Slides to Accompany *Programming Languages and Methodologies*

R. J. Schalkoff

Chapter 4, Part 1: `minic` and Parsing

minic (ver 1) Syntax, Part 1

```
<transunit> ::= <main-decl> <body>
<main-decl> ::= <type> main <arg>
<type> ::= int | void
<arg> ::= lparens <type> rparens
<body> ::= lbrace <decl> <statseq> rbrace
             /* forces declarations, if any,  first and only once */
<decl> ::= e | <type> <variable-list>;
<variable-list> ::= <variable> | <variable> , <variable-list>
<variable> ::= <identifier>
<statseq> ::= <return-stat> | <command-seq>; <return-stat>
<command-seq> ::= <command> | <command> ; <command-seq>
<command> ::= <variable> assign <expr>
<expr> ::= <numeral>
<return-stat> ::= return <return-arg>;
<return-arg> ::= lparens <variable> rparens  |
                              lparens <numeral>  rparens
```

minic (ver 1) Syntax, Part 2

```
/* lexical part (user identifiers and numerals) of the syntax */

<identifier> ::= <letter> | <identifier> <letter> |  <identifier> <digit>
<letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m
                  | n | o | p | q | r | s | t | u | v | w | x | y | z
<numeral> ::= <digit> | <digit> <numeral>
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

Syntax Notes

Note that some terminals in the grammar are shown using very 'unintuitive' names.

`minic terminal`	actually used	meaning
lparens	(	left parens
rparens	)	right parens
lbrace	{	left brace
rbrace	}	right brace
assign	=	assignment

Table 1: 'Equivalent' Terminals in `minic`

minic (ver 1) Sample Program

```
int main(void)
{
int t1;
t1=10;
return(t1);
}
```

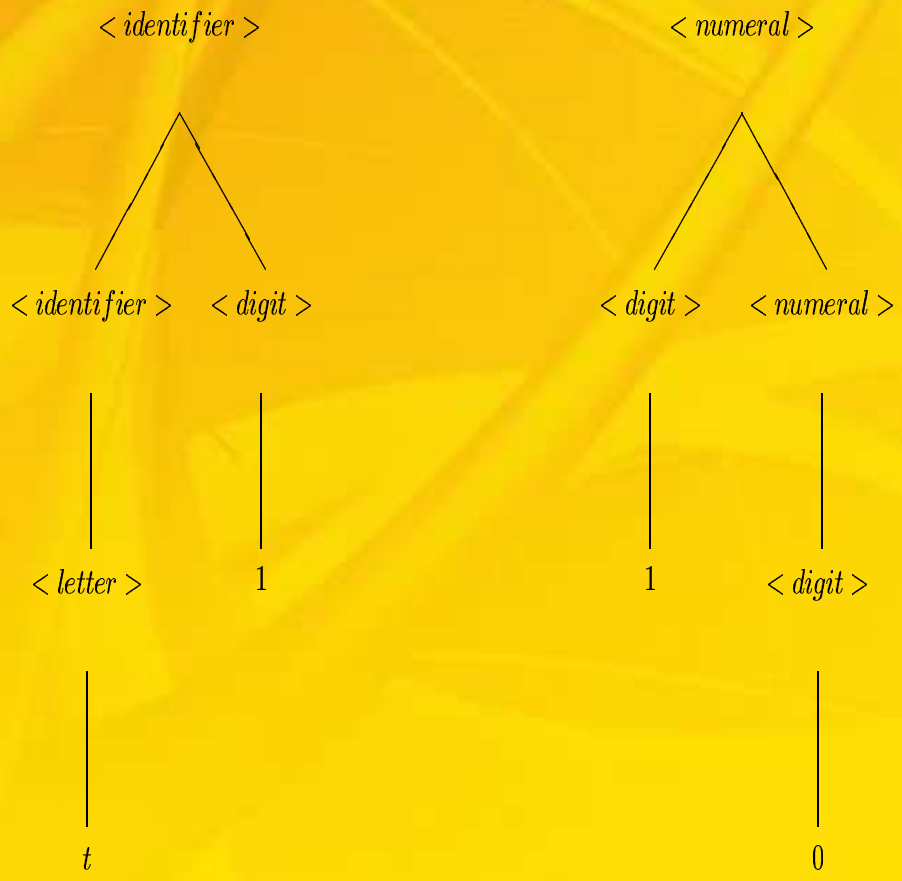
`minic` Syntax Subdivision

1. The productions which fundamentally determine the structure of major constructs in the language (main declarations, statement sequence, etc)
2. Reserved words and symbols
3. User-formed 'pseudo-terminals', i.e., strings, which are formed to represent numerals and identifiers

Reserved Words and Symbols

```
int
void
lparens
rparens
lbrace
rbrace
,
;
assign
return
a | b | c | d | e | ...
0|1|2|3|4|5|6|7|8|9
```

Sample Derivation Trees



Derivation Tree for Sample Program

Figure 2 shows the derivation tree for the following simple `minic` program:

```
int main(void)
{
int t1;
t1=10;
return(t1);
}
```

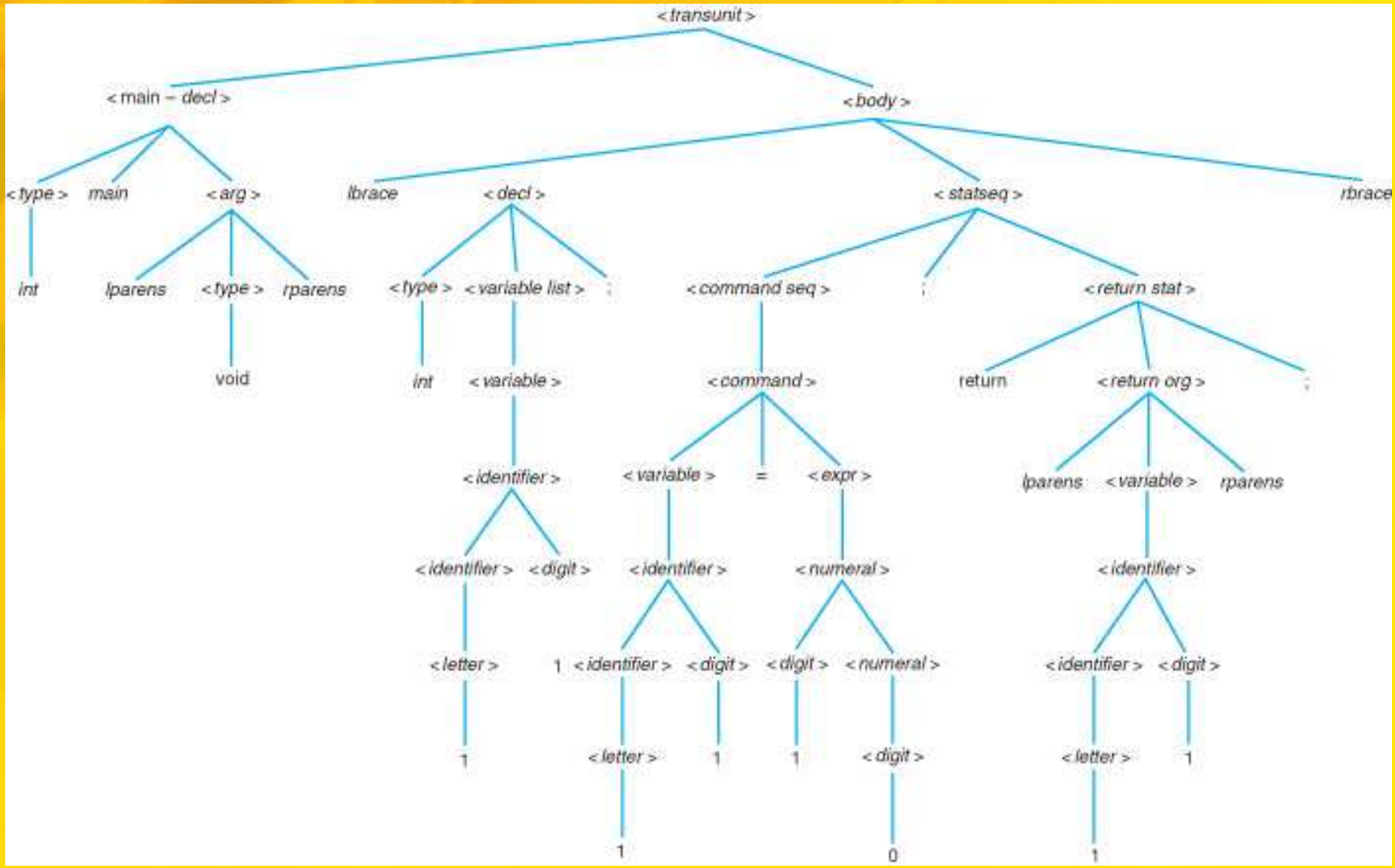


Figure 2: Derivation Tree for Simple `minic` Program

Concerns in the Development of a Parser for `minic`

We divide the overall syntax or the syntactic specification of a language into two parts:

- The lexical structure, which indicates the constraints on the **tokens**, or lexical units, which define the basic 'words' or (multi-symbol) terminals of the language. This includes reserved words and symbols (e.g., `void`, `while`, `if`, {, }, `;`, etc) as well as user-defined words such as variable and function names.
- The (remaining) syntactic structure, which is then input to the grammatical recognizer, or **parser**.

2 Essential Processes of Syntactic Analysis

1. **scanning:** To see if we can recognize all terminals in the string (program). Valid identifiers are recognized and converted to 'pseudo terminals'. At this time it is also convenient to convert the source file into **tokens** for subsequent syntactic analysis.

2. **parsing:** To see if the string (list, program) of tokens is derivable according to the (non-lexical) syntax of the language.

In summary, grammatical recognition is often accomplished in two parts: **scanning followed by parsing**. This corresponds to **lexical analysis** followed by **syntactic analysis.**

1. In practice, the parser must determine the extent of the elements which are derived from nonterminals. This is not simple, given a complex grammar.

2. The parser must find a use for all of string x, i.e., it cannot simply identify parts of the string with some structure and discard the rest. Thus, in a sense, the string must be 'consumed' in the parse.

Specification of the Parsing Problem

Given a string of terminals comprising a sentence x, and a grammar G, specified as:

$$G = (V_T, V_N, P, S)$$

- The process of creating the interior of the parse tree of productions which links S to x is called a parse.
- If we are successful, we have determined that x is a member of $L(G)$.
- If we fill the interior of the tree from the top down (i.e., from the root of the tree), a *top-down parse* results.
- If we work from the bottom (x) up, that is, begin with the terminal symbols, a *bottom-up parse* results.

The Cocke-Younger-Kasami (CYK) Parsing Algorithm

The CYK algorithm is a parsing approach which will parse string x in a number of steps proportional to $|x|^3$. The CYK algorithm requires the CFG be in Chomsky Normal Form (CNF). With this restriction, the derivation of any string involves a series of binary decisions.

In Chomsky Normal Form (CNF), each production of G must be in the form of either

$$A \rightarrow BC$$

or

$$A \rightarrow a$$

The CYK Parse Table

Given string $x = x_1, x_2, \ldots x_n$, where $x_i \in V_T$, $|x| = n$, and a grammar, G, we form a triangular table with entries t_{ij} indexed by i and j where $1 \leq i \leq n$ and $1 \leq j \leq (n - i + 1)$. The origin is at $i = j = 1$, and entry t_{11} is the lower left hand entry in the table. t_{1n} is the uppermost entry in the table. This structure is shown in Figure 3, for the case of $n = 4$.

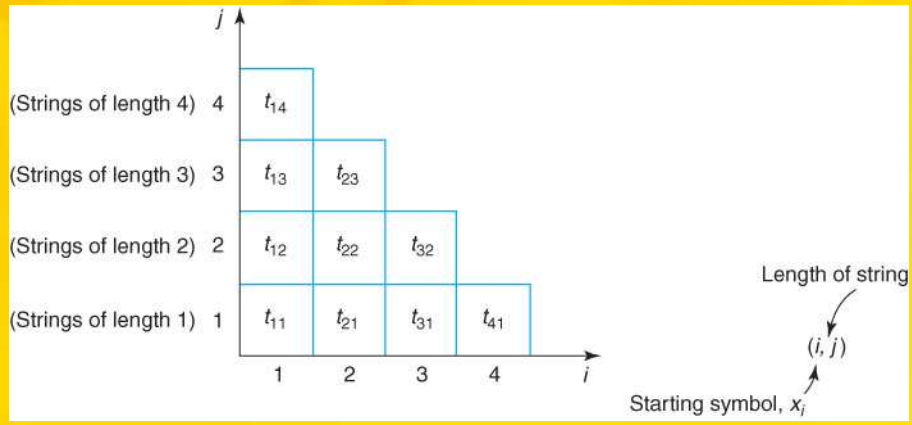


Figure 3: Basic Parse Table Structure for CYK Algorithm

Forming the CYK Table

- The CYK parse table is built, starting from location $(1, 1)$.
- *If a substring of x, beginning with x_i, and of length j can be derived from a nonterminal, this nonterminal is placed into cell (i, j).*
- If cell $(1, n)$ contains S, the table contains a valid derivation of x in $L(G)$.
- It is convenient to list the x_i, starting with $i = 1$, under the bottom row of the table.

Example of the CYK Approach

Sample Grammar Productions

$$S \rightarrow AB|BB$$

$$A \rightarrow CC|AB|a$$

$$B \rightarrow BB|CA|b$$

$$C \rightarrow BA|AA|b$$

We explore the parse (derivation) of string $x = aabb$ using the CYK approach.

Notice the grammar productions are already in CNF.

Resulting CYK Parse Table

Construction of the parse table for this example is shown in Figure 4(a).

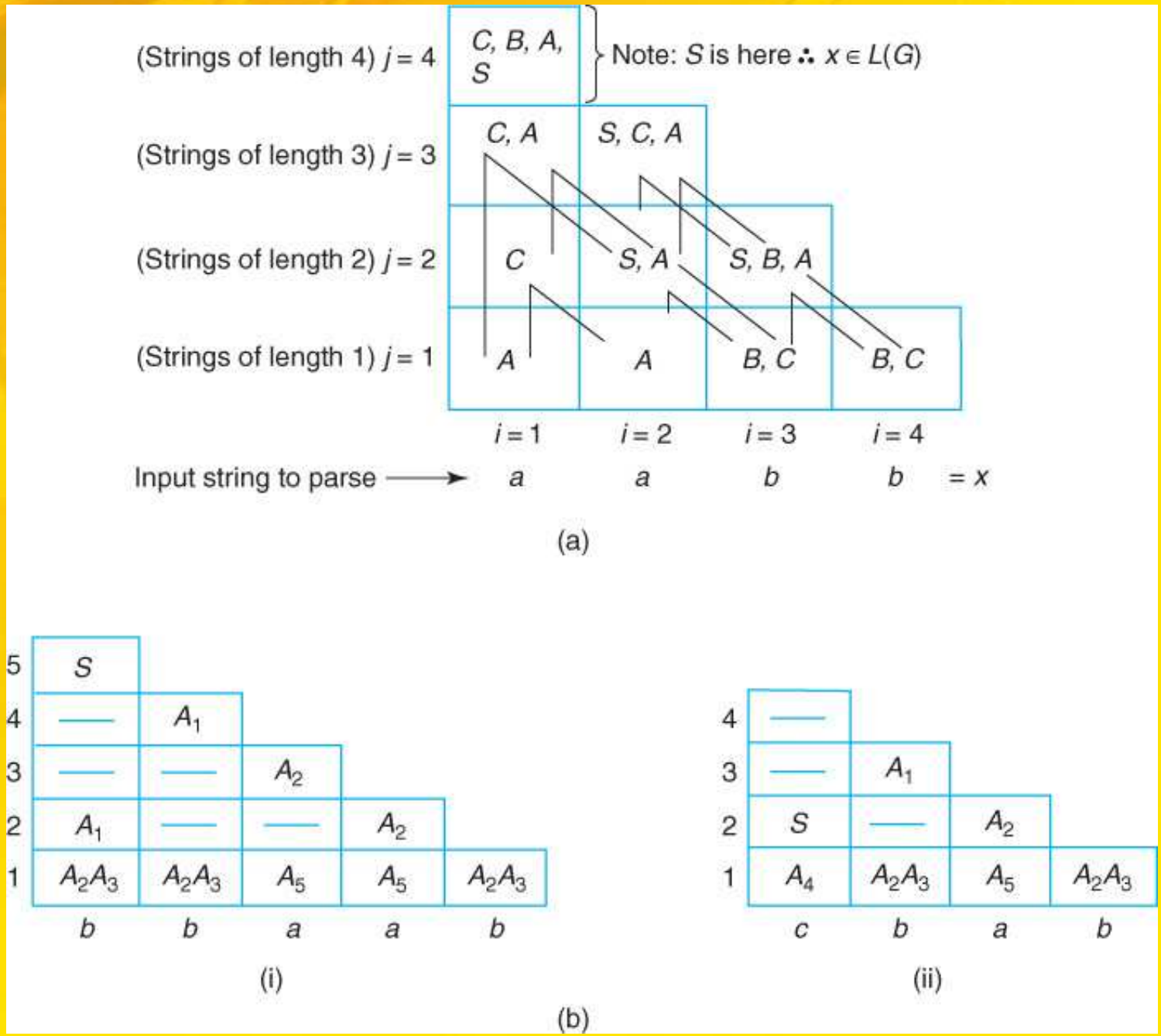


Figure 4: Sample Executions of the Parse Tables for Test Examples

Another Example of CYK of Parsing Suppose we are given the following finite state grammar (FSG):

$$S \rightarrow bA_1|cA_2$$

$$A_1 \rightarrow bA_2$$

$$A_2 \rightarrow b|aA_2$$

Converting to CNF yields

$$S \rightarrow A_3A_1$$

$$A_3 \rightarrow b$$

$$S \rightarrow A_4A_2$$

$$A_4 \rightarrow c$$

$$A_1 \rightarrow A_3 A_2$$

$$A_2 \rightarrow b$$

$$A_2 \rightarrow A_5 A_2$$

$$A_5 \rightarrow a$$

Parse tables for strings $x = bbaab$ and $x = cbab$ are shown in Figure 4(b), parts (i) and (ii) respectively.

Note that the existence of an empty cell (other than $(1, n)$) does not necessarily lead to a failure to parse.

Other Ways to (Scan and) Parse

- Using `flex` and `bison`
- Using Prolog and the LGN