Introductory Slides

5DV037 — Fundamentals of Computer Science

Umeå University
Department of Computing Science

Stephen J. Hegner
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

Alphabets

- An *alphabet* is a finite nonempty set.

Examples:

  - $\{A, B, \ldots, Z\}$
  - $\{A, B, \ldots, Z,\ a, b, \ldots, z,\ ,0, 1, \ldots, 9\}$
  - The ASCII character set
  - The printable ASCII characters
  - The ISO-8859-14 character set
  - $\{0, 1\}$
  - $\{1\}$

- The uppercase Greek letter Σ is often used to denote an alphabet.
- Usually each element of an alphabet is represented by a single symbol, but this is not necessary.
- Practical examples which use other representations will be given later.

Words

- A *word* over the alphabet Σ is any finite sequence of symbols from Σ. (Represented as a string.)

Examples:

- *Hello␣world!* is a word over the ASCII character set.
  - Note that a *word* in this sense is more general than a word in natural language.
- *Hejsan␣världen!* is a word over the ISO-8859-14 character set.
- 01101101 is a word over the character set $\{0, 1\}$.
- A program in most programming languages is a word over the ASCII character set.
- The contents of any file under UNIX is a word over the character set consisting of all possible byte values.
- The lowercase Greek letter λ is typically used to denote the *empty word* or *empty string* of length zero.

Languages

- A *language* over the alphabet Σ is any set of words over Σ.

Examples:

  - The set of all legal C programs ($\Sigma =$ printable ASCII).
  - $\{$*Hello␣world!*, *Hejsan␣världen!*$\}$ ($\Sigma =$ ISO-8859-14).
  - All strings containing *5DV037* as a substring.
  - All *palindromes* (strings which are the reverse of themselves; *e.g.*, *abba*, *amanaplanacanalpanama*).

- In theoretical work, abstract and seemingly meaningless languages are often used to illustrate points or prove results.

Examples:

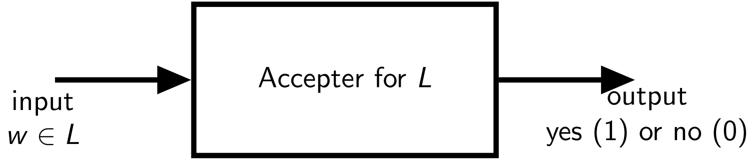
  - $\{a^n b^n \mid n \in \{0, 1, 2, \ldots\}\}$.
  - $\Sigma^* =$ all words over Σ.
  - $\Sigma^+ =$ all words over Σ except the empty word λ.

Questions about Languages

- The focus of this course is a theory of languages and their properties.
- A central question is the following.

The Membership Problem: Given a language L over an alphabet Σ, construct a device which will determine whether a string $w \in \Sigma^*$ is in L.

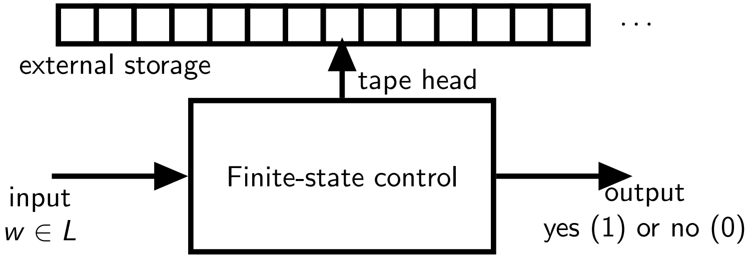
- Such a device is called an *accepter* for L.



- What is the structure of an accepter?

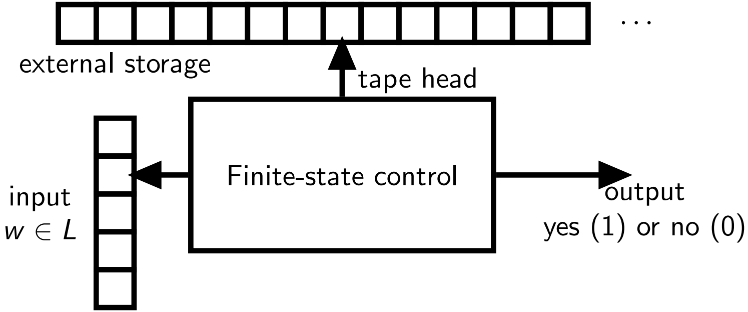
The Structure of Accepters

- An accepter consists of two main components:
  - The *finite-state control*
  - The *external storage*
- Often the external storage is regarded as lying on a tape of some sort, although this is not absolutely necessary.



The Structure of Accepters

- An accepter consists of two main components:
  - The *finite-state control*
  - The *external storage*
- Often the external storage is regarded as lying on a tape of some sort, although this is not absolutely necessary.
- The input may also be regarded as lying on a read-only tape.
- There will be other variations, introduced as needed.



Classes of Accepters to Be Studied in this Course

- Three main classes of accepters and the associated languages will be considered.

  Finite-state automata: No external storage.

  Pushdown automata: Stack as external storage.

  Turing machines: Semi-infinite read-write tape as external storage. (Effectively unbounded memory)

- For Turing machines, the distinction between a *decider* and a *semi-decider* will also be made.
  - A decider answers *yes* or *no* for every word w of the input language L.
  - A semi-decider always answers *yes* if $w \in L$, but it may loop forever instead of answering *no* in the case that $w \notin L$.
  - The latter is a consequence of the unsolvability of the *halting problem* — there exist languages which are semi-decidable but not decidable.

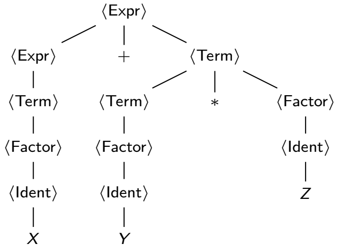
Beyond Simple Accepters

- Often, it is desirable to know more than just whether or not $w \in L$.

Example: Parsing a computer language or a natural language.

- If $w \in L$, it is desirable to know something of the structure of or information contained in w as well. (*e.g.*, *parse*).

$X + Y * Z \quad \rightsquigarrow$



- If $w \notin L$, it is useful to know why.
- To this end, it is important to introduce the notion of a *grammar*.

The Idea of a Grammar

- The ideas behind grammars are the following.

  Productions: The productions are rules which allow a (sub)string to be replaced by another string.

  Start symbol; The start symbol specifies the starting string to which the production rules are applied.

  Derivation: A string is derivable from the grammar if it may be obtained by applying the productions to the start symbol.

  Parsing: A parser for a given grammar is a program (algorithm) which takes strings and finds derivations for them.

  Accepter: An accepter runs a parser and answers yes if the parser finds a derivation.

Formalization of the Notion of a Grammar

Definition: A *(phrase-structure) grammar* is a four-tuple

$$G = (V, \Sigma, S, P)$$

in which

- V is a finite alphabet, called the *variables* or *nonterminal symbols*;
- Σ is a finite alphabet, called the set of *terminal symbols*;
- $S \in V$ is the *start symbol*;
- P is a finite subset of $(V \cup \Sigma)^+ \times (V \cup \Sigma)^*$ called the set of *productions* or *rewrite rules*;
- $V \cap \Sigma = \emptyset$;

- The production $(w_1, w_2) \in P$ is typically written $w_1 \underset{G}{\rightarrow} w_2$, or just $w_1 \rightarrow w_2$ if the context G is clear.
- The meaning of $w_1 \rightarrow w_2$ is that w_1 may be replaced by w_2 in a string.
- Usually, for $w_1 \rightarrow w_2$, w_1 will contain at least one variable, although this is not strictly necessary.

The Derivation of Words from a Grammar

Context: $G = (V, \Sigma, S, P)$

- Let $w_1 \underset{G}{\rightarrow} w_2$, and let $w \in (V \cup \Sigma)^+$ be a string which contains w_1; *i.e.*, $w = \alpha_1 w_1 \alpha_2$ for some $\alpha_1, \alpha_2 \in (V \cup \Sigma)^*$.
- A possible *single-step derivation* on w replaces w_1 with w_2.
- Write $\alpha_1 w_1 \alpha_2 \underset{G}{\Rightarrow} \alpha_1 w_2 \alpha_2$ (or just $\alpha_1 w_1 \alpha_2 \Rightarrow \alpha_1 w_2 \alpha_2$).
- Note that many derivation steps may be possible on a given string, and that applying one may preclude the application of another.
- This process is thus inherently nondeterministic.
- Write $w \overset{*}{\underset{G}{\Rightarrow}} u$ (or just $w \overset{*}{\Rightarrow} u$) if $w = u$ or else there is a sequence

  $$w = \alpha_0 \overset{*}{\underset{G}{\Rightarrow}} \alpha_1 \overset{*}{\underset{G}{\Rightarrow}} \alpha_2 \ldots \overset{*}{\underset{G}{\Rightarrow}} \alpha_k = u$$

  called a *derivation* of u from w (for G).
- The *language of G* is $\mathcal{L}(G) = \{w \in \Sigma^* \mid S \overset{*}{\underset{G}{\Rightarrow}} w\}$.
- The grammars G_1 and G_2 are *equivalent* if $\mathcal{L}(G_1) = \mathcal{L}(G_2)$.

An Example of Derivation

$$\text{Let} \quad G = (V, \Sigma, S, P) = (\{S\}, \{a, b\}, S, \{S \to aSb,\ S \to ab\}$$
$$= (\{S\}, \{a, b\}, S, \{S \to aSb \mid ab\}$$

- The symbol "|" is frequently used to specify alternatives for productions and save space.
- The string *aaabbb* has the derivation

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$$

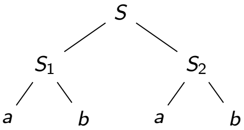
  and hence is in $\mathcal{L}(G)$.
- The string *aaaabbb* has no derivation and hence is not in $\mathcal{L}(G)$.
- It is easy to see that $\mathcal{L}(G) = \{a^n b^n \mid n \geq 1\}$.
- It is furthermore easy to see that every string in $\mathcal{L}(G)$ has a unique derivation.

Inessential Non-Uniqueness in Derivation

Let $G = (V, \Sigma, S, P) = (\{S, S_1, S_2\}, \{a, b\}, S,$
$\{S \to S_1 S_2,\ S_1 \to aS_1 b \mid ab,\ S_2 \to aS_2 b \mid ab\}.$

- Here $\mathcal{L}(G) = \{a^{n_1} b^{n_1} a^{n_2} b^{n_2} \mid n_1, n_2 \geq 1\}$.
- In this case even the simple string *abab* has two distinct derivations:
  $S \Rightarrow S_1 S_2 \Rightarrow abS_2 \Rightarrow abab$

  $S \Rightarrow S_1 S_2 \Rightarrow S_1 ab \Rightarrow abab$
- However, there is only one tree-like representation of the derivation.



- Such a tree, called a *derivation tree*, provides more useful information than just a linear derivation using $\Rightarrow$.
- Such trees are widely used in computer science.

Context-Free Grammars and Derivation Trees

- The grammars which have been presented as examples here (as well as in Chapter 1 of the book) are all *context free*.
- Such grammars are by far the most important kind in practice.
- The grammar $G = (V, \Sigma, S, P)$ is *context free* if every production in P is of the form $N \rightarrow \alpha$ for some $N \in V$. (*CFG* = *context-free grammar*).
- As shown on the previous slide, for a CFG, every derivation can be represented as a tree with ordered children.
  - The root of the tree is is the start symbol.
  - Every interior vertex is a nonterminal symbol.
  - Every leaf vertex is a terminal symbol.
  - For every interior vertex labelled with a nonterminal symbol N, the children of that vertex, from left to right, are labelled with the symbols defined by the string α for some production $N \rightarrow \alpha$.

A Real-World Example

Consider the problem of representing simple infix arithmetic expressions for a programming language.

- For simplicity, only addition and multiplication are considered.
- Want the parse tree to be unique.
- Want the tree to represent the precedence of the operations.
- Here is the standard example of such a grammar.
- G_{AExp} has:
  Nonterminals: $\{\langle\mathsf{Expr}\rangle, \langle\mathsf{Term}\rangle, \langle\mathsf{Factor}\rangle, \langle\mathsf{Ident}\rangle\}$.
  Terminals: $\{A, B, \ldots, Z, (,), +, *\}$.
  Start symbol: $\langle\mathsf{Expr}\rangle$
  Productions:

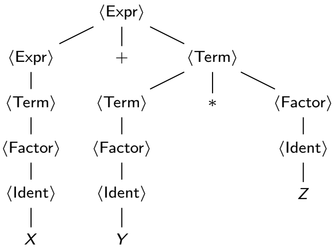
$$\begin{array}{rcl}
\langle \mathit{Ident} \rangle & \rightarrow & A \mid B \mid \ldots \mid Y \mid Z \\
\langle \mathit{Expr} \rangle & \rightarrow & \langle \mathit{Expr} \rangle + \langle \mathit{Term} \rangle \mid \langle \mathit{Term} \rangle \\
\langle \mathit{Term} \rangle & \rightarrow & \langle \mathit{Term} \rangle * \langle \mathit{Factor} \rangle \mid \langle \mathit{Factor} \rangle \\
\langle \mathit{Factor} \rangle & \rightarrow & (\langle \mathit{Expr} \rangle) \mid \langle \mathit{Ident} \rangle
\end{array}$$

A Real-World Example Continued

Nonterminals: $\{\langle\text{Expr}\rangle, \langle\text{Term}\rangle, \langle\text{Factor}\rangle, \langle\text{Ident}\rangle\}$.
Terminals: $\{A, B, \ldots, Z, (,), +, *\}$.
Start symbol: $\langle\text{Expr}\rangle$
Productions:

$$\begin{array}{rcl} \langle Ident\rangle & \rightarrow & A \mid B \mid \ldots \mid Y \mid Z \\ \langle Expr\rangle & \rightarrow & \langle Expr\rangle + \langle Term\rangle \mid \langle Term\rangle \\ \langle Term\rangle & \rightarrow & \langle Term\rangle * \langle Factor\rangle \mid \langle Factor\rangle \\ \langle Factor\rangle & \rightarrow & (\langle Expr\rangle) \mid \langle Ident\rangle \end{array}$$

- Here is the unique parse trees for $X + Y * Z$.
- Uniqueness will be discussed later in the course.
- Note here how the derivation is represented.
- Note also how it respects the standard arithmetic precedence operations.
- Subtrees can be evaluated and combined.



Standard Notation for Context-Free Grammars

- There is a standard notation known as BNF.
  - Backus Normal Form, or
  - Backus-Naur Form
- Identifiers are typically written enclosed in angle brackets, as already illustrated; *e.g.*, $\langle$Ident$\rangle$.
  - This is necessary because, in contrast to abstract theoretical examples, it is often the case that in real examples all of the usual Latin letters are terminal symbols.
  - In typesetting using the ASCII character set, the angle brackets may be written using $<$ and $>$; *e.g.*, <Ident>.
- The production symbol is sometimes written ::=, particularly in an ASCII description.

  Example: <Expr> ::= <Expr>+<Term> | <Term>

Some Supporting Notation and Notions

- It is useful to clarify and collect some notation.
- Some minor differences in mathematical notation:

In the textbook	In these slides	Meaning
$\{x \,:\, x \in S\}$	$\{x \mid x \in S\}$	set definition
$X - Y$	$X \setminus Y$	set difference
$\lvert x \rvert$	Length(x)	length of a string
$n_a(w)$	Count$\langle a, w\rangle$	number of a's occurring in w
$L(G)$	$\mathcal{L}(G)$	the language of G

- Some useful sets:

  The natural numbers: $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$

  The positive natural numbers: $\mathbb{N}^{>0} = \{1, 2, 3, \ldots\} = \mathbb{N} \setminus \{0\}$

  The integers: $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$

Some Supporting Concepts for Strings

- Some basic operations on strings:

  Concatenation: Concatenation simply appends one string to another.
  $(w_1, w_2) \mapsto w_1 w_2$ (also denoted $w_1 \cdot w_2$).
  Example: $(abc, def) \mapsto abcdef$.
  - Concatenation extends to finitely many strings in the obvious way: $(w_1, w_2, \ldots, w_k) \mapsto w_1 w_2 \ldots w_k$.

  Practical implementation: The UNIX *cat* command.

  Length: Length(w) just counts the number of elements in the string.
  Example: Length(Hello) = 5.
  Practical implementation: The UNIX *wc* command.

  Reversal: w^R is the string w with the letters in reverse order.
  Example: If $w = abc$, then $w^R = cba$.
  Practical implementation: The UNIX *rev* command.

Further Supporting Concepts for Strings

- Lisp-like operations on strings:
  - First$\langle w \rangle$ extracts the first element of a nonempty string. (Lisp *car*)
    - First$\langle a_1 a_2 \ldots a_k \rangle = a_1$
  - Rest$\langle w \rangle$ drops the first element of a nonempty string. (Lisp *cdr*)
    - Rest$\langle a_1 a_2 \ldots a_k \rangle = a_2 \ldots a_k$
- Other basic concepts of strings:

  Substring: A substring of w is any contiguous sequence extracted from w.

  Example: Let $w = abcdefg$. Then $bcdef$ and efg are substrings, as are λ and w itself. acd is not a substring.

  Prefix: A prefix is an initial substring. In the above, λ, a, abc, and $abcdefg$ are prefixes of $abcdefg$.

  Suffix: A suffix is a final substring. In the above, λ, f, def, and $abcdefg$ are prefixes of $abcdefg$.

Some Supporting Concepts for Languages

- First of all, as languages are sets, all set operations apply.
  Union: $L_1 \cup L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ or } w \in L_2\}$.
  Intersection: $L_1 \cap L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ and } w \in L_2\}$.
  Difference: $L_1 \setminus L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ and } w \notin L_2\}$.
  Complement relative to Σ^*: $\overline{L} = \{w \in \Sigma^* \mid w \notin L\}$.
- Many string operation extend to languages in a natural way.
  Concatenation: $L_1 L_2 = L_1 \cdot L_2 = \{w_1 w_2 \mid w \in L_1 \text{ and } w \in L_2\}$.
  Reversal : $L^R = \{w^R \mid w \in L\}$.
- Star and plus on a single language:
  - $L^0 = \{\lambda\}$.
  - $L^1 = L$.
  - $L^{k+1} = L^k \cdot L$.
  - $L^* = \bigcup\{L^k \mid k \in \mathbb{N}\} = L^0 \cup L^1 \cup L^2 \ldots L^k \cup \ldots$.
  - $L^+ = \bigcup\{L^k \mid k \in \mathbb{N}^{>0}\} = L^1 \cup L^2 \ldots L^k \cup \ldots = L^* \setminus \{\lambda\}$.
- Note finally that Σ^+ is defined to be $\Sigma^* \setminus \{\lambda\}$.