Slides to Accompany *Programming Languages and Methodologies*

R. J. Schalkoff

Chapter 1, Part 1: Introduction

Definition #1

- A programming language is a notational system intended primarily to facilitate human-machine interaction.
- The notation is both human and machine-readable.

Definition #2

From a linguistic viewpoint, a (programming) language has a *syntax*, and language elements have *semantics*.

Definition #3

- A program is something that is produced by a programming language.
- A program is a structured entity with semantics.

Formal Approaches

- A formal and quantitative characterization of a programming language is based upon a formal language which itself is based upon a formal grammar.
- Denote this grammar G.
- This viewpoint allows us to alternately and quantitatively define a program as a *sentence* or *string* produced by G, and a programming language as the set of all strings (programs) producible by G.

Choices, Choices, and Choices

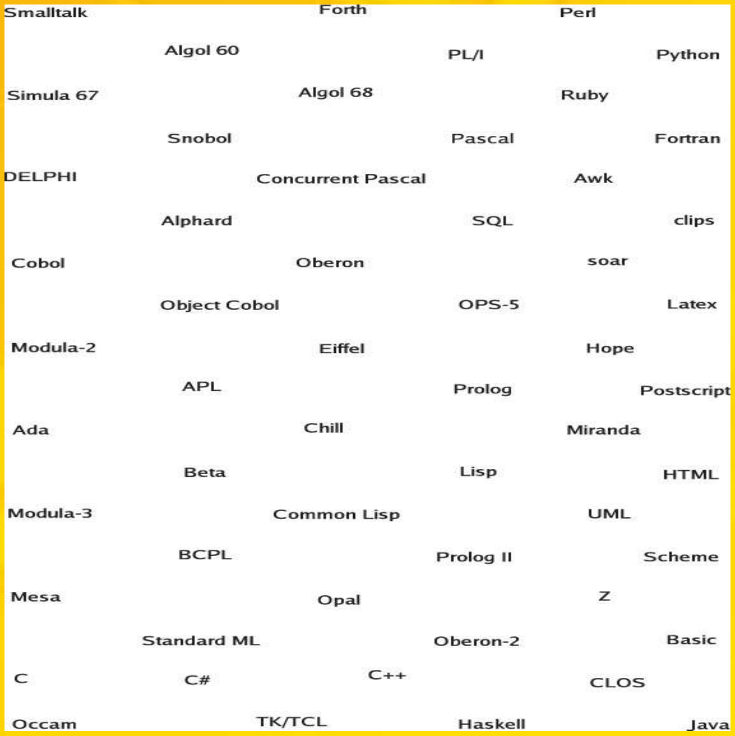


Figure 1: A Portion of the 'Sea of Languages'

Software and Moore's Law

Moore's law, stated in 1965 by Gordon Moore, postulates a doubling in computer hardware performance (measured by component density or gates on a chip) roughly every 18 months.

This translates into a factor of 100 every 10 years. While Moore's law is not a law of physics, it has been reasonably accurate in predicting combined computer processing, storage and communication capabilities for several decades.

There does not appear to be a corollary to Moore's law for software.

From Gears to Software Objects

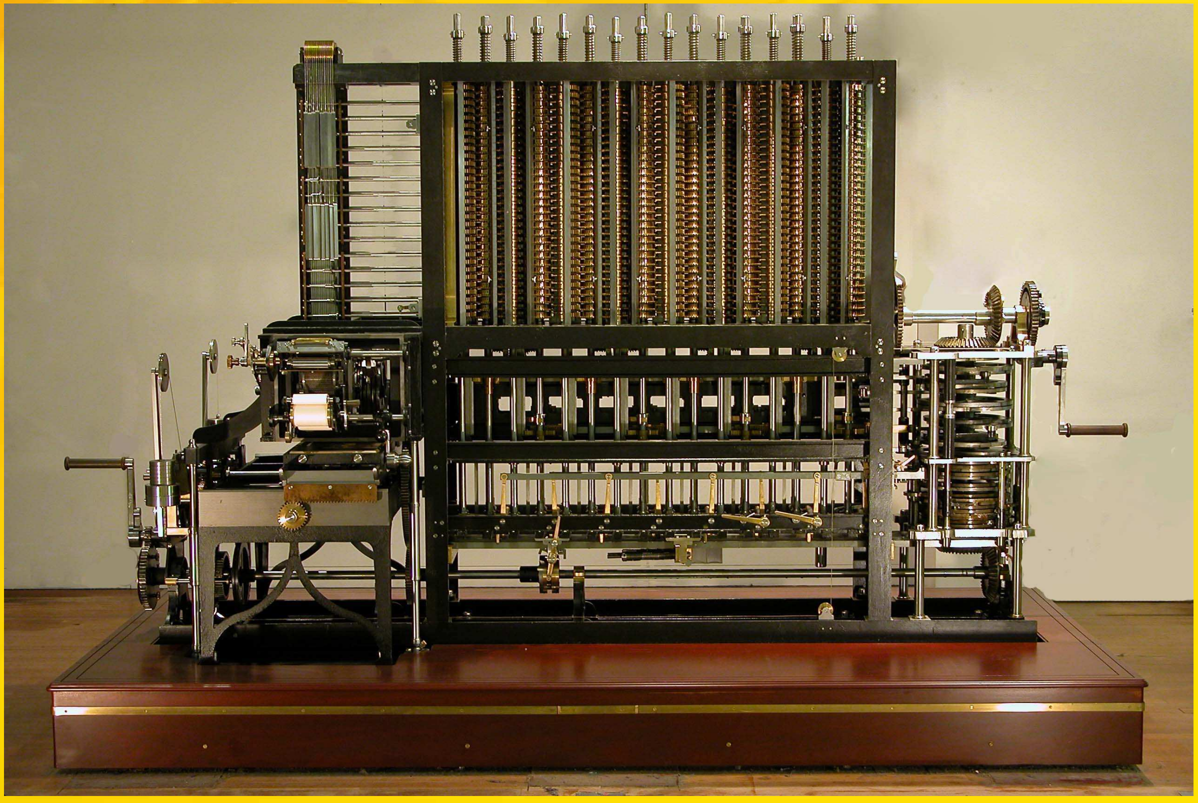


Figure 2: The Mechanical 'Difference Engine' Computer. (And you thought software development in linux was difficult?). Courtesy Doran Swade.

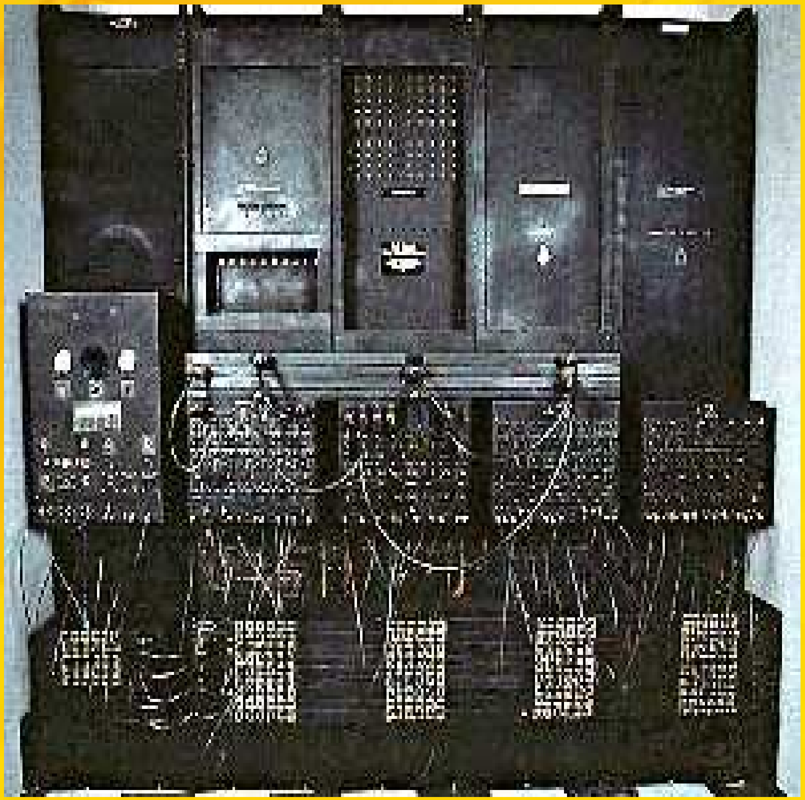


Figure 3: The ENIAC Computer, circa. 1945. ENIAC contained 20 electronic accumulators, each capable of storing a 10-digit decimal number and had a read-only memory of about 300 numbers.

hardware	software
gears	changing gears
relays/vacuum tubes	switches, cables, machine code
discrete transistors	assemblers
LSI	higher-level dev. systems
VLSI	paradigms chosen by *application*

Figure 4: Language Generations Parallel Hardware Evolution

A Point of Departure

Today, however, the situation is one of significantly greater independence. Computing hardware generally supports a number of operating systems and development tools, including language interpreters and compilers for programming languages.

- To a great extent, language choice is independent of hardware.
- A possible point of view: **software, not hardware, is holding back advances in computing.**

Language Chronology

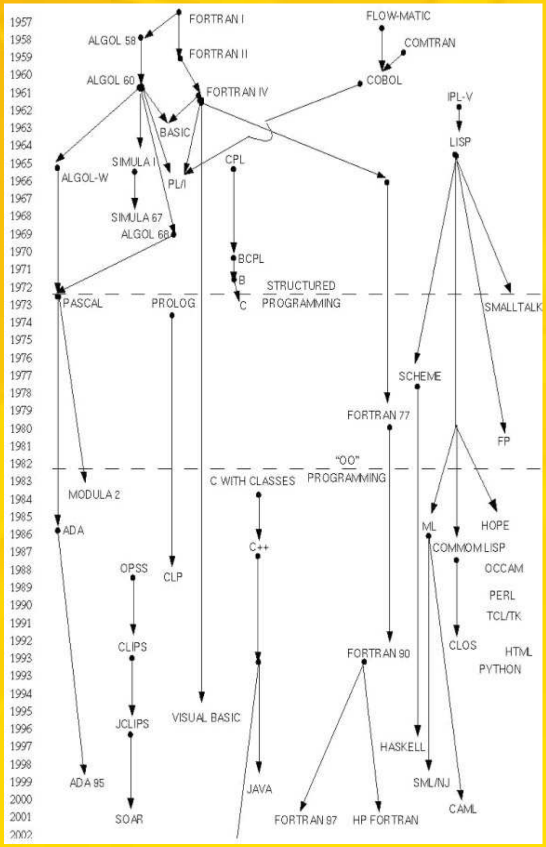


Figure 5: Programming Language 'Evolution'

Programming Languages by *Paradigm*

A wide variety of programming paradigms exist. Examples are:

1. Procedural or imperative (probably the best-known; found in languages such as c and Java)
2. Functional (or applicative)
3. Declarative
4. Object-oriented (allows complex systems to be modeled as modular components (which can be easily reused to model other systems or to create new components))
5. Rule-based
6. Event-driven
7. Parallel or concurrent

Syntax

Programming language syntax defines the allowable arrangement of symbols in programs, especially program fragments. For example, the syntax may constrain fragments to have matching parentheses, e.g.,

```
<main-decl> ::= <type> main <arg>
<type> ::= int | void
<arg> ::= lparens <type> rparens
```

Syntax also catalogs the basic elements of the language, most importantly the **structure** of the language.

Syntax is often indicated by a metalanguage, i.e., a language used to quantify another language. Examples are the language of regular expressions and BNF.

Reading Code Is Good for You

- Professional software developers spend a significant fraction of their time not in actually writing code, but instead in reading, understanding, and modifying existing code.
- An emerging notion is that learning how to produce good code is based upon reading good code.
- Lots of Open Source code corresponding to popular programs such as web browsers, graphics file manipulation programs, language interpreters/compilers, and even entire operating systems is available for reading and review.
- A key element for the success of this approach is the ability to distinguish good programming practices from 'not-so-good'.

Slides to Accompany *Programming Languages and Methodologies*

R. J. Schalkoff

Chapter 2, Part 1: Formal Grammars

Programs are Strings

Consider the following code fragment:

```
void main(void)
  {
    char *message[] = {"Hello ", "World"};
    int i;

    for(i = 0; i < 2; ++i)
      printf("%s", message[i]);
    printf("\n");
}
```

The code might as well appear as follows:

void main(void) { char *message[] = {"Hello ", "World"}; int i; for(i = 0; i < 2; ++i) printf("%s", message[i]); printf("\n"); }

Another Example

Consider the Lisp function definition:

```
(defun hello
   (print
     (cons 'Hello (list 'World))))
```

Again, as the compiler sees it:

(defun hello (print (cons 'Hello (list 'World))))

The Point(s)

- The source code (i.e., your program) is merely a sequence of symbols.
- This sequence (string) is characterized via formal grammars.
- The programming language which allows production of these strings exists to facilitate communication between the programmer and the compiler.

Formal Grammars

- Grammars generate languages
- The syntax of a programming language may be described through specification of a grammar.
- There are many different types of grammars (later).

Language Syntax Specification and Productions

- The syntax of a programming language may basically be written using a set of reserved words (primitives) and rules (productions) for combining these reserved words to form programs.
- For example, a part of the syntax of Pascal may be shown in BNF:

$$function-identifier ::= identifier$$

$$identifier ::= letter\{letter-or-digit\}$$

$$letter ::= a|b|c|\ldots|z|\ldots|A|\ldots|Z$$

$$digit ::= 0|1|2|3|4|\ldots|9$$

$$letter-or \; - \; digit ::= letter|digit$$

::= means 'is defined as'

| means 'or'

{} indicates items which may be repeated zero or more times

What About a Programming Language Compiler?

- Part of a compiler is a *grammatical recognizer* or *parser*.
- A compiler or interpreter for a (higher-level) language attempts to generate lower level machine instructions by determining the desired structure of the input high level language program through *parsing* the input or source code.
- The compiler/interpreter begins with processing of the program as an input string.
- This processing is based upon checking if the string meets a formal specification of the programming language syntax.

Slides to Accompany *Programming Languages and Methodologies*

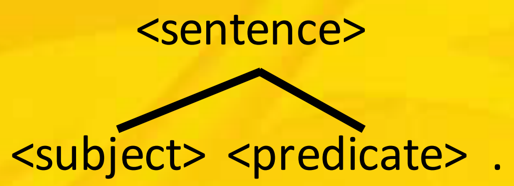
R. J. Schalkoff

Chapter 2, Part 2: Formal Grammars

What are grammars?

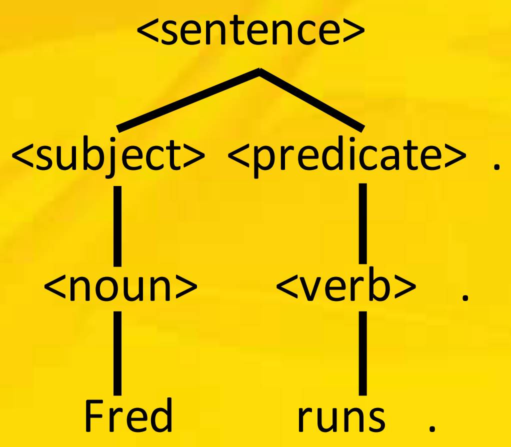
In natural language grammars define the syntactic structure of a sentence.

Fred runs.



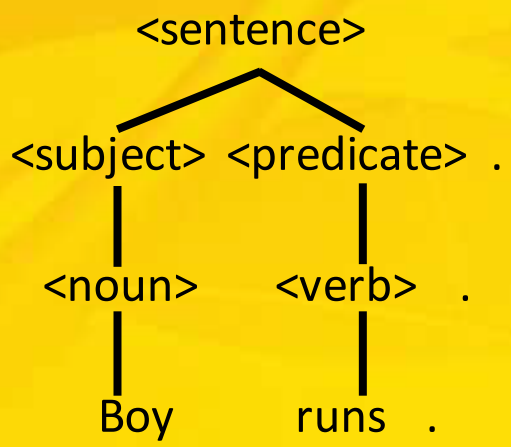
What are grammars?

In natural language grammars define the syntactic structure of a sentence.



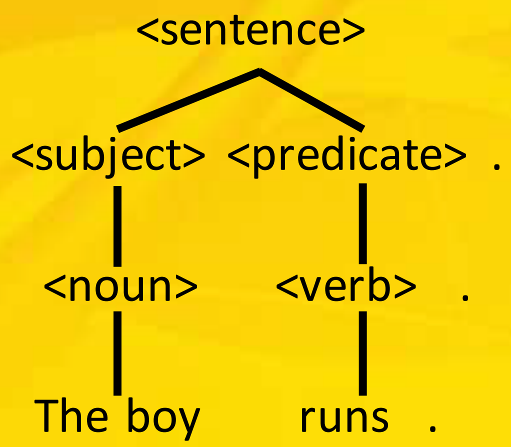
What are grammars?

In natural language grammars define the syntactic structure of a sentence.



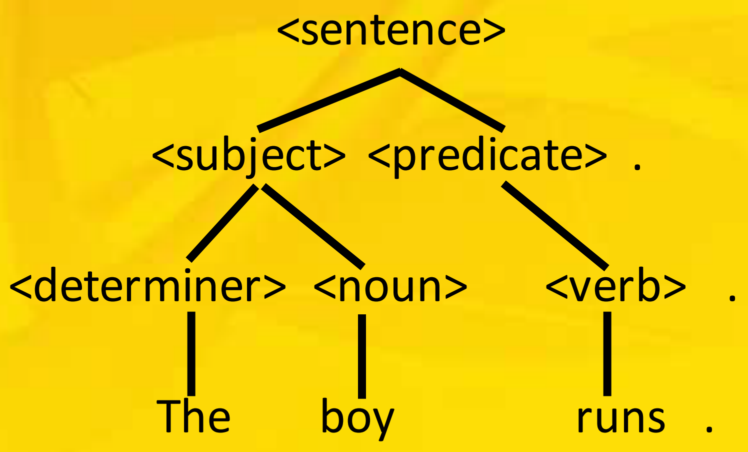
What are grammars?

In natural language grammars define the syntactic structure of a sentence.



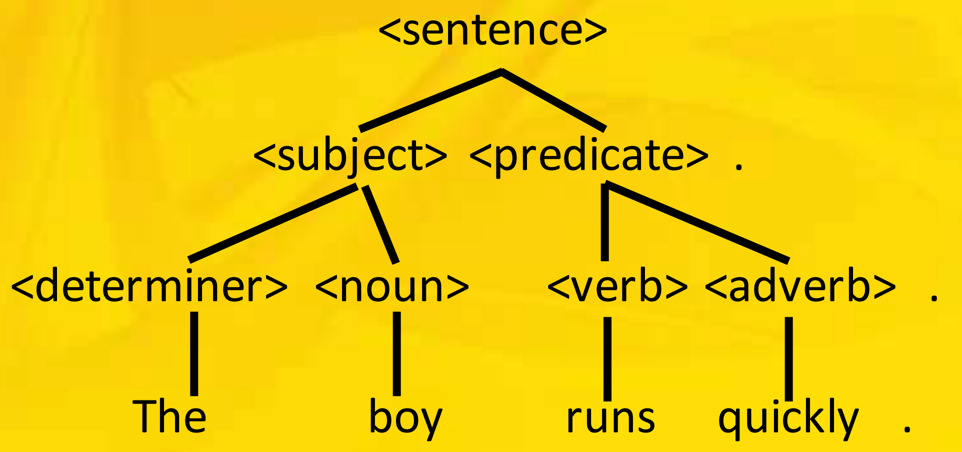
What are grammars?

In natural language grammars define the syntactic structure of a sentence.



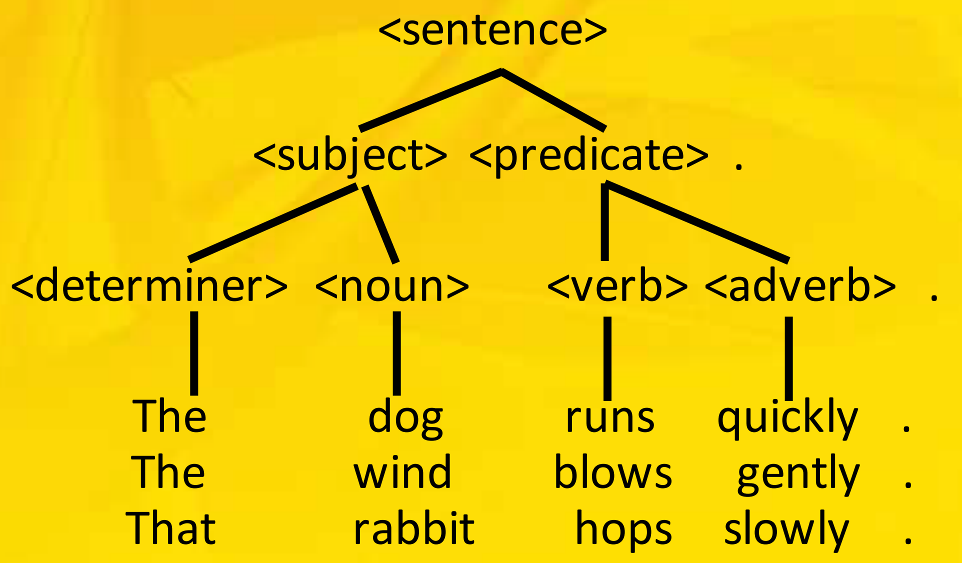
What are grammars?

In natural language grammars define the syntactic structure of a sentence.



What are grammars?

In natural language grammars define the syntactic structure of a sentence.



Alphabets

- In programming languages, as in natural language, we form sentences from words and symbols
- In language theory we don’t deal with words, only symbols.
- An alphabet is the list of symbols that can be used to form words and sentences.
- Some languages (natural, programming) have an infinite set of possible words.
- In language theory we will only work with finite sets of symbols.

Example: Alphabet V is a set of 4 symbols: V={a, b, c, d}

- Languages are sets of strings from V and can be finite or infinite

Example: Finite language L1(V) is a set of 5 strings: L={a, ab, aab, aaabc, aaaabcd}
Infinite langauge L2(V) consists of all strings from V where all a’s come before b’s and b’s before c’s and c’s before d’s. Note that L1 is a subset of L2.

An Alphabet and Forming Strings

- An alphabet (V) is a nite, nonempty set of *symbols*, e.g.:
  V = {a, b, c, …, z}
- The concatenation of a and b, denoted a∘b, produces a sequence of two symbols simply denoted hereafter as *ab*.
- A *string* over V is either a single symbol from V or a sequence of symbols formed by concatenation of zero or more symbols from V . Therefore, from V above, '*a*' '*ab*' '*az*' and '*azab*' are strings or *sentences* over V .
- The length of (or number of symbols in) string s is denoted |*s*|.

- A string has a natural ordering of elements from left to right.
- Often it is convenient to denote a string like $x = aaa \dots a$, where a symbol (or sequence of symbols) is repeated n times as $x = a^n$. For example:

$$aabbbcccc = a^2b^3c^4$$

The Closure Set

Define V^+ as

$$V^+ = V \cup V^2 \cup V^3 \cup \ldots$$

V^+ is the set of all nonempty sentences producible using V. Adding the empty string to V^+ produces V^*, i.e.,

$$V^* = \{\epsilon\} \cup V^+$$

V^* is denoted the *closure (set)* of V and $V^+ = V^* - \{\epsilon\}$ is often called the *positive closure* of V.

Languages Using Strings

Grammars are used with V to give some meaning to a subset of strings, $L \subseteq V^*$.

L is called a *language*. Furthermore:

- Languages are generated by grammars.
- Another viewpoint is that a grammar restricts the production of strings from V.
- Grammars are used to recognize (parse) elements of a language.

Grammar: Formal Definition

A *grammar*

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

consists of the following four entities:

1. A set of terminal or primitive symbols (primitives), denoted V_T (or, alternately, Σ). These are the elemental 'building blocks' of the grammar (and corresponding language).

2. A set of *non-terminal symbols, or variables*, which are used as intermediate quantities in the generation of an outcome consisting solely of terminal symbols. This set is denoted as V_N (or, alternately, N).

   Note that V_T and V_N are disjoint sets, i.e., $V_T \cap V_N = \emptyset$.

3. A set of *productions, or production rules or rewriting rules*

which govern how strings may be formed. It is this set of productions, coupled with the terminal symbols, which principally gives the grammar its 'structure.' The set of productions is denoted P.

4. A starting (or root) symbol, denoted S. $S \in V_N$.

Grammar Application Modes

A grammar may be used in one of two modes:

1. *Generative*: The grammar is used to create a string of terminal symbols using P; a *sentence* in the language of the grammar is thus generated.

2. *Analytic*: Given a sentence (possibly in the language of the grammar), together with specification of G, one seeks to determine:

   (a) If the sentence was generated by G; and, if so,

   (b) The structure (usually characterized as the sequence of productions used) of the sentence. This is where we begin to consider semantics.

Languages, Possible Strings and $L(G)$

- Any subset $L \subseteq V_T^*$ is a *language*.
- If $|L|$ is finite, the language is called finite, otherwise it is infinite.
- The *language generated by grammar* G, denoted $L(G)$, is the set of all strings which satisfy
  1. Each string consists solely of terminal symbols from V_T of G; and
  2. Each string was produced from S using P of G.

Grammar Types/Definitions

1. Symbols beginning with a capital letter (e.g., S_1 or S) are elements of V_N.
2. Symbols beginning with a lowercase letter (e.g., a or b) are elements of V_T.
3. n denotes the length of string s, i.e.,

$$n = |s|$$

4. Greek letters (e.g., α, β) represent (possibly empty) strings, typically comprised of terminals and/or nonterminals.

Grammar Types (Chomsky)

- T0 – All grammars, All Touring computable languages
- T1 – Context Sensitive Grammars (CSG)
- T2 – Context Free Grammars (CFG), push-down automata
- T3 – Regular Grammars, finite-state automata

T_1: Context-Sensitive (CS)

A T_1 or context sensitive grammar restricts productions to:

$$\alpha\alpha_i\beta \rightarrow \alpha\beta_i\beta$$

meaning β_i replaces α_i *in the context of* α *and* β, where α, $\beta \in (V_N \cup V_T)^*, \alpha_i \in V_N$ and $\beta_i \in (V_N \cup V_T)^* - \{\epsilon\}$. Note that α or β (or both) may equal ϵ.

T_2:Context Free (CFG)

In a T_2 or context-free grammar, the production restrictions are:

$$\alpha_1 = S_1 \in V_N$$

(α_1 *must be a single nonterminal*)

$$|S_1| \leq |\beta_2|$$

An alternate characterization is:

$$S_1 \longrightarrow \beta_2$$

where $\beta_2 \in (V_N \cup V_T)^+$. Note a CFG production allows S_1 to be replaced by string β_2 *independently or irrespective of the context in which* S_1 *appears.*

Examples

- T3 (a*b*)
  - S ::= aS
  - S ::= B
  - B ::= bB
  - B ::= <e>
- T2 ($a^n b^n$)
  - S ::= aSb
  - S ::= <e>
- T1 ($a^n b^n c^n$)

Examples

- T3 (a*b*)

  S ::= aS

  S ::= B

  B ::= bB

  B ::= <e>

S -> **aS** -> a**aS** -> aa**aS** -> aaa**B** -> aaa**bB** -> aaab**bB** -> aaabb

- T2 (a^nb^n)

  S ::= aSb

  S ::= <e>

S -> **aSb** -> a**aSb**b -> aa**aSb**bb -> aaa**aSb**bbb-> aaaabbbb

- T1 ($a^nb^nc^n$)

Why are They Important?

Context free grammars and the class of context-free languages they generate are paramount in the study of programming languages, since they are reasonably adequate for describing the syntax of many programming languages. It is also worth noting:

- Context-free grammars are important because they are the most descriptively versatile grammars for which effective (and efficient) parsers are available.

- *The production restrictions* <u>*increase*</u> *in going from context-sensitive* (T_1) *to context free* (T_2) *cases.*

A Sample G_s: Type and $L(G_s)$

$$S \rightarrow AB$$
$$S \rightarrow C$$
$$A \rightarrow C$$
$$A \rightarrow a$$
$$B \rightarrow b$$
$$B \rightarrow c$$
$$C \rightarrow d$$

In BNF:

```
<s> ::= <a><b> | <c>
<a> ::= <c> | a
<b> ::= b | c
<c> ::= d
```

S is the starting symbol and V_T and V_N may be deduced from the productions.

Regular (T3) Languages

- Type T3 Languages are known as “Regular Languages” and represent an important class in computer systems.
- Regular languages are exactly those that can be expressed using so-called “regular expressions” that are used for:
  - Lexical analysis (scanning) in compilers,
  - Pattern matching in a host of tools such as awk, sed, vi, grep, and others,
  - To simplify most command languages such as the unix shell (bash, etc.).
- Regular languages are exactly those that can be recognized by “Finite State Automata” or finite state *machines*.

Regular Expressions

- Regular expressions are a shorthand notation for describing the strings from regular languages, and thus the languages themselves.
- Regular expressions consist of the following constructs:
  - *Atoms*: A symbol from the alphabet is a RE: **a**
    - Represents matching the symbol (includes empty string)
  - *Concatenation*: two RE adjacent to one another is a RE: **ab**
    - Represents one RE followed by the other RE
  - *Alternation*: two RE separated by a vertical line is a RE: **a|b**
    - Represents selection of one RE or the other RE
  - *Kleene Star*: an RE followed by an asterisk is a RE: **a***
    - Represents zero or more copies of the RE
  - *Parenthesis*: a RE in parenthesis is a RE **(a)**
    - Represents grouping and scope for other operators

Example Regular Expresssions

a(bc)* -> a, abc, abcbc, abcbcbc, ...

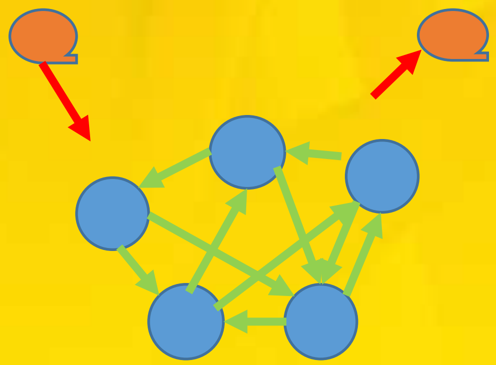
(a|b|c)* -> *e,* abcba, aaaccbb, cba, ... (any string with a, b, and c)

(abc)|(bca)|(cab) -> (just one of those three strings, no repeats)

- There are many notations added to regular expressions by tools like flex and awk designed to make Res even easier to use:
  - a+ : one or more copies of **a**
  - (ab)2,5 : between 2 and 5 copies of **ab**
  - [a-z] : range of characters (the lower case letters) also [A-Z], [0-9], etc.
  - These can be specific to the tool in question.
  - We will learn more when we get to flex and bison, but you can look these up online.

Finite State Automata

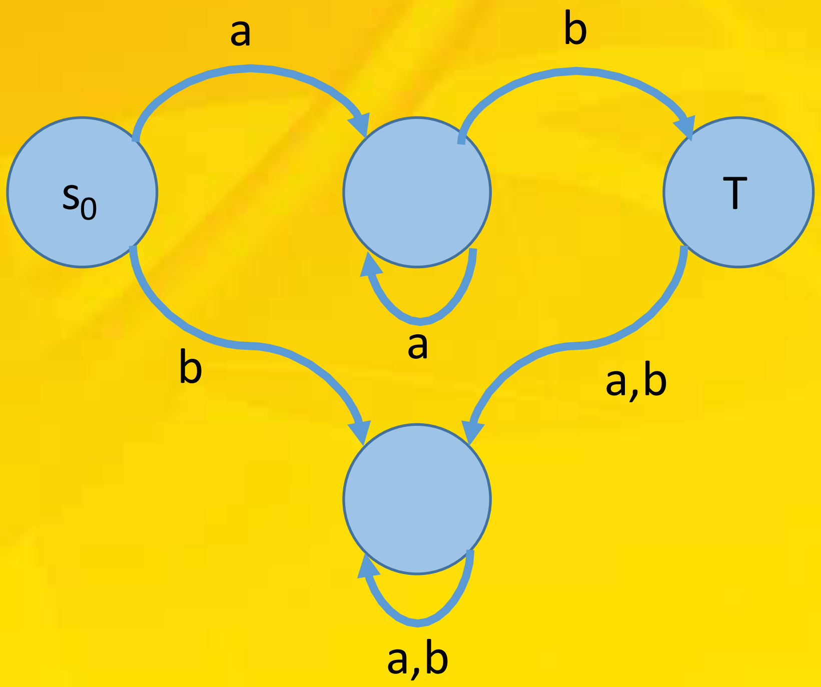
A classic finite state machine has a finite set of states, a set of transitions between states, and input tape and an output tape. The machine reads one symbol (which is subsequently discarded) each cycle from the input tape. The symbol read is used with the current state to decide what state to go to next, and a single symbol is written to the output tape. Neither tape can be read ahead or behind the current position on the tape.



FSA(V, S, s_0, T, F)
V – input alphabet
S – set of states
S_0 – initial state in S
T – Transitions S *X* V -> S
F – set of final states, subset of S

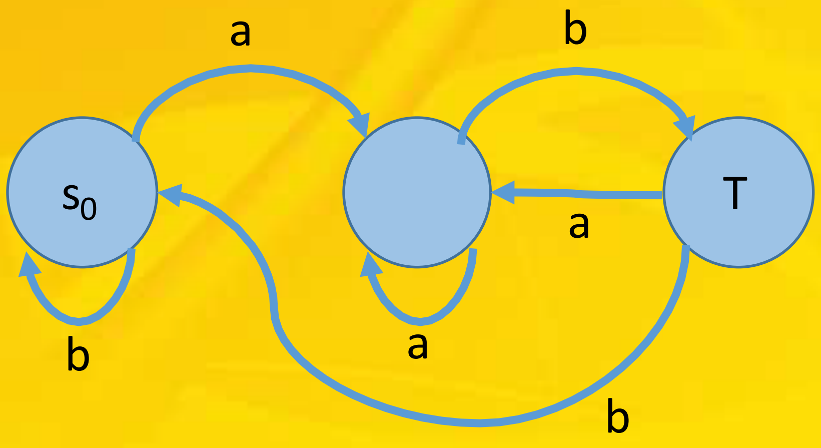
Example FSA

Example language: aa*b or “at least one a, ended with a b”



Example FSA

Example language: aa*b or “at least one a, ended with a b”

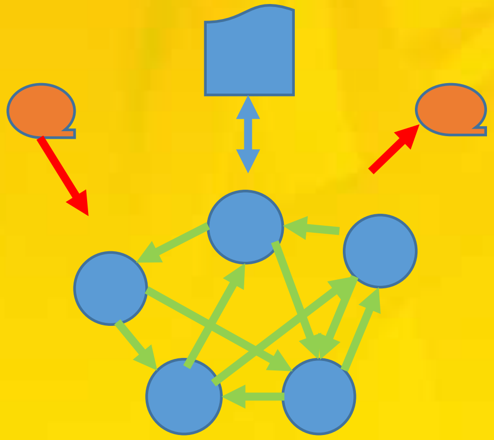

$$\begin{tikzcd}
s_0 \arrow[r, bend left, "a"] \arrow[loop below, "b"] & {} \arrow[r, bend left, "b"] \arrow[loop below, "a"] & T \arrow[l, "a"] \arrow[ll, bend left=60, "b"]
\end{tikzcd}$$

Context Free (T2) Languages

- Type T2 Languages represent an important class in computer systems.
- Context free languages are used for:
  - Parsing in compilers,
  - Pattern matching in a variety of applications including natural language recognition, text analysis, radar signatures, etc.
- Context free languages are exactly those that can be recognized by “Push Down Automata” or FSA with *stacks*.

Push Down Automata

A classic push down automata includes all of the same features as a finite state automata, plus a stack structure. In each cycle, in addition reading, writing, and changing state, the machine can either *push* or *pop* symbols onto or off of the stack. There is not normally any ability to peek at the stack, though in some compiler implementations the ability to peek at the stack and look ahead in the input affords some degree of optimization.



PDA(V, S, s_0, T, F)
V – input alphabet
S – set of states
S_0 – initial state in S
T – Transitions S *X* V -> S
F – set of final states, subset of S

Regular and Context Free Languages

In a later chapter we will learn about flex (lex) and bison (yacc). Flex takes regular expressions that represent the words and symbols of a programming language and converts them to a FSA representation in C. Bison takes context free grammars that represents the syntax of a programming language and converts them to a PDA representation in C. Together, with C code that implements semantics and code generation, they can be used to implement compilers for a wide range of programming languages.

Slides to Accompany *Programming Languages and Methodologies*

R. J. Schalkoff

Chapter 2, Part 3: Formal Grammars

The Derivation (or Parse) Tree

A derivation or parse tree, T, has the following characteristics:

1. The root of T is the starting symbol $S \in V_N$.
2. Leaf nodes of T are terminals $\in V_T$.
3. Interior[a] nodes are nonterminals $\in V_N$.
4. The children of any non-leaf node[b] represent the right hand side (RHS) of some production in P, where the parent node represents the corresponding LHS of the production.

Derivation trees are important when cataloging the production of x, since (except in cases of grammatical ambiguity) they show the structure of x *independently of the possible sequences.*

[a]neither root nor leaf

[b]recall leaf nodes have no children.

Derivation Tree Example

Recall the productions in an earlier grammar:

$$S \rightarrow AB$$

$$S \rightarrow C$$

$$A \rightarrow C$$

$$A \rightarrow a$$

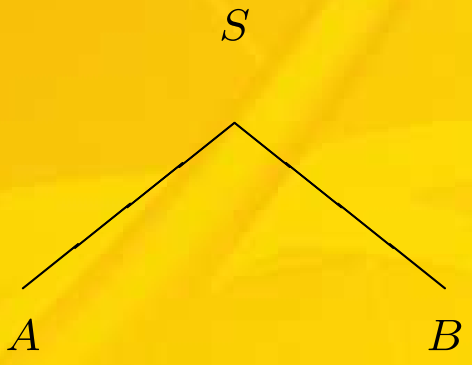
$$B \rightarrow b$$

$$B \rightarrow c$$

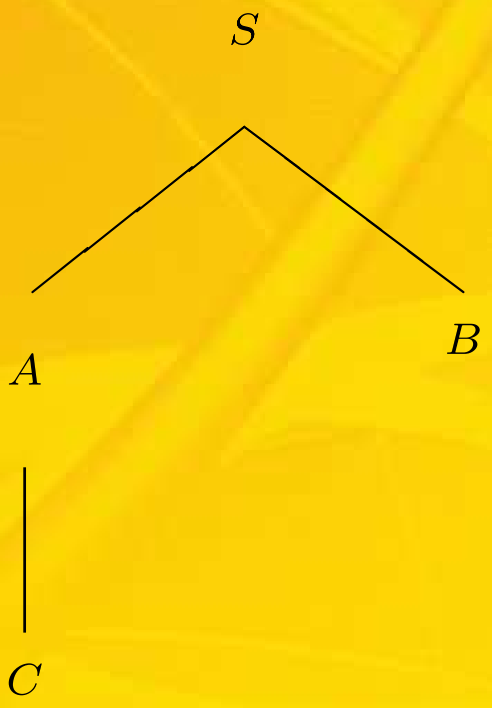
$$C \rightarrow d$$

First, we show the derivation of string dc, uisng the replacement sequence:

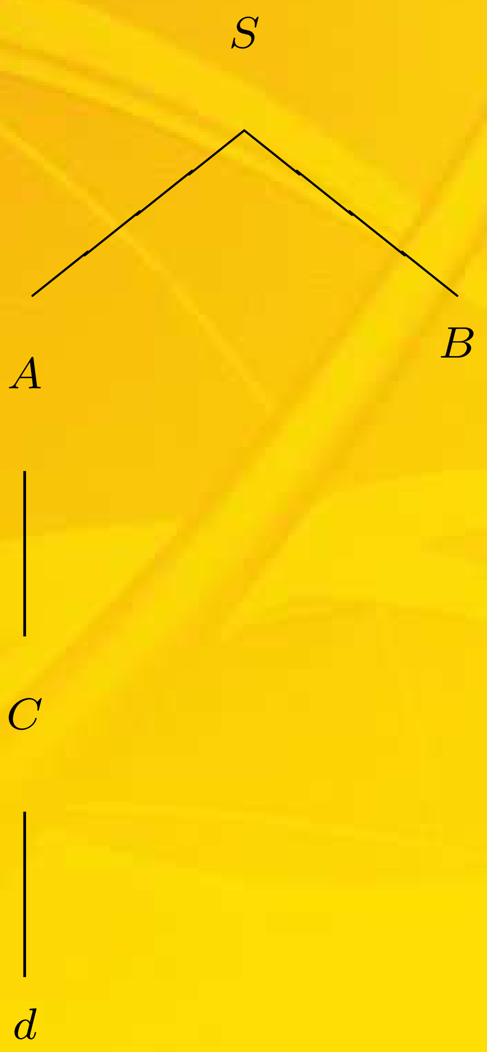
$$\boxed{S \Rightarrow AB \Rightarrow CB \Rightarrow dB \Rightarrow dc}$$



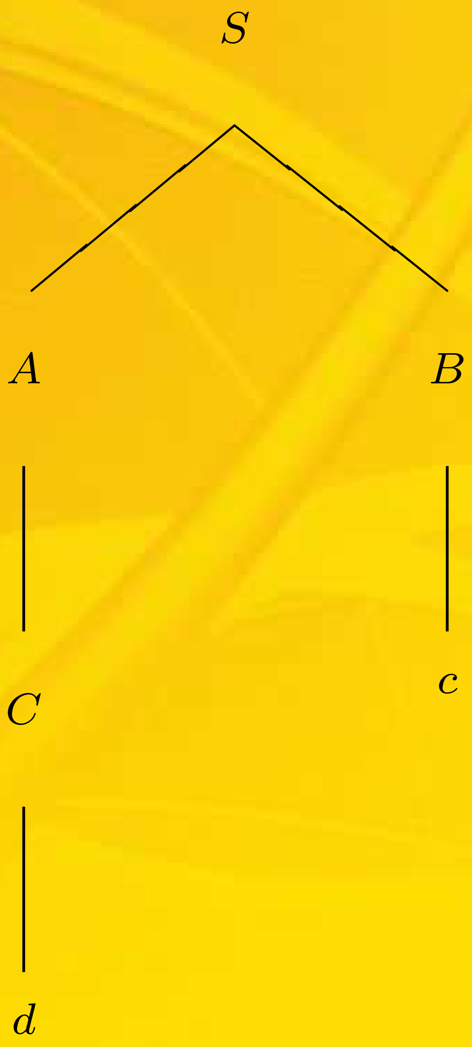
$$S \Rightarrow AB$$



$$AB \Rightarrow CB$$



$$CB \Rightarrow dB$$



$$dB \Rightarrow dc$$

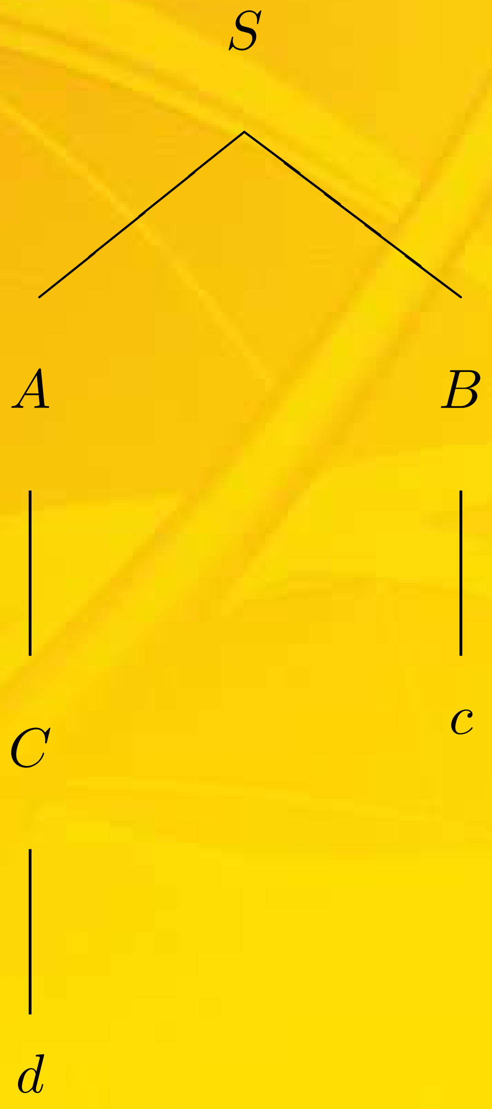
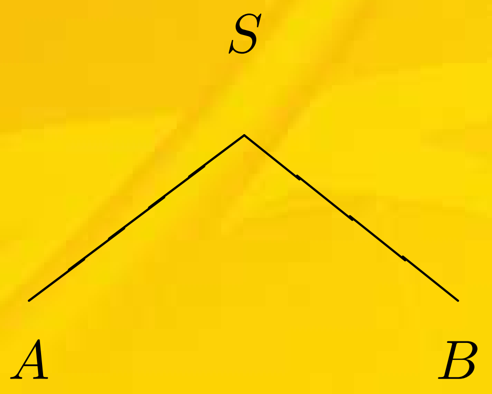


Figure 1: Derivation Tree Using the Production Sequence $S \Rightarrow AB \Rightarrow CB \Rightarrow dB \Rightarrow dc$

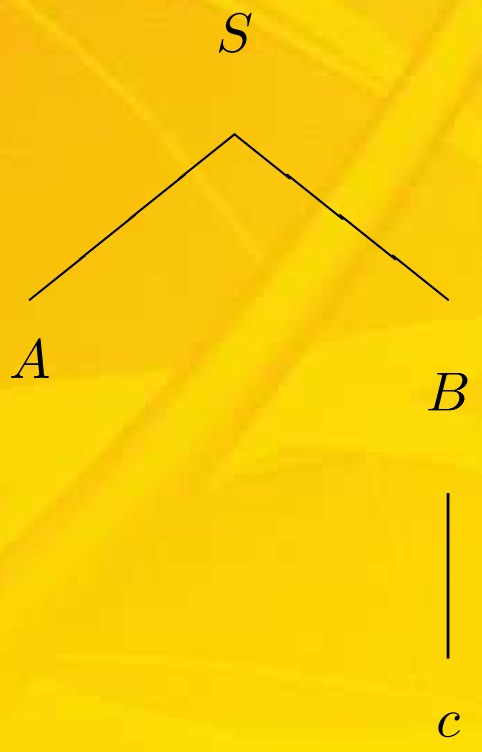
Now consider an alternative derivation *sequence* for the same string:

$$\boxed{S \Rightarrow AB \Rightarrow Ac \Rightarrow Cc \Rightarrow dc}$$

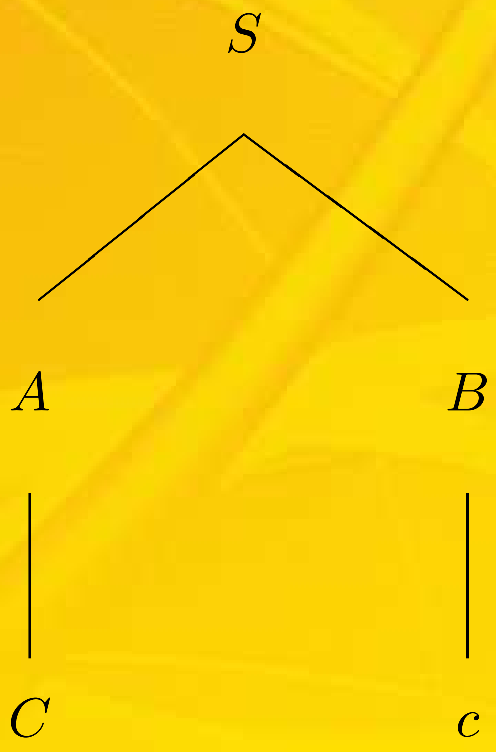
Development of the derivation tree with this sequence proceeds as follows:



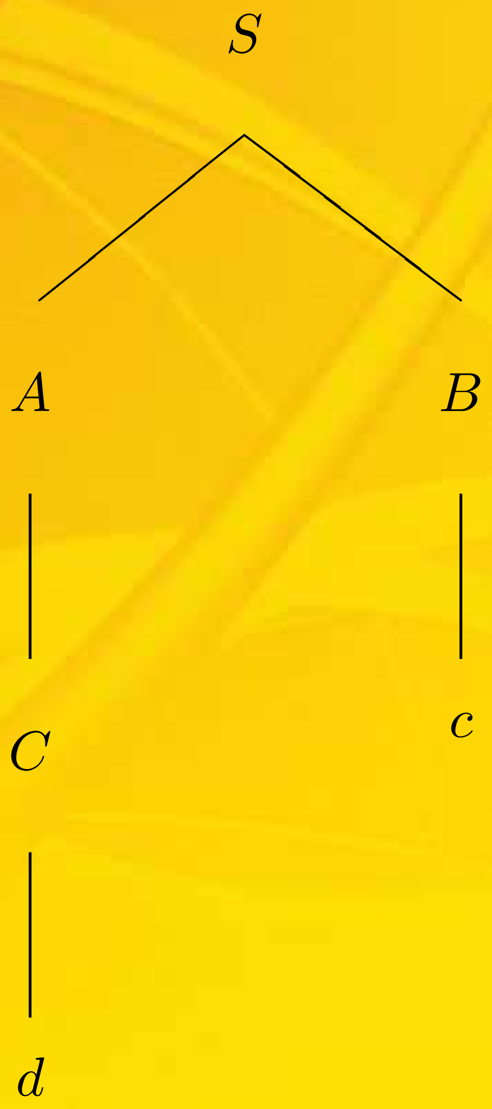
$$S \Rightarrow AB$$



$$AB \Rightarrow Ac$$



$$Ac \Rightarrow Cc$$



$$Cc \Rightarrow dc$$

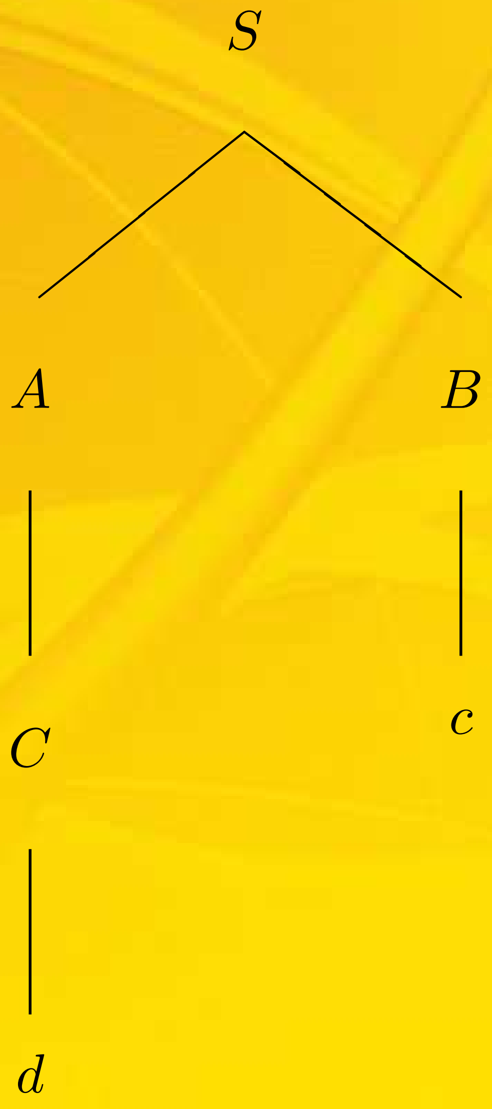


Figure 2: Derivation Tree Using the Production Sequence $S \Rightarrow AB \Rightarrow Ac \Rightarrow Cc \Rightarrow dc$

Grammatical/Syntactic Ambiguity

A grammar is ambiguous if any string in $L(G)$[a] *has two or more distinct derivation trees.*

[a] the language generated by grammar G

Classic Examples of Ambiguity

Classic examples of potential ambiguity in programming languages include:

- Strings employing binary mathematical operators. For example, what is the underlying structure of:

$$a = b + c * a$$

  Of course, specification of operator precedence together with the syntax specification may eliminate this problem.

- The use of nested "if-then-(optional) else" constructs. This is a major source of potential ambiguity, hence the syntax specification is usually augmented with a description of the corresponding interpretation algorithm; and

- Confusion between function invocation and array use. For example, how would (could) you interpret: $a(2)$? Is this function a called with argument 2 or the second (or third) element of array a?

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);
}
```

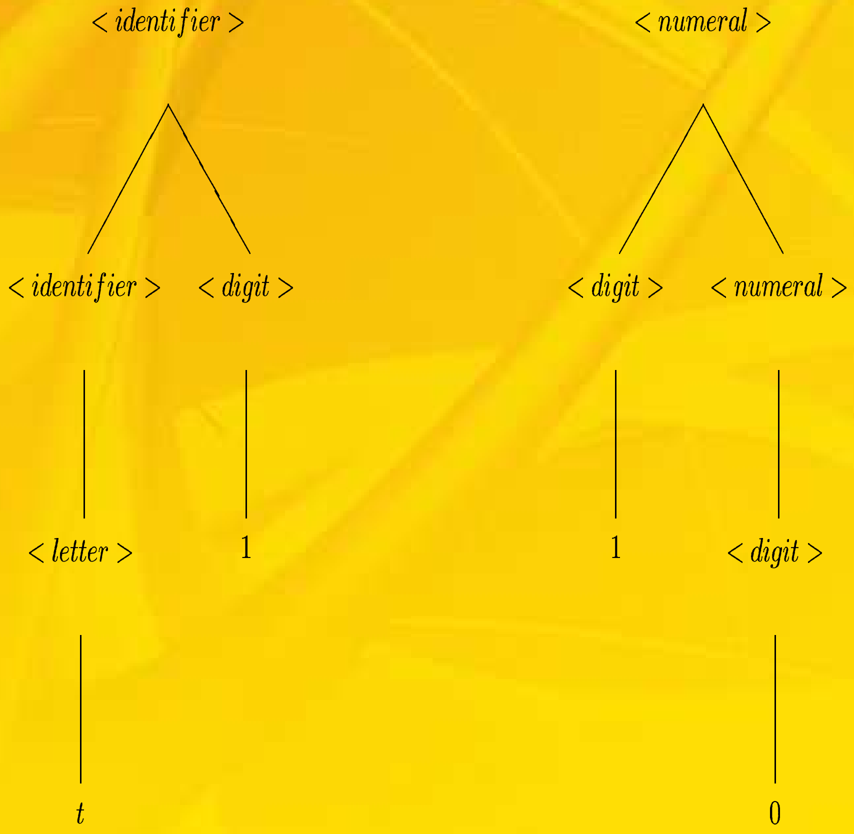
`minic` Syntax Subdivision

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


< identifier >
< identifier >
< digit >
< letter >
1
t
< numeral >
< digit >
< numeral >
1
< digit >
0

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);
}
```

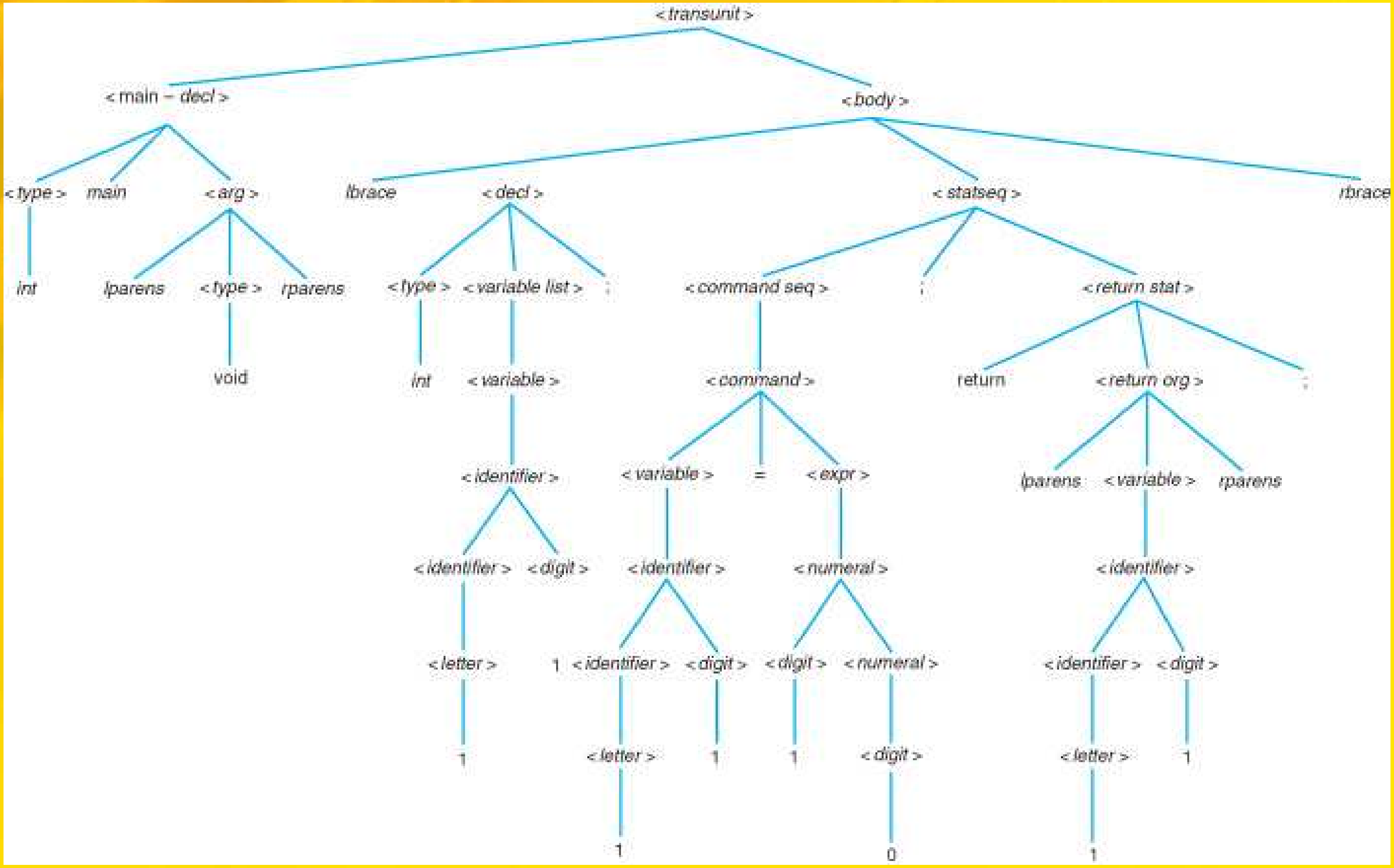


Figure 2: Derivation Tree for Simple `minic` Program

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 $n = 4$.

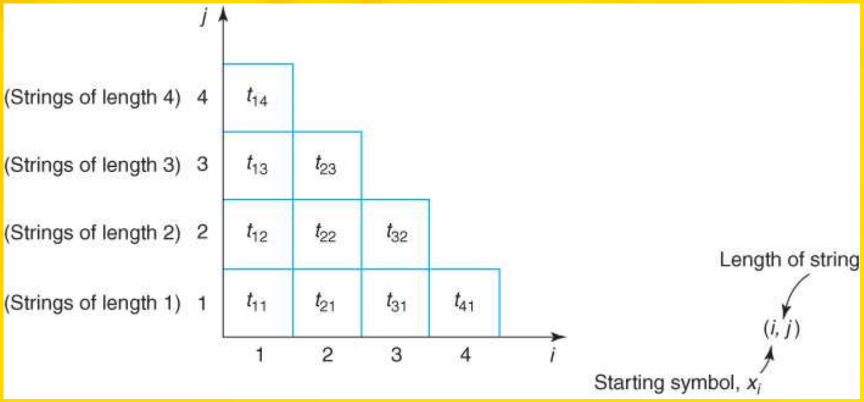


Figure 3: Basic Parse Table Structure for CYK Algorithm

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.

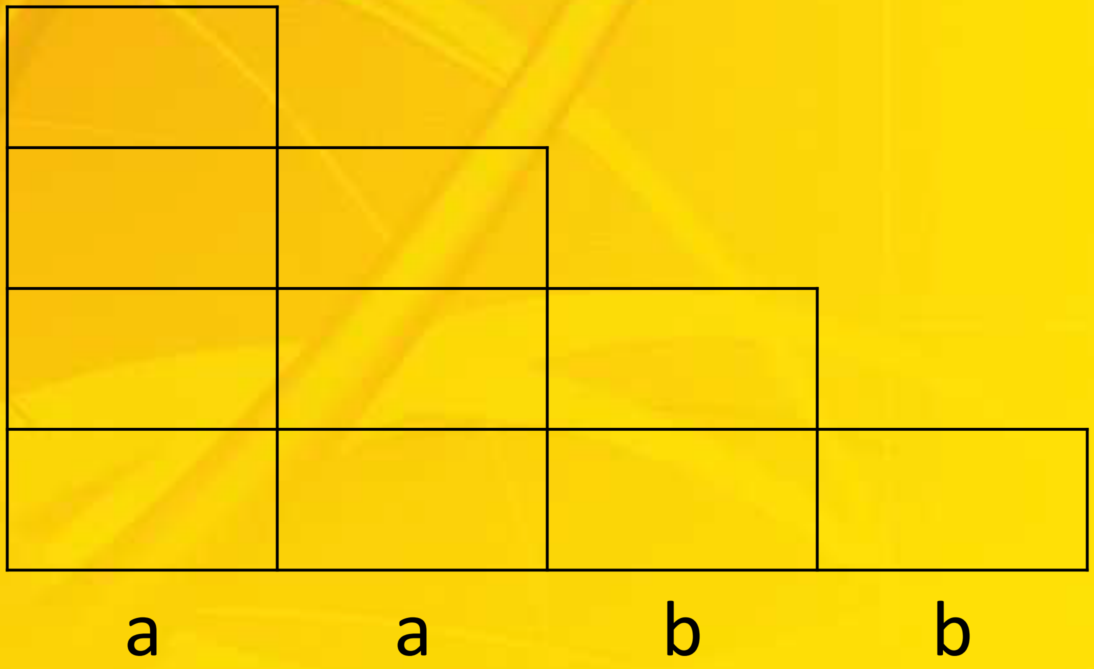
Notice the grammar productions are already in CNF.

Resulting CYK Parse Table

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

CYK Parse of a simple CFG

S → AB | BB
A → CC | AB | a
B → BB | CA | b
C → BA | AA | b



CYK Parse of a simple CFG

S → AB | BB
A → CC | AB | a
B → BB | CA | b
C → BA | AA | b

A	A		
a	a	b	b

CYK Parse of a simple CFG

S → AB | BB
A → CC | AB | a
B → BB | CA | b
C → BA | AA | b

A	A	B	B
a	a	b	b

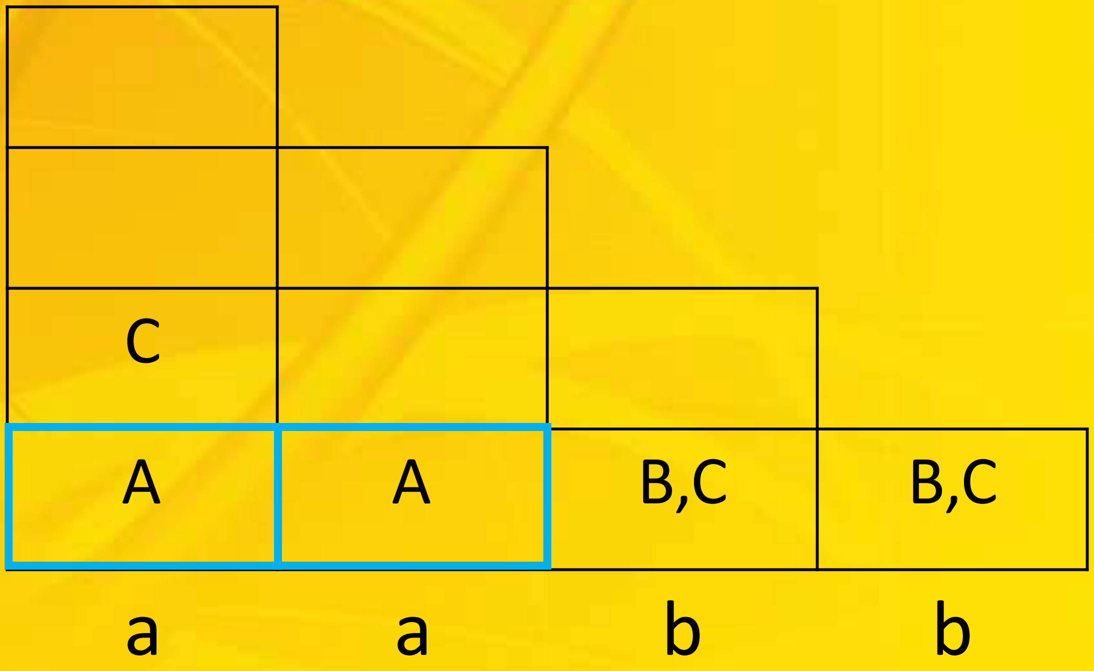
CYK Parse of a simple CFG

S → AB | BB
A → CC | AB | a
B → BB | CA | b
C → BA | AA | b

A	A	B,C	B,C
a	a	b	b

CYK Parse of a simple CFG

S → AB | BB
A → CC | AB | a
B → BB | CA | b
C → BA | AA | b



CYK Parse of a simple CFG

S → AB | BB
A → CC | AB | a
B → BB | CA | b
C → BA | AA | b

C	A		
A	A	B,C	B,C
a	a	b	b

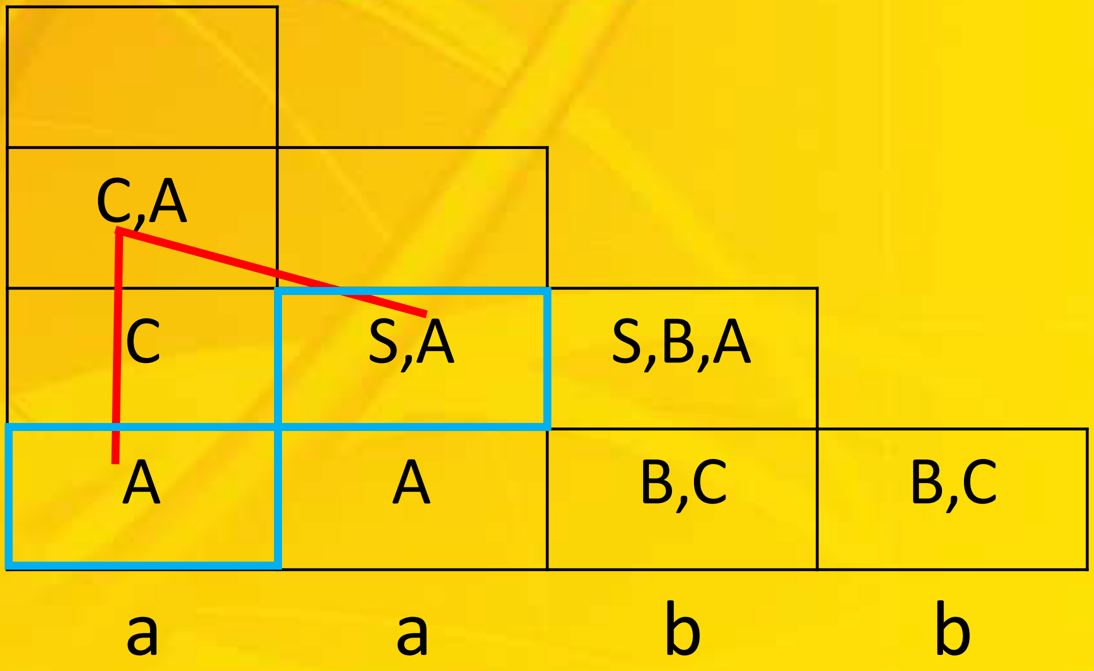
CYK Parse of a simple CFG

S → AB | BB
A → CC | AB | a
B → BB | CA | b
C → BA | AA | b

C	A	S,B,A	
A	A	B,C	B,C
a	a	b	b

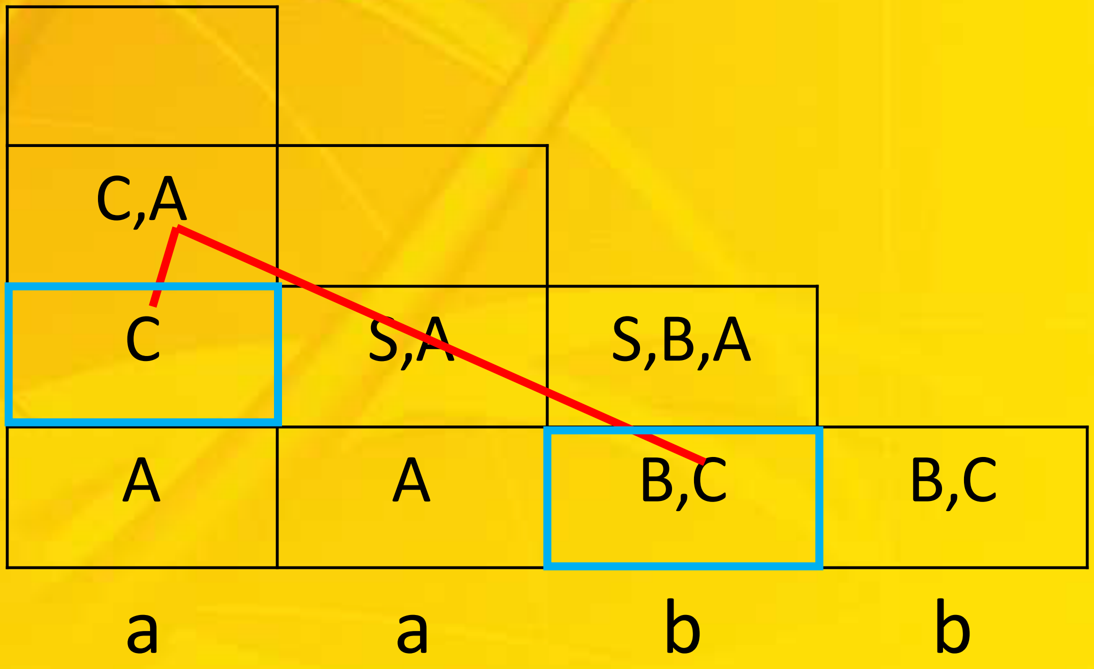
CYK Parse of a simple CFG

S → AB | BB
A → CC | AB | a
B → BB | CA | b
C → BA | AA | b



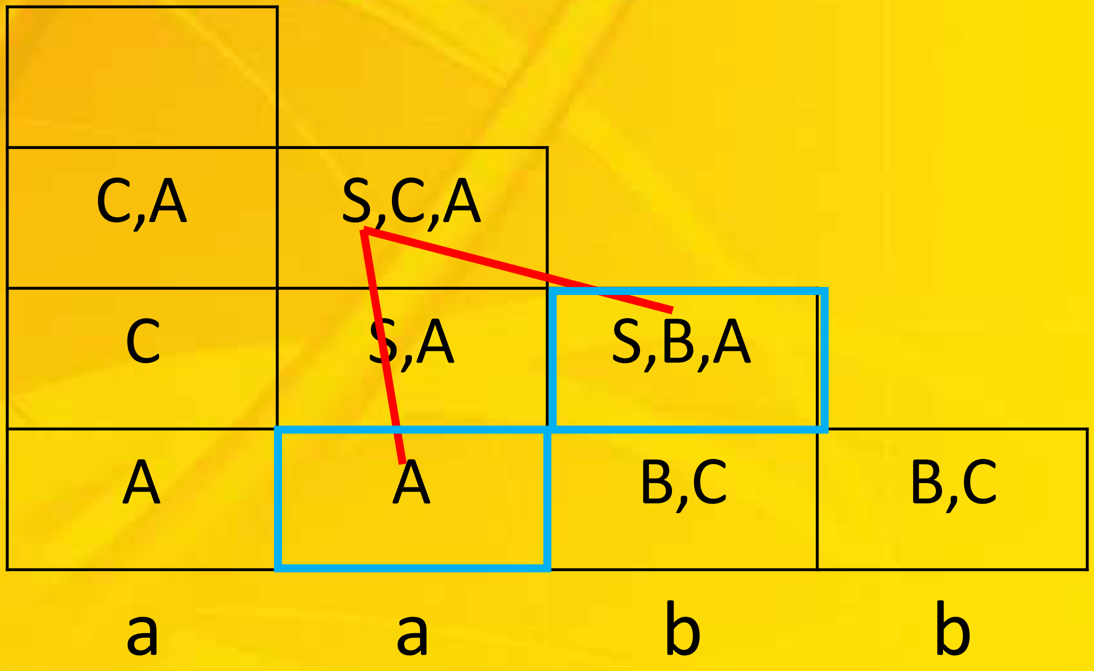
CYK Parse of a simple CFG

S → AB | BB
A → CC | AB | a
B → BB | CA | b
C → BA | AA | b



CYK Parse of a simple CFG

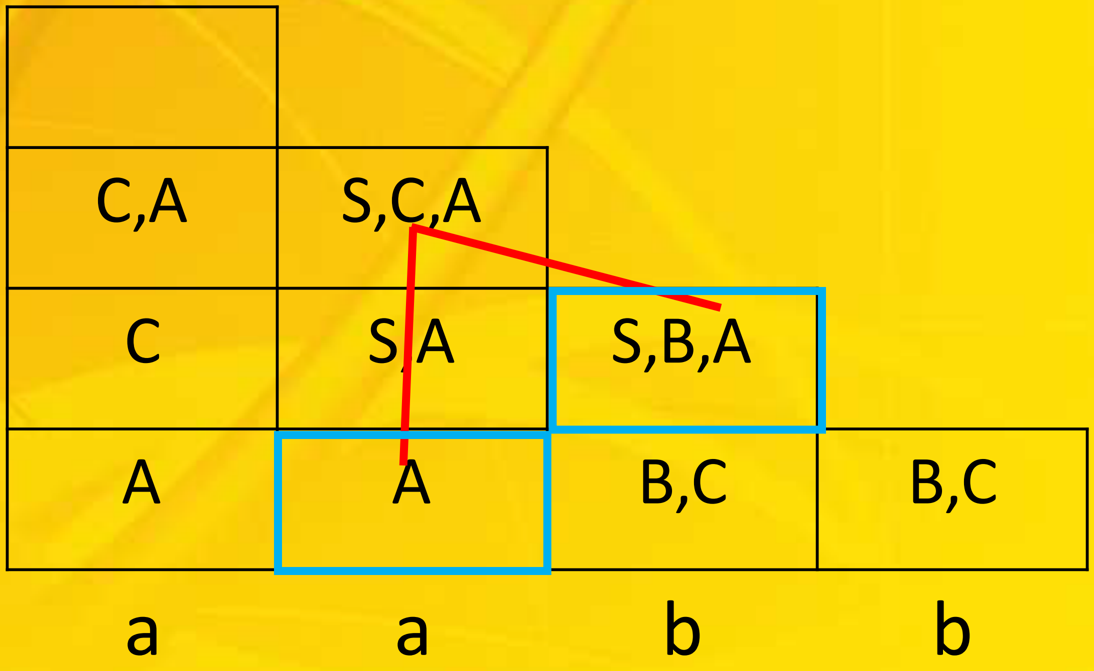
S → AB | BB
A → CC | AB | a
B → BB | CA | b
C → BA | AA | b



C,A	S,C,A		
C	S,A	S,B,A	
A	A	B,C	B,C
a	a	b	b

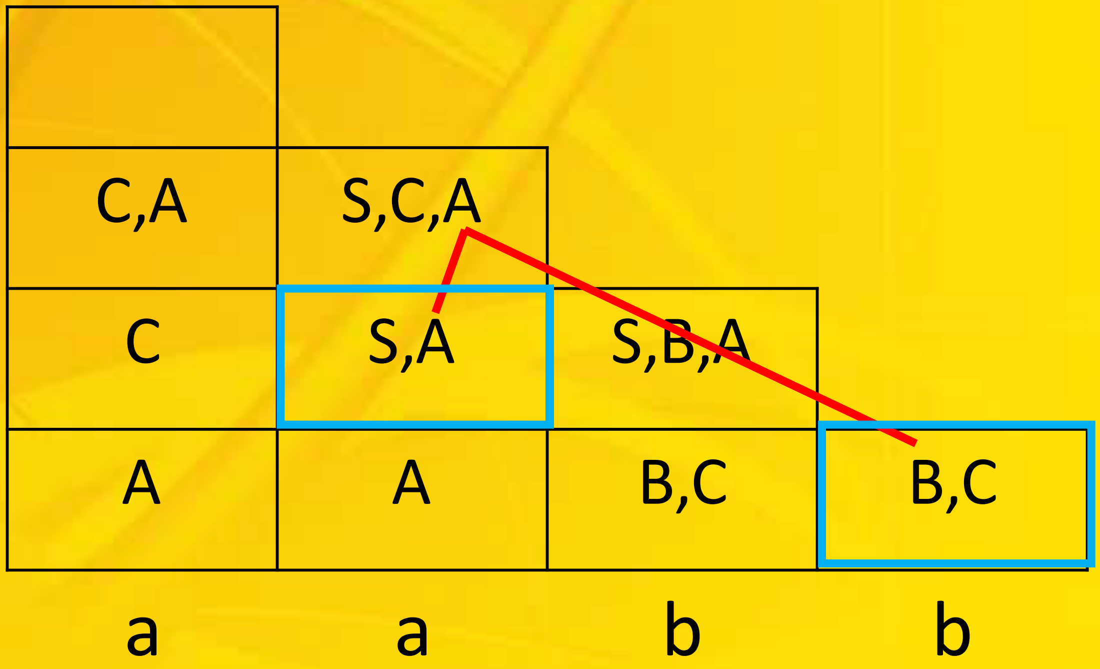
CYK Parse of a simple CFG

S → AB | BB
A → CC | AB | a
B → BB | CA | b
C → BA | AA | b



CYK Parse of a simple CFG

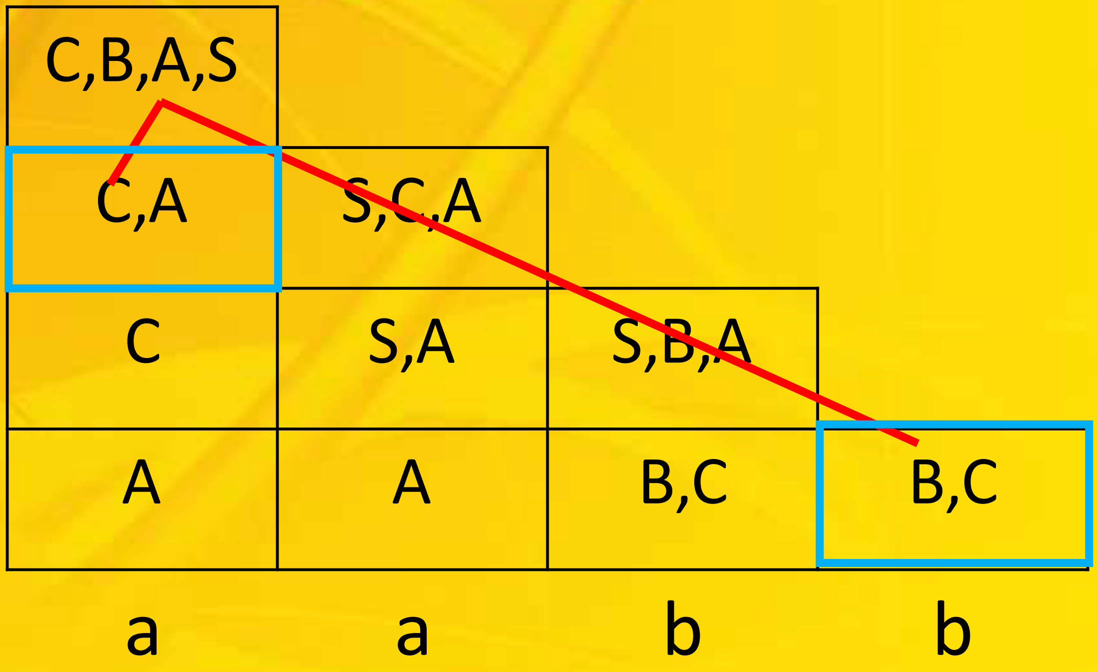
S → AB | BB
A → CC | AB | a
B → BB | CA | b
C → BA | AA | b



C,A	S,C,A		
C	S,A	S,B,A	
A	A	B,C	B,C
a	a	b	b

CYK Parse of a simple CFG

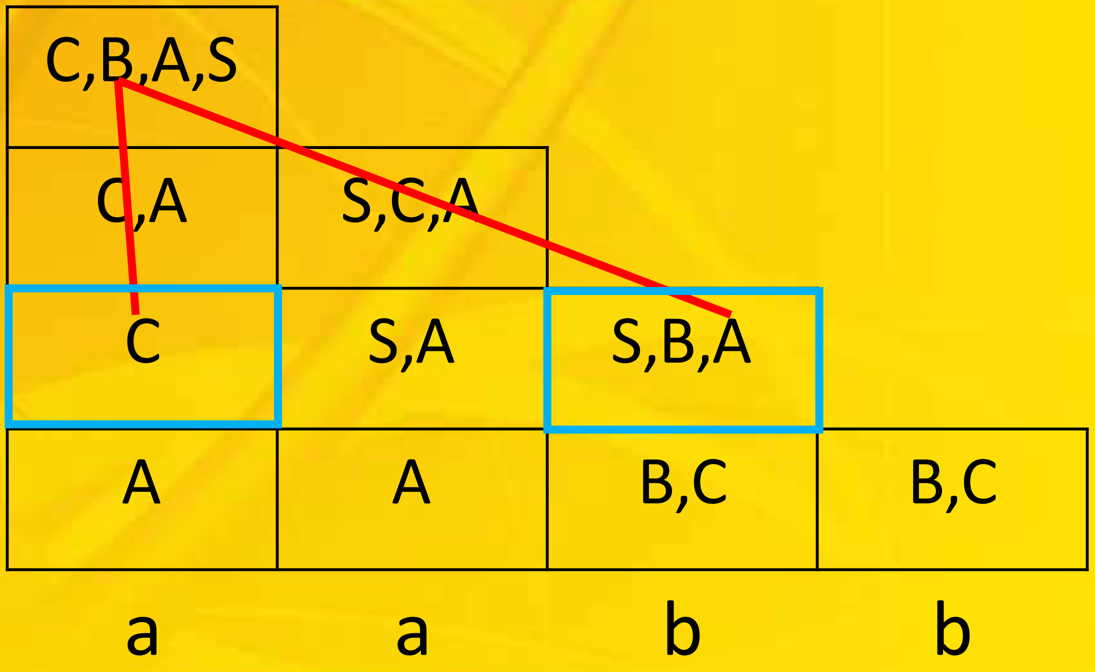
S → AB | BB
A → CC | AB | a
B → BB | CA | b
C → BA | AA | b



C,B,A,S			
C,A	S,C,A		
C	S,A	S,B,A	
A	A	B,C	B,C
a	a	b	b

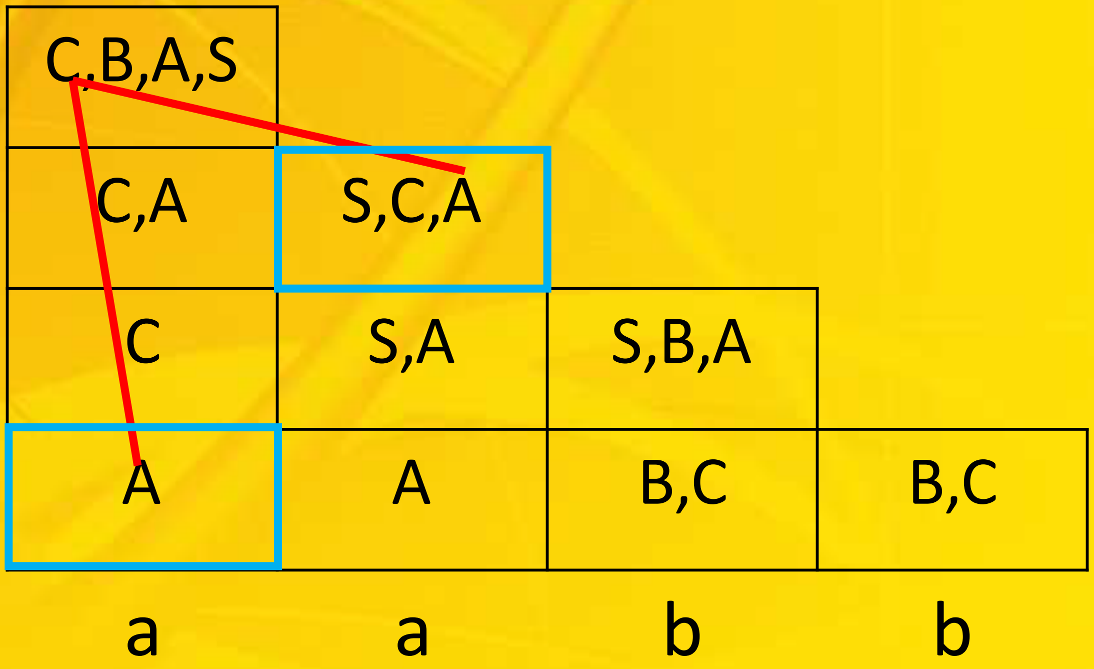
CYK Parse of a simple CFG

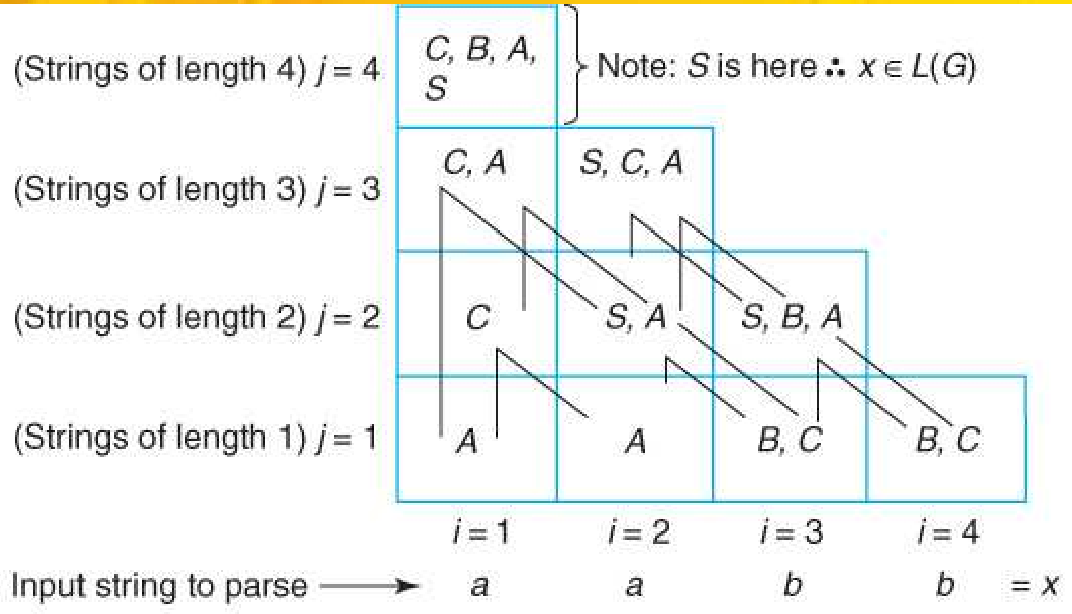
S → AB | BB
A → CC | AB | a
B → BB | CA | b
C → BA | AA | b



CYK Parse of a simple CFG

S → AB | BB
A → CC | AB | a
B → BB | CA | b
C → BA | AA | b




(Strings of length 4) $j = 4$
C, B, A, S
Note: S is here $\therefore x \in L(G)$
(Strings of length 3) $j = 3$
C, A
S, C, A
(Strings of length 2) $j = 2$
C
S, A
S, B, A
(Strings of length 1) $j = 1$
A
A
B, C
B, C
$i = 1$
$i = 2$
$i = 3$
$i = 4$
Input string to parse
a a b b $= x$

(a)

5	S				
4	—	A_1			
3	—	—	A_2		
2	A_1	—	—	A_2	
1	A_2A_3	A_2A_3	A_5	A_5	A_2A_3
	b	b	a	a	b

(i)

4	—			
3	—	A_1		
2	S	—	A_2	
1	A_4	A_2A_3	A_5	A_2A_3
	c	b	a	b

(ii)

(b)

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

$$S \longrightarrow bA_1|cA_2$$

$$A_1 \longrightarrow bA_2$$

$$A_2 \longrightarrow b|aA_2$$

Converting to CNF yields

$$S \longrightarrow A_3A_1$$

$$A_3 \longrightarrow b$$

$$S \longrightarrow A_4A_2$$

$$A_4 \rightarrow c$$

$$A_1 \rightarrow A_3 A_2$$

$$A_2 \rightarrow b$$

$$A_2 \rightarrow A_5 A_2$$

$$A_5 \rightarrow a$$

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

Slides to Accompany *Programming Languages and Methodologies*

R. J. Schalkoff

Chapter 6, Part 1: `minic` Scanning and Parsing with `flex` and `bison`

Resources: `flex` and `bison`

The `man` pages for GNU `flex` and GNU `bison` are essential and relatively complete references for these tools. In addition, web references for GNU `flex` and `bison`, including downloads of the executables and extensive documentation in various formats may be found at:

`http://www.gnu.org/software/flex/`

`http://www.gnu.org/software/flex/manual/`

`http://www.gnu.org/software/bison/bison.html`

`http://www.gnu.org/software/bison/manual/`

Regular Expressions (REs) in `flex`

Regular expressions[a] are commonly used in `bash`, `grep`, and `vi`.

Most importantly, they form the basis for the token recognizer implemented in `flex`.

Patterns to be recognized by `flex` in the input are denoted using extended regular expressions. The basic building blocks are summarized in Table 1.

[a] And extended regular expressions.

`flex` Syntax

- Most characters, including all letters and digits, are considered regular expressions that match themselves.
- When necessary, a meta character with special meaning may be quoted by preceding it with a backslash, as shown in Table 1.
- A bracketed entity denotes a regular expression that matches any single character enclosed in the brackets.
- Within a bracketed expression, two characters separated by a hyphen denote a range expression. Any single character in the range constitutes a match[a].
- If the first character in the range is a caret (^) then the bracketed entity matches any character not in the following range.

[a]Note this assumes an ordering.

character	action (matches)
x	match the character 'x'
.	match any character except newline
[xyz]	match either an 'x', a 'y', or a 'z'
[j-o]	match any letter from 'j' through 'o'
[^A-Z]	match any character except an uppercase letter
[^A-Z`\n`]	match any character except an uppercase letter or a newline
R*	match zero or more R, where R is a regular expression
R+	match one or more R
R?	match zero or one R
R{2,5}	match from two to five R
"[xyz]"	match the enclosed string *literally*, i.e., match [xyz]
`\X`	match the c interpretation of `\X` if X is a, b, f, n, r, t, or v
(R)	match R using parens to denote precedence
R | S	match regular expression R or regular expression S

Table 1: Primitive `flex` RE Building Blocks

`flex` Input File Structure

- The regular expressions to be matched against input are specified by the user in a `flex` source file (with the extension `.in`, by convention).
- This file consists of regular expressions to be matched against the input file together with corresponding `flex` actions.
- This file is translated by `flex` into a `c` program which reads an input character stream, and, where possible, converts the input into tokens which match the specified regular expressions.

flex Output

- The output of **flex** is a file, **lex.yy.c**, containing **c** source for a scanner.
- The scanning function within this file is denoted **yylex**.
- This process is shown below:

$$*.\texttt{in} \xrightarrow{\texttt{flex}} \underbrace{\texttt{lex.yy.c}}_{\text{contains } \texttt{yylex}()}$$

- **lex.yy.c** may be compiled into a freestanding executable using **gcc** with the **-lfl** library specification.

Returning Values and Variables

- Predefined global variables allow extended passing of values between `flex` and `bison`.
- In `flex`, following the match of a RE, the text corresponding to the match (in the case of `minic`, a token) is made available through the predefined global character pointer `yytext`.
- The length of the token is available in another global integer variable `yyleng`.

A simple use of `flex` with `minic`

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

`minic` Input (Source) File Used for Examples

```
/* this version does not gobble up whitespace */
ID [a-z][a-z0-9]*
NUM [0-9]+
%%
"(" printf(" lparens ");
")" printf(" rparens ");
"{" printf(" lbrace ");
"}" printf(" rbrace ");
"=" printf(" assign ");
";" printf(" semicolon ");
int printf(" int ");
void printf(" void ");
main printf(" main ");
return printf(" return ");
{ID} printf(" ide(%s) ",yytext);
{NUM} printf(" num(%s) ",yytext);
```

Input file `minicex0.in`

```
int   main  lparens  void  rparens
lbrace
int   ide(t1)  semicolon
ide(t1)  assign  num(10)  semicolon
return  lparens  ide(t1)  rparens  semicolon
rbrace
```

Use of `flex`*: Resulting File of Tokens*

```
/* this version gobbles up whitespace */
ID [a-z][a-z0-9]*
NUM [0-9]+
%%
[ \r\t\n]+ /* whitespace */
"(" printf(" lparens ");
")" printf(" rparens ");
"{" printf(" lbrace ");
"}" printf(" rbrace ");
"=" printf(" assign ");
";" printf(" semicolon ");
int printf(" int ");
void printf(" void ");
main printf(" main ");
return printf(" return ");
{ID} printf(" ide(%s) ",yytext);
{NUM} printf(" num(%s) ",yytext);
```

Input file `minicex0a.in`

```
 int  main  lparens  void  rparens  lbrace  int  ide(t1)
semicolon  ide(t1)  assign  num(10)  semicolon  return
lparens  ide(t1)  rparens  semicolon  rbrace
```

Use of **flex**: Resulting File of Tokens (wordwrapped for illustration)

Figure 1: Using `flex` for `minic`: Elimination of Whitespace

`bison` and Tokens for Parsing

- `flex`-creates function `yylex()`.
- The previous examples are based upon `printf` output of recognized tokens.
- `bison` produces the parser as the function `yyparse`.
- The parser requires `flex`-based generation of tokens.
- Specifically, `yyparse` expects single-digit tokens to be returned from the `flex`-generated scanner.

`bison`

- `bison` is a general purpose parser generator that converts a grammar description for an LALR(1) context free grammar into `c` (or `c++`) code to parse that grammar.
- `bison` produces the function `yyparse`.
- The user needs to also provide 'wrapper' code (i.e., `main`), an error reporting function which the parser calls to report an error, as well as a lexical analyzer (the latter from `flex`).

The Pragmatics of Using `bison`.

1. Specify the grammar in one or more `bison` grammar files.
2. Write or generate a lexical analyzer to process source code input and pass tokens to the parser.
3. Write a function that calls the `bison`-generated parser.
4. Write error reporting routines.
5. Run `bison` on the grammar to produce the parser function (`yyparse`).
6. Compile `yyparse` with other source files and link the object files to produce the overall parser.

Overview of bison Operation

- By convention, **bison** input files have the extension 'y'.

$$*.y \xrightarrow{\texttt{bison}} \underbrace{*.\texttt{tab.c}}_{\text{contains } \texttt{yyparse}()}$$

- The **bison**-generated **c** function **yyparse** calls **yylex** to provide tokens.

- **bison** may also be used with the **-d** option, in which case a file ***.tab.h** is generated for use with **flex**.

$$*.y \xrightarrow{\texttt{bison} \; -\texttt{d}} \left\{ \begin{array}{l} *.\texttt{tab.c} \\ *.\texttt{tab.h} \end{array} \right.$$

Conveying the grammar to `bison`

`bison` generates the parser from input in the form of a `bison` grammar file.

- A nonterminal symbol in the formal grammar is represented in `bison` input as an identifier, which, by bison convention, must be in lower case (i.e., all characters), e.g, `expr`.
- The bison representation for a terminal (also called a token type) is in upper case, e.g., `IDENTIFIER`.
- A terminal symbol that stands for a particular keyword in the language should be named after that keyword converted to upper case.

Using `flex` and `bison` Together for `minic`

- The `bison` input file ends in `.y`
- the corresponding parser created by `bison` is contained in the `c` file with extension `tab.c`

bison Input File for minic

```
%{
#include <stdio.h>
#include <ctype.h>
int yylex (void);
int yyerror (char *s);
%}

%token INT
%token VOID
%token MAIN
%token LPARENS
%token RPARENS
%token LBRACE
%token RBRACE
%token ASSIGN
%token SEMICOLON
%token COMMA
%token RETURN
%token NUM
%token IDE

%%  /* Grammar rules */
transunit: maindecl body
;
maindecl: type MAIN arg
;
type: INT | VOID
;
```

```
arg: LPARENS type RPARENS
;
body: LBRACE decl statseq RBRACE
;
decl:  /* empty */
     | type variablelist SEMICOLON
;
variablelist: variable
             | variable COMMA variablelist
;
statseq: returnstat
         | commandseq SEMICOLON returnstat
;
commandseq: command
           | commandseq SEMICOLON command
;
command: variable ASSIGN expr
;
variable: IDE
;
expr: NUM
;
returnstat: RETURN returnarg SEMICOLON
;
returnarg: LPARENS IDE RPARENS | LPARENS NUM RPARENS
;
%%
```

flex Input File for minic

```
%{
#include "minicex1.tab.h"
%}
ID [a-z][a-z0-9]*
A_NUM [0-9]+
%%
[ \r\t\n]+ /* whitespace */
"(" return LPARENS;
")" return RPARENS;
"{" return LBRACE;
"}" return RBRACE;
"=" return ASSIGN;
";" return SEMICOLON;
"," return COMMA;
int return INT;
void return VOID;
main return MAIN;
return return RETURN;
{ID} return IDE; /* more later */
{A_NUM} return NUM;
%%
```

Error Handling

```
int yyerror (s)  /* Called by yyparse on error */
     char *s;
{
  printf ("%s\n", s);
  return(-1);
}
```

Figure 2: Error Handling Function yyerror.c

The Overall Project and `main`

Recall:

- `flex` generates the `c`-based scanner in `lex.yy.c`
- `bison` generates the parser in `minicex1.tab.c`

```
#include "minicex1.tab.c"
#include "lex.yy.c"
#include "yyerror.c"
#define YYERROR_VERBOSE

int main ()
{
  yyparse ();
  return(1);
}
```

Figure 3: Overall Project Source File minicex1.c

Putting `flex` and `bison`- Generated Functions Together

Compiling the scanner/parser and sample operation is shown in the following:

```
$ gcc minicex1.c -lfl -o minicex1
$ ./minicex1 <minicex1.mc
$
```

Notes:

- In this case, the parse was successful, although it is not immediately apparent from the scarce scanner/parser output.
- See the book, Section 6.7, pages 180+ for more elaboration and extensions.

Slides to Accompany *Programming Languages and Methodologies*

R. J. Schalkoff

Chapter 7, Part 1: `minic` (Ver. 2)

The Complete Syntax for Enhanced minic Version 2

```
/* minic translational unit grammar */
/* file: transunit-bnf4r2nir.txt  */

<transunit> ::= <main-decl> <body>
<main-decl> ::= <type> main <arg>
<type> ::= int | void | boolean
<arg> ::= lparens <type> rparens
<body> ::= lbrace <dec-seq> <statseq> rbrace

<dec-seq> ::= e | <decl> <dec-seq>
<decl> ::= <type> <variable-list>;
<variable-list> ::= <variable> | <variable> , <variable-list>
<variable> ::= <identifier>

<statseq> ::= <return-stat> | <command-seq>; <return-stat>
<command-seq> ::= <command> | <command> ; <command-seq>
```

```
/* new commands */

<command> ::= <simple-assignment> | <while> | <for> | <if>

<while> ::= while lparens <boolean-expr> rparens <command>
/* note arguments to for optional */

<for> ::= for lparens <for-simple-assignment> ;
                        <for-boolean-expr>  ;
                        <for-simple-assignment>
                        rparens <command>

<if> ::= if lparens <expr> rparens <command>

<for-simple-assignment> ::= <simple-assignment> | e
<for-boolean-expr> ::= <boolean-expr> | e

<simple-assignment> ::= <identifier> = <expr>
<expr> ::= <integer-expr> | <boolean-expr>

<integer-expr> ::= <simple-integer-expr> | <comp-integer-expr>
```

```
<integer-expr> ::= <function-appl>
<function-appl> ::= <function-name> <fn-args>
<fn-args> ::= lparens <identifier> rparens | lparens <numeral> rparens

<boolean-expr> ::= <simple-boolean-expr> | <comp-boolean-expr>

<comp-integer-expr> ::= <simple-integer-expr> <int-oper> <rest-integer-expr>
<rest-integer-expr> ::= <simple-integer-expr> | <comp-integer-expr>
/* note to disambiguate need operator precedence rules */

<comp-boolean-expr> ::= <simple-integer-expr> <bool-oper> <simple-integer-expr>

<simple-integer-expr> ::= <identifier> | <integer-const>
<simple-boolean-expr> ::= <identifier> | <boolean-const>
<integer-const> ::= <numeral>
<int-oper> ::= * | / | + | -
<bool-oper> ::= < | <= | == | !=
<boolean-const> ::= true | false

<return-stat> ::= return <return-arg>;
```

```
<return-arg> ::= lparens <variable> rparens
                 | lparens <numeral>  rparens
                 | lparens <boolean-const>  rparens

/* end of updated syntax  */
/* lexical part for identifiers and numerals */

<identifier> ::= <letter> | <identifier> <letter> |  <identifier> <digit>
<letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m
                    | n | m | 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
```

Complete Prolog Scanning/Parsing Code

See the code listing in Section 7.4.3, pages 230+.

Contextual Constraints and Typed Languages

Consider a simplistic grammar intended to facilitate English-language sentence production[a] with a sample production of the form:

$$S \rightarrow NVO$$

where $S, N, V, O \in V_N$. The alternative BNF representation might be:

```
<sentence> ::- <noun> <verb> <object>
```

[a]Here S represents 'Sentence', N represents 'Noun Phrase', V represents 'Verb Phrase' and O represents 'Object'.

In using productions in this context-free grammar, there is no need for the noun, verb and object of the sentence to have any meaningful relationship, e.,g., it is possible to produce sentences like:

’computers eat ladders’

’ladders hate computers’

’books study students’

Such sentences, while *syntactically correct*, are *semantically meaningless*. To rectify this situation, we seek two things:

1. A way to 'globally' constrain the allowable replacements, rather than simply allowing any joint set of values which result from *independent* replacement of N, O, V.

2. A way to accomplish the above without using a context sensitive grammar.

Solution: Introducing Attribute Grammars

Consider the possibility that the nonterminals N, O, V have properties or *attributes* which are available for use during the generation or recognition phases of the grammar use.

These attributes are shown in Table 1.

nonterminal	attribute
N	'something used for computing'
V	'an action allowed in computing'
O	'the result of a computing action'

Table 1: Example of Nonterminal Attributes for Context-Sensitive Parse

- Suppose before attempting to parse an expression using these nonterminals, a function is available for checking attribute compatibility, e.g.,

  'This production may be used if the attributes of all nonterminals involve computing'.

- Although a simplistic example, it nonetheless conveys the idea:

  To implement contextual constraints, attributes are generated and checked during the parsing process.

Contextual Constraints in Programming Languages

- One of the most common constraints is that of *enforcing variable declarations and type checking.*
- Since many languages are typed, how do we enforce type checking in the parsing process?

- Consider two program[a] fragments:

```
int t1;
t1=10;
```

vs.

```
int t1;
t1=10.00001;
```

- Without modification, the parser cannot distinguish between these cases.
- Both would be considered syntactically correct. For example, in `minic` syntax:

```
<decl> ::= e | <type> <variable-list>;
<variable-list> ::= <variable> | <variable> , <variable-list>
<variable> ::= <identifier>
<command> ::= <variable> = <expr>
```

there is no checking to see if the type of `<expr>` matches the type declaration for the variable identifier used.

[a] Assume the lexical part of the language allows production of 10.00001.

Contextual Constraints and `minic`

What is wrong with the syntax of the following `minic` translational unit?

```
// miniccontext1.mc
boolean main(void)
{
boolean b1,b2;
int t2;
t2=true;
t3=t4+b3+20;
b2 = b1 + 20;
while(b2<t3) b2=t2+2;
return(t2);
}
```

The (hopefully not too surprising) answer is 'nothing'.

While we may question numerous (perceived) violations of variable and expression typing, including:

- The appearance of undeclared variables;
- The assignment of boolean values to variables declared as int;
- Variables used but not declared; and
- Mixing int and boolean types, etc.

  This is a result of familiarity with `c`, not `minic`. **we have not (yet), in the `minic` syntax, incorporated these contextual constraints**.

A scan/parse of this file also confirms the syntactic correctness of this file.

The Problem Shown with a 'Real' c Compiler

To further illustrate the problem and our objectives, consider a `c` formulation[a] of the preceding `minic` file:

```
// from miniccontext1.mc
#include<stdbool.h>
typedef bool boolean;
boolean main(void)
{
boolean b1,b2;
int t2;
t2=true;
t3=t4+b3+20;
b2 = b1 + 20;
while(b2<t3) b2=t2+2;
return(t2);
}
```

[a]Here we use the C99 extension for type `bool`.

Using `gcc`, this `c` source generates the warnings shown below.

```
bash-2.02$ gcc miniccontext1r.c -Wall
miniccontext1r.c:5: warning: return type of ‘main’ is not ‘int’
miniccontext1r.c: In function ‘main’:
miniccontext1r.c:9: ‘t3’ undeclared (first use in this function)
miniccontext1r.c:9: (Each undeclared identifier is reported only once
miniccontext1r.c:9: for each function it appears in.)
miniccontext1r.c:9: ‘t4’ undeclared (first use in this function)
miniccontext1r.c:9: ‘b3’ undeclared (first use in this function)
```

Potential Solution(s)

- Enforce the type of `<expr>` to match the type declaration for the variable identifier used.
- Easier said than done...

Possible Approaches

1. Lexical restrictions on identifier names for each type (e.g., ”all integers begin with i, j, or k”)
2. Constrain the production(s) involving `<variable>` in

   ```
   <decl> ::= <type> <variable-list>;
   <variable-list> ::= <variable>
   ```

   to be somehow constrained by the other production:

   ```
   <command> ::= <variable> = <expr>
   ```

   This suggests that the second production could only be used *in the context* of the first set of productions, e.g.,

   ```
   <decl> <command> ::= <type> <variable>; <variable> = <expr>
   ```

   As the above BNF-form production indicates, we have just entered the world of context-sensitive grammars.
3. $\boxed{\text{Use an attribute grammar.}}$

Attribute Grammars

> *An attribute grammar is a context-free grammar (CFG), together with a* **set of attributes for each nonterminal** *and a set of attribute evaluation procedures for each production.*

An important distinction now occurs: *not all nonterminals with the same name (e.g.,* `<variable>`*) are equal*, since they may have different attributes.

- An attribute attached to a nonterminal is either synthesized or inherited.
- For each production, there are auxiliary functions that define the synthesized attributes or check attributes.

A Simple Example: Contextual Constraints

Consider a grammar which generates $L(G)$, where

$$L(G) = \{a^n b^n c^n\}\ n \geq 0$$

Strings produced in this language must have equal numbers of a's, b's and c's.

The CFG grammar below is considered as a candidate for the task[a].

$$S \to ABC$$

$$A \to a|aA$$

$$A \to \epsilon$$

$$B \to b|bB$$

$$B \to \epsilon$$

$$C \to c|cC$$

$$C \to \epsilon$$

[a]Recall ϵ designates the empty string.

The corresponding Prolog implementation, using the LGN, is as follows:

```
%% example of CFG for a^n b^n c^n language
%% file: attrib0.pro

string --> getAs, getBs, getCs.

getAs --> [a], getAs.
getAs --> [ ].

getBs --> [b], getBs.
getBs --> [ ].

getCs --> [c], getCs.
getCs --> [ ].
```

Sample parsing operation is shown below.

```
?- string([a,a,a,b,b,b,c,c,c],[]).

Yes
?- string([a,a,b,b,c,c],[]).

Yes
?- string([a,a,b,b,c],[]).

Yes
?- string([a,a,a,b,b,c],[]).

Yes
?-
```

- Unfortunately, this is not the desired result.
- Notice that $L(G) = \{a^j b^k c^l\}$, which is different from the language desired.
- The obvious problem is that there is no constraint that the substrings of a's, b's and c's must be equal length.

Solution- An Attribute Grammar in Prolog

One alternative Prolog LGN formulation is shown below.

```
%% file: attrib1.pro

string --> getAs(M1), getBs(M2), getCs(M3),
{ M1=:=M2, M2=:=M3 }.

getAs(M) --> [a], getAs(N), { M is N+1 }.

getAs(0) --> [ ].

getBs(M) --> [b], getBs(N), { M is N+1 }.

getBs(0) --> [ ].

getCs(M) --> [c], getCs(N), { M is N+1 }.

getCs(0) --> [ ].
```

Sample Translations from LGN

```
string(A, B) :-
        getAs(C, A, D),
        getBs(E, D, F),
        getCs(G, F, H),
        C=:=E,
        E=:=G,
        B=H.

getAs(A, B, C) :-
        'C'(B, a, D),
        getAs(E, D, F),
        A is E+1,
        C=F.

getAs(0, A, A).
```

The solution which generates and parses the desired language is based upon predicate `getAs`, with sample operation shown below.

```
?- getAs(HowMany, [], []).

HowMany = 0
Yes
?- getAs(Number, [a], []).

Number = 1
Yes
?- getAs(Number, [a,a], []).

Number = 2

Yes
?- getAs(Number, [a,a,a], []).

Number = 3

Yes
```

```
?- getAs(Number, [a,a,b], []).

No
```

Overall Operation of the Modified Prolog Parser

Recall we required equal length strings of ’a’s, ’b’s and ’c’s in that order, and this is achieved.

```
?- string([a,a,a,b,b,b,c,c,c],[]).

Yes
?- string([a,a,a,b,b,c,c,c], []).

No
```