CS 230
Programming Languages

09 / 15 / 2022

Instructor: Michael Eckmann

Today's Topics

- Questions? / Comments?
- Continue describing the Syntax of a programming language
- Reminder of Context Free Grammars and BNF
- Ambiguity
- Extended BNF
- Larger grammar example
- Attribute grammars
- Start Parsers (Chapter 4) - Lexical Analyzer

Syntax

- What are Context Free Grammars (CFGs) and Backus-Naur Form (BNF)? And for what are they used?

Syntax

- What are Context Free Grammars (CFGs) and Backus-Naur Form (BNF)? And for what are they used?
- They are metalanguages used to describe syntax of a programming language

Syntax

- A Context Free Grammar is a four-tuple

(T, N, S, P) where

  - T is the set of terminal symbols
  - N is the set of non-terminal symbols
  - S is the start symbol (which is one of the non-terminals)
  - P is the set of productions of the form:
    - A -> X_1 . . . X_m where
      - A element of N
      - X_i element of N U T, 1 <= i <= m, m>=0

Syntax

- Reminder of
- Non-terminals and terminals
- Productions

Syntax

- **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`

Syntax

- Let's derive another sentence in the language described by the grammar on the previous slide.
- And the parse tree for that sentence

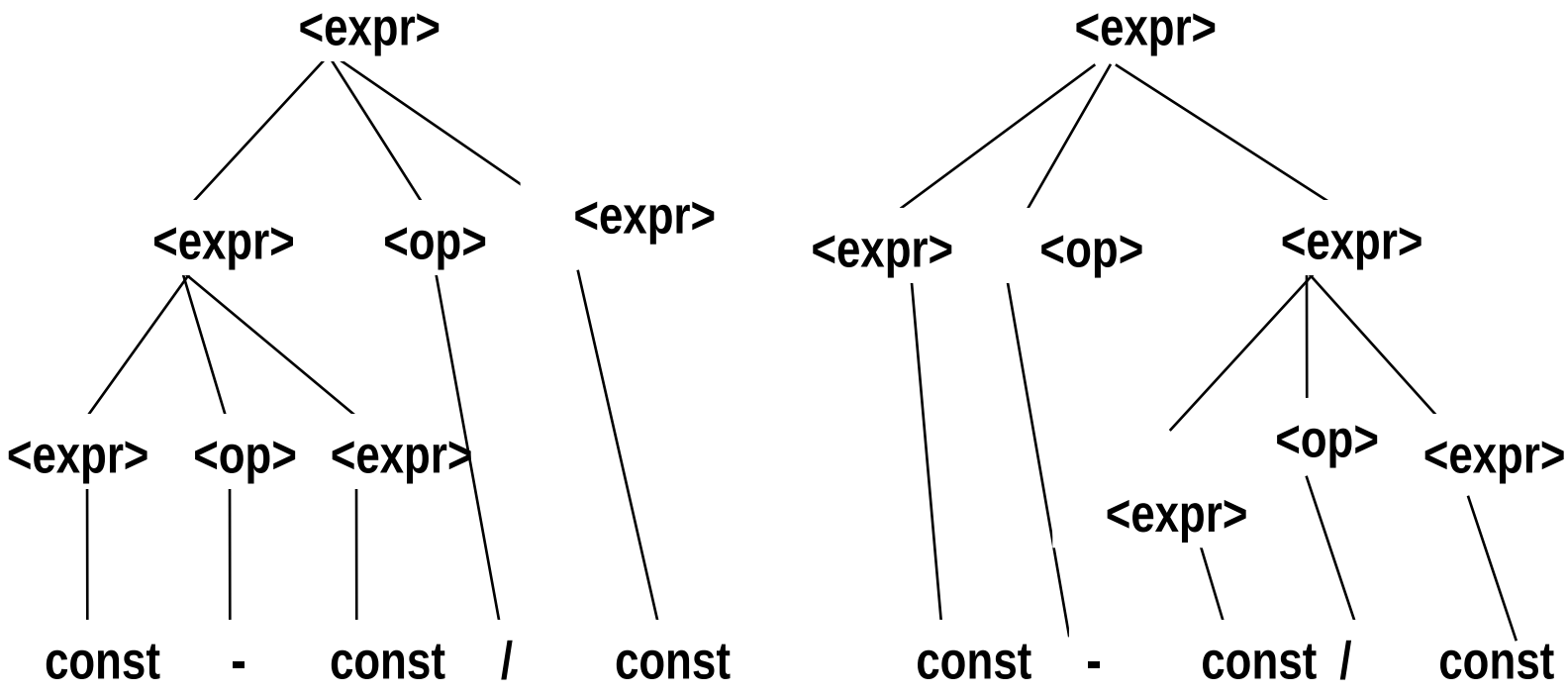
Ambiguity

- A grammar is ambiguous if one can generate a sentential form for which there is more than one parse tree.
- Compilers typically base the semantics of the language on their syntactic form. A compiler chooses the code to be generated for a statement from its parse tree.

An Ambiguous Expression Grammar

<expr> → <expr> <op> <expr> | const

<op> → / | -



Ambiguous

- Ambiguity is not able to be handled by compilers, so the language description must be unambiguous so the compiler can be written

- The compiler examines the parse tree to determine the code to generate. Two parse trees for the same syntax will cause the meaning (semantics) of the code to not be unique.

Ambiguous?

- Look at the if statement rules below

<if_stmt> → if <logic_expr> then <stmt>

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

<stmt> → <if_stmt> | ...

- Do you think this is ambiguous? That is, can more than one parse tree be generated from the same code?
- if (a==b) then if (c==d) then print_something() else print_something_else()

Ambiguous?

```
if (a==b) then if (c==d) then print_something() else
    print_something_else()
```

```
if (a==b) then
      if (c==d) then
           print_something()
      else
          print_something_else()
```

```
if (a==b) then
    if (c==d) then
        print_something()
else
    print_something_else()
```

Ambiguous?

- To make it unambiguous take a look at page 125 in our text.
- New Nonterminals are added and the other productions are changed in order to remove the ambiguity.

Extended BNF

- Common extensions to BNF include:

- Use of square brackets [ ] to enclose optional parts of RHS's. That is, the part in the brackets is used 0 or 1 time.
- Use of braces { } to enclose parts of RHS's that can be repeated indefinitely or left out. That is, the part in the braces is repeated 0 or more times.
- Use of parentheses ( ) around a group of items of which one is chosen. The items are separated by |.

Extended BNF (EBNF)

- It should be obvious that these new symbols are not terminal symbols in the language being described nor are they non-terminals. These new symbols are part of the EBNF meta language.
- If the language being described by the grammar does require brackets, braces or parentheses as terminal symbols (as many languages do) they have to be denoted in some way like underlining them to differentiate them from the EBNF symbols.

- What good are these extensions?

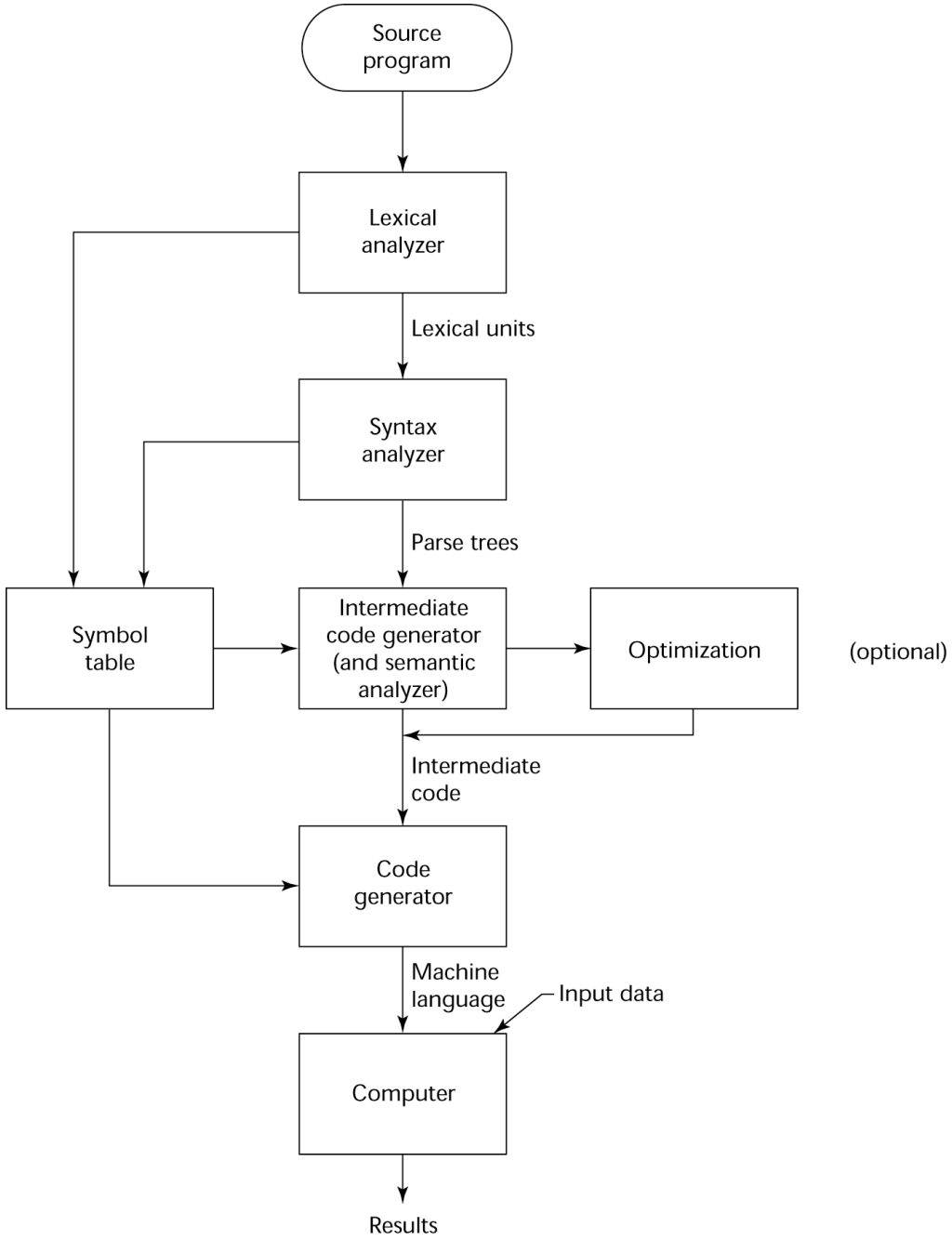
Extended BNF

- What good are these extensions?
  - They increase writability and readability of the BNF metalanguage. Often lead to fewer non-terminals and fewer productions.
  - They DO NOT enhance the descriptive power of BNF --- that is, they do not allow languages to be described that couldn't have been described without them.

CFG's and Recognizers

- Given a formal description of a language, a recognizer (syntax analyzer, aka parser) for that language can be algorithmically constructed. Therefore a program can be written to do this. yacc is an example of one.

Let's Revisit the Compilation Process



Compilation Process

- Lexical analyzer – groups the code into variable names, reserved words, punctuation, etc.
- Syntax analyzer – constructs parse trees to determine if the lexical units in order are syntactically correct
- Both create items in the **symbol table** to refer to these units
- Intermediate code generator (and semantic analyzer) generates assembly-like code from the symbol table and parse trees.

Chapter 3 so far

- Generator / recognizer
- Is a grammar a generator or recognizer, do you think?
- CFG and BNF are what?
- CFG and BNF are used for what?
- Derivation for a sentence (program)
- Parse tree for a sentence (program)
- What is ambiguity and why is it bad?

A more complex grammar

- Let's take a look at the mini-pascal Language.
- Let's first randomly generate a valid sentence (program) or two given this description.
- Then let's in our mind sort of create a parser from this EBNF description and use that to determine if some programs are syntactically correct.

Limitations of CFG and EBNF

- Do you think that the EBNF for mini-pascal is the complete description for the syntax of the language?
- Is anything missing? --- think of some syntax errors that you are used to seeing in say Java or other language.

Attribute Grammars

- Hence the creation of attribute grammars.
- An attribute grammar is an extension to a CFG.
- There are some rules of programming languages that cannot be specified in BNF (or by a CFG for that matter.)
- e.g. All variables must be declared before they are used.
- Also, there are things that are possible, but just too difficult to specify using CFG's, (e.g. Type compatibility) so Attribute Grammars are used.
- These kinds of things are termed “static semantics.”

Lexical & Syntax Analysis

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a **lexical analyzer** (mathematically, a finite automaton based on a regular grammar)
  - A high-level part called a syntax analyzer, or **parser** (mathematically, a push-down automaton based on a context-free grammar, or BNF)
    - The parser can be based directly on the BNF
    - Parsers based on BNF are easy to maintain

Lexical & Syntax Analysis

- Lexical and syntax analysis are separated because of
  - Simplicity - less complex approaches can be used for lexical analysis; separating them simplifies the parser
  - Efficiency - separation allows optimization of the lexical analyzer
  - Portability - parts of the lexical analyzer may not be portable, but the parser always is portable

Lexical Analysis

- Lexical analysis matches patterns and it is a front-end to the parser
- Identifies substrings of the source program that belong together - lexemes
  - Lexemes match a character pattern, which is associated with a token
  - Recall examples of lexemes and tokens

Lexemes and tokens

```
idx = 42 +
count;
```

Lexemes	*Tokens*
idx	identifier
=	equal_sign
42	int_literal
+	plus_op
Count	identifier
;	semicolon

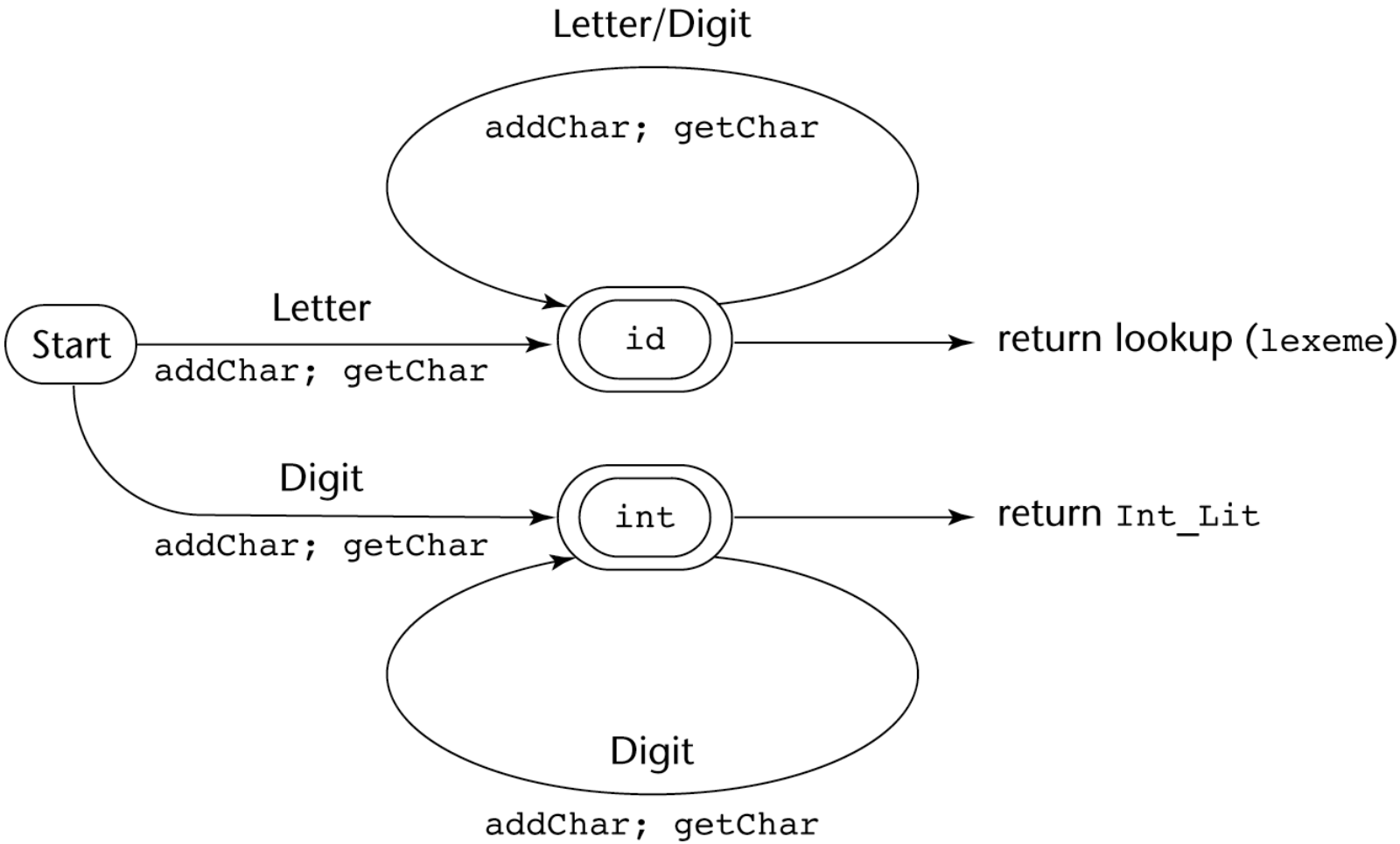
Lexical Analysis

- The lexical analyzer is usually a function that is called by the parser when it needs the next token
- Three approaches to building a lexical analyzer:
  - Write a formal description of the tokens and use a software tool (e.g. lex) that constructs table-driven lexical analyzers given such a description
  - Design a state diagram that describes the tokens and write a program that implements the state diagram
  - Design a state diagram that describes the tokens and hand-construct a table-driven implementation of the state diagram

Lexical Analysis

- Lexical analyzers are finite automatons
- A finite automaton is a 5-tuple (Q, Sigma, q0, delta, A) where
  - Q is a finite set of states
  - Sigma is a finite set of input symbols
  - q0 (the start state) is an element of Q
  - delta (the transition function) is a function from QxSigma to Q
  - A (the accepting states) is a subset of Q

Lexical Analysis



Lexical Analysis

- The preceding diagram is a state diagram for a lexical analyzer for only the program names and reserved words and literals in some language.
- The states are represented by ovals
  - Q = { Start, id, int }
- The input symbols are specified along the arrows
  - Sigma = { Letter, Digit }
- The arrows are the transitions among states
  - delta can be easily drawn into a chart with the headings:

    From-state, input symbol, To-state
- The start state is Start (that is, q0 = Start)
- The accepting states are drawn with two ovals
  - A = {id, int}

Lexical Analysis

- The way it works is, we start at the Start state and depending on what input symbol we get (a Letter or a Digit) we move to the appropriate state. And so on until we hit an accepting state (valid.)
- If we get an input symbol while in a state that isn't on an arrow leaving that state, then we end (return).
  - However, if we're in the Start state and get something other than a Letter or a Digit as the input symbol, we do not have a program name, reserved word nor a literal which is what that finite automaton was designed to detect.

Lexical Analysis

- Utility subprograms:
  - **`getChar`** - gets the next character of input, puts it in **`nextChar`**, determines its class and puts the class in **`charClass`**
  - **`addChar`** - puts the character from **`nextChar`** into the place the lexeme is being accumulated, **`lexeme`**
  - **`lookup`** - determines whether the string in **`lexeme`** is a reserved word (returns a code)