Language Specification and Translation

ICOM 4036

Spring 2004

Lecture 3

Language Specification and Translation Topics

- Structure of a Compiler
- Lexical Specification and Scanning
- Syntactic Specification and Parsing
- Semantic Specification and Analysis

Syntax versus Semantics

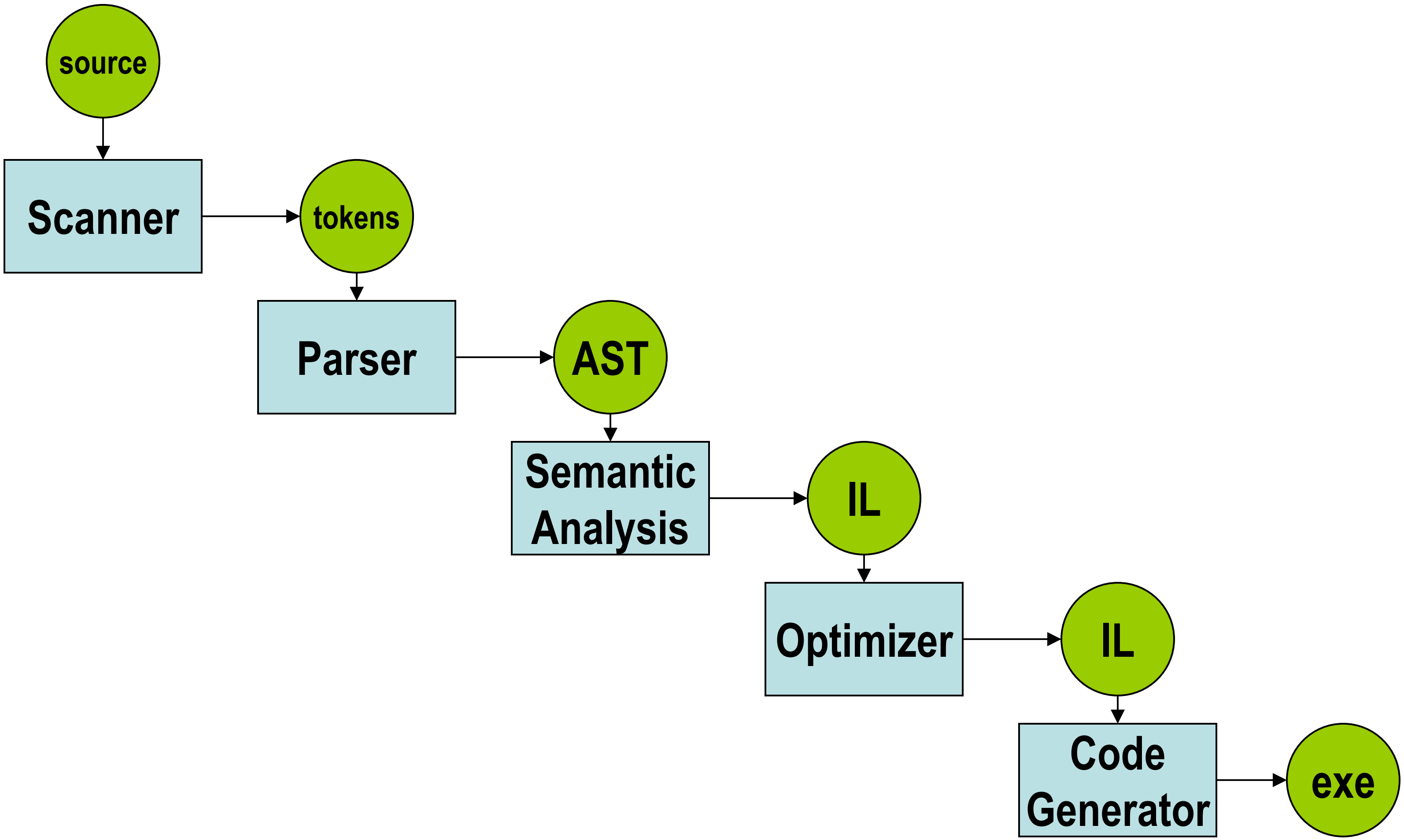
- Syntax - the form or structure of the expressions, statements, and program units
- Semantics - the meaning of the expressions, statements, and program units

The Structure of a Compiler

1. Lexical Analysis
2. Parsing
3. Semantic Analysis
4. Optimization
5. Code Generation

The first 3, at least, can be understood by analogy to how humans comprehend English.

A Prototypical Compiler



Introduction

- Reasons to separate compiler in phases:
  - 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

- First step: recognize words.
  - Smallest unit above letters

This is a sentence.

- Note the
  - Capital “T” (start of sentence symbol)
  - Blank “ “ (word separator)
  - Period “.” (end of sentence symbol)

Lexical Analysis

- Lexical analysis is not trivial. Consider:

  ist his ase nte nce

- Plus, programming languages are typically more cryptic than English:

  *p->f ++ = -.12345e-5

Lexical Analysis

- Lexical analyzer divides program text into "words" or "tokens"

  if x == y then z = 1; else z = 2;

- Units:

  if, x, ==, y, then, z, =, 1, ;, else, z, =, 2, ;

Lexical Analysis

- A lexical analyzer is a pattern matcher for character strings
- A lexical analyzer is a “front-end” for the parser
- Identifies substrings of the source program that belong together - lexemes
  - Lexemes match a character pattern, which is associated with a lexical category called a token
  - **`sum`** is a lexeme; its token may be **`IDENT`**

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 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
- We only discuss approach 2

State diagram = Finite State Machine

Lexical Analysis

- State diagram design:
  - A naïve state diagram would have a transition from every state on every character in the source language - such a diagram would be very large!
- In many cases, transitions can be combined to simplify the state diagram
  - When recognizing an identifier, all uppercase and lowercase letters are equivalent
    - Use a character class that includes all letters
  - When recognizing an integer literal, all digits are equivalent - use a digit class

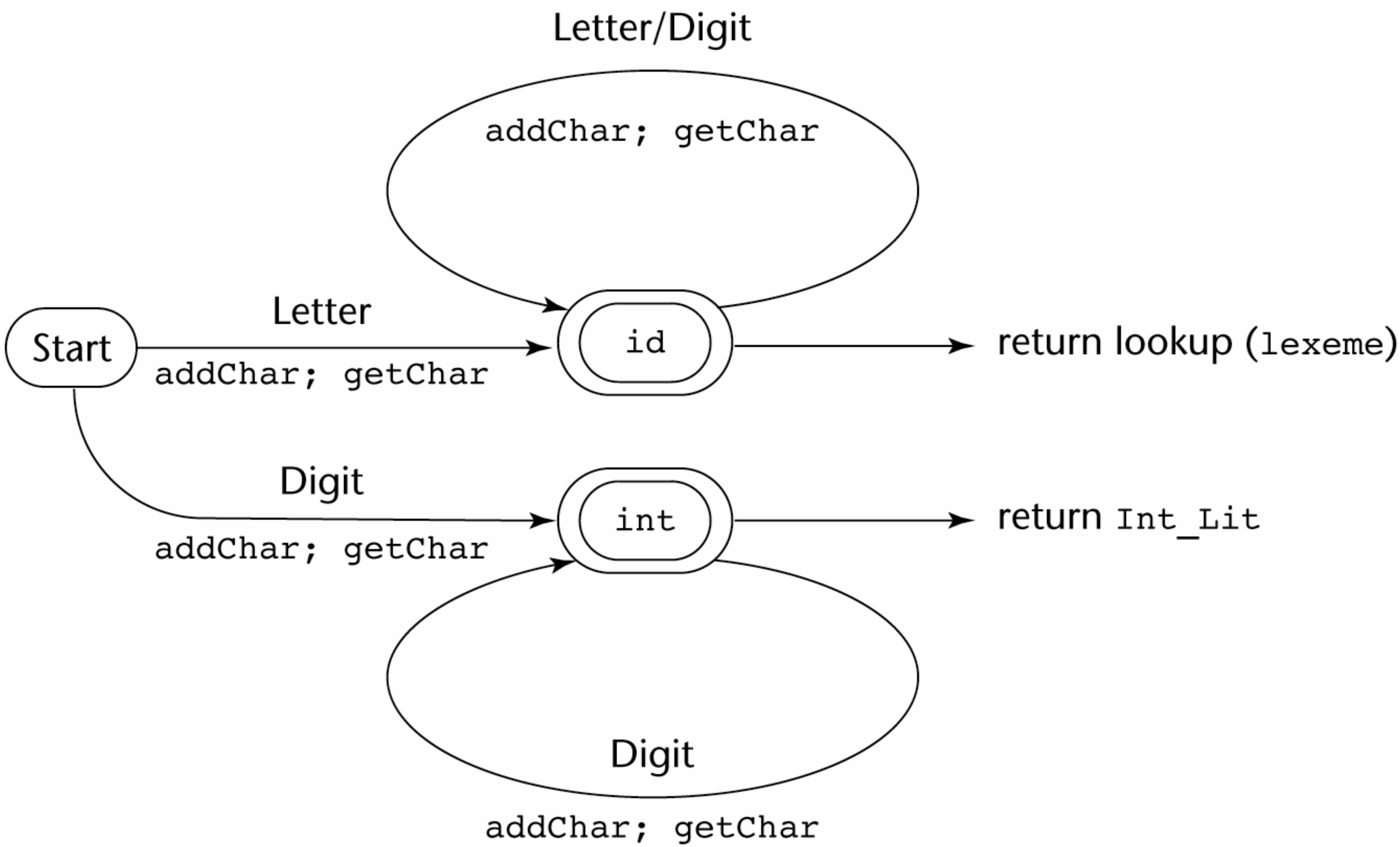
Lexical Analysis

- Reserved words and identifiers can be recognized together (rather than having a part of the diagram for each reserved word)
  - Use a table lookup to determine whether a possible identifier is in fact a reserved word

Lexical Analysis

- Convenient 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)

State Diagram



Lexical Analysis

- Implementation (assume initialization):

```
int lex() {
  getChar();
  switch (charClass) {
    case LETTER:
      addChar();
      getChar();
      while (charClass == LETTER || charClass == DIGIT)
      {
        addChar();
        getChar();
      }
      return lookup(lexeme);
      break;
       …
```

Lexical Analysis

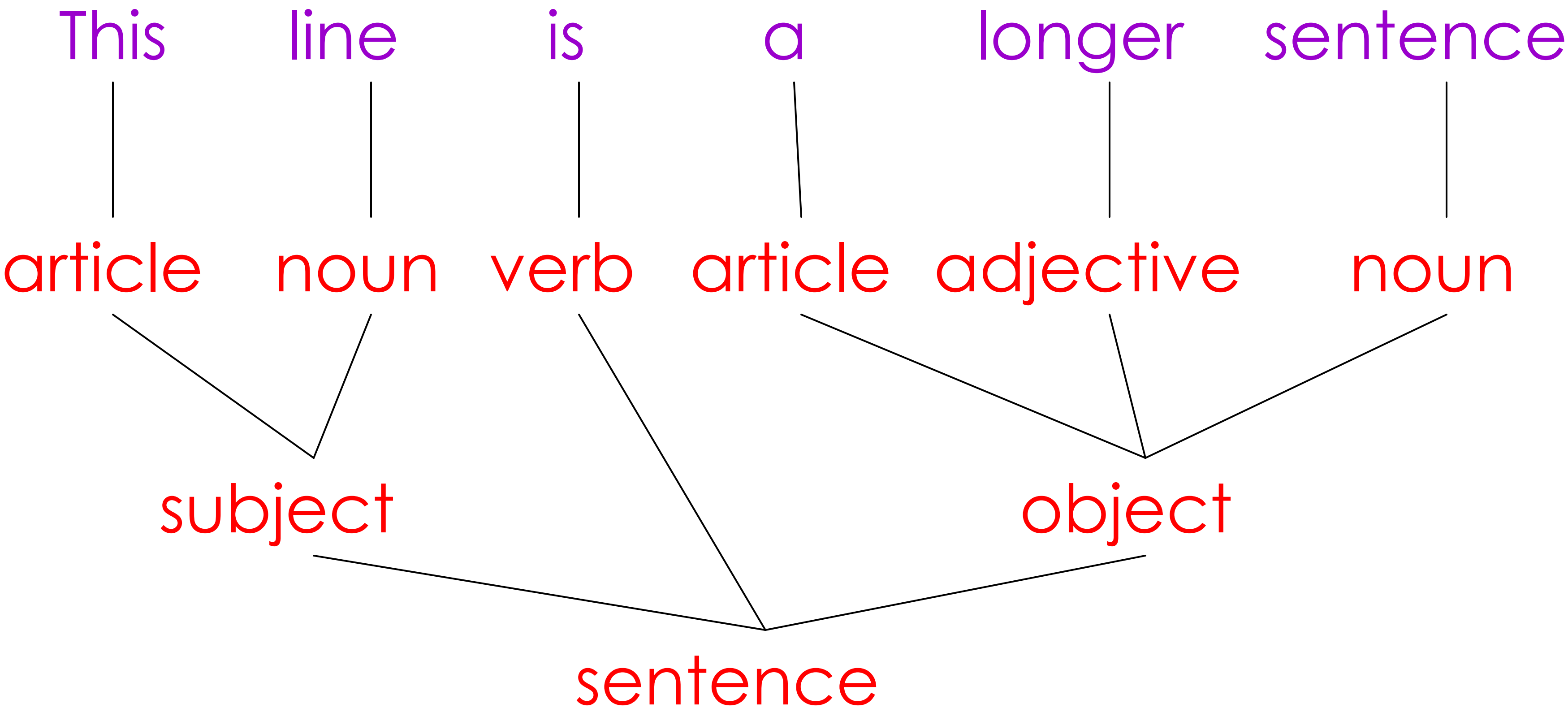
```
…
case DIGIT:
      addChar();
      getChar();
      while (charClass == DIGIT) {
        addChar();
        getChar();
      }
      return INT_LIT;
      break;
  }  /* End of switch */
}  /* End of function lex */
```

Parsing

- Once words are understood, the next step is to understand sentence structure

- Parsing = Diagramming Sentences
  - The diagram is a tree

Diagramming a Sentence

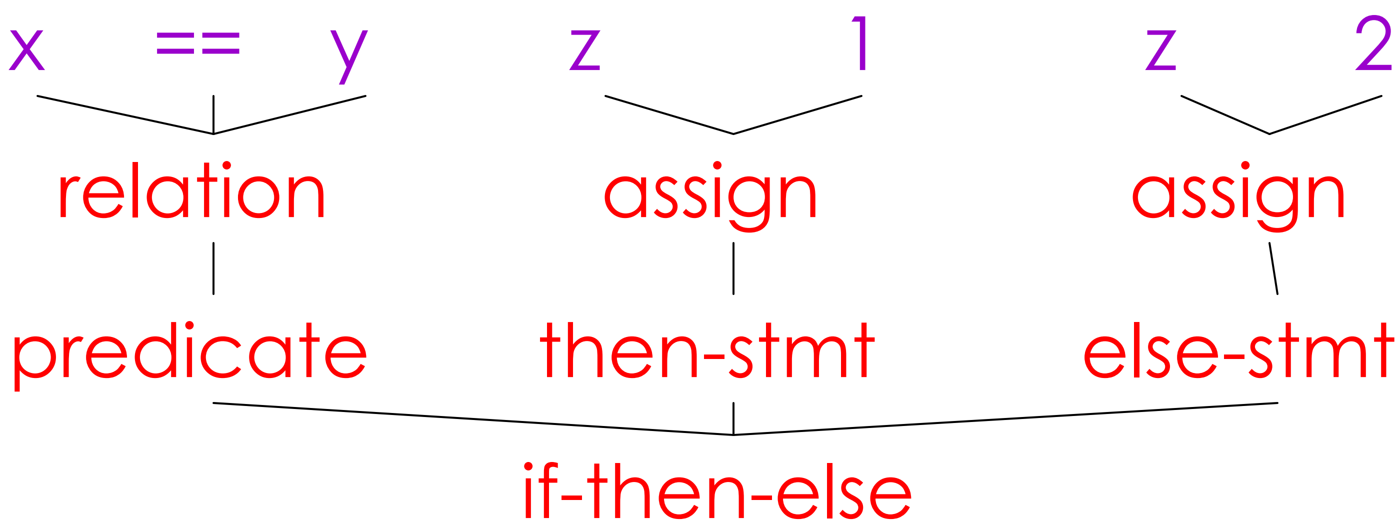


Parsing Programs

- Parsing program expressions is the same
- Consider:

  If x == y then z = 1; else z = 2;

- Diagrammed:



Describing Syntax

- 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

- Formal approaches to describing syntax:
  - Recognizers - used in compilers (we will look at in Chapter 4)
  - Generators – generate the sentences of a language (what we'll study in this chapter)

Formal Methods of Describing Syntax

- 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

Formal Methods of Describing Syntax

- Backus-Naur Form (1959)
  - Invented by John Backus to describe Algol 58
  - BNF is equivalent to context-free grammars
  - A metalanguage is a language used to describe another language.
  - In BNF, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (also called nonterminal symbols)

Backus-Naur Form (1959)

<while_stmt> $\rightarrow$ `while` (<logic_expr>) <stmt>

- This is a rule; it describes the structure of a while statement

Formal Methods of Describing Syntax

- A rule has a left-hand side (LHS) and a right-hand side (RHS), and consists of terminal and nonterminal symbols
- A grammar is a finite nonempty set of rules
- An abstraction (or nonterminal symbol) can have more than one RHS

  <stmt> $\rightarrow$ **<single_stmt>**

  | **`begin`** **<stmt_list>** **`end`**

Formal Methods of Describing Syntax

- Syntactic lists are described using recursion

  <ident_list> → `ident`

  | `ident`, **<ident_list>**

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

Formal Methods of Describing 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`

Formal Methods of Describing Syntax

- An example derivation:

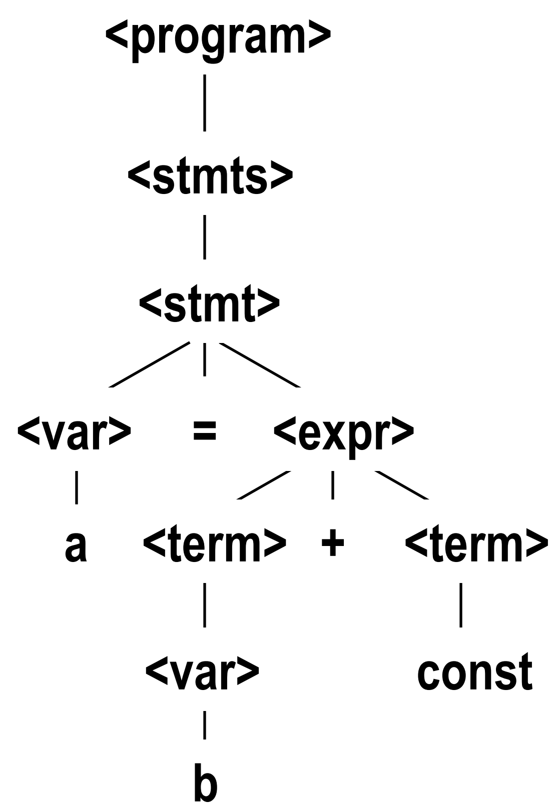
<program> => <stmts> => <stmt>
=> <var> = <expr> => a = <expr>
=> a = <term> + <term>
=> a = <var> + <term>
=> a = b + <term>
=> a = b + const

Derivation

- 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 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



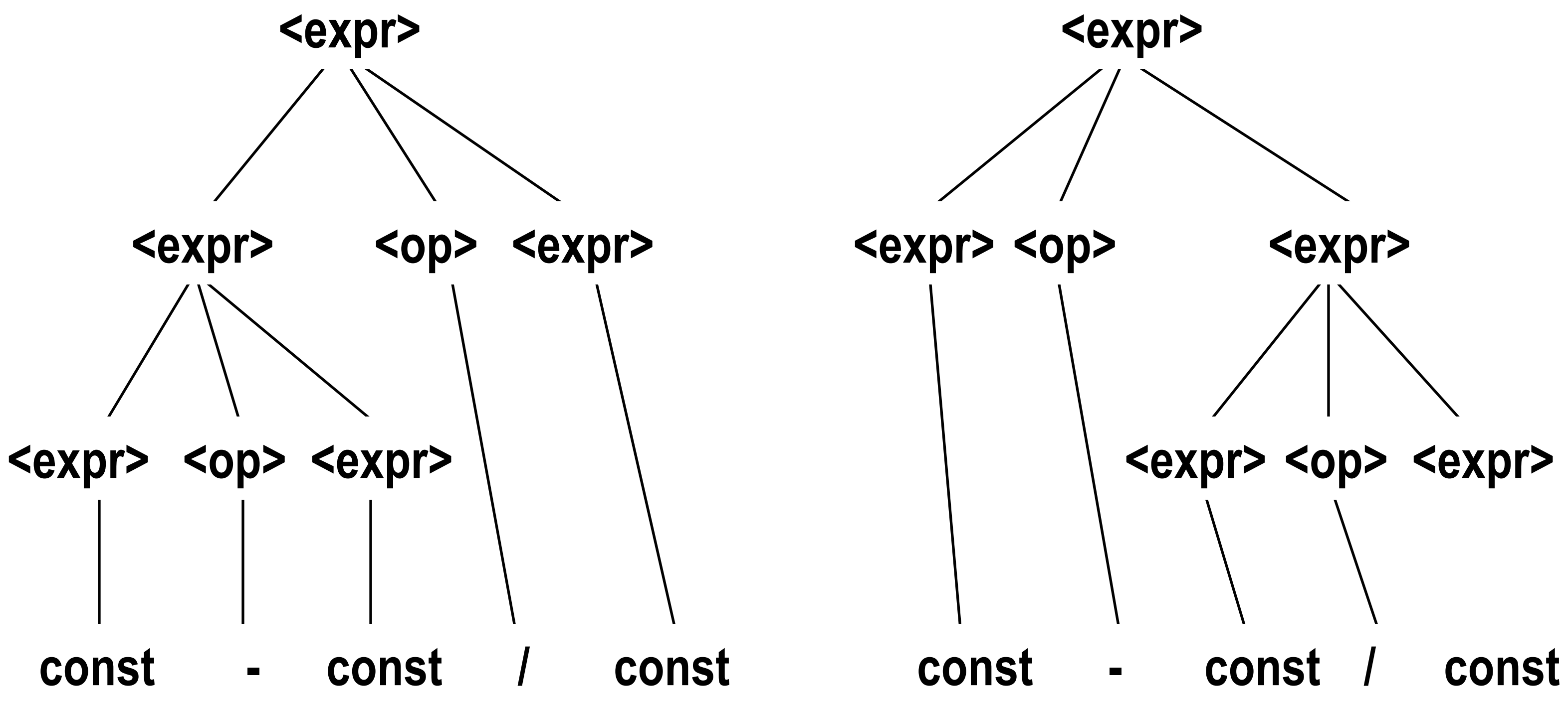
Formal Methods of Describing Syntax

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

An Ambiguous Expression Grammar

<expr> $\rightarrow$ <expr> <op> <expr> | const

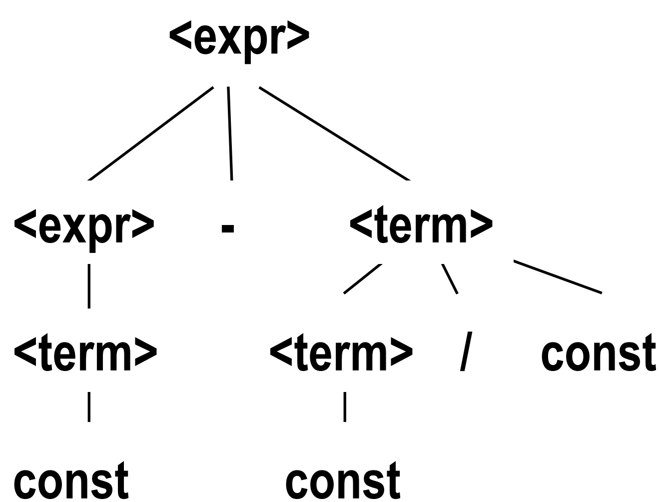
<op> $\rightarrow$ / | -



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`



Formal Methods of Describing Syntax

Derivation:

<expr> => <expr> - <term> => <term> - <term>

=> `const` - <term>

=> `const` - <term> `/ const`

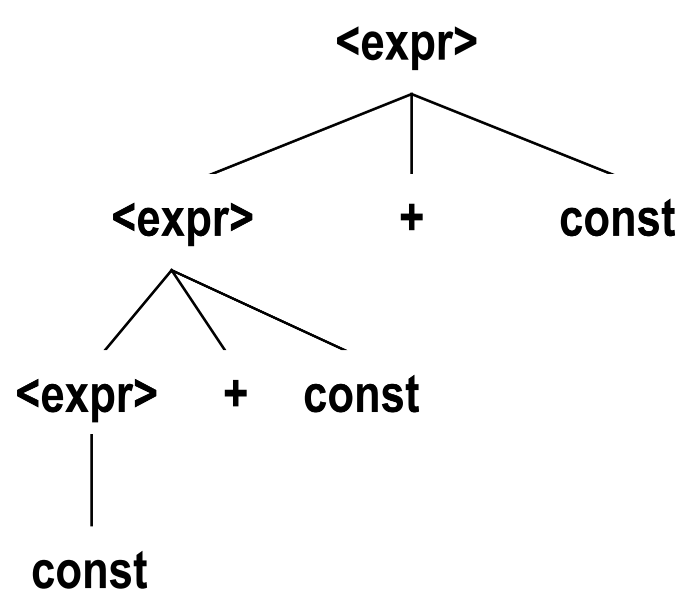
=> `const - const / const`

Formal Methods of Describing Syntax

- Operator associativity can also be indicated by a grammar

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

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



Formal Methods of Describing Syntax

- Extended BNF (just abbreviations):
  - Optional parts are placed in brackets ([ ])

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

  - Put alternative parts of RHSs 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>}

The Parsing Problem

- Goals of the parser, given an input program:
  - Find all syntax errors; for each, produce an appropriate diagnostic message, and recover quickly
  - Produce the parse tree, or at least a trace of the parse tree, for the program

The Parsing Problem

- Two categories of parsers
  - Top down - produce the parse tree, beginning at the root
    - Order is that of a leftmost derivation
  - Bottom up - produce the parse tree, beginning at the leaves
    - Order is that of the reverse of a rightmost derivation
- Parsers look only one token ahead in the input

The Parsing Problem

- Top-down Parsers
  - 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 token produced by A
- The most common top-down parsing algorithms:
  - Recursive descent - a coded implementation
  - LL parsers - table driven implementation

The Parsing Problem

- Bottom-up parsers
  - 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
  - The most common bottom-up parsing algorithms are in the LR family

The Parsing Problem

- The Complexity of Parsing
  - Parsers that work for any unambiguous grammar are complex and inefficient ($O(n^3)$, where n is the length of the input)
  - Compilers use parsers that only work for a subset of all unambiguous grammars, but do it in linear time ($O(n)$, where n is the length of the input)

Recursive-Descent Parsing

- Recursive Descent Process
  - There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal
  - EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of nonterminals

Recursive-Descent Parsing

- A grammar for simple expressions:

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


Recursive-Descent Parsing

- 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 the next input token; if they match, continue, else there is an error
  - For each nonterminal symbol in the RHS, call its associated parsing subprogram

Recursive-Descent Parsing

```
/* Function expr
   Parses strings in the language
   generated by the rule:
   <expr> → <term> {(+ | -) <term>}
 */

void expr() {

/* Parse the first term */

  term();
  …
```

Recursive-Descent Parsing

```
/* 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();
  }
}
```

- This particular routine does not detect errors
- Convention: Every parsing routine leaves the next token in **`nextToken`**

Recursive-Descent Parsing

- 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, it is a syntax error

Recursive-Descent Parsing

```
/* Function factor
   Parses strings in the language
   generated by the rule:
   <factor> -> id  |  (<expr>)  */

 void factor() {

 /* Determine which RHS */

   if (nextToken) == ID_CODE)

 /* For the RHS id, just call lex */

     lex();
```

Recursive-Descent Parsing

```
/* If the RHS is (<expr>) - call lex to pass
      over the left parenthesis, call expr,
  and
      check for the right parenthesis */

   else if (nextToken == LEFT_PAREN_CODE) {
      lex();
      expr();
      if (nextToken == RIGHT_PAREN_CODE)
        lex();
      else
        error();
   }  /* End of else if (nextToken == ...  */

   else error(); /* Neither RHS matches */
 }
```

Recursive-Descent Parsing

- The LL Grammar Class
  - 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 left recursion

Recursive-Descent Parsing

- The other characteristic of grammars that disallows top-down parsing is the lack of pairwise disjointness
  - The inability to determine the correct RHS on the basis of one token of lookahead
  - Def: FIRST(α) = $\{a \mid \alpha =>^* a\beta\}$

    (If $\alpha =>^* \varepsilon$, ε is in FIRST(α))

Recursive-Descent Parsing

- 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

    $$\text{FIRST}(\alpha_i) \;\; \text{FIRST}(\alpha_j) = \phi$$

- Examples:

$$A \rightarrow a \mid bB \mid cAb$$

$$A \rightarrow a \mid aB$$

Recursive-Descent Parsing

- Left factoring can resolve the problem

  Replace

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

with

<variable> → identifier <new>

<new> → ε | [<expression>]

or

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

(the outer brackets are metasymbols of EBNF)

Bottom-up Parsing

- The parsing problem is finding the correct RHS in a right-sentential form to reduce to get the previous right-sentential form in the derivation

Bottom-up Parsing

- Intuition about handles:
  - Def: β is the handle of the right sentential form $\gamma = \alpha\beta w$ if and only if $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$
  - Def: β is a phrase of the right sentential form γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1 \beta \alpha_2$
  - Def: β is a simple phrase of the right sentential form γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$

Bottom-up Parsing

- Intuition about handles:
  - The handle of a right sentential form is its leftmost simple phrase
  - Given a parse tree, it is now easy to find the handle
  - Parsing can be thought of as handle pruning

Bottom-up Parsing

- Shift-Reduce Algorithms
  - Reduce is the action of replacing the handle on the top of the parse stack with its corresponding LHS
  - Shift is the action of moving the next token to the top of the parse stack

Bottom-up Parsing

- Advantages of LR parsers:
  - They will work for nearly all grammars that describe programming languages.
  - They work on a larger class of grammars than other bottom-up algorithms, but are as efficient as any other bottom-up parser.
  - They can detect syntax errors as soon as it is possible.
  - The LR class of grammars is a superset of the class parsable by LL parsers.

Bottom-up Parsing

- LR parsers must be constructed with a tool
- Knuth's insight: A bottom-up parser could use the entire history of the parse, up to the current point, to make parsing decisions
  - There were only a finite and relatively small number of different parse situations that could have occurred, so the history could be stored in a parser state, on the parse stack

Bottom-up Parsing

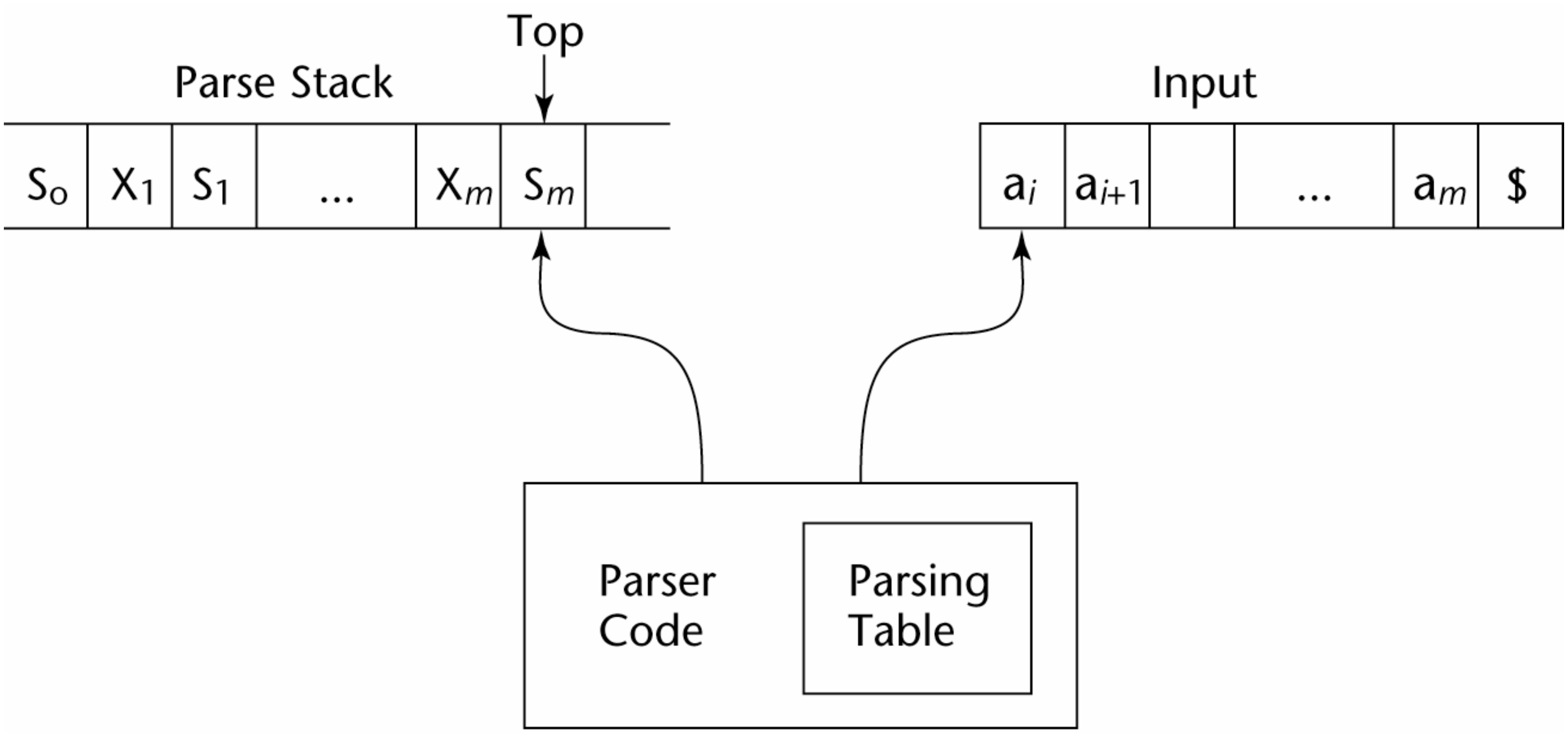
- An LR configuration stores the state of an LR parser

$(S_0X_1S_1X_2S_2\ldots X_mS_m,\ a_ia_i{+}1\ldots a_n\$)$

Bottom-up Parsing

- LR parsers are table driven, where the table has two components, an ACTION table and a GOTO table
  - The ACTION table specifies the action of the parser, given the parser state and the next token
    - Rows are state names; columns are terminals
  - The GOTO table specifies which state to put on top of the parse stack after a reduction action is done
    - Rows are state names; columns are nonterminals

Structure of An LR Parser



Bottom-up Parsing

- Initial configuration: $(S_0, a_1 \ldots a_n\$)$
- Parser actions:
  - If $\text{ACTION}[S_m, a_i]$ = Shift S, the next configuration is:

    $(S_0X_1S_1X_2S_2 \ldots X_mS_ma_iS, a_{i+1} \ldots a_n\$)$
  - If $\text{ACTION}[S_m, a_i]$ = Reduce $A \rightarrow \beta$ and S = $\text{GOTO}[S_{m-r}, A]$, where r = the length of β, the next configuration is

    $(S_0X_1S_1X_2S_2 \ldots X_{m-r}S_{m-r}AS, a_ia_{i+1} \ldots a_n\$)$

Bottom-up Parsing

- Parser actions (continued):
  - If ACTION[S_m, a_i] = Accept, the parse is complete and no errors were found.
  - If ACTION[S_m, a_i] = Error, the parser calls an error-handling routine.

LR Parsing Table

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Bottom-up Parsing

- A parser table can be generated from a given grammar with a tool, e.g., **`yacc`**

Semantic Analysis

- Once sentence structure is understood, we can try to understand "meaning"
  - But meaning is too hard for compilers
- Compilers perform limited analysis to catch inconsistencies
- Some do more analysis to improve the performance of the program

Semantic Analysis in English

- Example:

  Jack said Jerry left his assignment at home.

  What does “his” refer to? Jack or Jerry?

- Even worse:

  Jack said Jack left his assignment at home?

  How many Jacks are there?

  Which one left the assignment?

Semantic Analysis in Programming

- Programming languages define strict rules to avoid such ambiguities
- This C++ code prints “4”; the inner definition is used

```
{
   int Jack = 3;
   {
      int Jack = 4;
      cout << Jack;
   }
}
```

More Semantic Analysis

- Compilers perform many semantic checks besides variable bindings
- Example:

  Jack left her homework at home.

- A “type mismatch” between her and Jack; we know they are different people
  - Presumably Jack is male

Optimization

- No strong counterpart in English, but akin to editing
- Automatically modify programs so that they
  - Run faster
  - Use less memory
  - In general, conserve some resource
- The project has no optimization component

Optimization Example

X = Y * 0 is the same as X = 0

NO!

Valid for integers, but not for floating point numbers

Code Generation

- Produces assembly code (usually)
- A translation into another language
  - Analogous to human translation

Intermediate Languages

- Many compilers perform translations between successive intermediate forms
  - All but first and last are *intermediate languages* internal to the compiler
  - Typically there is 1 IL

- IL's generally ordered in descending level of abstraction
  - Highest is source
  - Lowest is assembly

Intermediate Languages (Cont.)

- IL's are useful because lower levels expose features hidden by higher levels
  - registers
  - memory layout
  - etc.

- But lower levels obscure high-level meaning

Issues

- Compiling is almost this simple, but there are many pitfalls.

- Example: How are erroneous programs handled?

- Language design has big impact on compiler
  - Determines what is easy and hard to compile
  - Course theme: many trade-offs in language design

Compilers Today

- The overall structure of almost every compiler adheres to our outline
- The proportions have changed since FORTRAN
  - Early: lexing, parsing most complex, expensive
  - Today: optimization dominates all other phases, lexing and parsing are cheap

Trends in Compilation

- Compilation for speed is less interesting. But:
  - scientific programs
  - advanced processors (Digital Signal Processors, advanced speculative architectures)
- Ideas from compilation used for improving code reliability:
  - memory safety
  - detecting concurrency errors (data races)
  - ...