

Organization of Programming Languages

CS320/520N

Lecture 03

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

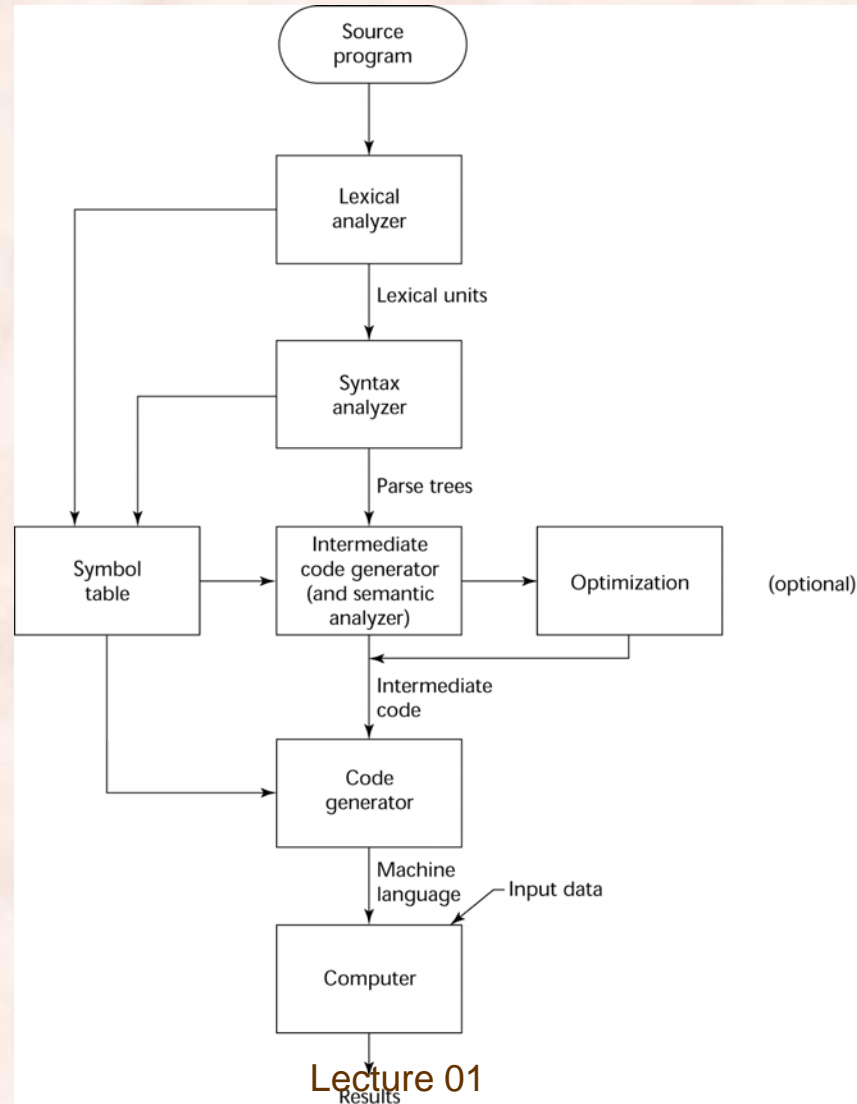
What is a programming language?

- A **programming language** is an artificial language designed for expressing algorithms on a computer:
 - Need to express an *infinite* number of algorithms i.e. Turing complete.
 - Requires an unambiguous **syntax**, specified by a *finite* context free grammar.
 - Should have a well defined compositional **semantics** for each syntactic construct: *axiomatic vs. denotational vs. operational*.
 - Often requires a practical implementation i.e. **pragmatics**:
 - Implementation on a real machine vs. virtual machine
 - *translation vs. compilation vs. interpretation*.

Implementation Methods: Compilation

- Translate high-level program (source language) into machine code (machine language)
- Slow translation, fast execution
- Compilation process has several phases:
 - **lexical analysis**: converts characters in the source program into lexical units (e.g. identifiers, operators, keywords).
 - **syntactic analysis**: transforms lexical units into *parse trees* which represent the syntactic structure of program.
 - **semantics analysis**: check for errors hard to detect during syntactic analysis; generate *intermediate code*.
 - **code generation**: machine code is generated.

The Compilation Process



Syntax vs. Semantics

- **Syntax:** specifies the form or structure of the expressions, statements, and program units.
 - `<if_stmt> → if <logic_expr> then <stmt>`
 - `<if_stmt> → if <logic_expr> then <stmt> else <stmt>`
 - `<while_stmt> → while (<logic_expr>) <stmt>`
- **Semantics:** the meaning of the expressions, statements, and program units.
 - what is the meaning of the Java while statement above?
- **Syntax vs. Semantics:**
 - semantics should follow directly from syntax.
 - formal specification easier for syntax than for semantics

Lexical Analysis: Terminology

- An **alphabet** Σ is a set of characters.
 - the English alphabet.
- A **lexeme** is a string of characters from Σ .
 - index = count + 1;
- A **token** is a category of lexemes:
 - index, count \rightarrow identifier
 - + \rightarrow plus_operator
 - 1 \rightarrow integer_literal
 - ; \rightarrow semicolon
- The **lexical rules** of a language specify which lexemes belong to the language, and their categories.

Syntactic Analysis: Terminology

- An **alphabet** Σ is a set of tokens.
 - $\Sigma = \{\text{identifier}, \text{plus_operator}, \text{integer_literal}, \dots\}$
- A **sentence** S is a string of tokens from Σ ($S \in \Sigma^*$).
 - $\text{index} = \text{count} + 1$;
- A **language** L is a set of sentences ($L \subseteq \Sigma^*$).
- The **syntactic rules** of a language specify which sentences belong to the language:
 - if $S \in L$, then S is said to be *well formed*.

Generative Grammars

- Formal grammars were first studied by linguists:
 - Panini (4th century BC): the earliest known grammar of Sanskrit.
 - Chomsky (1950s): first formalized generative grammars.
- A **grammar** is tuple $G = (\Sigma, N, P, S)$:
 - A finite set Σ of **terminal symbols**.
 - the tokens of a programming language.
 - A finite set N of **nonterminal symbols**, disjoint from Σ .
 - expressions, statement, type declarations in a PL.
 - A finite set P of **production rules**.
 - $P : (\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$
 - A distinguished **start symbol** $S \in N$.

Generative Grammars

- The language L associated with a formal grammar G is the set of strings from Σ^* that can be generated as follows:
 - start with the start symbol S ;
 - apply the production rules in P until no more nonterminal symbols are present.
- Example:
 - $\Sigma = \{a,b,c\}$, $N = \{S,B\}$
 - P consists of the following production rules:
 1. $S \rightarrow aBSc$
 2. $S \rightarrow abc$
 3. $Ba \rightarrow aB$
 4. $Bb \rightarrow bb$

Generative Grammars

- Production rules:
 1. $S \rightarrow aBSc$
 2. $S \rightarrow abc$
 3. $Ba \rightarrow aB$
 4. $Bb \rightarrow bb$
- Derivations of strings in the language $L(G)$:
 - $S \Rightarrow_2 abc$
 - $S \Rightarrow_1 aBSc \Rightarrow_2 aBabcc \Rightarrow_3 aaBbcc \Rightarrow_4 aabbcc$
 - $S \Rightarrow \dots \Rightarrow aaabbcc$
- $L(G) = \{a^n b^n c^n \mid n > 0\}$

Chomsky Hierarchy (1956)

- Type 0 grammars (unrestricted grammars)
 - Includes all formal grammars.
- Type 1 grammars (context-sensitive grammars).
 - Rules restricted to: $\alpha A \beta \rightarrow \alpha \gamma \beta$, where A is a non-terminal, and α , β , γ strings of terminals and non-terminals.
- **Type 2 grammars (context-free grammars).**
 - Rules restricted to $A \rightarrow \gamma$, where A is a non-terminal, and γ a string of terminals and non-terminals
- **Type 3 grammars (regular grammars).**
 - Rules restricted to $A \rightarrow \gamma$, where A is a non-terminal, and γ :
 - the empty string, or a single terminal symbol followed optionally by a non-terminal symbol.

Context Free Grammars (Type 2)

- Example:
 - $\Sigma = \{a,b\}$, $N = \{S\}$
 - P consists of the following production rules:
 1. $S \rightarrow aSb$
 2. $S \rightarrow \varepsilon$
 - $L(G) = ?$

CFGs provide the formal syntax specification of most programming languages.

Regular Grammars (Type 3)

- Example:
 - $\Sigma = \{a,b\}$, $N = \{S,A,B\}$
 - P consists of the following production rules:
 1. $S \rightarrow aS$
 2. $S \rightarrow cB$
 3. $B \rightarrow bB$
 4. $B \rightarrow \varepsilon$
 - $L(G) = ?$

Regular Grammars/Expressions provide the formal **lexical specification** of most programming languages.

Lexical Analysis (Chapter 4.2)

- A lexical analyzer is a “front-end” for the syntactic parser:
 - identifies substrings of the source program that belong together – **lexemes**.
 - lexemes are categorized into lexical categories called **tokens** such as: *keywords, identifiers, operators, numbers, strings, comments*.
- The lexemes of a PL can be formally specified using:
 - **Regular Grammars**.
 - **Regular Expressions**.
 - RE's \Leftrightarrow RG's (same generative power).

Lexical Analysis: Regular Expressions

- Operators:

- “+” { return (PLUS); } // PLUS = 201
- “-” { return (MINUS); } // MINUS = 202
- “*” { return (MULT); } // MULT = 203
- “/” { return (DIV); } // DIV = 204

- Identifiers:

- [a-zA-Z_][a-zA-Z_0-9]* { return (ID); } // ID = 200
- * is Kleene star and means *zero or more*.
- + means *one or more*
- . means *any character*.
- [^\t\n] means any character other than whitespaces.

Lexical Analysis: Regular Expressions

- Numbers:
 - `[1-9][0-9]*` { return (DECIMALINT); }
 - `0[0-7]*` { return (OCTALINT); }
 - `(0x|0X)[0-9a-fA-F]+` { return (HEXINT); }
 - + means *one or more*.
- Whitespaces:
 - `[\t\]+` { skip(); }
 - `[\r\n]` { newline++; skip(); }
- Each keywords is associated a token definition:
 - “bool” { return (BOOL); } // token 301
 - “break” { return (BREAK); } // token 302
 - ...

Lexical Analysis

- In practice, a **scanner generator** (e.g. *Lex*, *Flex*) reads such lexical definitions and automatically generates code for the lexical analyzer (**scanner**).
- The scanner is implemented as a deterministic **Finite State Automaton (FSA)**.
- An FSA is an abstract state machine that can be used to recognize tokens from a stream of characters.

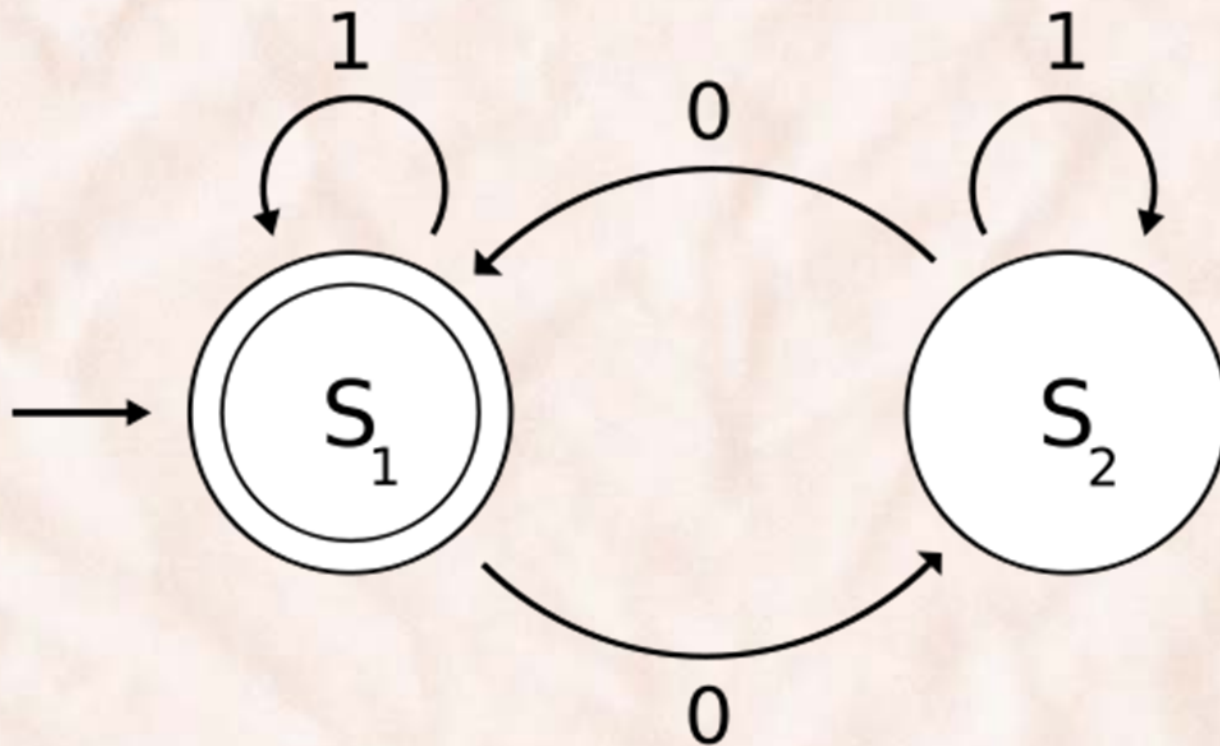
Finite State Automata

- A deterministic FSA is a tuple $(\Sigma, S, s_0, \delta, F)$:
 - Σ is the input alphabet (a finite set of symbols).
 - S is a finite set of states.
 - $s_0 \in S$ is the initial state.
 - $\delta: S \times \Sigma \rightarrow S$ is the state–transition function.
 - $F \subseteq S$ is the set of final states.

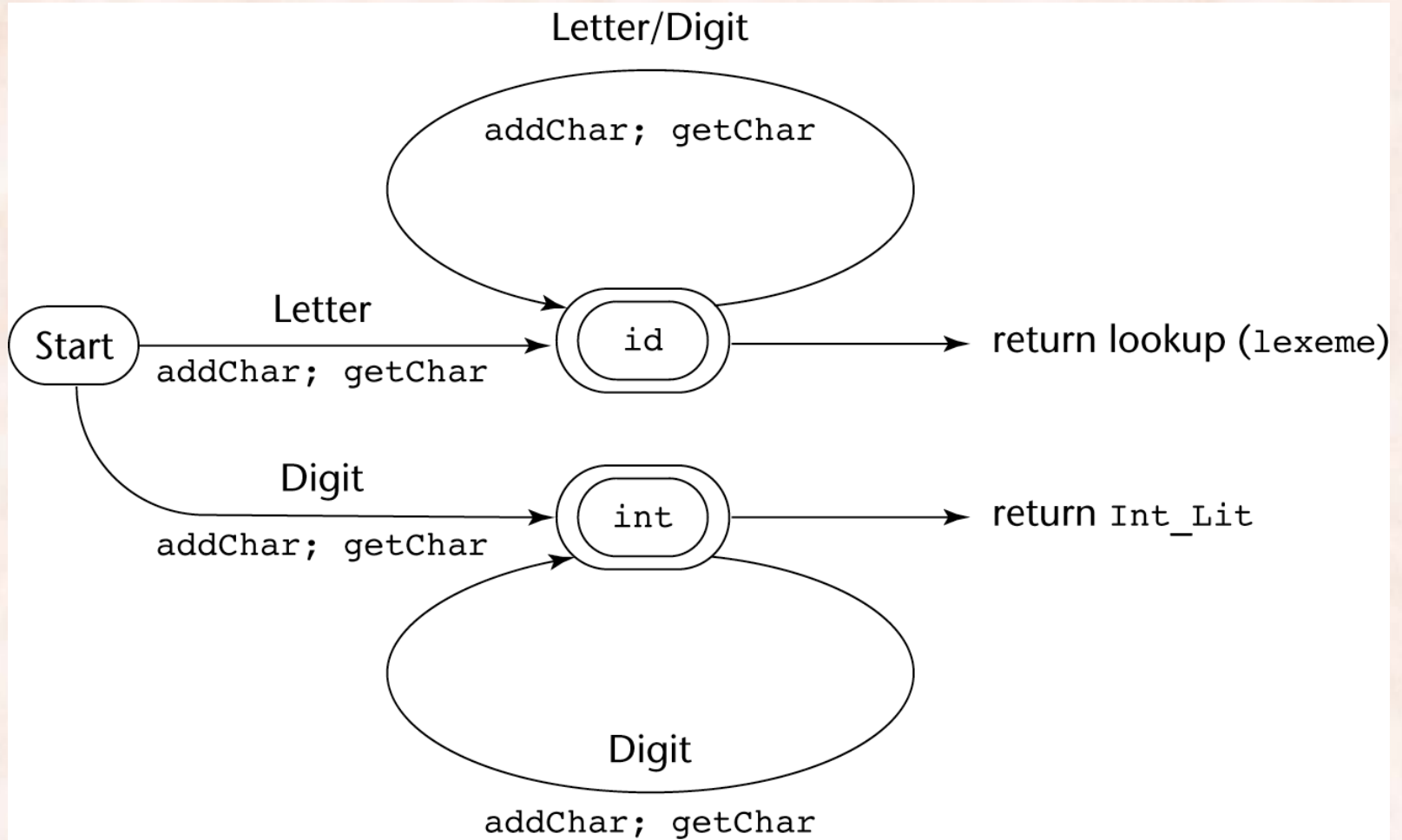
FSA: Representation & Implementation

- An FSA can be represented using **transition diagrams**.
- An FSA for recognizing integer literals, identifiers, and reserved words:
 - When recognizing an identifier, all uppercase and lowercase letters are equivalent \Rightarrow use a character class that includes all letters (`Letter`).
 - When recognizing an integer literal, all digits are equivalent \Rightarrow use a digit class (`Digit`).
 - Use a table lookup to determine whether a possible identifier is in fact a reserved word.

Transition Diagrams



Transition Diagrams



Syntax: Formal Specification using BNF

- Backus-Naur Form (BNF):
 - Invented by John Backus to describe Algol 58.
 - BNF is a metalanguage notation for Context Free Grammars, used for describing the syntax of programming languages.
 - Nonterminals are abstractions for syntactic constructs in the language (e.g. *expressions, statements, type declarations, etc.*)
 - Nonterminals are enclosed in angle brackets.
 - Terminals are lexemes or tokens.

Recursive Productions

- Syntactic lists are described using recursion:

$$\begin{aligned} \langle \text{ident_list} \rangle &\rightarrow \text{ident} \\ &\quad | \text{ident}, \langle \text{ident_list} \rangle \end{aligned}$$

- Simple expression grammar:

$$\begin{aligned} \langle \text{expr} \rangle &\rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle \\ &\quad | \langle \text{expr} \rangle * \langle \text{expr} \rangle \\ &\quad | a \quad | \quad b \quad | \quad c \end{aligned}$$

Grammars & Derivations

- A Simple Grammar:

$\langle \text{program} \rangle \rightarrow \langle \text{stmts} \rangle$

$\langle \text{stmts} \rangle \rightarrow \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle ; \langle \text{stmts} \rangle$

$\langle \text{stmt} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$

$\langle \text{var} \rangle \rightarrow a \mid b \mid c \mid d$

$\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle - \langle \text{term} \rangle$

$\langle \text{term} \rangle \rightarrow \langle \text{var} \rangle \mid \text{const}$

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

Grammars & Derivations

- An example derivation:

$\langle \text{program} \rangle \Rightarrow \langle \text{stmts} \rangle \Rightarrow \langle \text{stmt} \rangle$

$\Rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$

$\Rightarrow a = \langle \text{expr} \rangle$

$\Rightarrow a = \langle \text{term} \rangle + \langle \text{term} \rangle$

$\Rightarrow a = \langle \text{var} \rangle + \langle \text{term} \rangle$

$\Rightarrow a = b + \langle \text{term} \rangle$

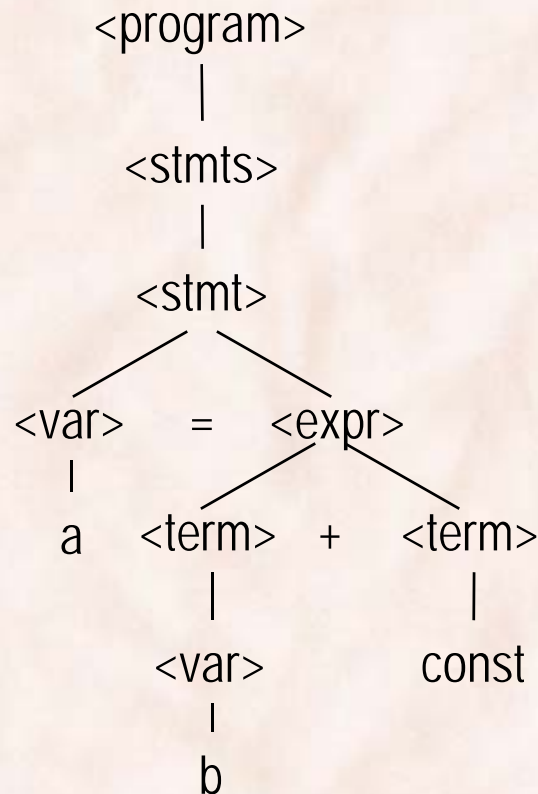
$\Rightarrow a = b + \text{const}$

Derivations

- A 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 *rightmost derivation* is one in which the rightmost nonterminal in each sentential form is the one that is expanded.
- A derivation may be neither leftmost nor rightmost

Parse Trees

- **Parse Tree** = a hierarchical representation of a derivation.



Parse Trees

- For any string from $L(G)$, a grammar G defines a recursive tree structure = Parse Tree.
- Parse Trees:
 - The root and intermediate nodes are nonterminals.
 - The leaf nodes are terminals.
 - For each rule used in a derivation step:
 - the LHS is a parent node.
 - the symbols in the RHS are children nodes (from left to right).

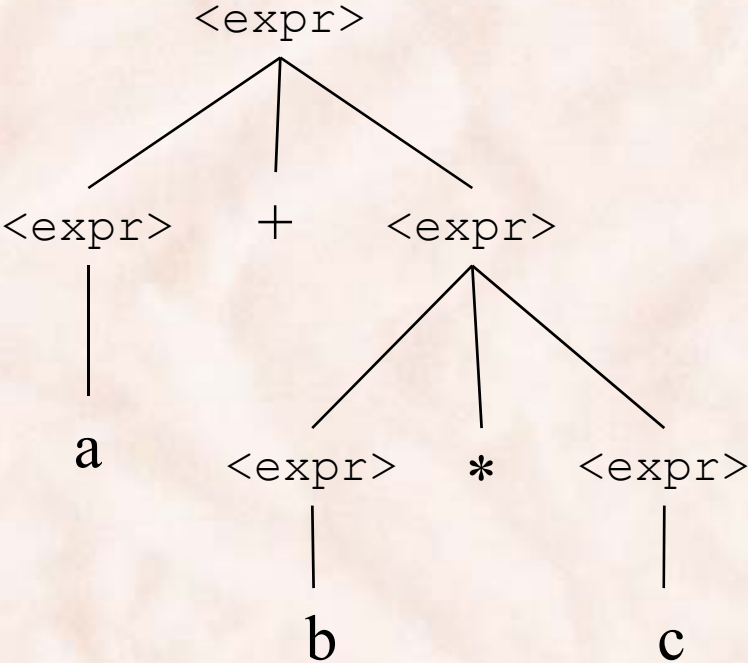
Syntactic Ambiguity

- A grammar is *ambiguous* if and only if it can generate a sentence that has two or more distinct parse trees.
- A grammar is *ambiguous* if a sentence has more than one leftmost derivations.
- This simple expression grammar is ambiguous :

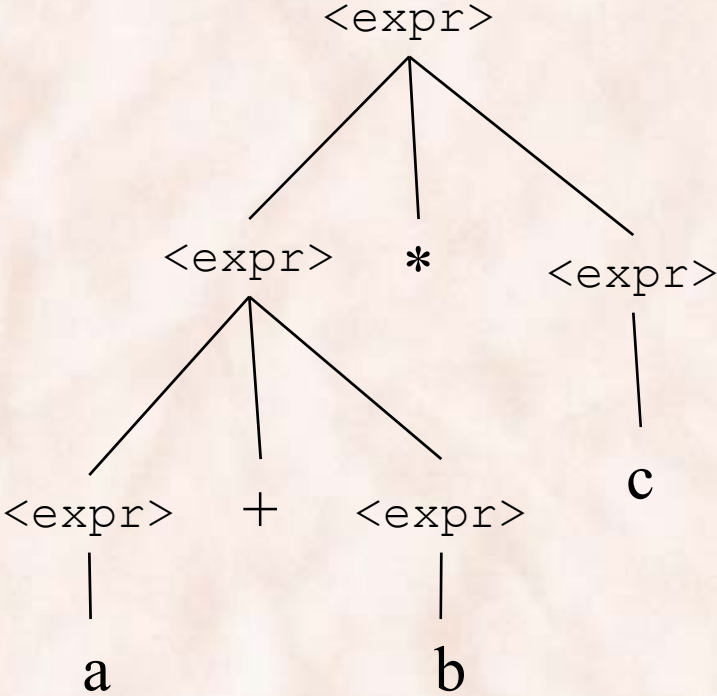
$$\begin{aligned} \langle \text{expr} \rangle &\rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle \\ &\quad | \langle \text{expr} \rangle * \langle \text{expr} \rangle \\ &\quad | a \quad | \quad b \quad | \quad c \end{aligned}$$

Syntactic Ambiguity

* lower than +



+ lower than *



Operator Precedence

- The expression string “ $a + b * c$ ” has two different parse trees:
 - Q: Which one is “correct”?
 - A: Both are syntactically correct, but we prefer the first one:
 - Its structure is closer to the the correct semantics of the expression.
 - Want meaning of the expression to be easily determined from its parse tree \Rightarrow need parse tree to encode precedence rules.
 - Operator ‘ $*$ ’ generated lower in the parse tree than ‘ $+$ ’ means that ‘ $*$ ’ has higher precedence than ‘ $+$ ’.

Operator Precedence

- Expression grammar that encodes precedence rules:

$$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle$$
$$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{fact} \rangle \mid \langle \text{fact} \rangle$$
$$\langle \text{fact} \rangle \rightarrow a \mid b \mid c$$

- What is the parse tree for “a + b * c”?
- What is the parse tree for “a + b + c”?
- Is this new grammar non-ambiguous?

Associativity of operators

- Associativity, like precedence, can be encoded in the grammar:

$$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle$$
$$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{fact} \rangle \mid \langle \text{fact} \rangle$$
$$\langle \text{fact} \rangle \rightarrow a \mid b \mid c$$

- Left recursive rules \Rightarrow left associative operators.
 - Right recursive rules \Rightarrow right associative operators.
- What are the parse trees for “a + b * c” & “a + b + c”?

Associativity of Operators

- Introducing the exponentiation operator ‘^’:

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle$

$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{fact} \rangle \mid \langle \text{fact} \rangle$

$\langle \text{fact} \rangle \rightarrow \langle \text{base} \rangle ^ \langle \text{fact} \rangle \mid \langle \text{base} \rangle$

$\langle \text{base} \rangle \rightarrow a \mid b \mid c$

- What is the precedence of ‘+’, ‘*’, ‘^’?
- What is the associativity of ‘^’?

Syntax vs. Semantics

- Operator precedence and associativity are **semantic rules**.
- CFGs are used to specify **syntactic rules**.
- The grammar can be written to encode semantic rules.
Why is this useful?

Syntax vs. Semantics

- The CFG specification is used to build a **Syntactic Analyzer**.
- The Syntactic Analyzer verifies that the input is a *syntactically correct* program.
- The Syntactic Analyzer generates a parse tree that is used in **Intermediate Code Generation** to eventually generate *semantically correct* machine code.
- Hence, the need for parse trees that are both *syntactically correct* and *semantically correct*.

The “Dangling Else” Ambiguity

- Initial grammar rules for the `if-then-else` statement :

```
<if_stmt> → if <logic_expr> then <stmt>  
          | if <logic_expr> then <stmt> else <stmt>
```

```
<stmt> → <if_stmt>  
        | <other_stmt>
```

- Why is this grammar ambiguous?
- Rewrite the grammar to reflect semantic constraints on the `if-then-else` statement (Chapter 3.1.1.10).

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 vs. EBNF

- BNF

`<expr> → <expr> + <term>`
`| <expr> - <term>`
`| <term>`

`<term> → <term> * <factor>`
`| <term> / <factor>`
`| <factor>`

- EBNF

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

Syntactic Analysis: The Problem

- **Syntactic Analysis (Parsing)** = a computing problem:
 - Input:
 - a context free grammar.
 - a sequence of tokens.
 - Output:
 - *YES* if the input can be generated by the CFG.
 - The parse tree \Rightarrow need unambiguous grammar.
 - *NO* if the input cannot be generated by the CFG.
 - Find all syntax errors; for each, produce an appropriate diagnostic message and recover quickly.

Syntactic Analysis: The Algorithms

- **Syntactic Analyzer (Parser)** = an algorithm/program that solves the syntactic analysis problem.
- Time Complexity of syntactic parsing algorithms:
 - Parsers that work for any unambiguous CFG are complex and inefficient – $O(n^3)$:
 - Cocke-Younger-Kasami (CYK) bottom-up parsing algorithm.
 - Compilers use parsers that only work for a subset of all unambiguous CFG grammars, but do it in linear time – $O(n)$:
- Two categories of parsers:
 - Top-down (LL)
 - Bottom-up (LR)

Top-down Parsers

- **Top down** – produce the parse tree, beginning at the root:
 - Traces or builds the parse tree in preorder.
 - Most common are LL(k):
 - L: a left-to-right scanning of the input.
 - L: corresponds to a leftmost derivation.
 - k: number of lookahead symbols.
 - Given a sentential form, $xA\alpha$, the parser must choose the correct A-rule to get the next sentential form in the leftmost derivation, using only the first k tokens produced by A.
 - Useful parsers look only one token ahead in the input \Rightarrow LL(1).

Top-down Parsers

- The most common top-down parsing algorithms:
 - Recursive descent – a coded implementation, based directly on the BNF description of the language.
 - Table driven implementation – a parsing table is used to implement the BNF rules.
- Implementation Methods:
 - Manually coded.
 - Generated automatically:
 - ANTLR is an LL(*) parser generator [www.antlr.org].
 - JavaCC is an LL(k) parser generator [javacc.dev.java.net]

Bottom-up Parsing

- **Bottom up** – produce the parse tree, beginning at the leaves:
 - Most common are LR(k):
 - L: a left-to-right scanning of the input.
 - R: corresponds to the reverse of a rightmost derivation.
 - k: number of lookahead symbols.
 - Given a right sentential form, α , determine what substring of α is the right-hand side of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation.
 - Useful parsers look only one token ahead in the input \Rightarrow LR(1).
- LR Parser generators:
 - yacc (Stephen Johnson for UNIX) , bison (GNU version of yacc).

Recursive Descent Parsing

- There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal.
- Assume we have a lexical analyzer named `lex()`, which puts the next token code in `nextToken`.
- The coding process when there is only one RHS:
 - For each terminal symbol in the RHS, compare it with `nextToken`;
 - if they match, continue;
 - else there is an error.
 - For each nonterminal symbol in the RHS, call its associated parsing subprogram \Rightarrow problem if grammar is Left Recursive.

Recursive Descent Parsing

- Left Recursive grammar:

```
<expr> → <expr> + <term>
        | <expr> - <term>
        | <term>
```

```
<term> → <term> * <factor>
        | <term> / <factor>
        | <factor>
```

```
<factor> → id
```

- Cannot do recursive descent parsing:

```
void expr() { expr(); ... } ⇒ infinite recursion!
```

Recursive Descent Parsing

- An expression grammar that has no left recursion:
 $\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle \{ (+ \mid -) \langle \text{term} \rangle \}$
 $\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ (* \mid /) \langle \text{factor} \rangle \}$
 $\langle \text{factor} \rangle \rightarrow \text{id} \mid (\langle \text{expr} \rangle)$
- Added support for parentheses.
- Left recursion can be eliminated automatically for any CFG.

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

```
void expr() {  
    /* Parse the first term */  
    term();  
    /* As long as the next token is + or -, call  
       lex to get the next token, and parse the  
       next term */  
    while (nextToken == PLUS_CODE ||  
           nextToken == MINUS_CODE) {  
        lex();  
        term();  
    }  
}
```

Recursive Descent Parsing

- Convention: Every parsing routine leaves the next token in `nextToken`.
- A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse:
 - The correct RHS is chosen on the basis of the next token of input (the lookahead).
 - The next token is compared with the first token that can be generated by each RHS until a match is found.
 - If no match is found, output a syntax error.

`<factor> → id | (<expr>)`

```
void factor() {
    /* Determine which RHS */
    if (nextToken) == ID_CODE)
        /* For the RHS id, just call lex */
        lex();
    else if (nextToken == LEFT_PAREN_CODE) {
        lex();
        expr();
        if (nextToken == RIGHT_PAREN_CODE)
            lex();
        else
            error();
    }
    else error(); /* Neither RHS matches */
}
```

The LL Grammars

- The Left Recursion problem:
 - If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser.
 - A grammar can be modified to remove direct left recursion.
For each nonterminal, A ,
 1. Group the A -rules as $A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$
where none of the β 's begins with A
 2. Replace the original A -rules with:
$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$$
 - [Aho et al., 1986] give an algorithm to remove left recursion from any CFG.

Eliminating Left Recursion

- Left Recursive grammar:

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

- Exercise: Transform into an equivalent grammar w/o left recursion.

The LL Grammars

- The lack of Pairwise Disjointness:
 - The inability to determine the correct RHS on the basis of one token of lookahead
 - Def: $FIRST(\alpha) = \{a \mid \alpha \Rightarrow_* a\beta\}$, where \Rightarrow_* means zero or more derivation steps.
 - [Aho et al., 1986] give an algorithm to compute $FIRST(\alpha)$.
- Pairwise Disjointness Test:
 - For each nonterminal A in the grammar that has more than one RHS, for each pair of rules, $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$, it must be true that:

$$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi$$

LL Grammars: Pairwise Disjointness

- Example:

$$A \rightarrow aB \mid bAb \mid Bb$$
$$B \rightarrow cB \mid d$$

- Example:

$$\langle \text{variable} \rangle \rightarrow \text{identifier} \mid \text{identifier} [\langle \text{expr} \rangle]$$

- Pairwise Disjointness hard to solve in general case.
- In some cases, Left Factoring can solve the problem.

LL Grammars: Left Factoring

- **Replace:**

`<variable> → identifier | identifier [<expr>]`

- **With:**

`<variable> → identifier <new>`

`<new> → ε | [<expression>]`

or

`<variable> → identifier [[<expression>]]`

(the outer brackets are metasymbols of EBNF)

Summary

- Generative Grammars
 - Regular Grammars (RG) for lexical analysis.
 - Context Free Grammars (CFG) for syntactic analysis.
- Lexical Analysis
 - RG, Regular Expressions, Finite State Automata (FSA).
 - Implementation: FSA.
- Syntactic Analysis:
 - CFGs specified using BNF.
 - Implementation:
 - Top-down parsing (e.g. Recursive Descent).
 - Bottom-up Parsing.