Designing a Lexical Analyser

- The key function in a lexical analyser is a routine called **GetNextToken** that extracts tokens one-by-one from the source file.
- The lexical analyser repeatedly makes calls to GetNextToken to process the whole file.
- When tokenising the input it is important to identify the **Token Separators**. These separators are special characters that delimit one token from an other. In many programming languages, the token separators are usially spaces, tab stops and carriage returns.
- The scanning loop can look like:

```
Initialise;
loop
    symbol = GetNextToken();
    Print symbol;
Until symbol = End_Of_File;
Clean_Up;
```


Recognising Tokens

- The scanner will start recognising a token after reading the first character:
- If the first character is:
  - A letter: then we're dealing with a keyword or identifier.
  - Numeric: then we're dealing with a number.
  - An Opening Quote (“): then we're dealing with a string.
  - Etc…

GetNextToken

```
Function GetNextToken
  CurrentChar = Get the next significant character

  If CurrentChar = EOF Then
       Deal with the end of file
  Else
       If CurrentChar = Digit Then
              CurrentToken = Deal With Number
       Else If CurrentChar = Letter Then
              CurrentToken = Deal With a Word
       Else If
              ...
       Else
              CurrentToken = Error – Illegal Character
       End IF
       GetNextToken = CurrentToken
   End If
End Function
```


GetWord

```
Function GetWord

  MyToken = CurrentChar
  CurrentChar = Get the next significant character

  While CurrentChar is Valid in a Word
      MyToken = MyToken + CurrentChar
      CurrentChar = Get next significant character
  Wend

  Return MyToken


End Function
```

Look Ahead

- Since we are reading characters until we find one that is not part of a word, the last one is essentially one extra character. We must not discard it. In fact *CurrentChar* is usually implemented as a global variable.
- This extra character is called the *LookAhead* character.

GetString

```
Function GetString

  MyToken = CurrentChar ' the opening quote
  CurrentChar = Get the next significant character

  While CurrentChar is not the Closing Quote
        MyToken = MyToken + CurrentChar
        CurrentChar = Get next significant character
  Wend

  MyToken = MyToken + CurrentChar ' The closing quote

  ' Again read an extra character for consistency
  CurrentChar = Get the next significant character

  Return MyToken

End Function
```

Error Reporting

- There are only a few errors that can be detected by the scanner. Such errors include:
  - Missing closing quote in a string. Missing quotes are a major issue, since characters will be read until the opening quote of the next sting are found, potentially 'eating-up' much of the actual code. This problem is typically alleviated by not allowing strings to span over multiple lines.
  - Illegal characters in the input file.

Parsing

- When analysing a programs syntax, a data structure called the **Parse Tree** is built to reflect the structure of the program.
- The nodes of the parse tree are the Non-Terminal symbols, whilst the leaves are the Terminals (Σ).
- The root node is the sentence symbol (S).
- There are two main methods of parsing:
  - Top-Down Parsing – the parse tree is build from the root downwards.
  - Bottom-Up Parsing – the parse tree is built from the leaves upwards to the root.

Simple Parsing Example

- Consider the BNF specification for a simple assignment statement:

```
<assign>      ::= <lhs> <ass-op> <rhs>;
<lhs>         ::= <ident>;
<ass-op>      ::= "=";
<rhs>         ::= <integer>
                    <arith-op> <integer>;
<arith-op>    ::= "+" | "-" | "*" | "/";
```

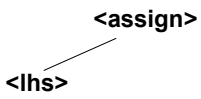

Top-Down Parse (1)

- In this example we will construct a parse tree for the assignment:

```
counter = 3 + 4
```

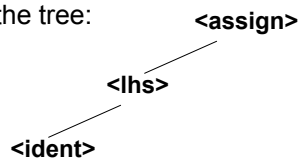
- The parser looks for the sentence symbol to create the root node:

- The first symbol in the rule is an LHS so we add it to the parse tree:

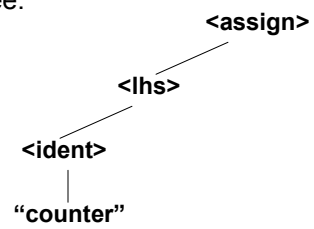


Top-Down Parse (2)

- An LHS is an IDENT which we add to the tree:

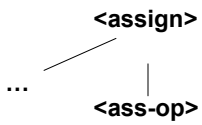


- In our example “counter” is an IDENT – we have a match and add it to the tree:

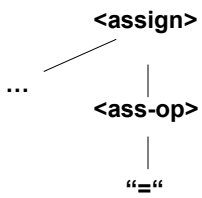


Top-Down Parse (3)

- The next expected item is the assignment operation, so it is added to the tree (at the current root because it is a non-terminal):

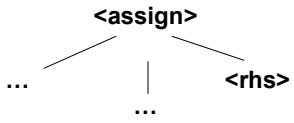


- An assignment operator is a non-terminal an looks at the next token and finds one:



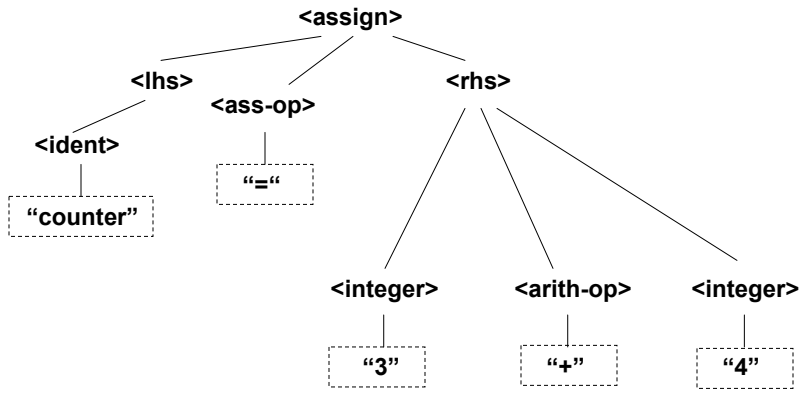
Top-Down Parse (4)

- The next item to be expected is an RHS which is a non-terminal so we add it to the current root:



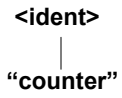
- The procedure is repeated until we complete the tree and find out that our assignment is structurally correct.

Top-Down Parse (5)

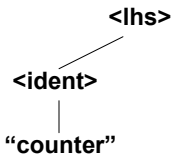


Bottom-Up Parse (1)

- The first token in the input is an IDENT so the leaf of the tree is obtained:

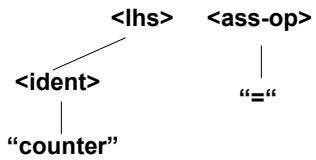


- By having a look at the rules we see that an IDENT is an LHS, so the tree grows up:



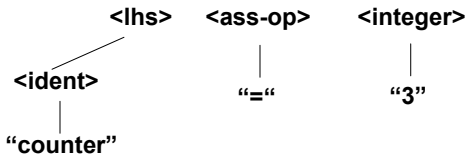
Bottom-Up Parse (2)

- An LHS on it's own cannot be resolved into anything else, so we continue reading from the input. We find an "=" sign so it's added as a leaf:

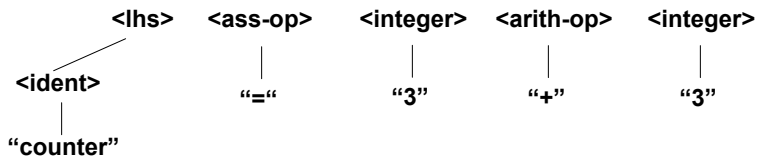


- The ASS-OP non-terminal cannot be resolved into anything else and neither can the LHS ASS-OP sequence so we continue reading the input and find a number which we add as a leaf:

Bottom Up Parse (3)

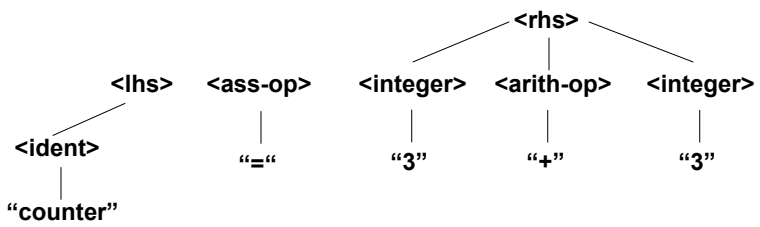


- This process continues until we consumed the whole input:



Bottom-Up Parse (4)

- After reading the last integer we see that the INTEGER ARITH-OP INTEGER sequence can be reduced to an RHS:



- Similarly in the next step we see that the resulting LHS ASS-OP RHS sequence can be further reduced to an ASSIGN, thus the parse is complete.

Things to Note…

- The grammar chosen for this example was purposely designed to keep the example simple.
- In reality parsing mechanisms are more sophisticated and grammars may really manifest properties that make parsing more complex. Consider the following grammar for a more elaborate assignment:

```
<assign>  ::= <lhs> <ass-op> <rhs>;
<lhs>     ::= <ident>;
<rhs>     ::= <factor> { <arith-op> <factor> };
<factor>  ::= <ident> | <integer>;
<ass-op>  ::= "="
<arith-op> ::= "+" | "-" | "*" | "/"
```


…Things to Note

- When parsing the assignment statement using the original grammar, when we read the identifier leaf, we saw that it could be reduced to an LHS (see Bottom-Up Parse (1)). Using the grammar presented above, we see that the identifier could be reduced to both an LHS or a FACTOR. The question here is – *Which path shall I follow?*
- Such issues are tackled by more sophisticated parsers.

Parsing a Variable Declaration (1)

- The simplest way to hand-code parsers is to provide programming language equivalents to BNF notational constructs:

Terminal Symbols	Test for the terminal symbol.
Non-Terminal Symbols	A procedure or function call.
Repetitions - { }	A while loop.
Alternatives - |	If-Then-Else statements.
Optional Items – [ ]	If-Then Statement.

Parsing a Variable Declaration (2)

- Consider simple variable declaration statements:

```
integer i,j,k
boolean isReady
```

- A suitable grammar to parse such statements would be:

```
<VarDecl>       ::= <TypeName> <VarNameList>;
<TypeName>      ::= INTEGER|BOOLEAN|REAL;
<VarNameList>   :: <VarName> {“,” <VarName>};
```

Parsing a Variable Declaration (3)

- For each Non-Terminal symbol, we define a function to parse it:

```
Function Parse_VarDecl(TOKEN)
  // Parse the type name
  LOOKAHEAD = Parse_TypeName(TOKEN)

  // Parse the variable name list
  LOOKAHEAD = Parse_VarNameList(LOOKAHEAD)

  Return LOOKAHEAD
End Function
```


Parsing a Variable Declaration (4)

- Since the Variable Declaration (VarDecl) is defined in terms of 2 other Non-Terminals (TypeName and VarNameList), it's parse is defined as calls to two other functions to parse each other non-terminal.
- Just as we had a look ahead character in the lexical analyser, we need a look ahead token in the parse tree.

Parsing a Variable Declaration (5)

- Parsing the Type Name:

```
Function Parse_TypeName(TOKEN)
  If TOKEN is INTEGER_TOKEN then
   Return NextToken // New Lookahead
  ElseIf TOKEN is BOOLEAN_TOKEN then
   Return NextToken
  ElseIf TOKEN is REAL_TOKEN then
   Return NextToken
  Else
   Print "Missing Type Name in Declaration"
   Return Error
  End If
End Function
```


Parsing a Variable Declaration (6)

- Note that in the previous example we used two of the transliteration mechanisms. We ised simple checks to see if a token is a non-terminal and we used If-Then-Else statements for alternatives.
- Parsing the Variable Name List:

```
Function Parse_VarNameList(TOKEN)
  LOOKAHEAD = Parse_VarName(TOKEN)
  While LOOKAHEAD = COMMA_TOKEN
   LOOKAHEAD = NextToken
   LOOKAHEAD = Parse_VarName(LOOKAHEAD)
  End While

  Return LOOKAHEAD
End Function
```

Parsing a Variable Declaration (7)

- In the preceding example we used a combination of parsing both terminals (comma's) and non-terminals (variable names). The repetition was handled by a while loop that allowed for zero-or more items enclosed in the braces { }.
- Parsing the variable name:

```
Function Parse_VarName(TOKEN)
  If TOKEN = IDENT_TOKEN then
   Return NextToken
  else
   Print "Missing Identifier"
   Return Error
  End If
End Function
```


Parsing an If-Then Statement (1)

- Consider the following definition for an if-then statement:

```
<IF_STMT> ::= "IF" <EXPRESSION> "THEN"
                  <STMT_BLOCK>
                  ["ELSE" <STMT_BLOCK>]
                  "ENDIF"
```

```
Function Parse_If(TOKEN)
   If TOKEN = IF_TOKEN Then
         LOOKAHEAD = NEXTTOKEN
         // Parse the Expression
         LOOKAHEAD = Parse_Expression(LOOKAHEAD)
         If LOOKAHEAD = THEN_TOKEN then
                  LOOKAHEAD = NEXTTOKEN
                  LOOKAHEAD = Parse_StmtBlock(LOOKAHEAD)
                  // see if we have an else part
                  If LOOKAHEAD = ELSE_TOKEN then
                            LOOKAHEAD = NEXTTOKEN
                            LOOKAHEAD = Parse_StmtBlock(LOOKAHEAD)
                  End If
                  // get the ENDIF
                  If LookAhead = ENDIF_TOKEN Then
                           Return NEXTTOKEN
                  Else
                           Print "Missing ENDIF in conditional"
                           Return Error
                  End If
         Else
                  Print "Missing THEN in conditional"
                  Return Error
         End If

   End If
End Function
```


Syntax Errors

- Consider the following assignment statement with a missing comma:

```
integer i,j k;
```

- The parse will proceed normally in the VarNameList part until the variable k is found instead of the comma. The function will "think" that the variable name list has terminated returning k as the look ahead token. The parser will look for the ending semi-colon and will find a k instead reporting a missing semi-colon instead of the missing comma.