Two ways of constructing the parse tree

• Predictive parser:
  – The lookahead symbol determines which production to apply, without backtracking

• Recursive-descent parser:
  – A special kind of top-down parser: single left-to-right scan, with one lookahead symbol.
  – Backtracking (trial-and-error) may happen

Challenge 1: Top-Down Parser Cannot Handle Left-Recursion

```
expr → expr – term | term
term → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Input: 3 – 4 – 5

Parse tree

Eliminating Left-Recursion

```
expr → expr – term | term
term → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

(Complete example in Fig. 4.12, page 105-107)
Challenge 2: Backtracking is Inefficient

- Backtracking: trial-and-error

\[ \text{type} \to \text{simple} | \text{array} \{ \text{simple} \} \text{ of type} \]

\[ \text{simple} \to \text{integer} | \text{char} \]

Input: array \{ integer \} \text{ of char}

Parse tree:

[Diagram]

Challenge 2: Backtracking is Inefficient

- Subscription: \text{term} \to \text{term} \text{ term}

- Term: \text{term} \to \text{0|1|2|3|4|5|6|7|8|9}

- We cannot avoid backtracking if the grammar has multiple productions to apply, given a lookahead symbol.

- Solution: Predictive Parser
  - If we cannot decide early, defer the decision until we have seen enough.
  - Change the grammar so that there is only one applicable production that is unambiguously determined by lookahead symbol.

Avoiding Backtracking by Left Factoring

\[ A \to \alpha \beta_1 | \alpha \beta_2 \]

\[ A \to \alpha A' \]

\[ A' \to \beta_1 | \beta_2 \]

\[ \text{expr} \to \text{term } | \text{term ... term} \]

\[ \text{term} \to 0|1|2|3|4|5|6|7|8|9 \]

Syntax Diagrams

- Written from EBNF, not BNF
- If-statement

[Diagram]

Review of Parsing

- Grammar for Integer Arithmetic Expression

\[ \text{expr} \to \text{expr } + \text{expr} | \text{expr } * \text{expr} | (\text{expr}) | \text{number} \]

\[ \text{number} \to \text{digit} | \text{digit} \]

\[ \text{digit} \to 0|1|2|3|4|5|6|7|8|9 \]

- Ambiguity 1: precedence

[eliminated by precedence cascade]

[Diagram]
Review of Parsing

- Ambiguity 2: Associativity (eliminated by recursion)

  \[
  \text{expr} \rightarrow \text{expr} + \text{term} | \text{term} \\
  \text{term} \rightarrow \text{term} * \text{factor} | \text{factor} \\
  \text{factor} \rightarrow \{ \text{expr} \} | \text{number} \\
  \text{number} \rightarrow \text{number digit} | \text{digit} \\
  \text{digit} \rightarrow 0|1|2|3|4|5|6|7|8|9
  \]

- Ambiguity 3: Dangling-Else

Review of Parsing

- Parsers
  - top-down
    - (construct the parse tree from root towards leaves)
  - recursive-descent
    - (left-to-right scan, using single lookahead symbol)
      - predicative parser
        - (no backtracking, the lookahead symbol unambiguously determines the production to apply)
      - bottom-up
        - (construct the parse tree from leaves towards root)

Review of Parsing

- Allowing Top-Down Parsers: Eliminating Left-Recursion
- Allowing Predicative Parsers: Removing Left-Factoring
- Alternative representations: make predicative parser easy to write
  - EBNF
  - Syntax Diagram

Semantics

Names

- Names: identify language entities
  - variables, procedures, functions, constants, data types, ...
- Attributes: properties of names
- Examples of attributes:
  - Data type:
    - \text{int } n = 5; \quad (\text{data type: integer})
  - Location:
    - \text{int } * y;
  - Value:
    - y = new int; \quad (a new value for y, a new location for *y)
  - Parameters, return value:
    - \text{int func(int n) \{ \ldots \}}

Binding Time

- Binding Time: the time when an attribute is bound to a name.
  - Static binding (static attribute):
    - occurs before execution
      - Language definition/implementation time: The range of data type \text{int}
      - translation time (parsing/semantic analysis): The data type of a variable
      - link time: The body of external function
      - load time: Location of global variable
  - Dynamic binding (dynamic attribute):
    - occurs during execution
      - \text{entry/exit from procedure or program: the value of local variables}
Binding

- Binding: associating attributes to names
  - declarations
  - assignments
  - declarations (prototype) and definition of a function

- The bindings can be explicit or implicit
  - e.g., int x;
    - Explicit binding: the data type of x
  - Implicit binding: the location of x (static or dynamic, depending on where the declaration is)

- The declaration itself can be implicit:
  - FORTRAN: no need to declare variables begin with IJKLNM

Chapter 5

C++ Example

```c++
const int Maximum = 100;
struct FullName {string Lastname, string FirstName};
class Student {
private:
  struct FullName name;
  int Age;
public:
  void setValue(const int a, struct FullName name);
  void setValue(const int a, string lName, string fName);
  int TStudent();
};
void Student::setValue(const int a, string lName, string fName) {
  int i;
  Age = a;
  { int j;
    name.LastName = lName;
    name.FirstName = fName;
  }
}
```

Scope of Binding

- Scope of Binding: the region of the program where the binding is maintained (is valid, applies).
  (Scope of Declaration: if all relevant bindings by the declaration have identical scope.)

- Block-structured language
  - lexical scope (static scope): from the declaration to the end of the block containing the declaration.
  - dynamic scope: introduced later.

Example

```
int x;
void p(void) {
  char y;
  ...
} // i

void q(void) {
  double z;
  ...
} // j

main() { 
  int w[10];
  ....
} // l
```

Declaration before Use

```
void p(void) { 
  int x;
  ....
  char y;
  ....
}
```

Exception in OO languages: Scope of local declarations inside a class declaration includes the whole class.

```
public class { 
  public int getValue() { return value; }
  int value;
};
```

Where can declarations happen?

- Blocks ({}), begin-end, ... Algol descendants: C/C++, Java, Pascal, Ada, ...
  - e.g., C
    - Function body
    - Anywhere a statement can appear (compound statement)

- External/global
- structured data type
- class
Scope Hole

• Scope Hole: Declarations in nested blocks take precedence over the previous declarations. That is, binding becomes invisible/hidden.

int x;
void p(void) {
  char x;
  x = 'a';
  
  main() {
    x = 2;
  }
}

Access Hidden Declarations

• scope resolution operator :: (C++)

int x;
void p(void) {
  char x;
  x = 'a';
  ::x = 42;
  
  main() {
    x = 2;
  }
}

Hide a Declaration

• File 1: File 2:
  extern int x;
  int x;

• File 1: File 2:
  extern int x;
  static int x;

Symbol Table

• Symbol Table: maintain bindings. Can be viewed as functions that map names to their attributes.

SymbolTable

Names    Attributes

The symbol table in p: the bindings available in p

int x = 1;
char y = 'a';

void q(void) {
  double a = 2.5;
  printf("%c\n", y);
}

int x = 1;
char y = 'a';

void p(void) {
  double a = 2.5;
  printf("%c\n", y);
}

int x = 1;
char y = 'a';

void q(void) {
  int y = 42;
  printf("%d\n", x);
  p();
}

main() {
  char x = 'b';
  q();
}

int x = 1;
char y = 'a';

void q(void) {
  double a = 2.5;
  printf("%c\n", y);
}

int x = 1;
char y = 'a';

void p(void) {
  double a = 2.5;
  printf("%c\n", y);
}

int x = 1;
char y = 'a';

void q(void) {
  int y = 42;
  printf("%d\n", x);
  p();
}

main() {
  char x = 'b';
  q();
}
The symbol table in \texttt{q}:
the bindings available in \texttt{q}
\begin{itemize}
\item integer, 1, global
\item character, 'a', global
\end{itemize}

The symbol table in \texttt{main}:
the bindings available in \texttt{main}
\begin{itemize}
\item integer, 1, global
\item character, 'a', global
\end{itemize}

\textbf{Static vs. Dynamic Scopes}
\begin{itemize}
\item \textbf{Static scope (lexical scope)}:
  \begin{itemize}
  \item scope maintained statically (during compilation)
  \item follow the layout of source codes
  \item used in most languages
  \end{itemize}
\item \textbf{Dynamic scope}:
  \begin{itemize}
  \item scope maintained dynamically (during execution)
  \item follow the execution path
  \item few languages use it (The bindings cannot be determined statically, may depend on user input).
  \end{itemize}
\end{itemize}

\textbf{What if it used dynamic scope?}
\begin{itemize}
\item integer, 1, global
\item character, 'a', global
\end{itemize}
What if it used dynamic scope?

```
int x = 1;
char y = 'a';
void p(void) {
    double x=2.5;
    printf("%c\n",y);
}

x character, 'b', local to main
The symbol table in q:
the bindings available in q

void q(void) {
    int y = 42;
    printf("%d\n",x);
p();
}
main() {
    char x = 'b';
    q();
}
```

The symbol table in q:
the bindings available in q

```
int x = 1;
char y = 'a';
void p(void) {
    double x=2.5;
    printf("%c\n",y);
}

x character, 'b', local to main
double, 2.5, local to p
The symbol table in p:
the bindings available in p

void q(void) {
    int y = 42;
    printf("%d\n",x);
    p();
}
main() {
    char x = 'b';
    q();
}
```

The symbol table in p:
the bindings available in p

x integer, 1, global
double, 2.5, local to p
character, 'b', local to main

y integer, 42, local to q
character, 'a', global

Main() {
    char x = 'b';
    q();
}

y integer, 42, local to q
character, 'a', global