COMPILER CONSTRUCTION

Principles and Practice

Kenneth C. Louden

3. Context-Free Grammars and Parsing

PART TWO

Contents

3.5 Extended Notations: EBNF and Syntax Diagrams

3.5.1 EBNF Notation

Special Notations for *Repetitive Constructs*

- **Repetition**
  - $A \rightarrow A\ \alpha \mid \beta$ **(left recursive), and**
  - $A \rightarrow \alpha\ A \mid \beta$ **(right recursive)**
    - **where α and β are arbitrary strings of terminals and non-terminals, and**
  - **In the first rule β does not begin with A and**
  - **In the second β does not end with A**

Special Notations for *Repetitive Constructs*

- **Notation for repetition as regular expressions use, the asterisk * .**

  $A \rightarrow \beta\, \alpha^*$ **, and**

  $A \rightarrow \alpha^*\, \beta$

- **EBNF opts to use curly brackets {. . .} to express repetition**

  $A \rightarrow \beta\{\alpha\}$ **, and**

  $A \rightarrow \{\alpha\}\, \beta$

- **The problem with any repetition notation is that it obscures how the parse tree is to be constructed, but, as we have seen, we *often* do not care.**

Examples

- **Example: The case of statement sequences**
- **The grammar as follows, in right recursive form:**

  stmt-Sequence $\rightarrow$ ***stmt*** **;** ***stmt-Sequence* / *stmt***

  stmt $\rightarrow$ **s**

- **In EBNF this would appear as**

  stmt-sequence $\rightarrow$ **{** ***stmt*** **; }** ***stmt*** **(right recursive form)**

  stmt-sequence $\rightarrow$ ***stmt*** ***{*** **;** ***stmt}*** **(left recursive form)**

Examples

- **A more significant problem occurs when the associativity matters**

  exp $\rightarrow$ ***exp addop term* | *term***

  exp $\rightarrow$ ***term { addop term }***

  (imply left associativity)

  exp $\rightarrow$ **{*term addop }* *term***

  (imply right associativity)

Special Notations for *Optional Constructs*

- **Optional construct are indicated by surrounding them with square brackets [...].**
- **The grammar rules for if-statements with optional else-parts would be written as follows in EBNF:**

  statement $\rightarrow$ ***if-stmt*** **/** other

  if-stmt $\rightarrow$ if (***exp***) ***statement*** [else ***statement***]

  exp $\rightarrow$ **0 | 1**

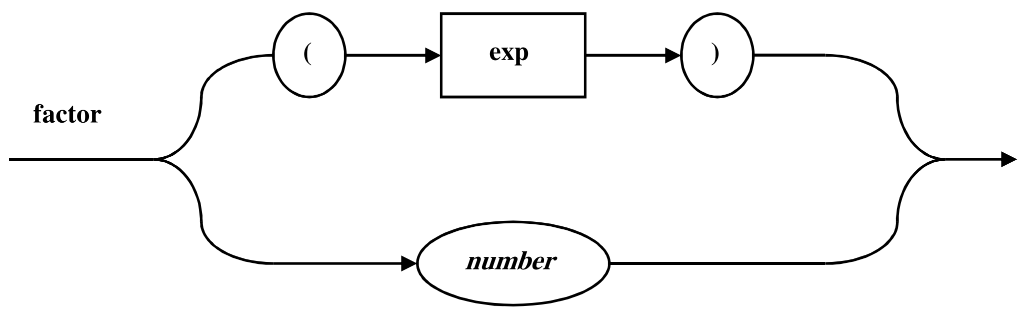
- ***stmt-sequence*** $\rightarrow$ ***stmt*****;** ***stmt-sequence*** **/** ***stmt*** **is written as**
- **stmt-sequence** $\rightarrow$ **stmt [; stmt-sequence]**

3.5.2 Syntax Diagrams

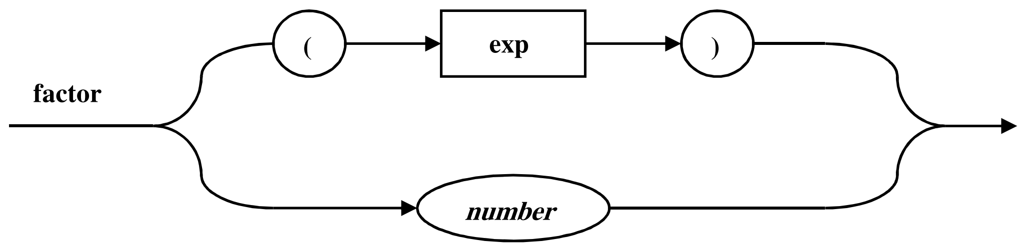
Syntax Diagrams

- **Syntax Diagrams:**
  - **Graphical representations for visually representing EBNF rules.**
- **An example: consider the grammar rule**

  ***factor*$\rightarrow$(*exp)* / number**
- **The syntax diagram:**



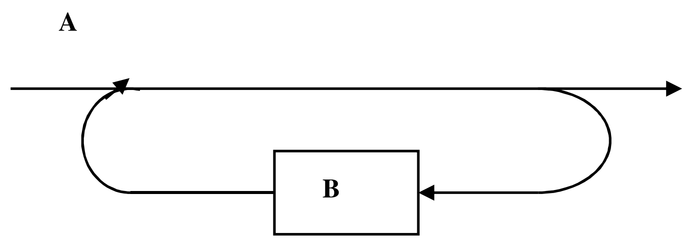
Syntax Diagrams



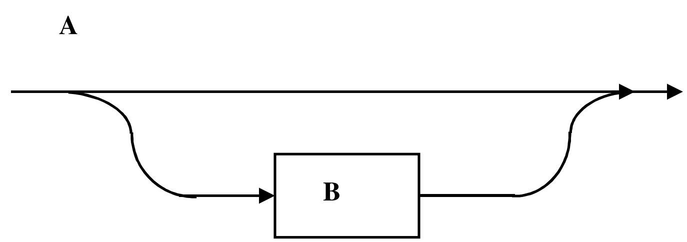
- Boxes representing terminals and non-terminals.
- Arrowed lines representing sequencing and choices.
- Non-terminal labels for each diagram representing the grammar rule defining that Non-terminal.
- A round or oval box is used to indicate terminals in a diagram.
- A square or rectangular box is used to indicate non--terminals.

Syntax Diagrams

- **A repetition : A $\rightarrow$ {B}**



- **An optional : *A* $\rightarrow$ *[B]***



Examples

- **Example: Consider the example of simple arithmetic expressions.**

  exp $\rightarrow$ ***exp addop term*** **/** ***term***

  addop $\rightarrow$ **+ | -**

  term $\rightarrow$ ***term mulop factor*** **/** ***factor***

  mulop $\rightarrow$ *******

  factor $\rightarrow$ **(** ***exp*** **) |** ***number***

- **This BNF includes associativity and precedence**

Examples

- **The corresponding EBNF is**

  ***exp* $\rightarrow$ *term* { *addop term* }**

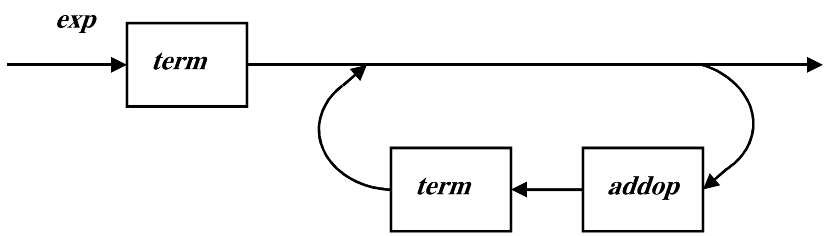
  ***addop*$\rightarrow$ + | -**

  ***term* $\rightarrow$ *factor* { *mulop factor* }**

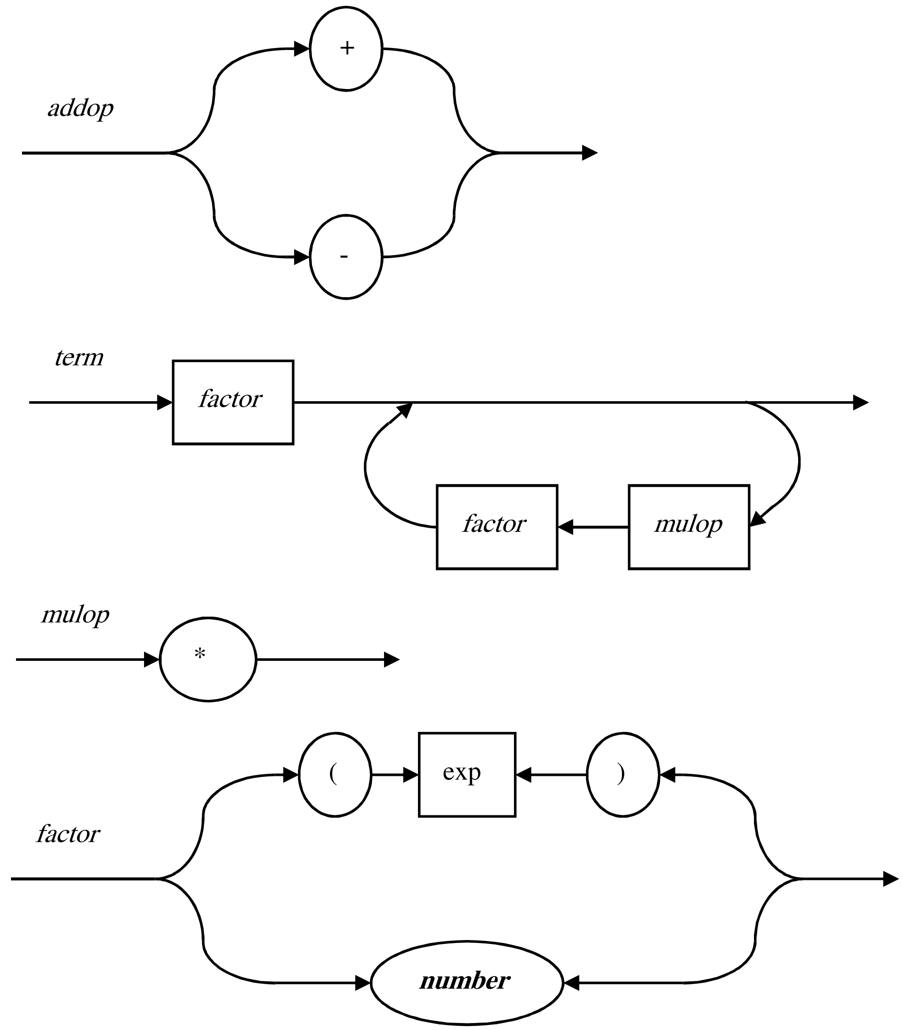
  ***mulop*$\rightarrow$ ***

  ***factor* $\rightarrow$ (*exp*) / *numberr* ,**

- **The corresponding syntax diagrams are given as follows:**



Examples



Examples

- **Example: Consider the grammar of simplified if-statements, the BNF**

  Statement* → *if-stmt* | *other

  if-stmt* → if (*exp*) *statement

  | if (*exp*) *statement* else *statement*

  ***exp* → 0 | 1**

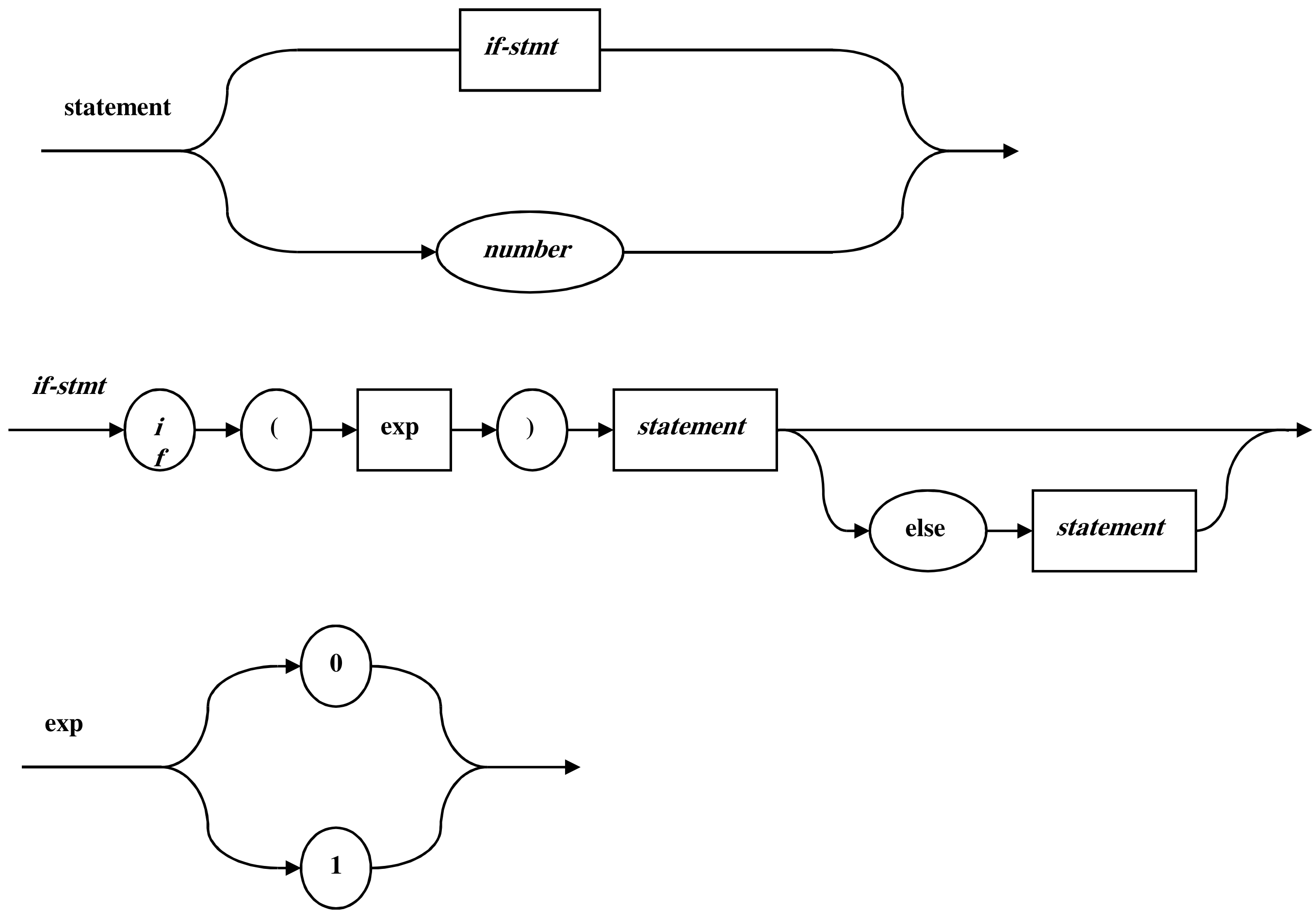
- **and the EBNF**

  statement* → *if-stmt* | *other

  if-stmt* → if (exp) *statement [* else *statement]

  ***exp* → 0 | 1**

The corresponding syntax diagrams are given in following figure.



3.6 Formal Properties of Context-Free Language

3.6.1 A Formal Definition of Context-Free Language

Definition

- **Definition: A context-free grammar consists of the following:**
  1. **A set T of terminals.**
  2. **A set *N* of non-terminals (disjoint from *T*).**
  3. **A set *P* of productions, or grammar rules, of the form $A \rightarrow a$, where *A is an* element of *N* and *a* is an element of $(T \cup N)^*$ (a possibly empty sequence of ter-minals and non-terminals).**
  4. **A start symbol *S* from the set *N*.**

Definition

- Let G be a grammar as defined above,

  $G = (T, N, P, S)$.

- A derivation step over G is of the form

  $a\ A\ \gamma \Rightarrow a\ \beta\ \gamma$,

  Where a and γ are elements of $(T \cup N)^*$, and

  $A \rightarrow \beta$ is in P.

- The set of symbols:
  - The union $T \cup N$ of the sets of terminals and non-terminals
- A sentential form:
  - a string a in $(T \cup N)^*$.

Definition

- **The relation α =>* β is defined to be the transitive closure of the derivation step relation =>;**
  - **tα =>* β if and only if there is a sequence of 0 or more derivation steps (n >= 0)**
    - **α_1=> α_2 =>…=> α_{n-1}=> α_n**
    - **such that α =-α_1, and β = α_n**

  (If n = 0, then α =β)

Definition

- A derivation over the grammar G is of the form
  - $S \Rightarrow^* w$,
  - where $w \in T^*$ (i.e., w is a string of terminals only, called a **sentence**), and S is the start symbol of G
- The **language generated by G**, written L(G), is defined as the set
  - $L(G) = \{ w \in T^* \mid \text{there exists a derivation } S \Rightarrow^* w \text{ of } G \}$.
  - L(G) is the set of sentences derivable from S.

Definition

- A l**eftmost derivation** S =>*lm w
  - is a derivation in which each derivation step
  - $a\, A\, \gamma \Rightarrow a\, \beta\, \gamma$,
  - is such that $a \in T^*$; that is, a consists only of terminals.
- A **rightmost derivation** is one in which each derivation step
  - $a\, A\, \gamma \Rightarrow a\, \beta\, \gamma$
  - has the property that $\gamma \in T^*$.

Parse Tree over Grammar G

A rooted labeled tree with the following properties:

1. Each node is labeled with a terminal or a non-terminal or ε.
2. The root node is labeled with the start symbol S.
3. Each leaf node is labeled with a terminal or with ε.
4. Each non-leaf node is labeled with a non-terminal.
5. If a node with label $A \in N$ has n children with labels $X1, X2, \ldots, Xn$

   (which may be terminals or non-terminals),

   then $A \rightarrow X1X2 \ldots Xn \in P$ (a production of the grammar).

Role of Derivation

- Each derivation gives rise to a parse tree.
- **In general, many derivations may give rise to the same parse tree.**
- **Each parse tree has a unique leftmost and right-most derivation that give rise to it.**
- The leftmost derivation corresponds to a preorder traversal of the parse tree.
- The rightmost derivation corresponds to the reverse of a postorder traversal of the parse tree.

CFG & Ambiguous

- A set of strings L is said to be a **context-free language**
  - if there is context-free grammar G such that $L = L$ (G).
- A grammar G is **ambiguous**
  - if there exists a string w $\in$ $L(G)$ such *that w* has two distinct parse trees (or leftmost or rightmost derivations).

3.6.2 Grammar Rules as Equations

Meaning of Equation

- The grammar rules use the arrow symbol instead of an equal sign to represent the definition of names for structures (non-terminals)
- Left and right-hands sides still hold equality in some extents, but the defining process of the language that results from this view is different.
- Consider, for example, the following grammar rule, which is extracted (in simplified form) from our simple expression grammar:
  - $exp \rightarrow exp + exp \mid$ ***number***

Rules as Equation

- A non-terminal name like *exp* defines a set of strings of terminals, called E;
  - (which is the language, of the grammar if the non-terminal is the start symbol).
- let **N** be the set of natural numbers;
  - (corresponding to the regular expression name ***number****).*
- Then, the given grammar rule can be interpreted as the set equation
  - $\mathbf{E = (E + E) \cup N}$
- This is a recursive equation for the set E:
  - $\mathbf{E = N \cup (N+N) \cup (N+N+N) \cup (N+N+N+N) \ldots\ldots}$

3.6.3 Chomsky Hierarchy and Limits of Context-Free Syntax

The Power of CFG

Consider the definition of a number as a sequence of digits using regular expressions:

***digit* = 0|1|2|3|4|5|6|7|8|9**

number* = *digit digit*

Writing this definition using BNF, instead, as

***Digit* → 0 |1|2|3|4|5|6|7|8|9**

number* → *number digit |digit

Note that the recursion in the second rule is used to express repetition only.

Regular Grammar

- A grammar is called **a regular grammar**
  - The recursion in the rule is used to express repetition only
  - Can express everything that regular expressions can
  - Can design a parser accepting characters directly from the input source file and dispense with the scanner altogether.
- A parser is a more powerful machine than a scanner but less efficient.
  - The grammar would then express the complete syntactic structure, including the lexical structure.
- The language implementers would be expected to extract these definitions from the grammar and turn them into a scanner.

Context Rules

- **Free of context rule**:
  - Non-terminals appear by themselves to the left of the arrow in context-free rules.
  - A rule says that A may be replaced regardless of where the A occurs.
- **Context-sensitive grammar rule**:
  - A rule would apply only if β occurs before and γ occurs after the non-terminal.
  - We would write this as
    - $\beta\, A\, \gamma \Rightarrow \beta\, a\, \gamma,\ (a \neq \varepsilon)$
- Context-sensitive grammars are more powerful than context-free grammars
  - but are also much more difficult to use as the basis for a parser.

Requirement of a Context-Sensitive Grammar Rule

- The C rule requires declaration before use
  - **First**: Include the name strings themselves in the grammar rules rather than include all names as identifier tokens that are indistinguishable.
  - **Second**: For each name, we would have to write a rule establishing its declaration prior to a potential use.
- In many languages, the length of an identifier is unrestricted,
  - The number of possible identifiers is (at least potentially) infinite.
- Even if names are allowed to be only two characters long,
  - The potential for hundreds of new grammar rules. Clearly, this is an impossible situation.

Solution like a Disambiguating Rule

- State a rule (declaration before use) that is not explicit in the grammar.
  - Such a rule cannot be enforced by the parser itself, since it is beyond the power of (reasonable) context-free rules to express.
- This rule becomes part of semantic analysis
  - Depends on the use of the symbol table (which records which identifiers have been declared).
- The **static semantics** of the language include type checking (in a statically typed language) and such rules as declaration before use.
- Regard as *syntax* only those rules that can be expressed by BNF rules. Everything else we regard as semantics.

Unrestricted Grammars

- More general than the context-sensitive grammars.
  - It have grammar rules of the form

    $\alpha \rightarrow \beta$,
- where there are no restrictions on the form of the strings *a* and β (except that *a* cannot be β)

Types of Grammars

- The language classes they construct are also referred to as the **Chomsky hierarchy**,
    - after Noam Chomsky, who pioneered their use to describe natural languages.

  type 0： unrestricted grammar, equivalent to Turing machines

  type 1： context sensitive grammar

  type 2： context free grammar, equivalent to pushdown automaton

  type 3： regular grammar , equivalent to finite automata
- These grammars represent distinct levels of computational power.

3.7 Syntax of the TINY language

3.7.1 A Context-Free Grammar for TINY

Grammar of the TINY language in BNF(1)

- *program* → *stmt-sequence*
- *stmt-sequence*→ *stmt-sequence*; *statement* | *statement*
- *statement* → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*
- *if-stmt*→ **if** *exp* **then** *stmt-sequence* **end**
- | **if** *exp* **then** *stmt-sequence* **else** *stmt-sequence* **end**
- *repeat-stmt*→ **repeat** *stmt-sequence* **until** *exp*
- *assign-stmt*→ **identifier :=** *exp*
- *read-stmt* → **read identifier**
- *write-stmt* → **write** *exp*

Grammar of the TINY language in BNF(2)

- *exp* → *simple-exp comparison-op simple-exp* | *simple-exp*
- *comparison-op* → **<** | **=**
- *simple-exp* → *simple-exp addop term* | *term*
- *addop* → **+**|**-**
- *term* → *term mulop factor factor* | *factor*
- *mulop* → *****|**/**
- *factor* → **(***exp***)** |**number** |**identifier**

3.7.2 Syntax Tree Structure for the TINY Compiler

P134-138

End of Part Two

THANKS