Topics

- Syntax
- Context-Free Grammars and BNF (Backus-Naur Form)
- Derivation and Parse Tree
- Ambiguity
- Extended BNF

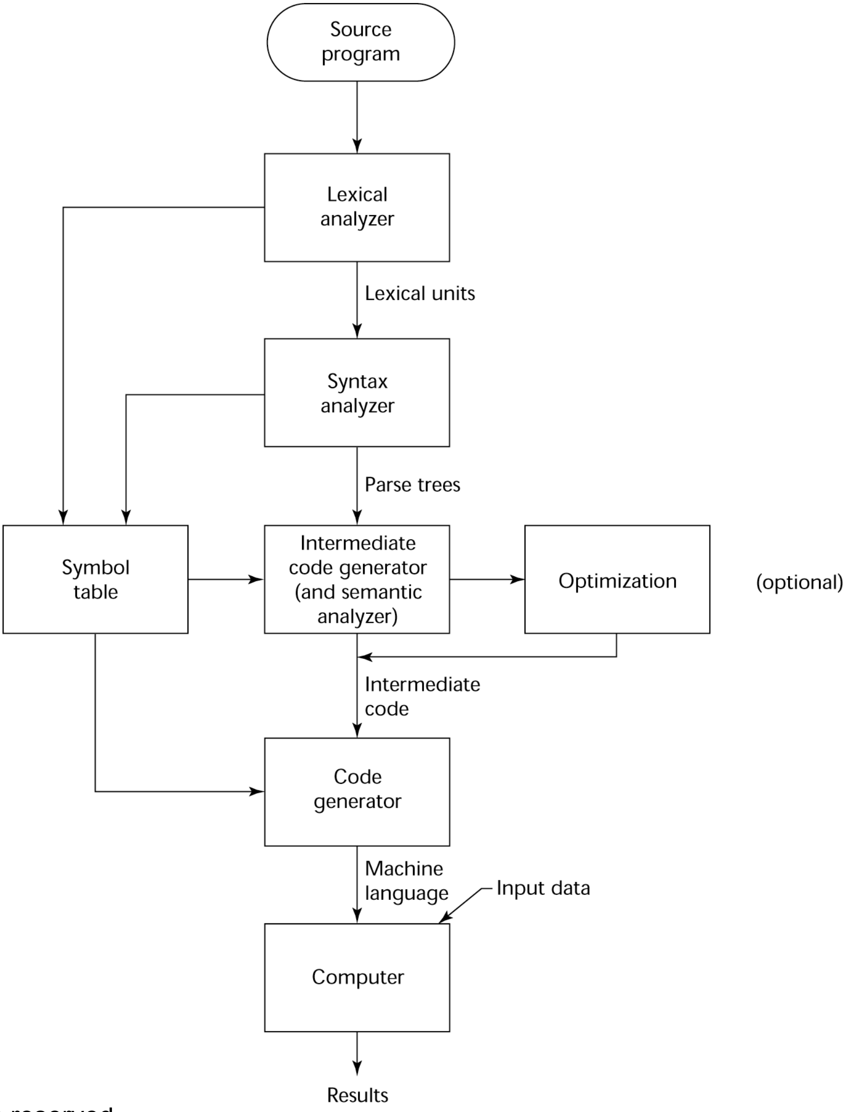
Syntax and Semantics

- **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
    - Initial evaluators
    - Implementers
    - Programmers (the users of the language)

The General Problem of Describing Syntax: Terminology

- 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`, `12.3`)
- A *token* is a category of lexemes (e.g., identifier)

The Compilation Process



Formal Definition of Languages

- **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
    - Detailed discussion of syntax analysis appears in Chapter 4
- **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 (a linguist) 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 Fundamentals

- In BNF, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (also called *nonterminal symbols,* or just *nonterminals*)

- *Terminals* are lexemes or tokens

- A rule has a left-hand side (LHS), which is a nonterminal, and 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

BNF Rules

- An abstraction (or nonterminal symbol) can have more than one RHS

```
<stmt> → <single_stmt>
       | begin <stmt_list> end
```

- Syntactic lists can be described using recursion

```
<ident_list> → ident
             | ident, <ident_list>
```

Derivation

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

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 Example Derivation

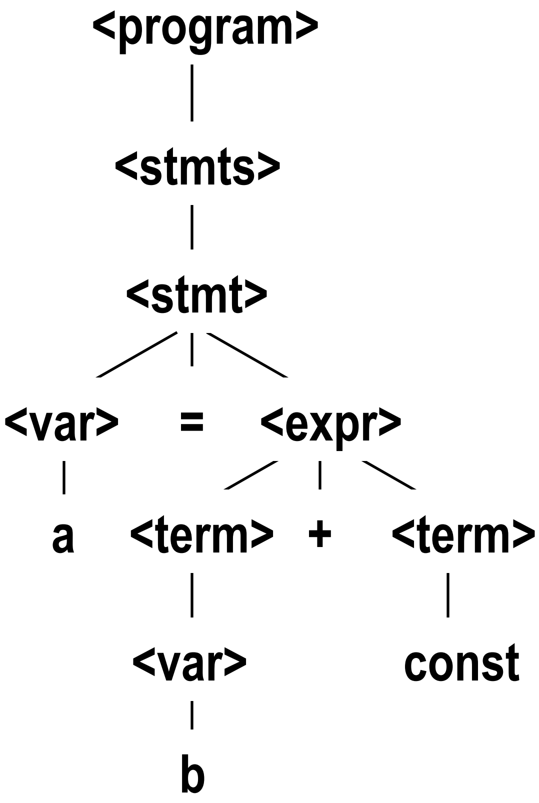
```
<program> => <stmts> => <stmt>
                    => <var> = <expr>
                    => a = <expr>
                    => a = <term> + <term>
                    => a = <var> + <term>
                    => a = b + <term>
                    => a = b + const
```

Derivation Terminology

- 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

Parse Tree

- A hierarchical representation of a derivation

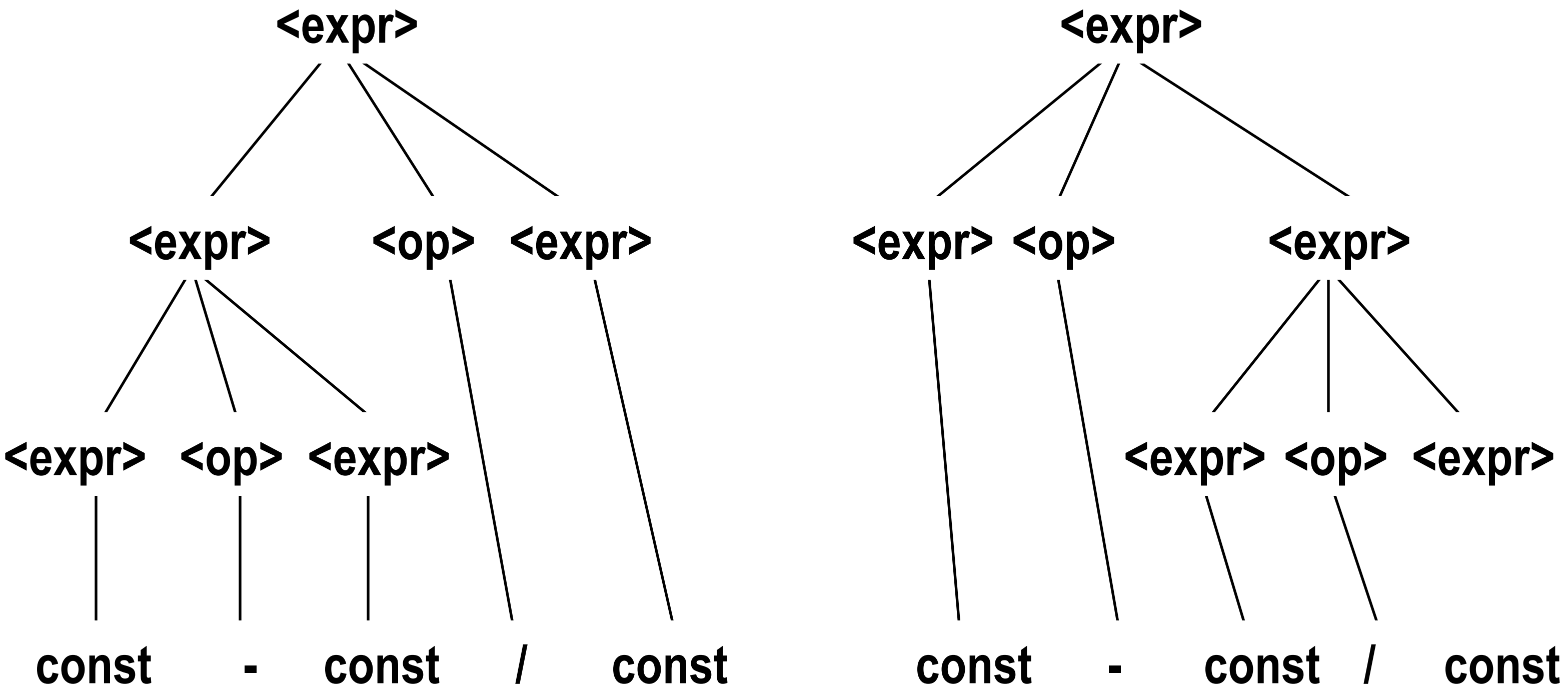


Ambiguity in Grammars

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

An Ambiguous Expression Grammar

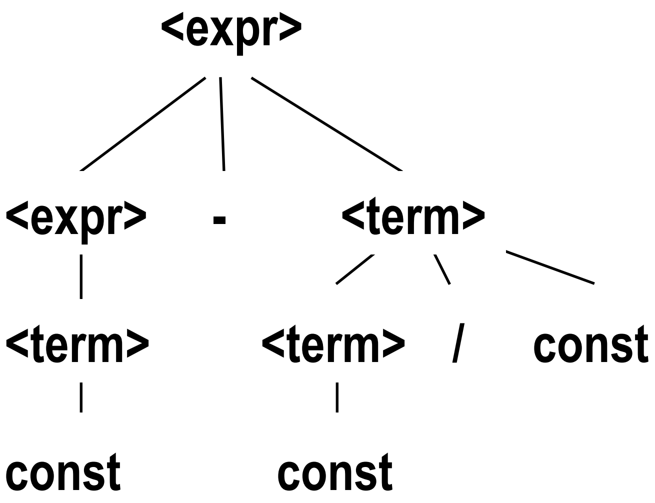
```
<expr> → <expr> <op> <expr>  |  const
<op> → /  |  -
What is 8-4/2?
```



An Unambiguous Expression Grammar

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

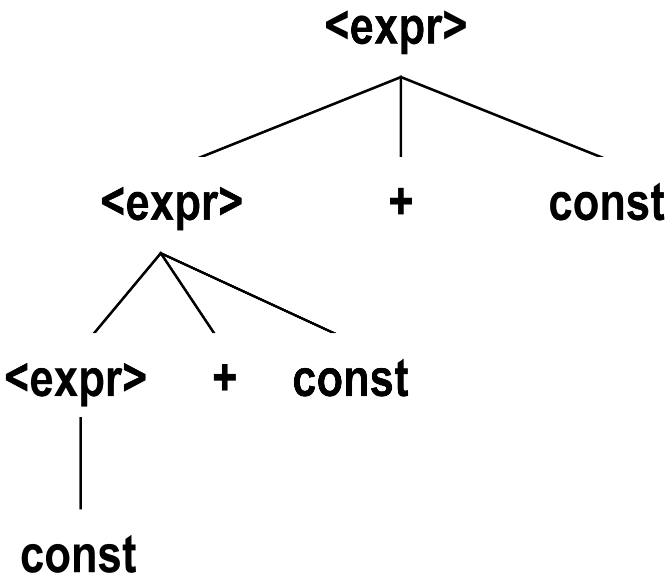
```
<expr> → <expr> - <term>  |  <term>
<term> → <term> / const| const
```



Associativity of Operators

- Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const  (ambiguous)
<expr> -> <expr> + const  |  const  (unambiguous)
How to compute 3+4+5?
```



Unambiguous Grammar for Selector

- Java `if-then-else` grammar

```
<if_stmt> -> if (<logic_expr>) <stmt>
           | if (<logic_expr>) <stmt> else <stmt>
```

Ambiguous!

if (<logic_expr>) **if** (<logic_expr>) <stmt> **else** <stmt>

- An unambiguous grammar for `if-then-else`

```
<stmt> -> <matched> | <unmatched>
 <matched> -> if (<logic_expr>) <matched> else <matched>
            | a non-if statement
 <unmatched> -> if (<logic_expr>) <stmt>
              | if (<logic_expr>) <matched> else <unmatched>
```

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>}
```

Reading Assignment

- Read Sections 3.1, 3.2 and 3.3