Program Syntax

In Text: Chapter 3 & 4

N. Meng, F. Poursardar

Overview

- Basic concepts
  - Programming language, regular expression, context-free grammars
- Lexical analysis
  - Scanner
- Syntactic analysis
  - Parser

Basic Definitions

- **Syntax**—the form or structure of the expressions, statements, and program units
- **Semantics**—the meaning of the expressions, statements, and program units

- Why write a language definition; who will use it?
  - Other language designers
  - Implementers (compiler writers)
  - Programmers (the users of the language)

What is a "Language"?

- A **sentence** is a string of characters over some alphabet
- A **language** is a set of sentences
- A **lexeme** is the lowest level syntactic unit of a language (e.g., *, sum, begin)
- A **token** is a category of lexemes (e.g., identifier)

Lexemes	*Tokens*
`index`	identifier
`=`	equal_sign
`2`	int_literal

Natural Languages Are Ambiguous

- "I saw a man on a hill with a telescope"
- Programming languages should be precise and unambiguous
  - Both programmers and computers can tell what a program is supposed to do

Recognizers vs. Generators

- We don't want to use English to describe a language (too long, tedious, imprecise), so …
- There are *two formal approaches to describing syntax*:
  - Recognizers
    - Given a string, a recognizer for a language L tells whether or not the string is in L (e.g.: Compiler – syntax analyzer)
  - Generators
    - A generator for L will produce an arbitrary string in L on demand. (e.g.: Grammar, BNF)
  - Recognition and generation are useful for different things, but are closely related

Programming Language Definition

- Syntax
  - To describe what its programs look like
  - Specified using **regular expressions** and **context-free grammars**
- Semantics
  - To describe what its programs mean
  - Specified using axiomatic semantics, operational semantics, or denotational semantics

Grammars

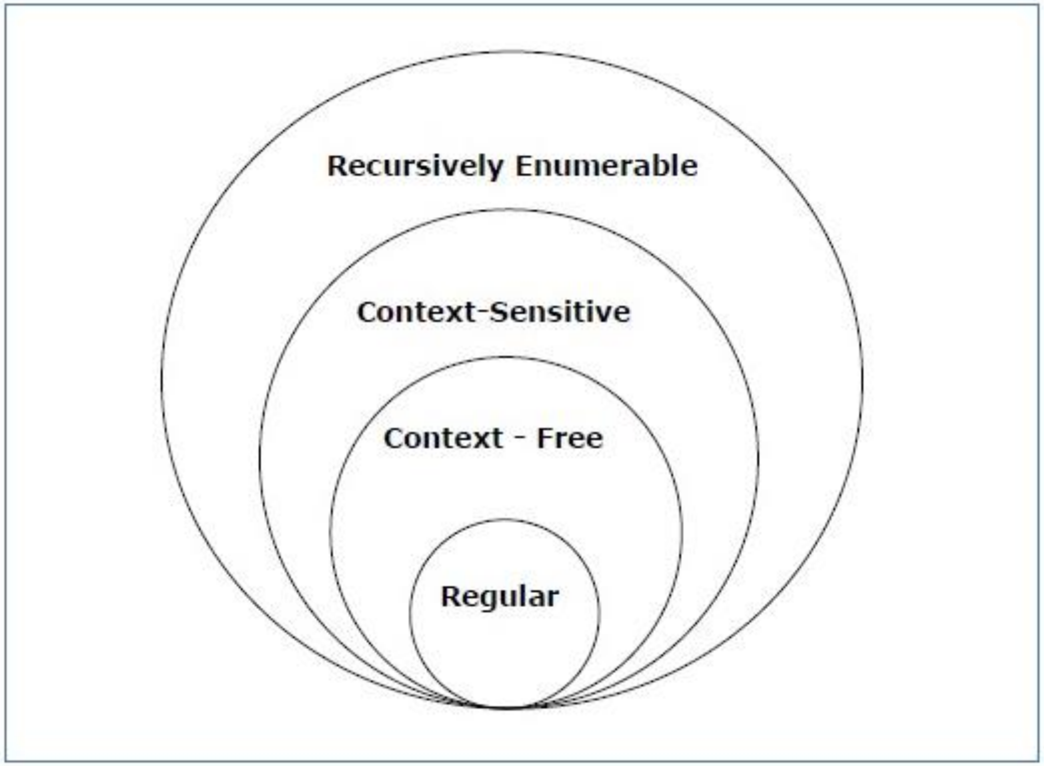
- Developed by Noam Chomsky in the mid-1950s
- 4-level hierarchy (0-3)
- Language generators, meant to describe the syntax of natural languages
- **Context-free** grammars define a class of languages called **context-free languages** (level 2)

Chomsky Classification of Grammars

Grammar Type	Grammar Accepted	Automaton
Type 0	**Unrestricted** grammar	Turing Machine
Type 1	Context-**sensitive** grammar	**Linear-bounded** automaton
Type 2	Context-free grammar	**Pushdown** automaton
Type 3	Regular grammar	**Finite** state automaton

Chomsky Classification of Grammars

The following illustration shows the scope of each type of grammar:



Type-2 grammars

- **Type-2 grammars** generate context-free languages.
- The productions must be in the form $\mathbf{A} \rightarrow \boldsymbol{\gamma}$
- where $\mathbf{A} \in \mathbf{N}$ (Non terminal)
- and $\boldsymbol{\gamma} \in \mathbf{(T \cup N)^*}$ (String of terminals and non-terminals).
- These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

- Example:

$S \rightarrow Xa$
$X \rightarrow a$
$X \rightarrow aX$
$X \rightarrow abc$
$X \rightarrow \varepsilon$

Regular Expressions

- A regular expression is one of the following:
  - A character
  - The empty string, denoted by ε
  - Two or more regular expressions concatenated
  - Two or more regular expressions separated by | (or)
  - A regular expression followed by the Kleene star (concatenation of zero or more strings)

Regular Expressions

- The pattern of numeric constants can be represented as:

$$digit \longrightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$unsigned\_integer \longrightarrow digit\ digit^{*}$$

$$unsigned\_number \longrightarrow unsigned\_integer\ ((\ .\ unsigned\_integer) \mid \epsilon)$$
$$(((\mathtt{e} \mid \mathtt{E})(+ \mid - \mid \epsilon)\ unsigned\_integer) \mid \epsilon)$$

What is the meaning of following expressions ?

- [0-9a-f]+
- b[aeiou]+t
- a*(ba*ba*)*

Define Regular Expressions

- Match strings only consisting of 'a', 'b', or 'c' characters
- Match only the strings "Buy more milk", "Buy more bread", or "Buy more juice"
- Match identifiers which contain letters and digits, starting with a letter

Context-Free Grammars

- Context-Free Grammars
  - Developed by Noam Chomsky in the mid-1950s
  - Describe the syntax of natural languages
  - Define a class of languages called context-free languages
  - Was originally designed for natural languages

Context-Free Grammars

- Using the notation Backus-Naur Form (BNF)
- A context-free grammar consists of
  - A set of *terminals* *T*
  - A set of *non-terminals* *N*
  - A *start symbol* *S* (a non-terminal)
  - A set of *productions* *P*

Terminals T

- The basic symbols from which strings are formed
- Terminals are tokens
  - if, foo, ->, ‘a’

Non-terminals **N**

- Syntactic variables that denote sets of strings or classes of syntactic structures
  - expr, stmt
- Impose a hierarchical structure on the language

Start Symbol **S**

- One nonterminal
- Denote the language defined by the grammar

Production **P**

- Specify the manner in which terminals and nonterminals are combined to form strings
- Each production has the format

  nonterminal -> a string of nonterminals and terminals

- One nonterminal can be defined by a list of nonterminals and terminals

Production **P**

- Nonterminal symbols can have more than one distinct definition, representing all possible syntactic forms in the language

  <if_stmt> -> if <logic_expr> then <stmt>

  <if_stmt> -> if <logic_expr> then <stmt> else <stmt>

Or

<if_stmt> -> if <logic_expr> then <stmt>
| if <logic_expr> then <stmt> else <stmt>

Backus-Naur Form

- Invented by John Backus and Peter Naur to describe syntax of Algol 58/60
- Used to describe the context-free grammars
- A **meta-language**: a language used to describe another language

BNF Rules

- A rule has a left-hand side(LHS), one or more right-hand side (RHS), and consists of **terminal** and **nonterminal** symbols
- For a nonterminal, when there is more than one RHS, there are multiple alternative ways to expand/replace the nonterminal
  - E.g., <stmt> -> <single_stmt>

    | begin <stmt_list> end

BNF Rules

- Rules can be defined using recursion

```
<ident_list> -> ident
              | ident, <ident_list>
```

- Two types of recursion
  - Left recursion:
    - id_list_prefix -> id_list_prefix, id | id
  - Right recursion
    - The above example

How does BNF work?

- It is like a mathematical game:
  - You start with a symbol **S**
  - You are given rules (**P**s) describing how you can replace the symbol with other symbols (**T**s or **N**s)
  - The language defined by the BNF grammar is the set of all terminal strings (**sentences**) you can produce by following these rules

Derivation

- A grammar is a generative device for defining languages
- The sentences of the language are generated through a sequence of rule applications
- The sequence of rule applications is called a **derivation**

An Example Grammar

```
<program> -> <stmts>
<stmts>   -> <stmt>
           | <stmt> ; <stmts>
<stmt>    -> <var> = <expr>
<var>     -> a | b | c | d
<expr>    -> <term> + <term>
           | <term> - <term>
<term>    -> <var>
           | const
```

An Exemplar Derivation

```
<program> => <stmts>
          => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

sentential forms (all lines above except the last)

sentence ← (the last line: `a = b + const`)

Sentential Forms

- Every string of symbols in the derivation is a **sentential form**
- A **sentence** is a sentential form that has only terminal symbols
- A **leftmost derivation** is one in which the leftmost non-terminal in each sentential form is the one that is expanded next in the derivation

Sentential Forms

- A **left-sentential form** is a sentential form that occurs in the leftmost derivation
- A **rightmost derivation** works right to left instead
- A **right-sentential form** is a sentential form that occurs in the rightmost derivation
- Some derivations are neither leftmost nor rightmost

Why BNF?

- Provides a clear and concise syntax description
- The parse tree can be generated from BNF
- Parsers can be based on BNF and are easy to maintain

Context-Free Grammars

- The syntax of simple arithmetic expression

```
expr -> id | number | -expr |(expr)
        |expr op expr
  op -> + | - | * | /
```

- What are the terminal symbols and nonterminal symbols?
- What is the start symbol?

One Possible Derivation

```
expr => expr op expr
     => …
     => id + number
```

Parse Tree

- A **parse tree** is
  - a hierarchical representation of a derivation
  - to represent the structure of the derivation of a terminal string from some non-terminal
  - to describe the hierarchical syntactic structure of programs for any language

An Example

- Given the simple assignment statement syntax

  <assign> -> <id> = <expr>

  <id> -> A | B | C

  <expr> -> <id> + <expr>

  | <id> * <expr>

  | (<expr>)

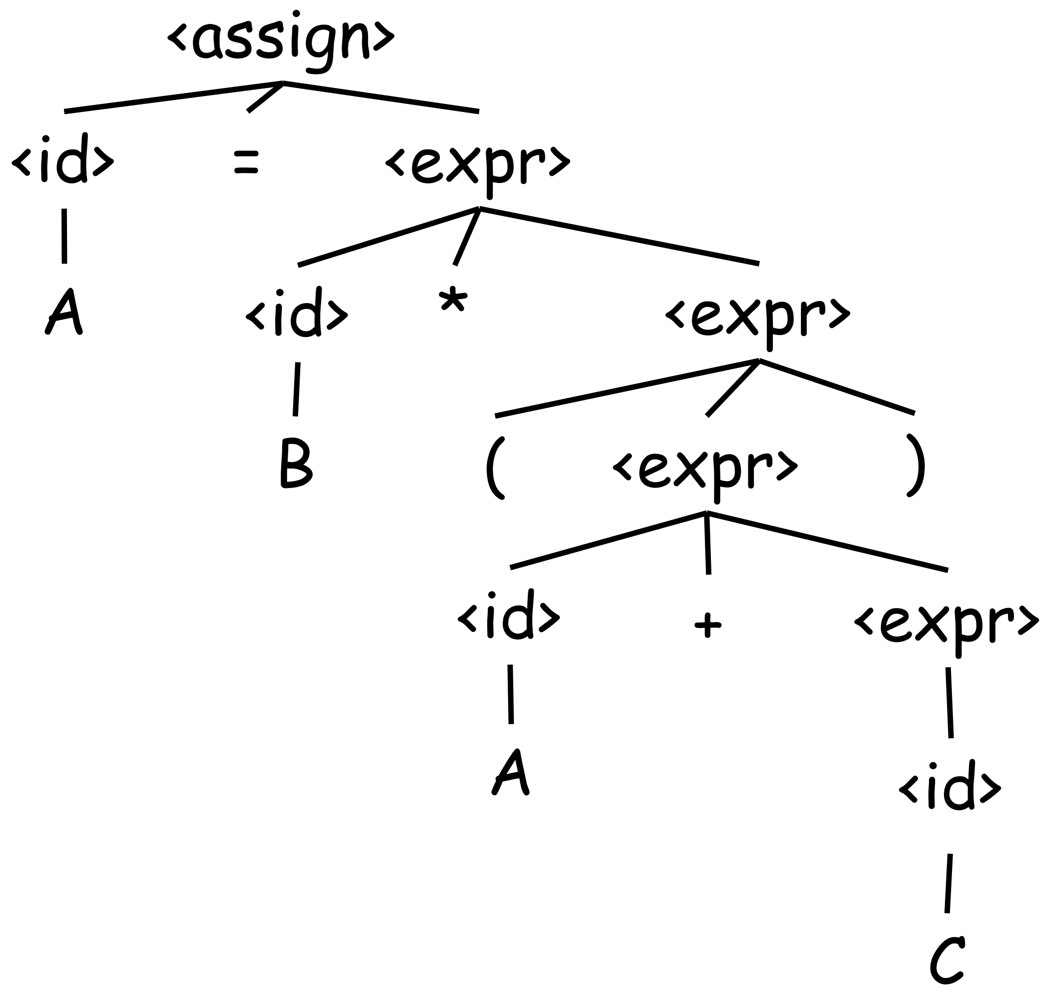
  | <id>

- With leftmost derivation, how is A = B * (A + C) generated?

Derivation for A = B * (A + C)

<assign> => <id> = <expr>
=> A = <expr>
=> A = <id> * <expr>
=> A = B * <expr>
=> A = B * (<expr>)
=> A = B * (<id> + <expr>)
=> A = B * (A + <expr>)
=> A = B * (A + <id>)
=> A = B * (A + C)

The Parse Tree for A = B * (A + C)



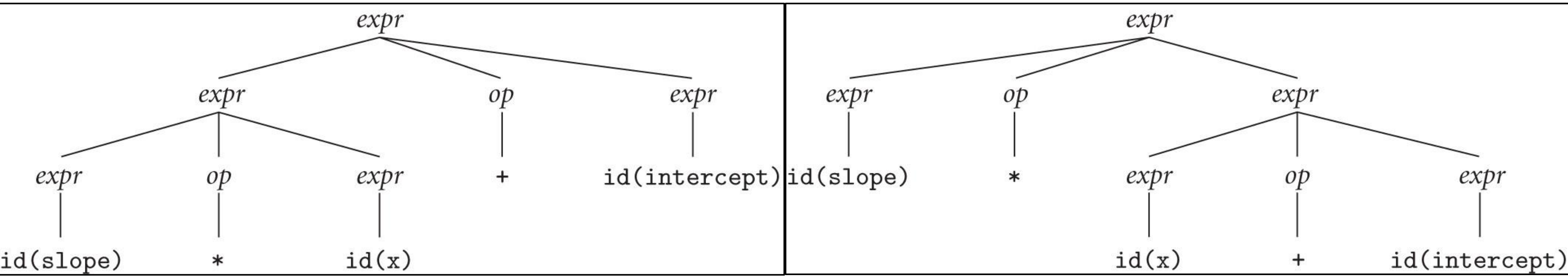
Parse Tree

- A grammar is **ambiguous** if it generates a sentential form that has two or more distinct parse trees

An Ambiguous Grammar

```
expr -> id | number | -expr |(expr)
        | expr op expr
  op -> + | - | * | /
```

- Parse trees for “slope * x + intercept”:



What goes wrong?

- The production rules do not capture the **associativity** and **precedence** of various operators
  - **Associativity** tells whether the operators group left to right or right to left
    - Is 10 – 4 – 3 equal to (10 - 4) – 3 or 10 – (4 – 3) ?
  - **Precedence** tells some operators group more tightly than the others
    - Is slope * x + intercept equal to (slope * x) + intercept or slope * (x + intercept)?

Operator Associativity

- Single recursion in production rules

```
<expr> -> <expr> - <expr> | const
```

✗ Ambiguous

```
<expr> -> <expr> - const | const
```

✓ Unambiguous

```
<expr> -> const - <expr> | const
```

✓ Unambiguous (less desirable)

Operator Precedence

- Use stratification in production rules
  - Intentionally put operators at different levels of parse trees

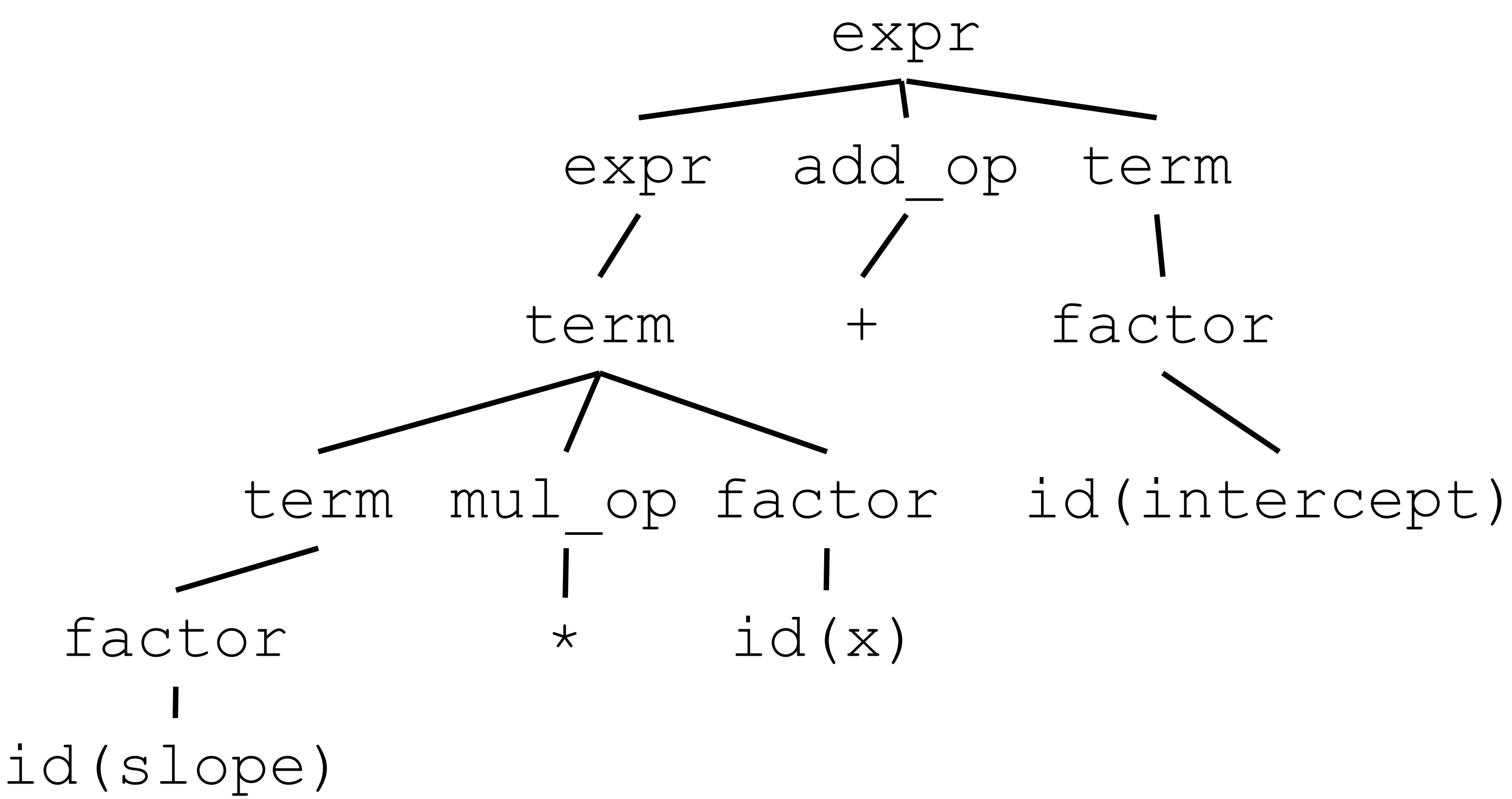
```
<expr> -> <expr> - <term> | <term>
<term> -> <term> / const  | const
```

Improved Unambiguous Context-Free Grammar

```
1. expr -> expr add_op term
          | term
2. term -> term mul_op factor | factor
3. factor -> id | number | -factor
                  | (expr)
3. add_op -> + | -
4. mul_op -> * | /
```

Revisit “slope * x + intercept”

- Parse Tree



Extended BNF (EBNF)

- Optional parts are placed in brackets (**[ ]**)

```
<proc_call> -> ident [ '('<expr_list>')' ]
```

- Put alternative parts of RHS in parentheses and separate them with vertical bars

```
<term> -> <term> (+ | -) const
```

- Put repetitions (0 or more) in braces (**{ }**)

```
<ident> -> letter { letter | digit }
```

BNF and EBNF

- BNF

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
```

- EBNF

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
```

Reference

- https://www.tutorialspoint.com/automata_theory/chomsky_classification_of_grammars.htm