Chapter 3

Syntax

Topics

- Introduction
- English Description
- Syntax
- Regular Expressions
- BNF
- Variations of BNF
- Chomsky Hierarchy
- Parsing
- Ambiguity, associativity, and precedence.

Introduction

- Language definition
  - Syntax
  - Semantics
- Syntax
  - Form, format, well-formedness, and compositional structure of the language.
  - Description of the ways different parts of the language may be combined to form other parts.
- Semantics
  - Meaning and interpretation of the language.
  - Description of what happens during the exsecution of a program.

Introduction: English Description

- Early days
  - Syntax and semantics: lengthy English explanations and many examples
  - Example (Syntax): the `if`-statement in Pascal may be described in words:

    An if-statement consists of the word "if" followed by a condition, followed by the word "then", followed by a statement, followed by an optional else part consisting of the word "else" and another statement.
- Example (Semantics): the `if`-statement in Pascal may be described in words:

  An if-statement is executed by first evaluating its conditions. If the condition evaluates to true, then the statement following the "`then`" is executed. If the condition evaluates to false, and there is an else part, then the statement following the "else" is executed.

Syntax

- (Programming language) Definition of what constitutes a grammatically valid program in that language.
- Syntax is specified as a set of rules, just as it is for natural languages.
- Clear, concise, and formal definition syntax is especially important for programmers, implementers, language designer, etc.
- The syntax of a language has a profound effect on the ease of use of a language.

Definition of a Language

- ***Formal language**:* set of finite string of atomic symbols.
- ***Alphabet**:* set of symbols.
- ***Sentences**:* the strings that belong to the language.

Alphabet: {a,b}

L_1 = {a, b, ab}

L_2 = {aa, aba, abba, abbba, … }

This language is finite: ju

This language is infinite

- More precise methods for defining languages are desirable than just using "…".

Definition of a Regular Expression

- Invented by a mathematical logician Stephen Kleene
- A ***regular expression*** over A denotes a language with alphabet A and is defined by the following set or rules:
  - 1. **Empty**. The symbol Æ is a regular expression, denoting the language consisting of no strings {}.
  - 2. **Atom**. Any single symbol of *aÎA* is a regular expression denoting the language consisting of the single string {a}.
  - 3. **Alternation**. If r_1 is a regular expression and r_2 is a regular expression, then (r_1+r_2) is a regular expression. The language it denotes has all the strings from the language denoted by r_1 and all the strings from the language denoted by r_2.

Regular Expressions

- 4. **Concatenation**. If r_1 and r_2 are regular expressions, then $(r_1 \cdot r_2)$ is a regular expression. The language it denotes is the set of all strings formed by concatenating a string from the set denoted by r_1 to the end of a string in the set denoted by r_2.
- 3. **Closure**. If *r* is a regular expression, then *r** is a regular expression. The language it denotes consists of all strings formed by concatenating zero or more strings in the language denoted by *r*.

- - Plus, dot, asterisk, the empty set symbol, and parentheses are part of the notation for regular expressions, not part of the languages being defined.
- - Definitions are recursive.

Conventions for Writing Regular Expressions

Regular Expression	Meaning
x	A character (stand for itself)
“xyz”	A literal string (stands for itself)
M | N	M or N
M N	M followed by N (concatenation)
M^*	Zero or more occurrences of M
M^+	One or more occurrences of M
[a-zA-Z]	Any alphabetic character
[0-9]	Any digit
.	Any single charcater

Regular Expressions: Example

- Sequences of ASCII characters make up a legal identifier
  - *l* stand for the regular expression denoting any lowercase or uppercase letter
  - *d* stand for the regular expression denoting any decimal digit.
- Modula-3 *l x (l+d+_)**
- ML *l x (l+d+_+’)**
- Ada *l x (l+d)* x (_ x (l+d) x (l+d)*)**

Regular Expressions

- Very popular tool in language design.
- Generic lexical-analyzer generator:
  - The regular expression is submitted directly.
  - Two commonly used generators:
    - “Lex” (generating C code).
    - “JIex” (for generating Java code).
- What is the problem with regular expressions?
- Why don't we use them to describe the syntax of programming languages?
  - Major shortcoming: bracketing is not expressible
  - Regular expressions are incapable of generating the language $\{a^n b^n\}$ where the number of *as* must be equal to the number of *bs*, very useful for matching nested beginning and ending tags, such as occur in expressions (parentheses) and statement lists (braces).

Formal Methods of Describing Syntax: BNF

- ***Metalanguage:*** is a language used to define other language.
- BNF: notation invented to describe the syntax of ALGOL 60
  - Backus-Naur Form in honor of John Backus and Peter Naur, who developed the notation of this metalanguage in unrelated research efforts.
  - ***Language:*** a sequence of tokens.
  - ***Tokens:*** identifiers, numbers, keywords, punctuation, etc. that constitute the lexicon of the language.

BNF: Grammar

- ***BNF grammar:*** a set of rewriting rules.
  - A a left-hand side: syntactic categories
    - A ***syntactic category*** is a name for a set of token sequences
  - A right-hand side: sequences of tokens and syntactic categories.

  <name> ::= sequence of tokens and syntactic categories
  - There may be many rules with the same left-hand side.
- A token sequence belongs to a syntactic category if it can be derived by taking the right-hand sides of rules for the category and replacing the syntactic category occurring in right-hand side with any token sequence belonging to that category.

BNF: Notation

- A BNF definition typically contains the following meta-symbols:
  - "::=" meaning "is defined to be"
  - "<>" to delimit syntactic categories
  - "|" meaning "or"

  Strictly speaking, the symbol for "or" is not necessary, but it is convenient for combining multiple right-hand sides for the same syntactic category.

BNF: Examples (1)

- Describe the syntax of regular expressions over the alphabet *{a,b}*

  *<RE> ::= Æ| a| b| (<RE>+<RE}) | (<RE>x<RE>) | <RE>**

  - *The syntactic category of regular expressions is defined to be either the symbol Æ, or the symbol a, or the symbol b, or an opening parenthesis, followed by a regular expression, followed by a plus sign followed by another regular expression followed by a closing parenthesis, and so on. The set of tokens used in this definition is { Æ,a,b,(,),+,x,*}*

BNF: Examples (2)

- Describe the ALGOL 60 `for` construct.

  <for statement> ::= <for clause> <statement>
  | <label> : <for statement>
  <for clause> ::= ***for*** *<variable> := <for list>* ***do***
  <for list> ::= <for list element>
  | <for list> , <for list element>
  <for list element> ::= <arith expr>
  | <ariht expr> ***step*** *<arith expr>* ***until*** *<arith expr>*
  | <arith expr> ***while*** *<boolean exp>*

- Sample `for` statements in ALGOL 60

  for *i := j* ***step*** *1* ***until*** *n* ***do*** *<statement>*
  A: B: ***for*** *k := 1* ***step*** *-1* ***until*** *n, i+1* ***while*** *j>1* ***do*** *<statement>*

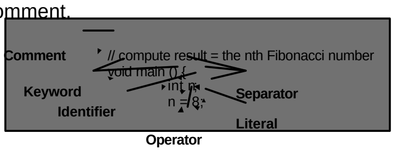
BNF: Examples (3)

- Describe simple integer arithmetic expressions with addition and multiplication.

  *<exp> ::= <exp> + <exp> | <exp> * <exp>*
  | (<exp>) | <number>
  <number> ::= <number> <digit> | <digit>
  <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- *Lexicon of a programming language contains the grammatical categories:*
  - *Identifier (variable names, function names, etc)*
  - *Literal or constants (integer and decimal numbers)*
  - *Operator (+,-,*,/,etc)*
  - *Separator (;,.,{,},etc)*
  - *Keyword or reserved words (int, main, if, for, etc)*

BNF: A Stream of Tokens

- Previous categories allow the compiler to look at a program as a stream of tokens.
  - Each one a member of a particular grammatical category
  - Separated from the next token by whitespaces or a comment.



```
// compute result = the nth Fibonacci number
void main () {
    int n;
    n = 8;
```

Variations of BNF: EBNF

- Several extensions to BNF have been proposed to make BNF definitions more readable.
  - Improve the clarity of syntax description and the efficiency of syntax analysis.
  - Do not add to the expressive power of the formalism, just to the convenience.
- **Extended BNF** (EBNF for short) was introduced to simplify the specification of recursion in grammar rules (curly brackets), and to introduce the idea of optional part in a rule's right-hand side (square brackets).

EBNF: Example

- Describe simple integer arithmetic expressions with addition and multiplication.

  <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  <number> ::= <digit> {<digit}
  <exp> ::= <term> {+ <term>} | <term> {- <term>}

  The curly brackets "{ }" denote zero or more repetitions.
  The square brackets enclose a series of alternatives from which one must choose.

- Definitions of language syntax in EBNF tend to be slightly clearer and briefer than BNF definitions.
- EBNF does not force the use of recursive definitions on the reader in very instance.

EBNF: Example

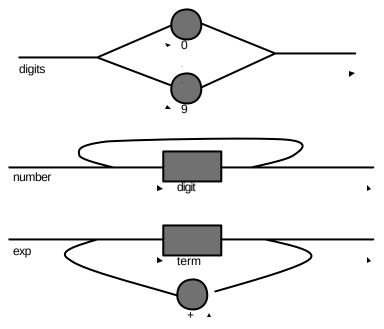
- The Ada reference manual use extended BNF.
  - Uses different convention to distinguish syntactic categories from terminals.
  - Syntactic categories are denoted by simple identifiers possibly containing underscores.
  - Keywords and punctuation are in bold face.

```
block ::=    [ block_identifier: ]
             [ declare {declaration} ]
             begin statement {statement}
             [ exception handler {handler} ]
             End [ block_identifier ];
```

Variations of BNF: Syntax Diagrams

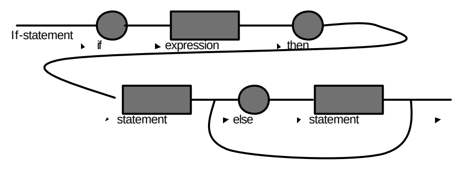
- Graphical representation that indicates the sequence of terminals and nonterminals encountered in the right-hand side of the rule.
  - Circles or ovals for terminals
  - Squares or rectangles for nonterminals
  - Connected with lines and arrows to indicate appropriate sequencing.
- Syntax diagrams can also condense several productions into one diagram.

Syntax Diagrams (1)



Syntax Diagrams (2)

- As an example of how to express the square brackets in syntax diagrams: the Pascal `if`-statement.



- Syntax diagrams are always written from the EBNF, not the BNF

Definition of a Grammar

- BNF notation is slightly different form of what are called context-free grammars. These grammars and others were developed independently by Chomsky.
- Grammar: a 4-tuple *<T,N,P,S>*
  - *T* is the set of terminal symbols
  - *N* is the set of nonterminal symbols *TÇN=Æ*
  - *S*, a nonterminal, is the start symbol
  - *P* are the productions of the grammar
    - A production has the form *a ? ß* where *a* and *ß* are strings of terminals and nonterminals *(a ¹ Î).*

Chomsky Hierarchy

- In the mid 1950s, Chomsky described a hierarchy that relates the power of different types of grammars.
  - ***Type-0 grammars (unrestricted grammars)*** include all formal grammars. They generate exactly all languages that can be recognized by a Turing Machine.
  - ***Type-1 grammars (context-sensitive grammars).*** These grammars have rules of the form $\alpha A\beta\rightarrow\alpha\gamma\beta$ with A a nonterminal and a, ß and ? strings of terminals and nonterminals. The strings a and ß may be empty, but ? must be nonempty. The rule $S\rightarrow\in$ is allowed if *S* does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognized by a non-deterministic Turing machine whose tape is bounded by a constant times the length of the input.

Chomsky Hierarchy

- ***Type-2 grammars (context-free grammars)*** generate the context-free languages. These are defined by rules of the form $A\rightarrow\gamma$ with *A* a nonterminal and ? a string of terminals and nonterminals. Context free languages are the theoretical basis for the syntax of most programming languages.
- ***Type-3 grammars (regular grammars)*** generate the regular languages. Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed (or preceded, but not both in the same grammar) by a single nonterminal. The rule is also here allowed if *S* does not appear on the right side of any rule. This family of formal languages can be obtained by regular expressions. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.

Parsing

- ***Parsing problem***: the interesting problem concerning grammars is how to efficiently recognize when a string is a sentence of a grammar.
- BNF: a simple arithmetic expression grammar

*<exp> ::= <exp> + <exp> | <exp> * <exp>*
| (<exp>) | <number>
<number> ::= <number> <digit> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Parsing: Example

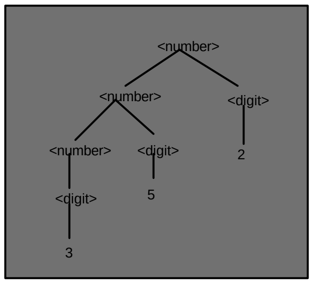
- Justify *352* as a *Number.*
  - Derive the string from the rules in a sequence of steps.
  - Begin with the start symbol *S=number*

1. Form the string *Number Digit* as a particular kind of *Number*, from the first alternative in the second rule.
2. Substitute *Number Digit* for *Number* in the string, again using the second rule, gaining the string *Number Digit Digit*.
3. Substitute *Digit* for *Number*, using the second alternative in the second rule, gaining *Digit Digit Digit*.
4. Substitute *3* as a particular kind of *Digit* from the third rule, achieving *3 Digit Digit*.
5. Substitute *5* for *Digit* in the string, achieving the string *3 5 Digit*.
6. Finally, substitute *2* for *Digit* in the string, achieving *3 5 2*

Parse Tree

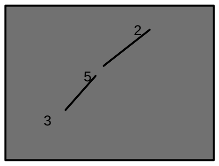
- We just parse the string *352* as an instance of the grammatical category *Integer.*
- Parsing process used in the design and analysis of programming language syntax.
  - Clearer style than English for expressing a sequence of steps.
  - Describe the parse graphically in the form of a ***parse tree*.**
- The parse tree is labeled by nonterminals at interior nodes and terminals at leaves.

Parse Tree: Example



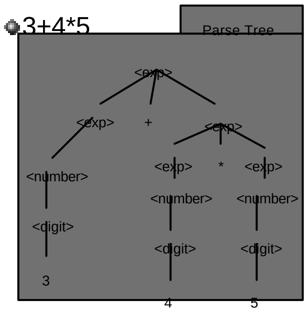
Abstract Syntax Tree

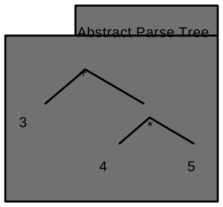
- Not all the terminals and nonterminals may be necessary to determine completely the syntactic structure of an expression.
- Example: the structure of the number 352



Abstract Syntax Tree: Example

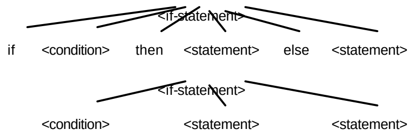
- 3+4*5





Abstract Syntax Tree

- Abstract syntax tree may also do away with terminals that are redundant once the structure of the tree is determined.
- Example:
  - *<if-statement> ::= if <condition> then <statement> else <statement>*

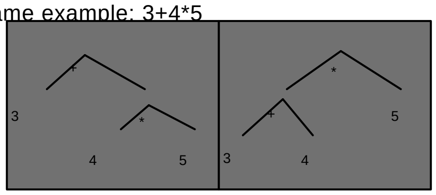


Ambiguity

- Parsing natural languages is difficult due to the ambiguity of language.
- ***Ambiguity***: the sentence can be understood in two different ways (two or more different parse trees).
- Either the grammar must be revised to remove the ambiguity, or a disambiguating rule must be stated to establish which structure is meant.

Ambiguity: Example

- Same example: 3+4*5



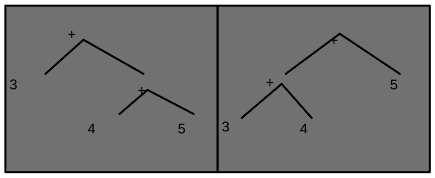
- Which of the two parse trees is the correct one for the expression 3+4*5?

Ambiguity: Resolution

- If operations are applied in a different order then the resulting semantics are quite different
  - First syntax tree: 23
  - Second syntax tree: 35
  - Meaning from mathematics: choose first tree, since multiplication has precedence over addition.
- State a disambiguation rule separately from the grammar or revise the grammar.
  - Usual way to revise the grammar is to write a new grammar rule that establishes a "precedence cascade" to force the matching of the "*" at a lower point in the parse tree.
  - *<exp> ::= <exp>* + *<exp>* | *<term>*
  - *<term> ::= <term>* * *<term>* | *(<exp>)* | *<number>*

Associativity

- There is still some ambiguity problem:
  - Rule for *<exp>* still allows to parse *3+4+5* as either *(3+4)+5* or *3+(4+5).*
    - Addition is either *right or left-associative.*



Associativity: Example (1)

- In the case of addition this does not affect the result.
- In the case of subtraction it surely would: 8-4-2=2 if minus is left-associative, but 8-4-2=6 if minus is right-associative.
- Replace rule

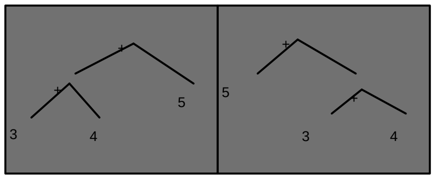
  <exp> ::= <exp> + *<exp>*

  with *<exp> ::= <exp>* + *<term>* or *<exp> ::= <term>* + *<exp>*

  The first rule is left-recursive while the second is right-recursive.

Associativity: Example (2)

- A left recursive rule for an operation causes it to left associate, while a right-recursive rule causes it to right-associate.



Ambiguity

- The BNF for simple arithmetic expressions is now unambiguous.
- Sometimes the process of rewriting a grammar to eliminate ambiguity causes the grammar to become extremely complex, and in such cases we prefer to state a disambiguation rule.

Dangling *else* problem

- A classical example of ambiguity in programming languages.
- Occurs when two adjacent *if* statements are followed by an *else* statement.

```
if (x<0)
    if (y<0) y = y-1;
    else y=0;
```

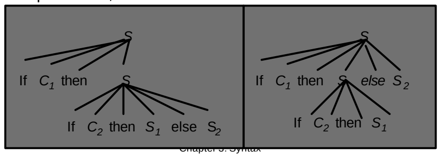
- Parse attaches the else clause to the second if statement
  - y will become 0 whenever x<0 and y>=0.
- Parse attaches the else clause to the first if statement
  - y will become 0 whenever x>=0.

Dangling *else* problem

- ALGOL 60 introduced the *if-then* and the *if-then-else* statements

  S ? ***if*** *C* ***then*** *S* | ***if*** *C* ***then*** *S* ***else*** *S* | *S’*

  - *Sequence of tokens:* ***if*** C_1 ***then*** S_1 ***else if*** C_2 ***then*** S_2 ***else*** S_3 *has only one interpretation.*
  - *Sequence of tokens:* ***if*** C_1 ***then if*** C_2 ***then*** S_1 ***else*** S_2 has two interpretations,



Dangling *else* problem: Solutions

- ALGOL 60:
  - Prohibited the nested *if* statement, as it could always be avoided by using the *begin/end* statement.
- PL/I and Pascal:
  - Adopted the solution of matching dangling *else* to the nearest unmatched *if* statement.
- ALGOL 68:
  - introduced the keyword *fi*.
  - Ada solves the problem with *end if*.
- “Terminating keyword” solution appears to be generally favored over the “nearest unmatched” solution in more recent programming languages.

Dangling *else* problem: Solutions

- Java solves the problem by expanding the BNF grammar for if statements in a rather bizarre way.
  - Separates the definition into two different syntactic categories, (*IfThenStatement,IfThenElseStatement)*, each which is a subcategory of the general category *Statement*.

    IfThenStatement ? if (Expression) Statement

    IfThenElseStatement ? if (Expression) StatementNoShortIf else Statement

Variations on BNF and EBNF

- In place of the arrow, a colon is used and the RHS is placed on the next line.
- Instead of a vertical bar to separate alternative RHSs, they are simply placed on separate lines.
- In place of square brackets to indicate something being optional, the subscript $_{opt}$ is used.
- Rather than using the | symbol in a parenthesized list of elements to indicate a choice, the words “one of” are used.

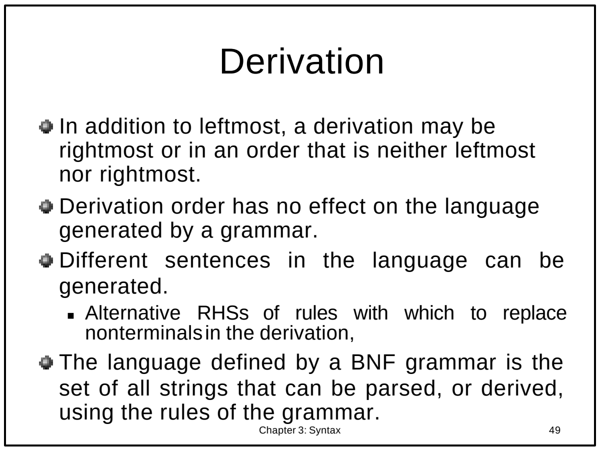
Derivation

- Method for describing the parse of a string.
- A ***derivation*** is a simple linear representation of a parse tree
  - more helpful when the string being derived has a simple grammatical structure
- Example: derivation of 352

  Number ⇒ Number Digit ⇒ Number Digit Digit ⇒ Digit Digit Digit ⇒ 3 Digit Digit ⇒ 3 5 Digit ⇒ 3 5 2

Derivation

- ***Sentential form*: each string on the right of a double arrow.**
  - **Contains terminal and nonterminals symbols.**
  - **Left end of the derivation is the start symbol *S***
  - **Each intermediate step creates a sentential form**
    - **Results from replacing the left-most nonterminal symbol by a string of terminals and nonterminals that appears on the right-hand side of some rule that has the same symbol on its left-hand side.**
  - **Derivations that use this order of replacement are called leftmost derivations.**



Derivation

- In addition to leftmost, a derivation may be rightmost or in an order that is neither leftmost nor rightmost.
- Derivation order has no effect on the language generated by a grammar.
- Different sentences in the language can be generated.
  - Alternative RHSs of rules with which to replace nonterminals in the derivation,
- The language defined by a BNF grammar is the set of all strings that can be parsed, or derived, using the rules of the grammar.