Session Objectives

- Define the Chomsky hierarchy of grammar types
- Define the parse tree for a derivation
- Read and understand languages defined in EBNF

Chomsky Grammar Hierarchy

Type	Name	Restrictions on productions $\alpha \longrightarrow \beta$
0	phrase structure	α contains a non-terminal
1	context-sensitive	α contains a non-terminal and the length of α is less than or equal to the length of β
2	context-free	α consists of a single non-terminal
3	right-linear grammar	only producations of the form $\alpha \longrightarrow a\beta$ or $\alpha \longrightarrow a$ can be used where α and β are single non-terminals and a is a terminal

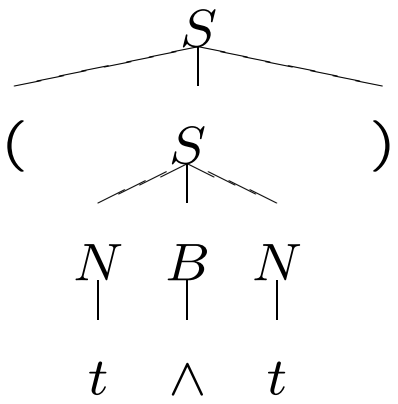
- The type 3 Chomsky grammars (right-linear grammars) are in fact just our old friend the regular languages again.
- The type 2 grammars (context-free grammars) can be parsed by a DFA supplied with a push-down stack; the resulting machine is called a push-down acceptor.

Context-Free Grammar: Example

- The grammar for Boolean expressions is a type 2 grammar since every production rule has a single non-terminal on the left-hand side.
- The main thing that makes this a non-regular grammar is the need to balance pairs of brackets (it's our old nemesis 0^n1^n in another guise), and that can be done with a stack, so the fact that this language can be parsed by a push-down acceptor makes sense.

Derivation Trees

- We include a vertex for each symbol, and use edges to denote production rules.
- The tree structure is one reason why it is reasonable to be able to write good parsing algorithms for context-free grammars.



Parser Generation

- Parser generators (or compiler compilers) are programs that take a context-free grammar as input and generate code for the corresponding parser.
- YACC (Yet Another Compiler Compiler), a standard Unix utility developed by Bell labs, and the Gnu version, wittily called Bison, generate C programs. The Java Compiler Compiler, JavaCC, originally developed by Sun Microsystems, can be downloaded free from `http://www.webgain.com/products/java_cc/`
- We can compare this with the situation for regular expressions; some programming languages (for example, Perl, Tcl, PHP) offer inbuilt support for parsing regular expressions.
- By contrast, for context-free grammars, parsers are more complicated, and are generated as code that must be integrated by the programmer with other parts of a system.

Backus–Naur Form

- Chomsky's work on grammars influenced John Backus, who realised that many programming languages are context free (or nearly so).
- He developed Backus-Naur Form (BNF) in order to specify the Algol 60 programming language.
- Since then, the grammar has been revised, and many languages are specified in Extended Backus-Naur Form (EBNF).

EBNF Grammars

- Each rule of an EBNF grammar defines one symbol of the form

$$\text{symbol} ::= \text{expression} \tag{1}$$

  In (1) both the symbol and the expression are non-terminals and represent syntactic categories.

- For example, in the Boolean grammar, N represents truth values and B represents logical operators.

- Terminal symbols (also known as string literals) are given inside quotes; this is useful in grammars that are to be read by a machine so that it can distinguish between the grammar and the language it generates.

EBNF Metacharacters

In a similar way to regular expressions, there are some metacharacters used in EBNF to make the representation more compact.

Symbol	Semantics
$A?$	A or nothing: 'optional A'
AB	A followed by B
$A \mid B$	A or B but not both
$A - B$	any string that matches A but does not match B
$A+$	one or more repetitions of A
$A*$	zero or more repetitions of A

Note that the or operator $\mid$ is an exclusive or.

Example: Boolean Expressions

The grammar defined earlier can be written in EBNF as follows:

$$S ::= \text{“(”}\, S \,\text{“)”}\, (B(\text{“(”}\, S \,\text{“)”} \mid N))? \mid N(B(\text{“(”}\, S \,\text{“)”} \mid N))?$$

$$N ::= \text{“0”} \mid \text{“1”}$$

$$B ::= \text{“}\wedge\text{”} \mid \text{“}\vee\text{”}$$

I think that it is arguable that the original context-free grammar was clearer than this!

Example: Logic

wffs in propositional logic

$$\text{formula} ::= \text{atomic formula} \mid (\neg \text{formula}) \mid (\text{formula} \wedge \text{formula}) \mid$$

$$(\text{formula} \rightarrow \text{formula}) \mid (\text{formula} \vee \text{formula})$$

$$\text{atomic formula} ::= \bot \mid p \mid q \mid r \mid p_0 \mid p_1 \mid \ldots$$

predicate calculus We defined a term in predicate logic by

- Any variable or constant symbol is a term.
- If f is an m-ary function and t_1, t_2, ..., t_m are terms, then $f(t_1, t_2, \ldots, t_m)$ is a term.

We can write this in EBNF as follows:

$$\text{term} ::= \text{variable} \mid \text{constant} \mid f_1(\text{term}) \mid f_2(\text{term}, \text{term}) \mid \ldots$$

$$\text{variable} ::= x \mid y \mid z \mid x_1 \mid \ldots$$

$$\text{constant} ::= 0 \mid 1 \mid \ldots$$

Example: XML

```
 1  <?xml version = "1.0"?>
 2  <!DOCTYPE letter SYSTEM "letter.dtd">
 3  <letter>
 4  <contact type = "to">
 5    <name>John Smith</name>
 6    <address>123 High St.</address>
 7    <postcode>B99 1AA</postcode>
 8  </contact>
 9
10  <contact type = "from">
11    <name>Jane Smith</name>
12    <address>321 Low St.</address>
13    <postcode>B11 9ZZ</postcode>
14  </contact>
15
16  <salutation>Dear John,</salutation>
17  <para>What time will you return?</para>
18  <para>Your dinner is in the oven.</para>
19  <closing>Yours faithfully</closing>
20  <signature>Jane</signature>
21  </letter>
```

- XML is rapidly becoming the standard for specifying the structure of data presented on the Web. As such, it is very likely that you will have to write your own XML language definitions.
- There are two ways of specifying XML document structure: Document Type Definitions (DTDs) and schemas.
- A DTD expresses the set of rules for document structure using an EBNF grammar.

DTD for Letter

```
 1   <!ELEMENT letter ( contact+, salutation, para+, closing, signature )>
 2   <!ELEMENT contact ( name, address, postcode )>
 3   <!ATTLIST contact type CDATA #IMPLIED>
 4
 5   <!ELEMENT name ( #PCDATA )>
 6   <!ELEMENT address ( #PCDATA )>
 7   <!ELEMENT postcode ( #PCDATA )>
 8   <!ELEMENT salutation ( #PCDATA )>
 9   <!ELEMENT para ( #PCDATA )>
10   <!ELEMENT closing ( #PCDATA )>
11   <!ELEMENT signature ( #PCDATA )>
```

- Each `ELEMENT` type defines an EBNF rule (with fairly obvious syntax). The `ATTLIST` element type declaration on line 3 defines an attribute (here `type`) for the `contact` element.
- The keyword `#IMPLIED` means that if the parser finds a `contact` element without a type attribute it can ignore the attribute.
- The type `CDATA` specifies character data.

Example: Programming Languages

In Ada, the rule to define an expression has the form

expression ::= relation{**and** relation}

| relation{**and then** relation} | relation{**or** relation}

| relation{**or else** relation} | relation{**xor** relation}

This depends on the definition of a relation:

relation ::=

simple_expression[relational_operator simple_expression]

| simple_expression[**not**]**in** range | simple_expression[**not**]**in** subtype_mark

Summary

1. Phrase structure grammars use rules to generate words.
2. Words are made up of terminal symbols; non-terminal symbols represent intermediate states or grammatical constructs.
3. Chomsky defined a four-level hierarchy of grammars. Regular languages are the smallest class, and the next most complex are context-free grammars.
4. A context-free language can be parsed by a DFA with a push-down stack.
5. Computer scientists use Extended Backus-Naur Form (EBNF) to express context-free grammars.
6. Compiler compilers generate a parser from a machine-readable context-free grammar.

Session Objectives

- Define the Chomsky hierarchy of grammar types
- Define the parse tree for a derivation
- Read and understand languages defined in EBNF