a **Deterministic Finite-State Automaton** (DFA) which is a special case of a NFA with the additional requirements that:

1. There are no transitions involving ε
2. No state has two outgoing transitions based on the same symbol

Thus, when we are in some state in a DFA, there is no choice to be made; the operation of a DFA can very easily be converted into a program.

It is vital to note that: **every NFA can be converted to an equivalent DFA**

We will define an algorithm to do this, for arbitrary NFAs; the basic idea here is that sets of states in the NFA will correspond to just one state in the DFA.

1.3.1 Constructing a DFA from an NFA ("Subset Construction")

We merge together NFA states by looking at them from the point of view of the input characters:

- From the point of view of the input, any two states that are connected by an ε-transition may as well be the same, since we can move from one to the other without consuming any character. Thus states which are connected by an ε-transition will be represented by the same states in the DFA.
- If it is possible to have multiple transitions based on the same symbol, then we can regard a transition on a symbol as moving from a state to a set of states (ie. the union of all those states reachable by a transition on the current symbol). Thus these states will be combined into a single DFA state.

To perform this operation, let us define two functions:

- The **ε-closure** function takes a state and returns the set of states reachable from it based on (one or more) ε-transitions. Note that this will always include the state tself. We should be able to get from a state to any state in its ε-closure without consuming any input.
- The function **move** takes a state and a character, and returns the set of states reachable by one transition on this character.

We can generalise both these functions to apply to sets of states by taking the union of the application to individual states.

Eg. If A, B and C are states, move({A,B,C},'a') = move(A,'a') $\cup$ move(B,'a') $\cup$ move(C,'a').

The Subset Construction Algorithm

1. Create the start state of the DFA by taking the ε-closure of the start state of the NFA.
2. Perform the following for the new DFA state:
   For each possible input symbol:
   (a) Apply move to the newly-created state and the input symbol; this will return a set of states.
   (b) Apply the ε-closure to this set of states, possibly resulting in a new set.

   This set of NFA states will be a single state in the DFA.
3. Each time we generate a new DFA state, we must apply step 2 to it. The process is complete when applying step 2 does not yield any new states.
4. The finish states of the DFA are those which contain any of the finish states of the NFA.

1.4 DFA minimisation

Since we are interested in translating a DFA into a program, we will want to ensure that this program is as efficient as possible. In automata terms, one aspect of this will be to ensure that the constructed DFA has as few states as possible. This is achieved by an algorithm known as **DFA minimisation**.

We take the "optimistic" approach: we start by assuming that all states are actually the same, and only distinguish those which we can prove are different. As for subset construction, the states of the minimised DFA will be sets of states from the original DFA; these will be states that we couldn't tell apart!

Two states are different if:

- one is a final state and the other isn't, or
- the transition function maps them to different states, based on the same input character

We base our algorithm on partitioning the states using this criterion.

1. Initially start with two sets of states: the final, and the non-final states.
2. For each state-set created by the previous iteration, examine the transitions for each state and each input symbol. If they go to a different state-set for any two states, then these should be put into different state-sets for the next iteration.
3. We are finished when an iteration creates no new state-sets.

1.5 Further Properties of Regular Languages

Since we have shown the equivalence of regular languages and FSAs, we can prove facts about language membership by describing constructions on their corresponding automata. Also, we show that it is possible to prove facts *other* than language membership using a FSA.

1.5.1 Total DFA

The last type of FSA that we need to look at is the *total* DFA - this has properties which we will need in the next section.

A total FSA is one where each state has exactly one transition to some other state for *every* input symbol. Thus there are no "gaps" in the transition table for the machine: given any state and any input symbol, it will always be possible to make a transition.

We can convert any ordinary DFA to a total DFA as follows:

1. Create some new state S, which we will use for "stuck" computations
2. For each input symbol, make a transition from state S to itself: thus it is never possible to leave this state.
3. For each other state S', and for every possible input symbol a, check to see if S' has an outgoing transition on a; if it doesn't, then make a new transition from S' to S on a.

It can be easily seen that this DFA is total.

1.5.2 Closure of Regular Languages

By the definition of regular expressions (and by Thompson's algorithm), we know that the union, concatenation and Kleene closure of regular languages must also be a regular language. It is reasonable to ask: does this hold for other "set-theoretic" operations, such as intersection, difference and complement? We prove that these languages are regular by constructing the appropriate automata.

Let L_1 and L_2 be regular languages over the alphabet Σ, and let M_1 and M_2 be their corresponding *total* DFAs. Let us define the components of M_1 and M_2 as follows:

- let the set of states of M_1 be P, from which we distinguish:
  - p_0, the start state of M_1
  - P_f, the set of final states of M_1
- let D_1 be the transition function for M_1; that is, the (partial) function: $D_1 : P \times \Sigma \longrightarrow P$
- let the set of states of M_2 be Q, from which we distinguish:
  - q_0, the start state of M_2
  - Q_f, the set of final states of M_2
- let D_2 be the transition function for M_2; that is, the (partial) function: $D_2 : Q \times \Sigma \longrightarrow Q$

Now we can define:

1. The intersection of L_1 and L_2 is the machine $M_1 \cap M_2$ defined by:
   - the set of states is $P \times Q$, the Cartesian product of P and Q. Thus for any $p \in P$ and $q \in Q$, the pair (p, q) is a state. We distinguish:
     - (p_0, q_0), the start state of $M_1 \cap M_2$
     - $\{(p, q) \mid p \in P_f \text{ and } q \in Q_f\}$, the set of final states of $M_1 \cap M_2$
   - the transition function: $D_{P \times Q} : (P \times Q) \times \Sigma \longrightarrow (P \times Q)$ is defined, for any character 'a', by: $D_{P \times Q}((p, q), a) = (D_1(p, a), D_2(q, a))$

2. The difference of L_1 and L_2 is the machine $M_1 - M_2$, defined as for intersection, except that the set of final states is now: $\{(p, q) \mid p \in P_f \text{ and } q \notin Q_f\}$

3. The complement of the language L_1 is the machine M_1', which is simply M_1 with the final and non-final states swapped; ie. the set of final states is: $P - P_f$

1.5.3 Other Decision Problems for FSAs

We use a FSA to answer a question of the form: "is x in the language L?". This is known as a **decision problem**, since the answer will be "yes" or "no". We can pose other decision problems for regular languages:

DP1 For any regular language L, is $L = \emptyset$?

DP2 For any two regular languages L_1 and L_2, is $L_1 \subset L_2$?

DP3 For any two regular languages L_1 and L_2, is $L_1 = L_2$?

Once again, we approach these problems by examining the corresponding FSAs. The corresponding FSA decision problem, and the decision algorithm, for the above are:

DP1 Given a FSA M, does M accept *any* strings?
A FSA accepts no strings if either there are no final states, or if there is no path from the start state to a final state. To see if this is so, we need to work out which states are "reachable" from the start state. Construct the following sets of states:

- R_0 just contains the start state
- R_i is all the states in R_{i-1}, along with any states reachable in one transition from a state in R_{i-1}

We stop the construction when $R_i = R_{i-1}$ (this has to happen, as there are only a *finite* number of states altogether). When this happens, just examine the states in R_i; if this set doesn't contain any final states, then M accepts no strings.

DP2 Given two FSAs M_1 and M_2, is the language accepted by M_1 a subset of that accepted by M_2?
We note that $L_1 \subset L_2$ if and only if $L_1 - L_2 = \emptyset$. Thus our algorithm is: construct the machine $M_1 - M_2$ (as in the last section), and apply the algorithm for $[DP1]$

DP3 Given two FSAs M_1 and M_2, is the language accepted by L_1 equal to that accepted by M_2?
We note that $L_1 = L_2$ if and only if they are subsets of each other. Thus we simply apply $[DP2]$ twice.

1.6 Grammars

Basically, a grammar consists of a set of **production rules** of the form:

$$A \rightarrow \alpha$$

where A is a symbol known as a **non-terminal**, and α is a sequence of symbols.

Each production rule can be seen as giving the definition of the non-terminal on the left-hand-side; the above rule states that "whenever we see an A, we can replace it with α".

A non-terminal may have more than one definition, in which case we use "|", the union operator; eg.

$$\begin{array}{rcl} A & \rightarrow & \alpha \\ & | & \beta \end{array}$$

means that "whenever we see an A, we can replace it with α or with β".

We distinguish one special non-terminal in the grammar called the **start symbol**, usually written S. The production rules for this symbol are usually written first in a grammar.

Any symbol used in the grammar which does not appear on the left-hand-side of some rule (ie. has no definition) is called a **terminal** symbol.

The following is an example of a grammar:

$$\begin{array}{rcl} S & \rightarrow & X\mathbf{c} \\ X & \rightarrow & YX \\ & | & \varepsilon \\ Y & \rightarrow & \mathbf{a} \\ & | & \mathbf{b} \end{array}$$

The non-terminals of the grammar are S, X and Y; the terminals are **a, b, c**.

A grammar works by starting with the start symbol, and successively applying the production rules (replacing the l.h.s. with the r.h.s.) until we get a word which contains no non-terminals; this is known as a **derivation**.

Anything which we can derive from the start symbol by applying the production rules is called a **sentential form**. The last sentential form which contains no non-terminals is called a **sentence**.

Note that any grammar may have an infinite number of sentences which can be derived; the set of all these sentences is the language defined by the grammar.

Some experimentation with the above grammar will show that it can derive all words which start with arbitrarily many 'a's or 'b's and finish with a 'c' - this is the language defined by the regular expression `(a|b)*c`.

In fact, every regular expression can be converted to a grammar.

However, not every grammar can be converted back to a regular expression; any grammar which *can* is called a **regular grammar**; the language it defines is a regular language.

$$\textit{Regular Expression} \longrightarrow \textit{Grammar}$$

$$\textit{Regular Expression} \longleftarrow \textit{Regular Grammar}$$

1.7 Regular Grammars

A regular grammar is a grammar where all of the production rules are of one of the following forms:

$$\begin{array}{rcl} & A & \rightarrow & \mathbf{a}B \\ or & A & \rightarrow & \mathbf{a} \end{array}$$

where A and B represent any single non-terminal, and **a** represents any single terminal, or the empty string.

If a grammar can only derive regular languages, but is not of the above form, then (we claim) it can be converted to this form. In order to justify our assertion that we can represent all regular expressions in this manner, we give an algorithm for converting from an FSA to a regular grammar.

1.8 Converting an FSA to a Regular Grammar

We will let each state in the NFA correspond to a non-terminal in the grammar. Each symbol used on the transitions corresponds to a terminal symbol.

Let S_i denote the non-terminal corresponding to state i of the NFA. Then

1. The start symbol of the grammar is S_0, the non-terminal corresponding to the start state of the NFA
2. For each transition from state i to state j on some symbol 'a', create a production rule of the form: $S_i \rightarrow \mathbf{a}S_j$
3. For each state i of the NFA which is a finish state, create a production rule of the form: $S_i \rightarrow \varepsilon$

Note that ε-transitions between states would generate production rules of the form $S_i \rightarrow S_j$, which says that "wherever we see S_i we can replace it with S_j" - another indication that ε-transitions denote equivalence.

This process will work equally well with a DFA, and can be applied in reverse to a regular grammar to produce the corresponding NFA.

$$\textit{Regular Expression} \longleftrightarrow \textit{NFA or DFA} \longleftrightarrow \textit{Regular Grammar}$$

1.9 Limitations of Regular Languages

Since regular grammars are a special type of grammar, it is reasonable to assume that the set of regular languages is a subset of all possible languages. So what type of languages aren't regular?

First of all, we note one very important point:

Every finite language is regular

Any language is just a set of strings; a finite language is a set which contains a finite number of strings. We know this is regular, since if we have a finite language $\{w_1, w_2, ..., w_n\}$, we can rewrite it as the regular expression: `w1 | w2 | ... | wn`.

How do we form an infinite regular language? The union or concatenation of two finite languages will still be a finite language; thus, any *infinite* regular language must be formed from a regular expression which uses Kleene closure.

That means the regular expression which represents it must be of the form `X(Y*)Z`, where `X, Y` and `Z` are all regular expressions (possibly involving the use of Kleene-closure themselves). One implication of this is that every infinite regular language must have as a subset some set of words of the form xy^iz, for all $i \geq 0$, where x, y and z are strings which correspond to `X, Y` and `Z` respectively.

This conclusion is known as the **Pumping Lemma** for regular languages; ie:

if an infinite language L is regular,
then there exist three strings x, y and z such that
$y \neq \varepsilon$ and $\{xy^iz \mid i \geq 0\} \subseteq L$

1.9.1 The Pumping Lemma: Examples

Consider the following three languages:

$$\begin{array}{lcl} L_1 & = & \{a^nb^n \mid 0 \leq n \leq 100\} \\ L_2 & = & \{a^nb^n \mid n \geq 0\} \\ L_3 & = & \{a^nb^m \mid n, m \geq 0\} \end{array}$$

The first language is regular, since it contains only a finite number of strings.

The third language is also regular, since it is equivalent to the regular expression `(a*)(b*)`.

The second language consists of all strings which contain a number of 'a's followed by an *equal* number of 'b's.

Lemma: L_2 is not regular
Strategy: Proof by contradiction
Proof: Let us assume that it is regular; then we must have some set of strings of the form $\{xy^iz \mid i \geq 0\} \subseteq L$. Suppose such a subset did exist.

- Obviously, y could not contain a mixture of 'a's and 'b's, since this would mean that xy^iz would have 'b's before 'a's. Thus, y must consist solely of 'a's or solely of 'b's.
- Let us assume then that y consists solely of 'a's. Then if, for some n we have that xy^nz is in the language, there is no way that $xy^{n+1}z$ can be in the language, since this will contain extra 'a's without any extra 'b's. Thus there will be no way for xy^iz to be in the language for every $i \geq 0$.
- We can use exactly the same argument to refute the supposition that y can consist entirely of 'b's.

Thus we have shown that y cannot consists of 'a's 'b's or their mixture; ie no such y exists, and so the Pumping Lemma is not satisfied. Thus L_2 is not regular.

QED

(Note that we cannot draw any conclusions from the fact that $L_1 \subseteq L_2 \subseteq L_3$).

2 CONTEXT-FREE LANGUAGES

In this section we examine the most popular type of grammars - context-free grammars - and we consider methods of making them more efficient, as well as their inherent limitations.

2.1 The Chomsky Hierarchy

Obviously languages exist which are not regular; Noam Chomsky categorised regular and other languages as follows:

Language Class	Grammar	Automaton
3	Regular	NFA or DFA
2	Context-Free	Push-Down Automaton
1	Context-Sensitive	Linear-Bounded Automaton
0	Unrestricted (or *Free*)	Turing Machine

This is a hierarchy, so every language of type 3 is also of types 2, 1 and 0; every language of type 2 is also of types 1 and 0 etc.

The distinction between languages can be seen by examining the structure of the production rules of their corresponding grammar, or the nature of the automata which can be used to identify them.

- **Type 3 - Regular Languages**

  As we have discussed, a regular language is one which can be represented by a regular grammar, described using a regular expression, or accepted using an NFA or a DFA.

- **Type 2 - Context-Free Languages**

  A Context-Free Grammar (CFG) is one whose production rules are of the form:

  $$A \quad \rightarrow \quad \alpha$$

  where A is any single non-terminal, and α is any combination of terminals and non-terminals.

  A NFA/DFA cannot recognise strings from this type of language since we must be able to "remember" information somehow. Instead we use a Push-Down Automaton which is like a DFA except that we are also allowed to use a stack.

- **Type 1 - Context-Sensitive Languages**

  Context-Sensitive grammars may have more than one symbol on the left-hand-side of their production rules (provided that at least one of them is a non-terminal). However, the production rules must now obey the following:

  CS1 The number of symbols on the left-hand-side must not exceed the number of symbols on the right-hand-side

  CS2 We do not allow rules of the form $A \quad \rightarrow \quad \varepsilon$ unless A is the start symbol and does not occur on the right-hand-side of any rule.

  Since we allow more than one symbol on the left-hand-side, we refer to those symbols other than the one we are replacing as the **context** of the replacement.

  The automaton which recognises a context-sensitive language is called a linear-bounded automaton: this is basically a NFA/DFA which can store symbols in a list.

  Conditions CS1 and CS2 above mean that the sentential form in any derivation must always increase in length every time a production rule is applied. This basically means that the size of a sentential form is bounded by the length of the sentence (ie. word) we are deriving.

  Since the sentential form cannot thus grow infinitely large before deriving a sentence, a linear-bounded automaton always uses a *finitely*-long list as its store.

- **Type 0 - Unrestricted (Free) Languages**

  Free grammars have absolutely no restrictions on their grammar rules, (except, of course, that there must be at least one non-terminal on the left-hand-side).

  The type of automata which can recognise such a language is basically a NFA/DFA with an infinitely-long list at its disposal to use as a store; this is called a Turing machine.

2.2 Grammars for Programming Languages

Of the four grammar types, context-free grammars (CFG) represent the best compromise between power of expression and ease of implementation.

Regular languages are obviously too weak, since they can't even describe situations such as checking that the number of "begin" and "end" statements in a text are equal (this would be a variant of the language $a^n b^n$)

However, there are some structures which even context-free languages cannot describe.

For example, the language `(a|b)*c(a|b)*` is regular and thus context-free. However, if we add in the condition that the string before the 'c' must be the same as the string after the 'c', we get the language $\{\alpha c \alpha \mid \alpha \in (\mathtt{a}|\mathtt{b})*\}$, which is not even context-free.
This language abstracts the problem of having a declaration of a variable name before its use (with α being the variable name and 'c' the intervening program text).
In situations such as these, we would usually formulate a context-free grammar for the language in so far as it is possible, and leave details such as declaration/usage checking to the "semantic" phase.

2.2.1 BNF Notation

One of the earliest forms of notation used for describing the syntax of a programming language was **Backus-Naur Form** (BNF); this is basically just a variant of a context-free grammar, with the symbol "::=" used in place of "→" to mean "is defined as". Additionally, non-terminals are usually written in angle-brackets and terminals in quotes.

For example, we could describe a block of statements in Pascal as:

$$
\begin{array}{lll}
\langle \text{ block } \rangle & ::= & \text{“BEGIN” } \langle \text{ opt-stats} \rangle \text{ “END”}. \\
\langle \text{ opt-stats } \rangle & ::= & \langle \text{ stats-list } \rangle \mid \varepsilon. \\
\langle \text{ stats-list } \rangle & ::= & \langle \text{ statement } \rangle \mid \langle \text{ statement } \rangle \text{ “;” } \langle \text{ stats-list } \rangle.
\end{array}
$$

An extension of this notation, known as **extended BNF** (EBNF), allows us to use regular- expression-like constructs on the right-hand-side of the rules: for any sequence of symbols α, we write "[α]" to say that α is optional, and "$\{\alpha\}$" to denote 0 or more repetitions of α (ie. the Kleene closure of α).

Thus we could re-write the above rules in EBNF as:

$$
\begin{array}{lll}
\langle \text{ block } \rangle & ::= & \text{“BEGIN” } [\langle \text{ stats-list } \rangle] \text{ “END”}. \\
\langle \text{ stats-list } \rangle & ::= & \{\langle \text{ statement } \rangle \text{ ”;”}\} \ \langle \text{ statement } \rangle.
\end{array}
$$

Using EBNF has the advantage of simplifying the grammar by reducing the need to use recursive rules.

We could describe the complete syntax of a programming language using EBNF (or equivalently a CFG), but it tends to make things easier if we break it up into two sections:

1. The lexical syntax, which defines which patterns of characters make up which words. This can be described using regular expressions

2. The context-free syntax, which defines which sequences of words which make up phrases from the programming language; we describe this using a context-free grammar.

Thus the first task in writing a parser for a language is breaking up its syntax into the "regular" and "context-free" part. The choice is arbitrary but a simple guideline is that:

characters from the input should not appear in the context-free syntax;
these should be tokenised by the lexical phase.

As well as modularising our task, this approach will facilitate debugging in parser generators such as `yacc`.

2.3 Derivations and Parse Trees

Parsing is the process where we take a particular sequence of words and check to see if they correspond to the language defined by some grammar.

Basically what we have to do is to show how we can get:

from	the start symbol of the grammar
to	the sequence of words that supposedly belong to the language
by	using the production rules

We can do this by constructing the appropriate parse tree for the sequence of words, which is simply a graphical way of listing the production rules that we have used.
Considering the following simple grammar for arithmetic expressions:

$$
\begin{array}{rcl}
E & \rightarrow & E\ OP\ E \ \mid\ (E)\ \mid\ -E\ \mid\ \mathbf{num} \\
OP & \rightarrow & + \mid - \mid * \mid \div
\end{array}
$$

Here there are just two non-terminals, E and OP (where E is the start symbol); the terminal symbols are: +, -, *, ÷, (,), **num**.

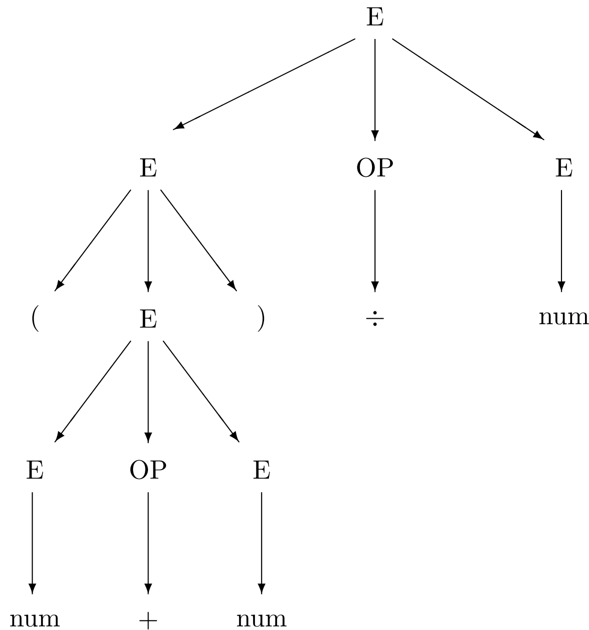
Suppose the lexical analysis phase has given us the sequence of tokens which correspond to: **(num + num)** ÷ **num**. We can parse this as follows:

Rule Applied	*Sent. Form*
	E
$E \rightarrow E\ OP\ E$	$E\ OP\ E$
$E \rightarrow (E)$	$(E)\ OP\ E$
$E \rightarrow E\ OP\ E$	$(E\ OP\ E)\ OP\ E$
$E \rightarrow \mathbf{num}$	$(\mathbf{num}\ OP\ E)\ OP\ E$
$OP \rightarrow +$	$(\mathbf{num}+ E)\ OP\ E$
$E \rightarrow \mathbf{num}$	$(\mathbf{num} + \mathbf{num})\ OP\ E$
$OP \rightarrow \div$	$(\mathbf{num} + \mathbf{num})\ \div\ E$
$E \rightarrow \mathbf{num}$	$(\mathbf{num} + \mathbf{num})\ \div\ \mathbf{num}$

We say that E derives **(num + num)** ÷ **num**, and write:

$$E \rightarrow^{*} (\mathbf{num} + \mathbf{num}) \div \mathbf{num}$$

We can represent the above derivation graphically by means of a **parse tree**. The root of the tree is the start symbol E, and its leaves are the terminal symbols in the sentence which has been derived. Each internal node is labelled with a non-terminal, and the children are the symbols obtained by applying one of the production rules.



There are two ways of constructing a parse tree whose root is the start symbol and whose leaves are the sentence:

Top-Down As we did earlier, we start with the start symbol and apply the production rules (replacing l.h.s. with r.h.s.) until we derive the sentence

Bottom-Up We start with the sentence, and apply the production rules in reverse (ie. replacing r.h.s. of the rule with the l.h.s.) until we derive the start symbol.

Both of these methods will be discussed in detail later.

2.3.1 Leftmost and Rightmost Derivations

At any stage during a parse, when we have derived some sentential form (that is not yet a sentence) we will potentially have two choices to make:

1. which non-terminal in the sentential form to apply a production rule to
2. which production rule for that non-terminal to apply

Eg. in the above example, when we derived $E\ OP\ E$, we could then have applied a production rule to any of these three non-terminals, and would then have had to choose among all the production rules for either E or OP.

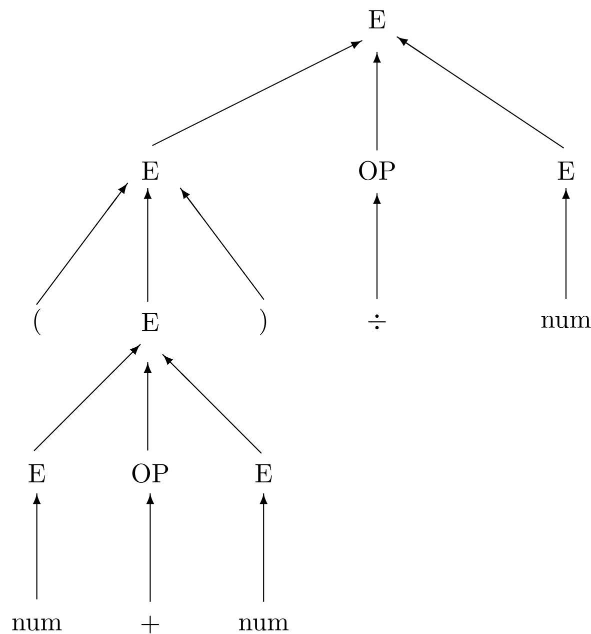
The first decision here is relatively easy to solve: we will be reading the input string from left to right, so it is our own interest to derive the leftmost terminal of the resulting sentence as soon as possible. Thus, in a top-down parse we always choose the leftmost non-terminal in a sentential form to apply a production rule to - this is called a **leftmost derivation**.

If we were doing a bottom-up parse then the situation would be reversed, and we would want to do apply the production rules in reverse to the leftmost symbols; thus we are performing a **rightmost derivation** in reverse.

For example, a bottom-up rightmost derivation would look like:

Rule Applied (in reverse)	*Sent. Form*
	(**num** + **num**) ÷ **num**
$E \rightarrow$ **num**	(E +**num**) ÷ **num**
$OP \rightarrow +$	(E OP **num**) ÷ **num**
$E \rightarrow$ **num**	(E OP E) ÷ **num**
$E \rightarrow E\ OP\ E$	(E) ÷ **num**
$E \rightarrow (E)$	E ÷ **num**
$OP \rightarrow \div$	E OP **num**
$E \rightarrow$ **num**	$E\ OP\ E$
$E \rightarrow E\ OP\ E$	E

Note that this has no effect on the parse tree; we still get:



2.4 Modifying Grammars

Once we have decided between a leftmost or rightmost derivation strategy, we will then need some method of deciding which production rule to choose for a given non-terminal. There are some modifications that we can make to any grammar in order to make this task a little easier.

2.4.1 Left Factoring

In general, if A is a single non-terminal, and α, β and γ are strings of terminals or non- terminals, then a production rule of the form: $A \rightarrow \alpha\beta \mid \alpha\gamma$ can be left-factored into two rules of the form:

$$\begin{aligned} A &\rightarrow \alpha\, B \\ B &\rightarrow \beta \mid \gamma \end{aligned}$$

where B is some new non-terminal.

2.4.2 Eliminating Ambiguity

A grammar is said to be **ambiguous** if it can produce more than one parse tree for a particular sentence; this occurs when two different sequences of leftmost (or rightmost) derivations can produce the same sentence from the same start symbol.

Consider the following example:

$$
\begin{array}{rcll}
STAT & \rightarrow & \textbf{if } EXPR \textbf{ then } STAT\ ESTAT & Rule\ 1 \\
 & | & \textbf{st} & Rule\ 2 \\
ESTAT & \rightarrow & \textbf{else } STAT & Rule\ 3 \\
 & | & \varepsilon & Rule;\ 4 \\
EXPR & \rightarrow & \textbf{num} & Rule\ 5
\end{array}
$$

Suppose we wanted to parse the sentence: **if num then if num then st else st**

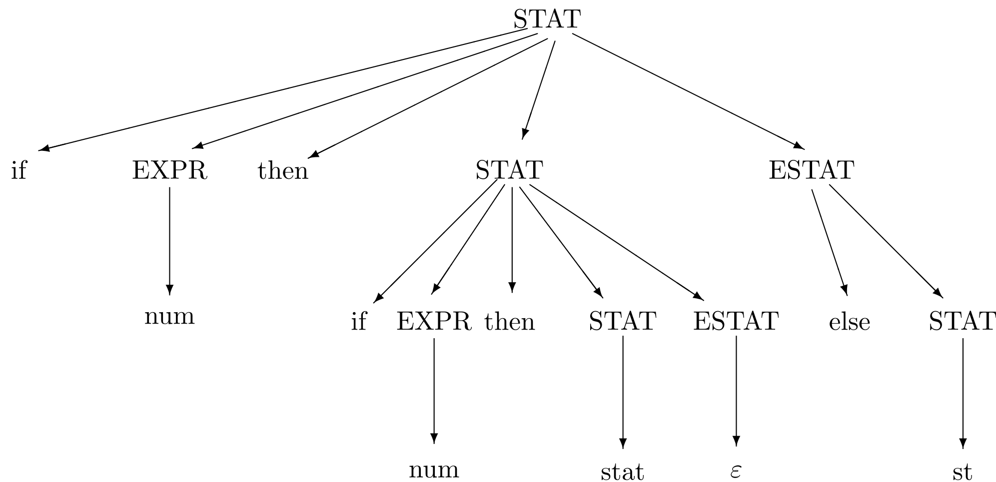
We could try to derive this (in the manner of a leftmost, top-down parse) as follows:

Rules	Sentential Form
	STAT
Rule 1	**if** *EXPR* **then** *STAT ESTAT*
Rule 5	**if num then** *STAT ESTAT*
Rule 1	**if num then if** *EXPR STAT ESTAT ESTAT*
Rule 5	**if num then if num then** *STAT ESTAT ESTAT*
Rule 2	**if num then if num then st** *ESTAT ESTAT*

So far, so good. But in order to finish the derivation, we could choose either of two routes:

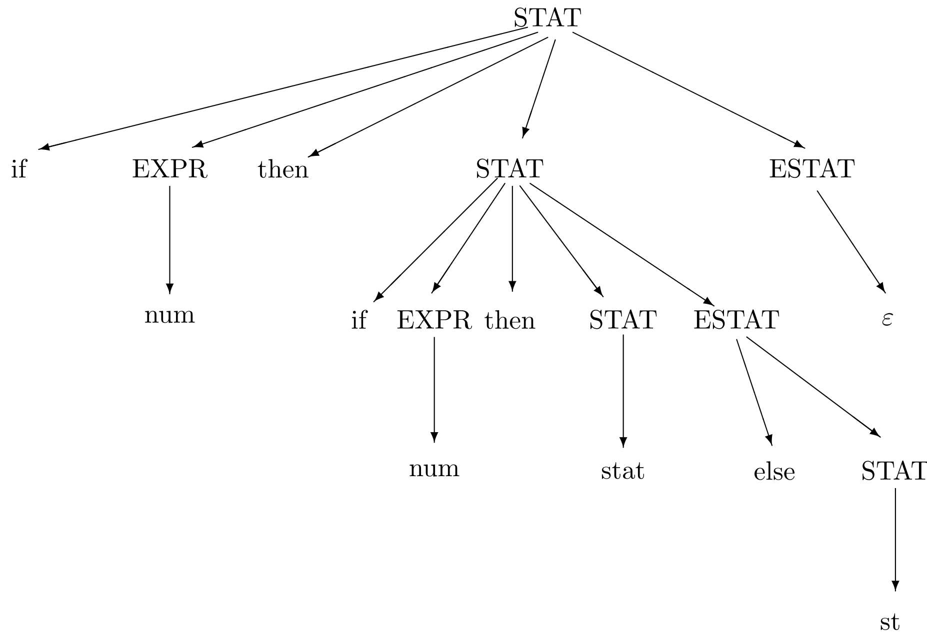
1. We apply rule 4, 3 and 2 to get:

Rules	Sentential Form
Rule 4	**if num then if num then st** *ESTAT*
Rule 3	**if num then if num then st else** *STAT*
Rule 2	**if num then if num then st else st**



2. Alternatively, we could apply rule 3 first, and then 2 and 4 to get:

Rules	Sentential Form
Rule 3	**if num then if num then st else** *STAT ESTAT*
Rule 2	**if num then if num then st else st** *ESTAT*
Rule 4	**if num then if num then st else st**



In both cases the correct sentence is derived, but the difference in the structure of the parse trees reflects the fact that the grammar is ambiguous. In this case, the ambiguity arises because we have not specified whether the `else` is to be associated with the first `if` (as in the first derivation) or with the second `if` (as in the second); ie. which of the following it is semantically equivalent to:

1. `if num then` { `if num then st` } `else st`
2. `if num then` { `if num then st else st` }

The latter is preferable, since in most programming languages we will want to be able to match each `else` with the closest previous unmatched `if`.

In order to incorporate this convention into our grammar, we must change the rules so that we can tell the difference between matched statements (an equal number of if and else terminals) and unmatched statements. This gives us:

$$
\begin{array}{rcl}
STAT & \rightarrow & MSTAT \\
 & | & USTAT \\
MSTAT & \rightarrow & \textbf{if}\ EXPR\ \textbf{then}\ MSTAT\ \textbf{else}\ MSTAT \\
 & | & \textbf{st} \\
USTAT & \rightarrow & \textbf{if}\ EXPR\ \textbf{then}\ UELSE \\
UELSE & \rightarrow & STAT \\
 & | & MSTAT\ \textbf{else}\ USTAT \\
EXPR & \rightarrow & \textbf{num}
\end{array}
$$

Unlike left-factoring, we cannot provide any general procedure which will eliminate ambiguity; instead, it will depend on our knowledge of the semantics of the language.

In fact, it is not possible to write an algorithm which will take an arbitrary grammar and decide whether or not it is ambiguous. Even if we do identify ambiguity, we may not always be able to remove it, since some languages cannot be described by a grammar which is unambiguous.

2.4.3 Eliminating Immediate Left Recursion

A grammar is left recursive if we can find some non-terminal A which will eventually derive a sentential form with itself as the left-most symbol.

We have immediate left recursion of there is a rule of the form:

$$A \quad \rightarrow \quad A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

where each α_i and β_j is any mixture of terminals and non-terminals, with the condition that each β_j does not begin with A.

The essential problem with such productions is that a parser could keep applying some rule $A \rightarrow A\alpha_i$ which will not ultimately generate some terminal symbol that we can compare with the input string.
We fix this problem by introducing some new non-terminal A', and replacing the above rule with two new rules:

$$\begin{array}{rcl} A & \rightarrow & \beta_1 A' \mid \ldots \mid \beta_m A' \\ A' & \rightarrow & \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon \end{array}$$

The first rule cannot be left-recursive since no β_j begins with A, and the second rule will not be left-recursive as long as no α_i is ε.

This process will always work if the left recursion is immediate; however, it will not work if the recursion involves more than one step.

2.4.4 Eliminating Indirect Left Recursion

Indirect left recursion occurs as follows:

Suppose we have non-terminals $A_0, A_1, \ldots, A_n$ and that $\alpha_1, \alpha_1, \ldots, \alpha_{n+1}$ are mixtures of terminals and non-terminals, then the production rules:

$$\begin{array}{rcl} A_0 & \rightarrow & A_1\alpha_1 \mid \ldots \\ A_1 & \rightarrow & A_2\alpha_2 \mid \ldots \\ & \ldots & \\ A_n & \rightarrow & A_0\alpha_{n+1} \mid \ldots \end{array}$$

could result in a leftmost derivation of the form:

$$A_0 \quad \rightarrow \quad A_1\alpha_1 \quad \rightarrow \quad A_2\alpha_2\alpha_1 \quad \rightarrow \quad \ldots \quad \rightarrow \quad A_0\alpha_{n+1}\ldots\alpha_2\alpha_1$$

Again, such a derivation would cause problems during a top-down parse.

If we have a grammar that does not have a derivation of the form $B \rightarrow^* B$, for any non-terminal B (ie. the grammar has no cycles), and there are no ε-productions, then we can apply the following algorithm to eliminate all left recursion.

Let $\alpha, \beta_1, \ldots, \beta_n$ represent any mixture of terminals and non-terminals. Then:

1. Arrange all the non-terminals into some arbitrary order: call them $A_1, A_2, \ldots, A_n$
2. For each non-terminal A_i in turn, do:
   (a) For each terminal A_i such that $1 \leq j < i$ and we have a production rule of the form $A_i \rightarrow A_j\alpha$, where the A_j productions are $A_j \rightarrow \beta_1 \mid \ldots \mid \beta_n$, do:
   Replace the production rule $A_i \rightarrow A_j\alpha$ with the rule $A_i \rightarrow \beta_1\alpha \mid \ldots \mid \beta_n\alpha$
   (b) Eliminate any immediate left recursion among the A_i productions

This algorithm is preventative in nature - it does not look for left recursion and then eliminate it; instead, it rearranges the grammar so that no left recursion can possibly occur. Thus, even non-left-recursive grammars may be rewritten using this algorithm.

The effect of step 2(a) above is to eliminate all production rules of the form $A_i \rightarrow A_j\alpha$ for every $j < i$; in other words, if we have a chain of derivations of the form:

$$A_0 \quad \rightarrow \quad A_1\alpha_1 \quad \rightarrow \quad A_2\alpha_2\alpha_1 \quad \rightarrow \quad \ldots \quad \rightarrow \quad A_i\alpha_i\ldots\alpha_2\alpha_1$$

We can be sure that the index i of A_i is increasing all the time, and so can never get back around to 0 at any stage in the derivation (which would have caused recursion).

Note: the replacement technique used in step 2(a) is a standard procedure: if we ever have a production rule involving a non-terminal on the r.h.s. that we "don't like" for some reason,then we get rid of it by replacing it in the rule with the r.h.s. of its production rule(s).

2.4.5 Eliminating ε-productions

An ε-production is a rule of the form $A \rightarrow \varepsilon$ for some non-terminal A (this is also called an **erasing rule** for A). Usually the main motivation for removing this type of rule is that we need to apply some algorithm which only works for grammars with no ε-productions (such as the elimination of indirect left-recursion).

A non-terminal is said to be **nullable** if it can derive the empty string in one or more steps; that is, if B is nullable, then there exists some derivation $B \rightarrow^* \varepsilon$. We need this concept, since the elimination of an erasing rule for a non-terminal will have a knock-on effect on every other non-terminal that can derive it.

In order to eliminate ε-productions from a grammar we perform the following:

For every nullable non-terminal B:

- For every production rule of the form: $C \rightarrow \alpha B \beta$ add in a new rule of the form $C \rightarrow \alpha\beta$, where C is any non-terminal, and α and β are any sequences of terminals or non-terminals.

ie. since we've removed the facility which allows B to go to ε, we must compensate for this everywhere B is used on the r.h.s of a production rule..

2.4.6 Eliminating Unit Productions

A **unit production** is one of the form $A \rightarrow B$, where both A and B are single non-terminals. What this basically says is that A is the same as B, since any where we see one we can replace it with the other. Thus we can shorten or grammar without changing the accepted language by eliminating this type of production rule.

For any non-terminal A, we say that a non-terminal B is A**-derivable** if there exists some derivation $A \rightarrow^* B$. Obviously this type of derivation will be effected by the elimination of unit productions.

To eliminate unit productions from a grammar, we perform the following:

1. Eliminate all ε-productions from the grammar.
   (This makes the task of finding unit derivations much easier).
2. For each non-terminal A, and For every other non-terminal B which is A-derivable (ie $A \rightarrow^* B$)
   - For every B production rule of the form $B \rightarrow \alpha$, add a new rule to the grammar of the form $A \rightarrow \alpha$
3. Delete all the unit productions.

2.5 PUSH-DOWN AUTOMATA

Just as finite-state automata corresponded to regular languages, the context-free languages (CFLs) have corresponding machines called push-down automata (PDA). In this section we formally define PDAs and examine the connection with CFLs.

We have previously seen how to construct a deterministic FSA for a regular language; it is an important result of this section that:

> while every CFG has a corresponding PDA,
> not every CFG has a corresponding *deterministic* PDA

2.5.1 Formal Definition

Basically, a PDA is an NFA with a stack; formally we define a PDA as consisting of:

- An alphabet Σ of *input symbols*
- A set of *states*, of which we distinguish:
  - a unique start state
  - a set of final states
- An alphabet Γ of *stack symbols*

- A transition relation which, for any given state, input symbol and stack symbol, gives a new state and stack symbol; i.e. it has the form:

$$(State,\ InputSymbol,\ StackSymbol) \ \mapsto \ (State,\ StackSymbol)$$

Basically, if $a \in \Sigma$, $t, u \in \Gamma$ and P and Q are states, a transition of the form

$$(P, a, t) \mapsto (Q, u)$$

means "read the symbol 'a' from the input, move from state P to state Q, and replace the symbol t on top of the stack with the symbol u".

Obviously for this to happen we must be in state P, we must have 'a' as the next symbol on the input tape, and t must be on top of the stack.

The actions *push* and *pop* are instances of the above transitions; for any stack symbol t, we have

- push t is $(P, \varepsilon, \varepsilon) \mapsto (Q, t)$
- pop t is $(P, \varepsilon, t) \mapsto (Q, \varepsilon)$

A PDA is said to *accept* some string x if and only if we can get:

- **from** the start state with an empty stack and x as input
- **to** a finish state with an empty stack and an empty input.

2.5.2 Converting a CFG to a PDA

Every CFG can be converted to an equivalent PDA. The constructed PDA will perform a leftmost derivation of any string accepted by the CFG.

Suppose we are given a CFG and let T and N be its sets of terminal and non-terminal symbols respectively. Let S be the start symbol. Then we construct the PDA as follows:

- The input alphabet is T
- There are only two states, let us call them P and Q, where
  - P is the start state
  - Q is the only final state
- The stack alphabet is $T \cup N$
- The transitions are as follows:
  - $(P, \varepsilon, \varepsilon) \mapsto (Q, S)$
  - For each grammar rule of the form $A \rightarrow \alpha$, there is a transition $(Q, \varepsilon, A) \mapsto (Q, \alpha)$
  - For each terminal symbol 'a', there is a transition $(Q, a, a) \mapsto (Q, \varepsilon)$

2.5.3 Converting a PDA to a CFG

We will proceed in a manner analogous to Kleene's theorem for regular languages: that is, we will try to slice up the machine into various components (each of which has a corresponding language), and then put them back together again using a CFG.

For any PDA, let us define the language

$$\langle P, Q, t \rangle = \left\{ \ x \in \Sigma^* \ \middle| \ \begin{array}{l} x \text{ is consumed in moving from state } P \text{ to state } Q \text{ in the} \\ \text{machine, with the symbol } t \text{ being taken from the top of the} \\ \text{stack in the process} \end{array} \right\}$$

Now, given any PDA, we construct a context-free grammar which accepts the same language as follows:

- The terminal symbols are just the input symbols of the PDA
- The non-terminal symbols are all triples of the form $\langle P, Q, t \rangle$, for each state P and Q, and each stack symbol t

- If S and F are the start and finish states respectively of the PDA, then the start symbol of the CFG is $\langle S, F, \varepsilon \rangle$.
- The production rules are as follows:
  - For each transition of the form $(P, a, t) \mapsto (R, \beta_1 \cdots \beta_j)$, add all rules of the form

    $$\langle P, Q_j, t \rangle \to a.\langle R, Q_1, \beta_1 \rangle.\langle Q_1, Q_2, \beta_2 \rangle \ldots \langle Q_{j-1}, Q_j, \beta_j \rangle$$

    where the Q_i can be any state in the machine
  - For each transition of the form $(P, a, t) \mapsto (R, \varepsilon)$, add all rules of the form

    $$\langle P, R, t \rangle \to a$$

2.5.4 Deterministic PDAs

In general terms, a *deterministic* PDA is one in which there is at most one possible transition from any state based on the current input.

Formally, a deterministic PDA is a PDA where:

- for every state P, input symbol 'a' and stack symbol t, there is at most one transition of the form $(P, a, t) \mapsto (Q, u)$ for any state Q and stack symbol u.

Any context-free language that can be converted to a deterministic PDA is called a *deterministic CFL.*

2.5.5 Non-Deterministic CFLs

The important point then is that:

not every context-free language is determinstic

To see this, consider the language $\{x^n y^n \mid n \geq 0\} \cup \{x^n y^{2n} \mid n \geq 0\}$ We can easily show that this context-free by giving its CFG. However, we will show that it is not *deterministic* context-free.

Proof: *(by contradiction)*

Suppose that this language *is* deterministic context-free; then it has a corresponding deterministic PDA.

Let us create two copies of this PDA called M_1 and M_2. Call any two states "cousins" if they are copies of the same state in the original PDA. Now we construct a new PDA as follows:

- The states of the new PDA is the union of the states in M_1 and M_2, where
  - the start state of M_1 is the new start state
  - the final states of M_2 are the new final states
- The transition relation is that of M_1 and M_2 with the following alterations:
  - Change any transition originating from a final state in M_1 so that it now goes to its "cousin" state in M_2
  - Change all those 'y' transitions which cause a move into some state from M_2 into 'z' transitions

This is a PDA over the alphabet $\{x, y, z\}$. To see what language it recognises, consider its actions on an input of $x^k y^k z^k$ for some fixed $k \geq 0$.

Initially it will move from the start state to a final state of M_1 while consuming the input $x^k y^k$. Because it is deterministic, there is no other state which it could reach while consuming this input. But we know that by its construction M_1 can now also go on to accept k more copies of 'y'; therefore if we run the new PDA on the rest of the input, it will consume k more copies of 'z' as it moves through M_2.

Thus the constructed PDA accepts the language $\{x^n y^n z^n \mid n \geq 0\}$ - but this is impossible, as this language is not context free. Thus our assumption that the PDA for the original language could be deterministic is contradicted.

$\mathcal{QED}$

2.6 Proving that a Language is not Context-Free

Since context-free languages are so important, it is desirable to have some way of proving when a language *isn't* context-free. We did this for regular languages by choosing those which were "big enough" (ie. infinite) to cause a particular type of repetition (or *periodicity*) in their derivation. The nature of the "pump" (for regular expressions it was Kleene Closure) could then be used to characterise these languages.

So we consider some questions about context-free grammars:

- In what way does a CFG "pump"?
  Repetition in a CFG is achieved by recursion in the rules. This can occur directly or indirectly, but will result in a derivation of the form: $A \rightarrow^* vAy$, where A is a non-terminal, and v and y are strings.

- What is the effect of this "pumping"?
  Each time we apply the sequence of rules which recurse back to A, we also generate an extra copy of both v and y. Thus if we go through the sequence say n times, we get the derivation $A \rightarrow^* v^n A y^n$

  This "matching" or "balancing" of two patterns is the hallmark of context-free languages.

- When does this "pumping" happen?
  This can be broken down into two sub-questions:

  - For what sort of parse tree does this "pumping" happen?
    If a derivation "pumps" on some non-terminal A, then this non-terminal must occur twice in some path from the root of the parse tree to one of the leaves. This is guaranteed to happen if we know that the grammar has, say, m non-terminals altogether, and the parse tree for a string has a path with length greater than m (we simply run out of non-terminals to label the nodes with, and so must repeat one!). So a parse tree contains a "pump" if its height is greater than m.
  - For what sort of strings does this "pumping" happen?
    Suppose the maximum number of symbols on the r.h.s. of any grammar rule of the language is p. Then each application of any rule generates at most p symbols; (ie. each node of the parse tree has at most p children). Thus a path of length m or less can generate at most p^m symbols. This means that if we have a string of length greater than p^m, there must have been a "pump" in the parse tree which generated it.

  Thus the ability to generate strings of length greater than p^m indicates that the language represented by the grammar is infinite.

Let us suppose that u, x and z are strings such that:

$$S \rightarrow^* uAz \quad \textit{and} \quad A \rightarrow^* x$$

and if, as assumed above, we have also that:

$$A \rightarrow^* vAy$$

The it should be clear that for any $i \geq 0$:

$$S \rightarrow^* uAz \rightarrow^* uvAyz \rightarrow^* \ldots \rightarrow^* uv^iAy^iz \rightarrow^* uv^ixy^iz$$

Then we can generate the set $\{uxz, uvxyz, uv^2xy^2z, \ldots, uv^ixy^iz, \ldots\}$

Tying all this together we get:

2.6.1 The Pumping Lemma for context-free languages

For any context-free grammar G, there is a number K, depending on G, such that any string generated by G which has length greater than K can be written in the form $uvxyz$, so that either v or y is non-empty, and uv^nxy^nz is in the language generated by G for all $n \geq 0$.
Proof:
Based on the above discussion, we can see that the number K is simply p^m.

QED

2.6.2 The Pumping Lemma: Examples

Lemma: The language $L = \{a^n b^n c^n \mid n \geq 1\}$ is not context free.
Proof *(By contradiction)*
Suppose this language is context-free; then it has a context-free grammar.
Let K be the constant associated with this grammar by the Pumping Lemma.
Consider the string $a^K b^K c^K$, which is in L and has length greater than K.
By the Pumping Lemma this must be representable as $uvxyz$, such that all uv^ixy^iz are also in L.
This is impossible, since:

- either v or y cannot contain a mixture of letters from $\{a, b, c\}$; otherwise they would be in the wrong order for uv^2xy^2z
- if v or y conatin just 'a's, 'b's or 'c's, then uv^2xy^2z cannot maintain the balance between the *three* letters (it can, of course maintain the balance between *two*)

QED

Lemma: The language $L = \{a^i b^j c^k \mid i < j \text{ and } i < k\}$ is not context free.
Proof *(By contradiction)*
Suppose this language is context-free; then it has a context-free grammar.
Let K be the constant associated with this grammar by the Pumping Lemma.
Consider the string $a^K b^{K+1} c^{K+1}$, which is in L and has length greater than K.
By the Pumping Lemma this must be representable as $uvxyz$, such that all uv^ixy^iz are also in L.

- By the same argument as for the previous lemma neither v nor y may contain a mixture of symbols.
- Suppose v consists entirely of 'a's.
  Then there is no way y, which cannot have both 'b's and 'c's, can generate enough of these letters to keep their number greater than that of the 'a's (it can do it for one or the other of them, not both).
  Similarly y cannot consist of just 'a's.
- So suppose then that v or y contains only 'b's or only 'c's.
  Consider the string uv^0xy^0z which must be in L. Since we have dropped both v and y, we must have at least one 'b' or one 'c' less than we had in $uvxyz$, which was $a^K b^{K+1} c^{K+1}$. Consequently, this string no longer has enough of either 'b's or 'c's to be a member of L.

QED

2.6.3 Limitations of the Pumping Lemma

The Pumping Lemma is very unspecific as to *where* in the generated string the pumping is to occur. This can make some proofs quite difficult.

For example, to prove that $\{a^m b^n \mid m > n \text{ or } m \text{ is a prime number}\}$ is impossible by *direct* application of the Pumping Lemma.

In such cases, specialised versions, such as *Parikh's Theorem*, are required.

2.7 The Closure of Context-Free Languages

We have seen that the regular languages are closed under common set-theoretic operations; the same, however, does not hold true for context-free languages.

Lemma: The context-free languages are closed under union, concatenation and Kleene closure.
That is, if L_1 and L_2 are context-free languages, so are $L_1 \cup L_2$, L_1L_2 and L_1^*.
Proof:
We will prove that the languages are closed by creating the appropriate grammars.
Suppose we have two context-free languages, represented by grammars with start symbols S_1 and S_2 respectively. First of all, rename all the terminal symbols in the second grammar so that they don't conflict with those in the first. Then:

- To get the union, add the rule $S \rightarrow S_1 \mid S_2$
- To get the concatenation, add the rule $S \rightarrow S_1\ S_2$
- To get the Kleene Closure of L_1, add the rule $S \rightarrow S_1\ S \mid \varepsilon$ to the grammar for L_1.

QED

Lemma: The context-free languages are <u>not</u> closed under intersection
That is, if L_1 and L_2 are context-free languages, it it not always true that $L_1 \cap L_2$ is also.
Proof:
We will prove the non-closure of intersection by exhibiting a counter-example.
Consider the following two languages:

$$\begin{array}{lcll} L_1 & = & \{a^i b^j c^k & \mid \ i < j\} \\ L_1 & = & \{a^i b^j c^k & \mid \ i < k\} \end{array}$$

We can prove that these *are* context-free by giving their grammars:

$$\begin{array}{lccl} L_1 = & S & \rightarrow & ABC \\ & A & \rightarrow & \mathbf{a}A\mathbf{b} \mid \varepsilon \\ & B & \rightarrow & \mathbf{b}B \mid \mathbf{b} \\ & C & \rightarrow & \mathbf{c}C \mid \varepsilon \end{array}$$

$$\begin{array}{lccl} L_2 = & S & \rightarrow & ASC \mid B\mathbf{c} \\ & A & \rightarrow & \mathbf{a} \\ & B & \rightarrow & \mathbf{b}B \mid \varepsilon \\ & C & \rightarrow & \mathbf{c}C \mid \mathbf{c} \end{array}$$

The intersection of these languages is:

$$L_1 \cap L_2 \quad = \quad \{a^i b^j c^k \quad \mid \quad i < j \ and \ i < k\}$$

We proved in the last section that this language is not context-free.

QED

Lemma: The context-free languages are <u>not</u> closed under complementation.
That is, if L is a context-free language, it it not always true that L' is also.
Proof: *(By contradiction)*
Suppose that context-free languages *are* closed under complementation.
Then if L_1 and L_2 are context-free languages, so are L_1' and L_2'. Since we have proved closure under union, $(L_1' \cup L_2')$ must also be context-free, and, by our assumption, so must its complement $(L_1' \cup L_2')'$.
However, by de Morgan's laws (for sets), $(L_1' \cup L_2')' \equiv (L_1 \cap L_2)$, so this must also be a context-free language.
Since our choice of L_1 and L_2 was arbitarary, we have contradicted the non-closure of intersection, and have thus proved the lemma.

QED

2.8 Decision Problems for Context-Free Languages

Based on the Pumping Lemma, we can state some decidability results for context-free languages.

Lemma: It is decidable whether or not a given string belongs to a context-free language.
Proof:
If we eliminate all ε-productions, and all unit productions from the context-free grammar for the language, then each time we apply a production rule we either:

- make the sentential form grow longer
- increase the number of terminal symbols in the sentential form

Therefore the number of production rules applied (and thus the height of the parse tree) will be bounded by the length of the input string. Since this is finite, an effective (though inefficient) decision algorithm need only enumerate all such parse trees, and check the leaves to see if they correspond to the string.

QED

Without examining the issue in detail, we also note the following:

- The constraints placed on the rule-format of **context-sensitive** grammars allow us to use the same argument as above to assert that language membership is decidable.
- In order to decide if a string is a member of a **free** language, we run the corresponding Turing Machine, and see if it halts on "yes". Clearly this is the **Halting Problem**, which is undecidable. Thus it is not possible in general to construct a parser for a free language.

Lemma: It is decidable whether or not a context-free language is empty.
Proof:
We need to show that the corresponding context-free grammar can generate a string.
If a CFG can generate a string, then it should be able to do so without using recursion (since we could just skip the recursion and generate a shorter string). If there are m non-terminals altogether, then some tree of height less than m must have leaves which are only terminals.
Thus we need only examine all such trees to see if the CFG generates a sentence.

$\mathcal{QED}$

Some decision problems for CFGs (and accordingly for push-down automata) are not solvable.

We state, without proof, that the following problems are undecidable in general:

- Given two context-free languages L_1 and L_2, is $L_1 \cap L_2 \equiv \emptyset$?
  ie. do two context-free grammars generate the same string?
- Given some alphabet of terminal symbols Σ, does a grammar over this alphabet generate *all* the strings of Σ^*?
- Is the context-free grammar for a language ambiguous?
- Given two context-free languages L_1 and L_2, is $L_1 \subset L_2$?
- Given two context-free languages L_1 and L_2, is $L_1 = L_2$?

The proof of these theorems follows from a famous decision problem called **Post's correspondance problem**.

3 TOP-DOWN PARSING

In this section we examine some algorithms for the implementation of a top-down parser, including recursive descent, the $LL(k)$ family, Earley's parser and Unger's method.

3.1 Recursive Descent Parsing

Recall that we have a grammar (set of production rules) and a sentence; our job is to show how the production rules can be applied to the start symbol in order to derive the sentence.

The simplest way to construct a top-down parser is to regard:

- each production rule as defining a function, where:
  - the name of the function is the non-terminal on the l.h.s. of the rule
  - each instance of a non-terminal on the r.h.s. is a call to the corresponding function,
  - each instance of a terminal on the r.h.s. tells us to match it with the input symbol

This is fairly easy to do if each r.h.s. starts with a different terminal symbol - this means that we needed only to read one character from the input in order to know which rule to pick.

Suppose we were given the following production rules:

$$X \rightarrow AB \mid \mathbf{a}A\mathbf{b} \mid C\mathbf{c} \mid \varepsilon$$

Since we can't make easy predictions, the simplest approach here is to try the first rule; if that doesn't work, try the second, and so on. Really, this type of algorithm is more suitable for implementation in a goal-directed language such as Prolog (where we can backtrack after a failure), rhather than a conventional one like C. Note that if we used a Prolog version with an ambiguous grammar, then the program should respond with an answer for each possible derivation (since Prolog will try to find all ways of satisfying its goal).

3.2 Lookahead

The basic problem in parsing is choosing which production rule to use at any stage during a derivation. Any approach to parsing which involves backtracking will probably involve creating complex or inefficient algorithms in a conventional programming language. To make implementation easier, we introduce the notion of lookahead.

Lookahead means attempting to analyse the possible production rules which can be applied, in order to pick the one most likely to derive the current symbol(s) on the input.

Suppose we are in the middle of a derivation, where we have:

- Current Sentential Form = $\alpha X \beta$
- Input string = $\alpha \mathbf{a} \gamma$

where:

- α consists entirely of terminals, and corresponds to the piece of the input that has already been matched
- X is the leftmost non-terminal, β is the rest of the sentential form
- **a** is the next symbol on the input, γ is the rest of the input

Our job then is to figure out which production rule will allow $X\beta$ can derive $\mathbf{a}\gamma$; thus we examine the r.h.s. of the production rules for X as follows:

1. If the r.h.s. starts with a terminal symbol, then it must be of the form $X \rightarrow \mathbf{a}\delta$ for this rule to be of any use
2. If the r.h.s. starts with a non-terminal, ie. it looks like $X \rightarrow Y\delta$ say, then
   (a) we must examine the production rules for Y to see if any of these can derive an **a**.
   (b) If Y can derive ε, then we must see what δ can derive; ie. apply steps 1 and 2 to the production rules for δ.
3. If we find that X can derive ε, then we must check to see what the symbols following X in the sentential form (ie. the first symbol in β) can derive.

3.2.1 FIRST and FOLLOW

We formalise the task of picking a production rule using two functions, FIRST and FOLLOW.

FIRST is applied to the r.h.s. of a production rule, and tells us all the terminal symbols that can start sentences derived from that r.h.s. It is defined as:

1. For any terminal symbol **a**, $FIRST(\mathbf{a}) = \{\mathbf{a}\}$.
   Also, $FIRST(\varepsilon) = \{\varepsilon\}$.
2. For any non-terminal A with production rules $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$, $FIRST(A) = FIRST(\alpha_1) \cup FIRST(\alpha_2) \cup \ldots \cup FIRST(\alpha_n)$
3. For any r.h.s. of the form: $\beta_1\beta_2\ldots\beta_n$ (where each β_i is a terminal or a non-terminal) we have:
   (a) $FIRST(\beta_1)$ is in $FIRST(\beta_1\beta_2\ldots\beta_n)$
   if β_1 can derive ε, then $FIRST(\beta_2)$ is also in $FIRST(\beta_1\beta_2\ldots\beta_n)$
   if both β_1 and β_2 can derive ε, then $FIRST(\beta_3)$ is also in $FIRST(\beta_1\beta_2\ldots\beta_n)$
   ...
   if $\beta_1\beta_2\ldots\beta_i$ can derive ε, then $FIRST(\beta_{i+1})$ is also in $FIRST(\beta_1\beta_2\ldots\beta_n)$

   (b) ε is in $FIRST(\beta_1\beta_2\ldots\beta_n)$ only if ε is in $FIRST(\beta_i)$, for all $0 \leq i \leq n$

FIRST can be applied to any r.h.s., and returns a set of terminal symbols.

Thus, if X is the current non-terminal (ie. leftmost in the current sentential form), and **a** is the next symbol on the input, then we want to look at the r.h.s. of the production rules for X and choose the one whose FIRST set contains **a**.

FOLLOW is used only if the current non-terminal can derive ε; then we're interested in what could have followed it in a sentential form. (NB: A string can derive ε if and only if ε is in its FIRST set.)

1. If S is the start symbol, then put $\perp$ into $FOLLOW(S)$
2. Examine all rules of the form $A \rightarrow \alpha X \beta$; then
   (a) $FIRST(\beta)$ is in $FOLLOW(X)$
   (b) If β can derive the empty string then put $FOLLOW(A)$ into $FOLLOW(X)$

The "$\perp$" is a symbol which is used to mark the end of the input tape.

FOLLOW can be applied to a single non-terminal only, and returns a set of terminals.

Thus, if X is the current non-terminal, **a** is the next symbol on the input, and we have a production rule for X which allows it to derive ε, then we apply this rule only if **a** is in the FOLLOW set for$X$.

FIRST and FOLLOW help us to pick a rule when we have a choice between two or more r.h.s. by predicting the first symbol that each r.h.s. can derive. Even if there is only one r.h.s. we can still use them to tell us whether or not we have an error - if the current input symbol cannot be derived from the only r.h.s. available, then we know immediately that the sentence does not belong to the grammar, without having to (attempt to) finish the parse.

3.3 Implementing a Parser

The rules we developed above do not change during a parse. Hence, instead of writing them into our parsing program, we can store them in a two-dimensional array, called a parse table. This parse table tells us the rule to apply for the current non-terminal in the sentential form and the current input symbol. Any blank entries in the parse table denote "error".

For any non-terminal X and terminal **z**, the fact that some production $X \rightarrow \alpha$ is in the parse table entry for [X,**z**] means that we have worked out that this is the production most likely to lead to a derivation of **z** from X.

Thus, to put entry $X \rightarrow \alpha$ into the parse table entry [X,**z**] we must have either:

1. **z** is in $FIRST(\alpha)$, or

2. ε is in $FIRST(\alpha)$ and **z** is in $FOLLOW(X)$

Using these rules we can systematically construct a parse table for any grammar.

The most common way to implement this type of parser is using a stack. At any stage during the parse, the leftmost non-terminal in the sentential form is on top of the stack, with the rest of the symbols (to the right) underneath it. Applying a production rule involves popping the non-terminal off the stack and pushing on the r.h.s. of the rule (with the symbols in reverse order) back on.

3.4 The $LL(k)$ Parsers

In general we can define a class of grammars known as the $LL(k)$ grammars, where k represents the number of lookahead symbols needed to eliminate any element of choice from a parse.

L = Scanning the input from **L**eft to right

L = Conducting a **L**eftmost derivation

k = Using k symbols of lookahead

Any (regular) grammar constructed from a DFA will be $LL(0)$ since we do not need to use FIRST or FOLLOW.

The class of grammars for which we need only one symbol of lookahead are called$LL(1)$ grammars - the definitions for FIRST and FOLLOW given earlier correspond to this case.

Sometimes when we construct an parse table there will be two or more rules in the same box. In this situation we say that the grammar is "not $LL(1)$", since the algorithm has clearly failed to indicate a non-deterministic parse in all cases.

For any k in $\{2, 3, 4, \ldots\}$, we can adjust the definitions of FIRST and FOLLOW so that they use k symbols of lookahead; this will give us an $LL(k)$ parse table. We say that a language is $LL(k)$ if there are no duplicate entries in its $LL(k)$ parse table.

Making k greater means that we will be able to parse more languages, but it also means that the algorithm will be more complex, and the resulting parse table will be larger. In general $LL(1)$ grammars represent the most acceptable compromise between power of expression and ease of implementation for top-down parsers.

3.5 Early's Parser

Early describes his parser as a "breadth–first top-down parser with bottom-up recognition" (!) While we will regard it as a top-down parser it does in fact exhibit many similarities to LR parsing.

Note that any similarity with the LR algorithms is superficial however:

- The LR algorithm generates a set of states from the grammar; this machine can then be used to parse *any* input (Parser construction)
- Early's algorithm is executed for a particular string; each string will cause different state-sets to be generated (Parser operation)

At any stage in a top-down parse we have a "current" non-terminal symbol and a choice of rules to apply to it. If we cannot make a definite choice (eg. based on lookahead), we must adopt a search strategy.

Conventionally, this will be

1. Depth first, where we choose the first alternative and try it; if this doesn't work we must backtrack and try the next one. (this is how Prolog's DCGs[1] work)
2. Breadth first, where we hive off one "subprocess" for each alternative, and proceed with trying each of them in parallel.

The major problem with breadth-first parsing is that it takes exponential time (relative to the input string length) in general. Early's approach seeks to limit the options chosen, resulting in an algorithm of order n^3, where n is the length of the input string.

[1]See the SICStus Prolog manual pages 69-72

3.5.1 Early's Items

This is achieved by enhancing the concept of an **item**. In Early's parser an item has the form:

$$i: \ {}_kX \ \rightarrow \ \alpha_\triangle\beta$$

where i and k are numbers, and $X \rightarrow \alpha\beta$ is a production rule. An item of this form is interpreted as meaning:

- We are at position i on the input (when $i = 0$ we are at the start of the string).
- The non-terminal X corresponds to some sub-string of the input which starts at position k
- We have recognised an α on the input, and should we go on to recognise β, we can then make the reduction to X

If the start symbol is S, we can represent this situation as:

$$\overbrace{x_0 \ldots \overbrace{\underbrace{x_k \ldots x_{i-1}}_{}\triangle\ \underbrace{x_i}_{}}^{X} \ldots\ldots x_{n-1}}^{S}$$

$$x_0 \ldots \overbrace{x_k \ldots x_{i-1}}^{\alpha} \ \triangle \ \overbrace{x_i}^{\beta} \ldots\ldots x_{n-1}$$

Thus at any stage during the parse we always have that $0 \leq k \leq i \leq n$ where n is the length of the input string.

We will need to define a version of the closure operation for these items:

- **State Closure** (Marker before a non-terminal)
  For each state with an item of the form
  $$i: \ {}_kX \ \rightarrow \ \alpha_\triangle Y\beta$$
  where Y is any non-terminal, take all rules of the form $Y \ \rightarrow \ \gamma$, and add the states for:
  $$i: \ {}_iY \ \rightarrow \ {}_\triangle\gamma$$

3.5.2 Early's Algorithm

Early's algorithm proceeds as follows:

Given any grammar with start symbol S, and some input string $x_0 \ldots x_{n-1}$ of length n,

1. Form the start state, which consists of the item
   $$0: \ {}_0 \ \rightarrow {}_\triangle S$$
   and its closure
2. For each new state (for some position i) that we create, and for each item in that state:
   - **Shift Input** (Marker before a terminal)
     If the item is of the form
     $$i: \ {}_kX \ \rightarrow \ \alpha_\triangle a\beta$$
     where a is a terminal symbol matching the input symbol x_i, create the state formed by taking the closure of:
     $$i+1: \ {}_kX \ \rightarrow \ \alpha a_\triangle\beta$$
   - **Apply Rule** (Marker at end of rule)
     If the item is of the form
     $$i: \ {}_kX \ \rightarrow \ \gamma_\triangle$$
     look at all those states for k that have the marker to the left of X; ie items of the form
     $$k: \ {}_jY \ \rightarrow \ \alpha_\triangle X\beta$$
     and create new states consisting of the closure of items of the form:
     $$i: \ {}_jY \ \rightarrow \ \alpha X_\triangle\beta$$

3. Keep applying step (2) until all states up to n have been generated. If one of these then contains the item

$$n: \ {}_0 \rightarrow S_\triangle$$

the parse has been successful.

3.6 Unger's Method

When faced with a choice of production rules, Unger's Method works by evaluating all possible partitions of the input string that can correspond to the symbols on the right-hand-side of each option.

As an example, we take the grammar

$$\begin{array}{lcl} E & \rightarrow & E + T \\ E & \rightarrow & T \\ T & \rightarrow & T * a \\ T & \rightarrow & a \end{array}$$

Suppose we are trying to match the string "a*a+a".
We may represent the initial situation as an attempt to match E to this:

E
$a * a + a$

Table 1

As there are two options for the right-hand-side, we enumerate these as:

E	$+$	T
$a * a$	$+$	a
$a*$	$a+$	a
a	$*a+$	a
a	$*a$	$+a$
a	$*$	$a + a$

Table 1.1

T
$a * a + a$

Table 1.2

If we consider table 1.2 first, we can expand the production rules for T to get two new tables:

T	$*$	a
$a * a$	$+$	a
$a*$	$a+$	a
a	$*a+$	a
a	$*a$	$+a$
a	$*$	$a + a$

Table 1.2.1

a
$a * a + a$

Table 1.2.2

Now by just looking at the columns involving terminal symbols, we can tell that both of these are impossible. The only correct rows in table 1.2.1 should have a "*" in the second column and an "a" in the third: there are none. Similarly, in table 1.2.2 it is clearly impossible for "a" to be the same as "a*a+a", and so we reject this also.

Going back then to table 1.1, we can see that some of its rows can also be eliminated. The middle column has the terminal "+" in it, so the only acceptable rows are those with a "+" in this column: that is row 1. So the table now looks like:

E	$+$	T
$a * a$	$+$	a

Now for each row that is left, we must expand both E and T, and consider *their* possible partitions.

Since we know that T can generate "a" (we could write out the tables to prove this), we can proceed with analysing E. Expanding out we get:

E	$+$	T
a	$*$	a

Table 1.1.1

T
$a * a$

Table 1.1.2

We can throw away table 1.1.1 immediately since "+" won't match with "*". Expanding the remining table gives us

T	$*$	a
a	$*$	a

Table 1.1.2.1

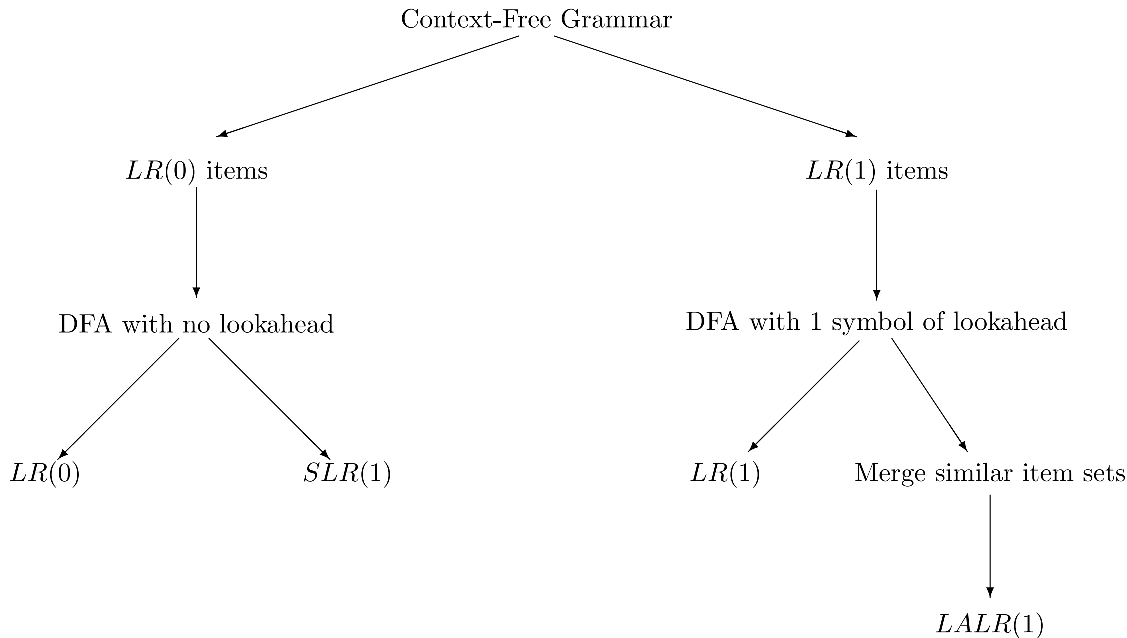
a
$a * a$

Table 1.1.2.2

The second table can be thrown away, and expanding T shows that table 1.1.2.1 does indeed represent a correct parse.

4 BOTTOM-UP PARSING

In this section we describe bottom-up parsing and examine the (deterministic, directional) $LR(k)$ family of parsers; i.e. $LR(0)$, $SLR(1)$, $LR(1)$ and $LALR(1)$. We also examine the Tomita and CYK approaches to bottom-up parsing.



4.1 Bottom-Up (Shift-Reduce) Parsing

In bottom-up parsing we start with the sentence and try to apply the production rules in reverse, in order to finish up with the start symbol of the grammar. This corresponds to starting at the leaves of the parse tree, and working back to the root.

- Each application of a production rule in reverse is known as a **reduction**.
- The r.h.s. of a rule to which a reduction is applied is known as a **handle**.

Thus the operation of a bottom-up parser will be as follows:

1. Start with the sentence to be parsed as the initial sentential form
2. Until the sentential form is the start symbol do:
   (a) Scan through the input until we recognise something that corresponds to the r.h.s. of one of the production rules (this is called a handle)
   (b) Apply a production rule in reverse; ie. replace the r.h.s. of the rule which appears in the sentential form with the l.h.s. of the rule (an action known as a reduction)

In step 2(a) above we are "shifting" the input symbols to one side as we move through them; hence a parser which operates by repeatedly applying steps 2(a) and 2(b) above is known as a **shift-reduce parser**.

A shift-reduce parser is most commonly implemented using a stack, where we proceed as follows:

- start with the sentence to be parsed on top of the stack
- a "shift" action correponds to pushing the current input symbol onto the stack
- a "reduce" action occurrs when we have a handle on top of the stack. To perform the reduction, we pop the handle off the stack and replace it with the terminal on the l.h.s. of the corresponding rule.

In a top-down parser, the main decision was which production rule to pick. In a bottom-up shift-reduce parser there are two decisions:

1. Should we shift another symbol, or reduce by some rule?

2. If reduce, then reduce by which rule?

Observe that the basic operation of a shift reduce parser is going through the input symbols from left-to-right looking for one of a particular set of strings (the r.h.ss. of the production rules).

Thus the basis of any shift-reduce parser will be a machine which can shift input symbols until it recognises a handle. We have already met such a machine - what we need is simply a (handle-recognising) DFA.

- A parser action such as "shift **a**" corresponds to making a transition from one state to another based on symbol 'a' in the DFA
- An action such as reduce by $A \rightarrow \alpha$ means that we have recognised all the symbols in α and have now reached what is effectively a final state for that handle.

In order to construct the DFA from a grammar we need some way of representing the current status of the parse as a state in the DFA.

4.2 $LR(0)$ items

An $LR(0)$ item is a production rule with the marker "$\triangle$" somewhere on the r.h.s.

An $LR(0)$ item of the form $A \rightarrow \alpha_1\alpha_2\ldots\alpha_i{\triangle}\alpha_{i+1}\ldots\alpha_n$ means that:

- we have already recognised terminal symbols on the input which correspond to (ie. which are reducible to) $\alpha_1\alpha_2\ldots\alpha_i$
- should we now recognise symbols corresponding to $\alpha_{i+1}\ldots\alpha_n$, then we will have moved to a final state in the DFA, and will be ready to reduce by the rule $A \rightarrow \alpha_1\ldots\alpha_n$

Note that an item which has the "$\triangle$" on the far right of the r.h.s. of a rule implies that it is time to perform a reduction by that rule (we've recognised everthing on the r.h.s.).

4.2.1 Kernel Items

When we have an item with the marker immediately to the left of some non-terminal, then this tells us that the next step in the parse is to recognise some sequence of symbols which correspond to this non-terminal. To do this, we have to examine the rules for that non-terminal, and proceed from there.

For instance, the item $B \rightarrow \mathbf{c}{\triangle}C$ tells us that the next step is to recognise a C; we can do this by recognising one of the two r.h.s. of the production rules for C. This item is thus equivalent to: $C \rightarrow {\triangle}B\mathbf{d}$ and $C \rightarrow {\triangle}\mathbf{d}$.

An item with the "$\triangle$" on the far left of the r.h.s. is called a **non-kernel item**. Anywhere we have the "$\triangle$" to the immediate left of a non-terminal is equivalent to each of the non-kernel items for that non-terminal.

4.2.2 The closure and move operations on items

The process of building a handle-recognising DFA from $LR(0)$ items is very similar to converting a NFA to a DFA using the "subset construction" algorithm of section 1. Hence we will need some equivalent of ε-closure and the NFA *move* function.

To get the closure of a set of items, take all those items which have the marker immediately to the left of a non-terminal, and add in all the non-kernel items for that non-terminal. This operation is very similar to the ε-closure operation since we can "get to" all items in the closure set of an item without shifting any symbols (ie. without consuming any input).

The transitions in the DFA will correspond to moves between item-sets based on some symbol. Basically move works as follows:

Let x represent any symbol (terminal or non-terminal), α and β are any sequence of symbols, then:

$$\begin{aligned} \mathsf{move}\{A \rightarrow \alpha{\triangle}x\beta, x\} &= \{A \rightarrow \alpha x{\triangle}\beta\} \\ \mathsf{move}\{A \rightarrow \alpha{\triangle}x\beta, b\} &= \{\},\ for\ any\ b \neq x \end{aligned}$$

4.3 Handle-recognising DFAs and LR(0) Parsers

Using these idea, we are ready to construct a handle-recognising DFA. DFA states are sets of items; making a transition in the DFA corresponds to moving the marker one place to the right in an item.

We make things a little easier if we introduce a production rule of the form $S' \rightarrow S$ to the grammar, where S is the start symbol. Whenever we reduce by this rule, we know that we are finished. (A grammar which has had this rule added is called an **augmented grammar**).

Since our new start symbol is now S', the "goal" of the parse will be to recognise an S and reduce this to S'; this is represented by the $LR(0)$ item: $S' \rightarrow {}_{\triangle}S$. This item is called the **initial item**.

4.3.1 Construction of a handle-recognising DFA

To construct a handle-recognising DFA from a context-free grammar, perform the following:

1. Prepare the grammar by:
   (a) Eliminating any ε-productions
   (b) If S is the start symbol, add a new rule to the grammar $S' \rightarrow S$.
2. The start state of the DFA is $\mathsf{closure}\{S' \rightarrow {}_{\triangle}S\}$
3. Each time we create a new state in the DFA perform the following:
   For each grammar symbol (terminal or non-terminal):
   work out the closure of the move operation applied to newly created state and this symbol; this may create new states.

Note the similarities between this and the "subset construction" algorithm which we used to convert a NFA to a DFA.

4.3.2 Operation of a handle-recognising DFA

Any handle-recognising DFA works as follows:

1. Start in state I_0.
2. **SHIFT:** As we read symbols from the input, make the appropriate transitions in the DFA from the current state.
3. **REDUCE:** If we are in a state that has a production rule of the form
   $A \rightarrow \alpha_{\triangle}$ for any non-terminal A and r.h.s. α, then we have recognised the handle α, and can reduce by the rule. We do this as follows:
   (a) move backwards along the DFA transitions for each symbol in α
   (b) from the state you end up in, make a forwards transition on A
4. **ACCEPT:** The final DFA state is the one containing the item $S' \rightarrow S_{\triangle}$. If we reach this state and have consumed all the input, then we accept the string as being valid.

4.3.3 The $LR(k)$ parsers

Any parser constructed using the above method will be a member of the family of $LR(k)$ parsers:

L = scanning the input from **L**eft to right

R = Conducting a **R**ightmost derivation (in reverse)

k = using k symbols of lookahead.

As usual, lookahead is indicated by the presence of FIRST or FOLLOW somewhere in the construction. We did not use them above, so we have constructed a $LR(0)$ parser.

4.3.4 Inadequte States

It is possible that some of the DFA states constructed using the previous algorithm will contain items which suggest both "shift" and "reduce" actions. This situation is known as a **shift/reduce conflict**.

Because we have conflicts in the handle-recognising DFA, the parser will not know which action to take; it will have to pick one arbitrarily and try it out. Because of this non-determinism, we say that the grammar was not $LR(0)$.

An alternative problem could arise if we have two or more items in a state which have the "$\triangle$" marker to the right of their r.h.s.; thus we would not know which one to reduce by. This situation is called a **reduce/reduce conflict**. Instances of this are rarer than shift/reduce conflicts.

[Note that no state can have a **shift/shift conflict** since we are using a DFA; this sort of problem would arise only as a result of using an NFA.]

Any state in a handle-recognising DFA which contains either a shift/reduce or a reduce/reduce conflict is known as an **inadequate state**.

It is the ideal of LR parser design to minimise the number of inadequate states without sacrificing efficiency. The most common way of doing this is to introduce lookahead into the parser construction, leading to the $LR(1)$ family of parsers.

4.4 SLR(1) Parsers

$SLR(1)$ stands for $SimpleLR(1)$; this is basically a method of adding lookahead to $LR(0)$ parsers as simply as possible.

The technique is based on the following observation:

If we are in a DFA state containing the item: $A \rightarrow \alpha_\triangle$ then a possible action will be to reduce by this rule. Doing this reduction would involve:

going from a sentential form that looks like: $\ldots \alpha_\triangle \ldots$
to one that looks like: $\ldots A_\triangle \ldots$

By looking at examples, we can see that the symbol immediately to the right of the marker in a sentential form should correspond to the next input symbol: we can rephrase this as: the symbol following A should be the next symbol in the input.

Since we already have a method of characterising the set of symbols which can follow a non-terminal in a sentential form, we can formulate the $SLR(1)$ **reduction rule**:

- Reduce by the rule $A \rightarrow \alpha$ only if the current state contains $A \rightarrow \alpha_\triangle$ and the next input symbol is in $FOLLOW(A)$

This provides a quick and easy way to incorporate lookahead into the parser; however, there are many languages which are not $SLR(1)$.

4.5 $LR(1)$ Parsers

In order to introduce lookahead as early as possible into the parser construction, we define the following:

An $LR(1)$ **item** is an $LR(0)$ item plus some lookahead symbols.
Thus an $LR(1)$ item looks like: $[A \rightarrow \alpha_1\alpha_2 \ldots \alpha_{i\triangle}\alpha_{i+1} \ldots \alpha_n,\ a]$

This means that:

- we have just recognised $\alpha_1\alpha_2 \ldots \alpha_i$, and
- should we go on to recognise $\alpha_{i+1} \ldots \alpha_n$ (or something which corresponds to them), then we can reduce by the rule $A \rightarrow \alpha_1 \ldots \alpha_n$,
- with the understanding that, when we do, the next symbol on the input will be 'a'.

[Note: Given that the next input symbol will be following A in the current sentential form, we know that the lookahead symbol 'a' must be in $FOLLOW(A)$.]

4.5.1 Constructing LR(1) Item sets

In order to construct the sets of items for an $LR(1)$ parser, we just need to make three modifications:

1. The Initial Item
   The initial item is now of the form $[S' \rightarrow {}_\triangle S,\ \perp]$. That is we want to recognise an S (or something corresponding to it) and reduce by this rule, on the understanding that the next symbol is the end of the input (ie. that there is no more input).

2. The move operation
   As might be expected, $\mathsf{move}([A \rightarrow \alpha_\triangle B\gamma,\ a], B) = [A \rightarrow \alpha B_\triangle \gamma,\ a]$; ie. move does not change the lookahead symbol

3. The closure Operation
   Previously we had constructed the closure of the $LR(0)$ item $[A \rightarrow \alpha_\triangle B\gamma]$ by adding in all non-kernel items of the form $[B \rightarrow {}_\triangle\beta]$. This will be the same for the $LR(1)$ item $[A \rightarrow \alpha_\triangle B\gamma, a]$, but we need to compute a new lookahead set.

   Since we are only interested in finding β and reducing it to B as a kind of a "sub-goal" of finding $B\gamma$, we can state that after reducing to B, the next input symbol should match whatever is derived from γ; ie. it should be in $FIRST(\gamma)$. If γ can derive ε, then the next input symbol should match what had previously been expected to follow A, ie. the lookahead symbol 'a'.

   Thus: $\mathsf{closure}([A \rightarrow \alpha_\triangle B\gamma, a])$ contains all items of the form $[B \rightarrow {}_\triangle\beta,\ b]$, where 'b' is a terminal in $FIRST(\gamma a)$

The algorithm for constructing a handle-recognising DFA is exactly the same as before, except that we use the above modifications.

One result of these changes is to increase the number of possible items, and also the number of likely states in the handle-recognising DFA; thus $LR(1)$ parsing tables for a grammar tend to be significantly larger than the $LR(0)$ or $SLR(1)$ tables for the same grammar.
However, the parsing is improved, because now we can modify the $SLR(1)$ reduction rule to get:

The LR(1) reduction rule:

- Reduce by the rule $A \rightarrow \alpha_1 \ldots \alpha_n$ if and only if the item $[A \rightarrow \alpha_1 \ldots \alpha_{n\triangle},\ a]$ is in the current state, and 'a' is the next input symbol.

Since the number of 'a' symbols may be smaller than $FOLLOW(A)$, there can be less reductions in an $LR(1)$ parse table, and thus less potential conflicts.

4.6 $LALR(1)$ Parsers

While $LR(1)$ parsers are quite powerful, they can involve a large number of states, and this can cause complications for the implementation. We can reduce the number of states without paying too much of a price in terms of increased conflicts by noting that there will often be two different states whose basic items are the same, but which differ only in lookahead sets.
We can reduce the size of an $LR(1)$ parse table if we combine such states together; the resulting lookahead is the union of the two individual lookaheads. Parse tables constructed from this type of DFA are known as $LALR(1)$ parsers, for Look-Ahead $LR(1)$ parsers. [Stupid name - all $LR(1)$ parsers have lookahead]

Merging two states in this manner will not adversely effect the move function, since this does not depend on the lookahead.

Merging states will not add any extra shift/reduce conflicts that were not in the $LR(l)$ parser originally, since shift actions do not depend on lookaheads. However, the number of reduce/reduce conflicts may be increased.

4.7 Tomita's Parser

Tomita's Algorithm amounts to a simple generalisation of the standard LR algorithm. The LR parse-table is constructed as usual, and we proceed with the parsing algorithm in the normal way.

In order to keep a record of the parse-state, we maintain a stack consisting of symbol/state pairs. This is maintained as follows:

1. Each time we make a shift action we push the symbol and the new state onto the stack.
2. A reduce action for some rule whose right-hand-side has length k involves popping k symbol/state pairs off the stack, and then pushing the symbol and state corresponding to the rule's left-hand-side.

Thus at any stage during the parse, the current state will be on top of the stack. When we meet a conflict arising from an inadequate state, we simply *duplicate* the stack, and split the parse into a different process for each stack. The input string is accepted if *any* of the parallel parsings terminate successfully.

In general the effectiveness of this algorithm is proportional to the "strength" of the initial parse table; e.g. a Tomita Parser constructed from an LALR(1) table can expect to be more efficient than one constructed from an LR(0) table.

4.8 CYK Method

The Cocke-Younger-Kasami parsing method is based on a bottom-up approach to the partitioning of the input. In its most general form it is identical to *chart parsing*.

The algorithm proceeds as follows:

Phase 1 For each symbol in the input string, construct a set of all those non-terminals that can generate that symbol on its own. (This can be calculated from the grammar)

Phase 2 For each adjacent pair of symbols, construct a set consisting of all those non-terminals that can derive that pair, or some pairing of their phase 1 non-terminals.

Phase i For each substring of the input of length i, construct a set of non-terminals which can generate that string, or some combonation of the terminals in the string and any phase k non-terminals (for any $k < i$).

We are finished if the start symbol is a phase n non-terminal, where n is the string length.