National University
Of Computer and Emerging Sciences

Theory of Programming Languages

Describing Syntax and Semantics

Sajid Anwer

Department of Computer Science,
FAST-NUCES, CFD Campus

Introduction

- **Syntax:** the *form or structure* of the expressions, statements, and program units

- **Semantics:** the *meaning* of the expressions, statements, and program units

- Syntax and semantics provide a language's definition
  - » Users of a language definition
    - Other language designers
    - Implementers
    - Programmers (the users of the language)

Describing Syntax: Fundamentals

- 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)

Describing Syntax: Fundamentals

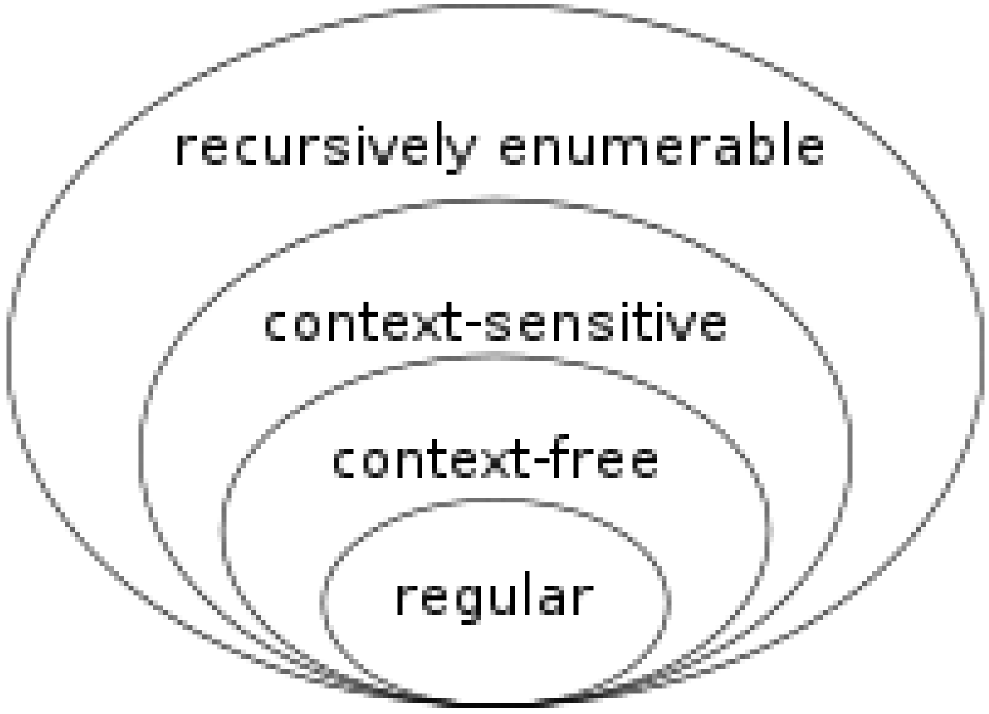
- Recognizers
  - A recognition device reads input strings *over the alphabet* of the language and decides whether the input strings belong to the language
  - Example: syntax analysis part of a compiler

- Generators
  - A device that generates sentences of a language
  - One can determine if the syntax of a *particular sentence* is syntactically correct by comparing it to the structure of the generator

BNF and Context-Free Grammars

- Context-Free Grammars
  - Developed by Noam Chomsky in the mid-1950s
  - Language generators, meant to describe the *syntax* of natural languages
  - Define a class of languages called context-free languages
- Backus-Naur Form (1959)
  - Invented by John Backus to describe the syntax of Algol 58
  - BNF *is equivalent* to context-free grammars

BNF and Context-Free Grammars



BNF Fundamentals

- In BNF, *abstractions* are used to represent classes of syntactic structures
  - » *nonterminal*
  - » *terminals*
- *Terminals* are lexemes or tokens
- A statement is BNF is called *production rule* and it has a:
  - » left-hand side (LHS), which is a nonterminal,
  - » A right-hand side (RHS), which is a string of terminals and/or nonterminals.

BNF Fundamentals (continued)

- Nonterminals are often enclosed in angle brackets
  - Examples of BNF rules:

```
<ident_list> → identifier | identifier, <ident_list>
<if_stmt> → if <logic_expr> then <stmt>
```

- Grammar: a *finite* non-empty set of rules
- A *start symbol* is a special element of the nonterminals of a grammar

An Example Grammar

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

Generate a=b;

National University
Of Computer and Emerging Sciences

An Example Derivation

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

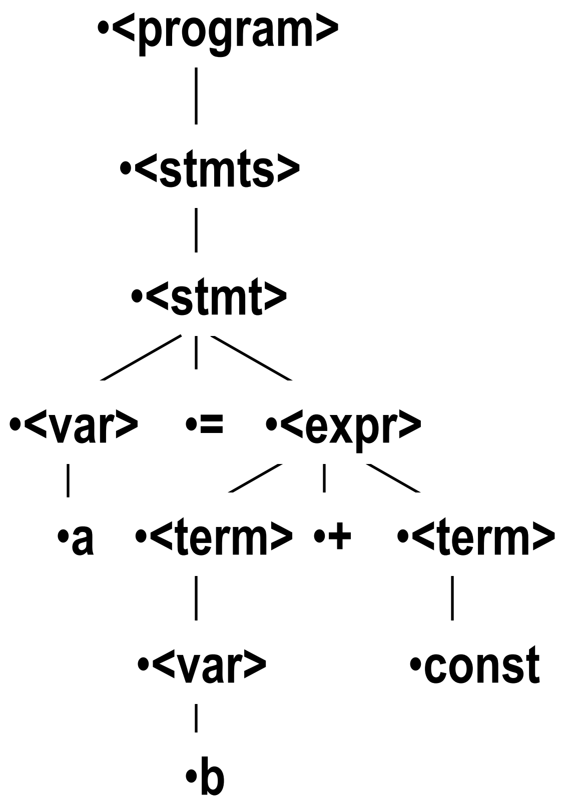
Derivations

- Every string of symbols in a 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 nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

National University
Of Computer and Emerging Sciences

Parse Tree

- A hierarchical representation of a derivation

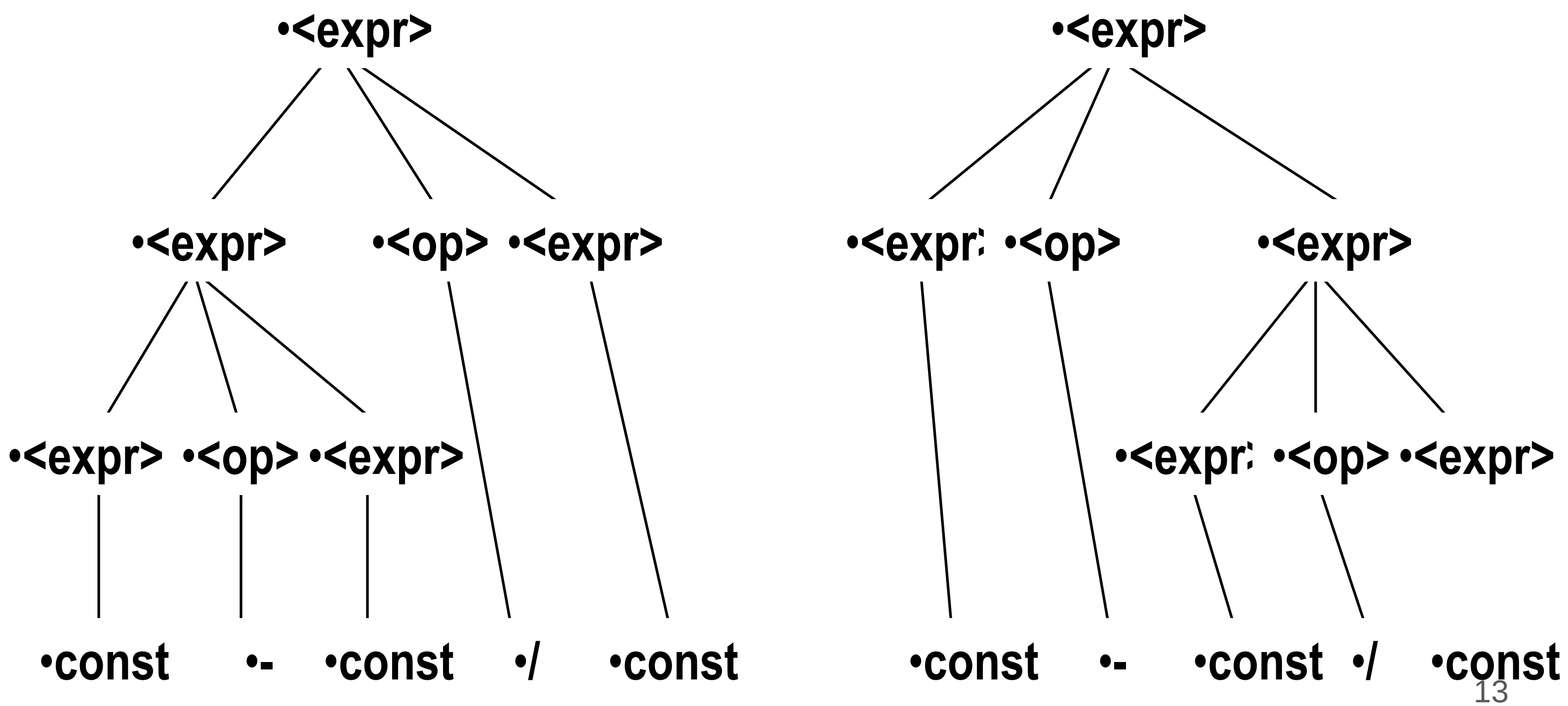


National University Of Computer and Emerging Sciences

An Ambiguous Expression Grammar

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

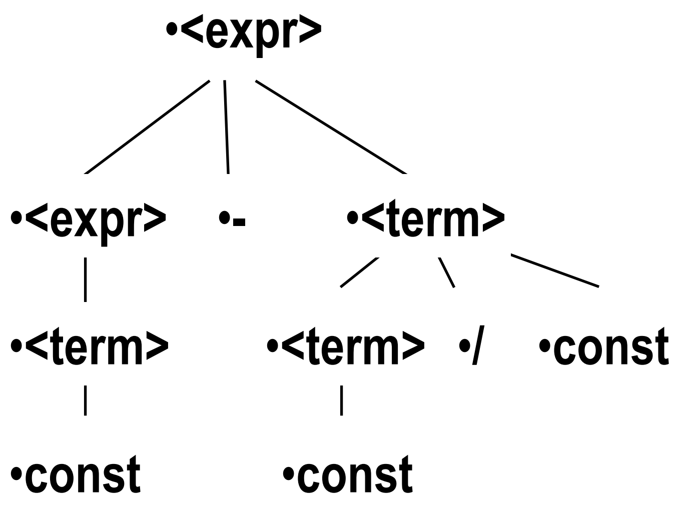
```
<expr> → <expr> <op> <expr>  |  const
<op> → /  |  -
```



An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

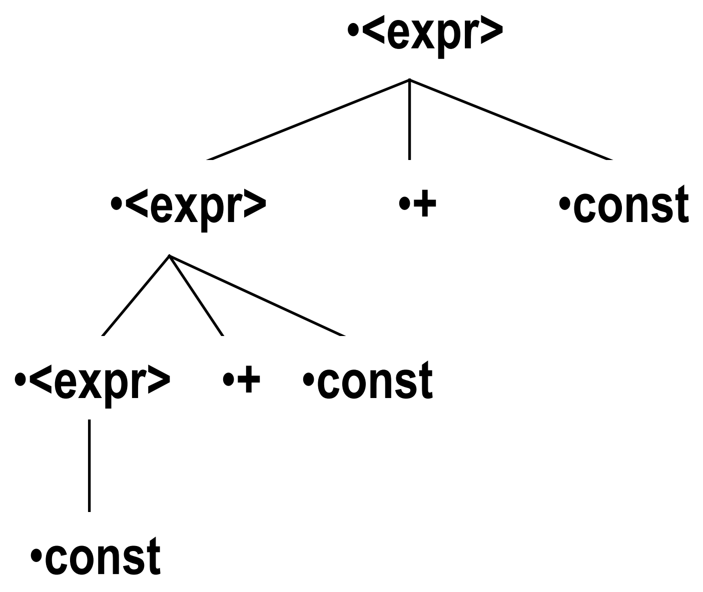
```
<expr> → <expr> - <term>  |  <term>
<term> → <term> / const| const
```



Associativity of Operators

- Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const  (ambiguous)
<expr> -> <expr> + const  |  const  (unambiguous)
```



Extended BNF

- Optional parts are placed in brackets [ ]

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

- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

```
<term> → <term> (+|-) const
```

- Repetitions (0 or more) are placed inside 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>}
```

Semantics

- There is *no single widely acceptable notation* or formalism for describing semantics
- Several needs for a methodology and notation for semantics:
  - Programmers need to know *what statements mean*
  - *Compiler writers* must know exactly what language constructs do
  - Correctness proofs would be possible
  - *Compiler generators* would be possible
  - Designers could *detect* ambiguities and inconsistencies

Operational Semantics

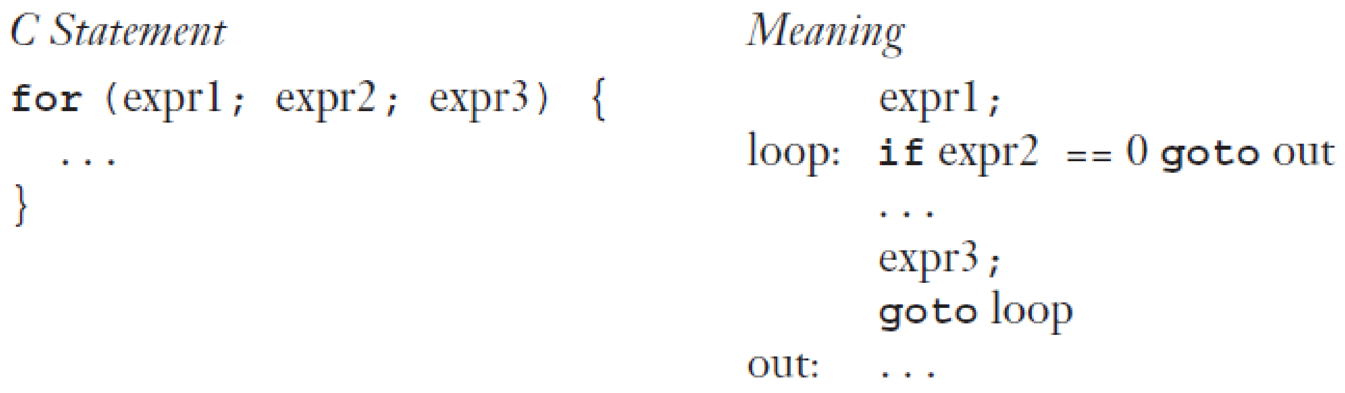
- Operational Semantics
  - Describe the meaning of a program by executing its *statements on a machine*, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement

- A *hardware* pure interpreter would be too expensive
- A *software* pure interpreter also has problems
  - The detailed characteristics of the particular computer would make actions difficult to understand
  - Such a semantic definition would be *machine- dependent*

- To use operational semantics for a high-level language, *a virtual machine* is needed.

Operational Semantics (continued)

- A better alternative: A complete computer simulation
- The process:
  - Build a translator (translates source code to the *machine code of an idealized computer*)
  - Build a simulator for the idealized computer



```
C Statement                          Meaning

for (expr1; expr2; expr3) {                  expr1;
   ...                               loop:   if expr2 == 0 goto out
}                                            ...
                                             expr3;
                                             goto loop
                                     out:    ...
```

Denotational Semantics

- Based on recursive function theory
- The most abstract semantics description method
- Originally developed by Scott and Strachey (1970)

- The process of building a denotational specification for a language:
  - Define a *mathematical object* for each language entity
  - Define a *function* that *maps instances* of the language entities onto instances of the corresponding mathematical objects

- The meaning of language constructs are defined by only the values of the program's variables
- Denotational vs operational semantics?

Denotational Semantics

```
<bin_num> → '0'
          | '1'
          | <bin_num> '0'
          | <bin_num> '1'
```

M_{bin}('0') = 0
M_{bin}('1') = 1
M_{bin}(<bin_num> '0') = 2 * M_{bin}(<bin_num>)
M_{bin}(<bin_num> '1') = 2 * M_{bin}(<bin_num>) + 1

Decimal Numbers

```
<dec_num> → '0' | '1' | '2' | '3' | '4' | '5' |
            '6' | '7' | '8' | '9' |
            <dec_num> ('0' | '1' | '2' | '3' |
                       '4' | '5' | '6' | '7' |
                       '8' | '9')
```

M_{dec}('0') = 0, M_{dec} ('1') = 1, …, M_{dec} ('9') = 9

M_{dec} (<dec_num> '0') = 10 * M_{dec} (<dec_num>)

M_{dec} (<dec_num> '1') = 10 * M_{dec} (<dec_num>) + 1

…

M_{dec} (<dec_num> '9') = 10 * M_{dec} (<dec_num>) + 9

Axiomatic Semantics

- Based on *formal logic* (predicate calculus)
- Original purpose: *formal program verification*
- Axioms or inference rules are defined for each statement type in the language (to allow transformations of logic expressions into more formal logic expressions)
- The logic expressions are called assertions

Axiomatic Semantics (continued)

- An assertion before a *statement (a precondition)* states the relationships and constraints among variables that are true at that point in execution
- An assertion *following a statement is a postcondition*
- A *weakest precondition* is the least restrictive precondition that will guarantee the postcondition
- Pre-, post form: `{P} statement {Q}`
- An example
  - » `a = b + 1  {a > 1}`
  - » One possible precondition: `{b > 10}`
  - » Weakest precondition: `{b > 0}`

Program Proof Process

- The postcondition for the entire program is the desired result
  - » Work back through the program to the first statement. If the precondition on the first statement is the same as the program specification, the program is correct.

Axiomatic Semantics: Assignment

- An axiom for assignment statements (x = E): $\{Q_{x\text{->}E}\}$ x = E {Q}

- The Rule of Consequence:

$$\frac{\{P\}\,S\,\{Q\},\ P' \Rightarrow P,\ Q \Rightarrow Q'}{\{P'\}\,S\,\{Q'\}}$$

Evaluation of Axiomatic Semantics

- Developing axioms or inference rules for all of the statements in a *language is difficult*
- It is a good tool for *correctness proofs*, and an excellent framework for reasoning about programs, but it is not as useful for language users and compiler writers
- Its usefulness in describing the meaning of a programming language is limited for language users or compiler writers