When someone says: 'I want a programming language in which I need only say what I wish done', give him a lollipop.

-Alan Perlis

CSE341
Programming Languages

Lecture 3 – October 2019

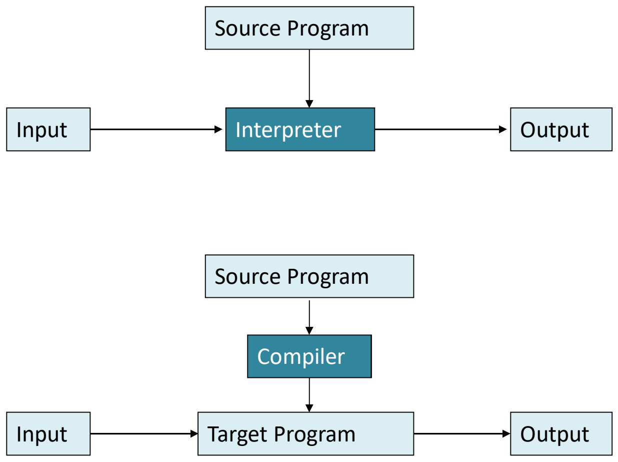
Describing Syntax and Semantics

Largely adapted from V. Shmatikov, J. Mitchell and R.W. Sebesta

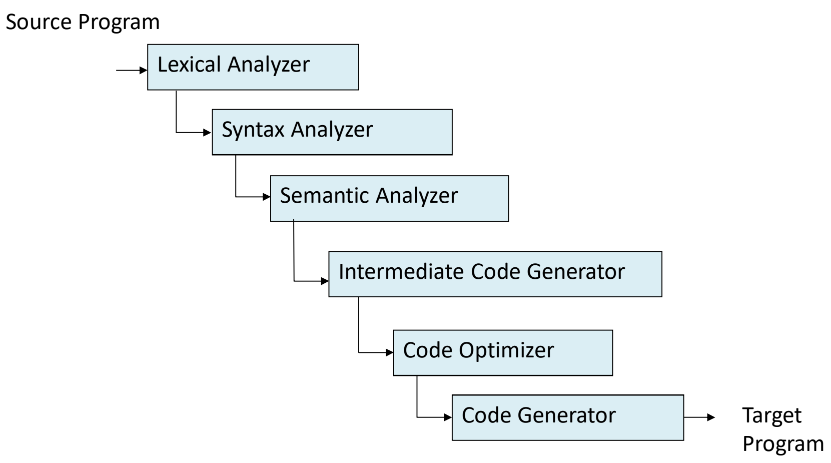
Syntax and Semantics of Programs

- Syntax
  - The symbols used to write a program
- Semantics
  - The actions that occur when a program is executed
- Programming language implementation
  - Syntax → Semantics
  - Transform program syntax into machine instructions that can be executed to cause the correct sequence of actions to occur

Interpreter vs Compiler



Typical Compiler



Syntax

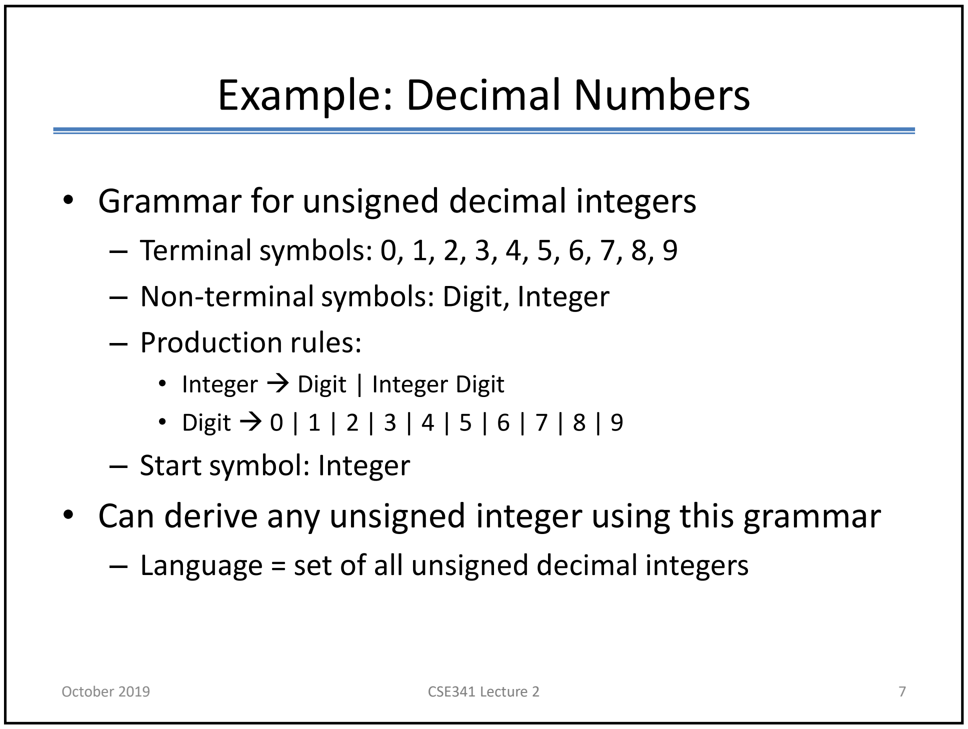
- Syntax of a programming language is a precise description of all grammatically correct programs
  - Precise formal syntax was first used in ALGOL 60
- Lexical syntax
  - Basic symbols (names, values, operators, etc.)
- Concrete syntax
  - Rules for writing expressions, statements, programs
- Abstract syntax
  - Internal representation of expressions and statements, capturing their "meaning" (i.e., semantics)

Grammars

- A **meta-language** is a language used to define other languages
- A **grammar** is a meta-language used to define the syntax of a language. It consists of:
  - Finite set of terminal symbols
  - Finite set of non-terminal symbols
  - Finite set of production rules
  - Start symbol
  - Language = (possibly infinite) set of all sequences of symbols that can be derived by applying production rules starting from the start symbol

Backus-Naur Form (BNF)

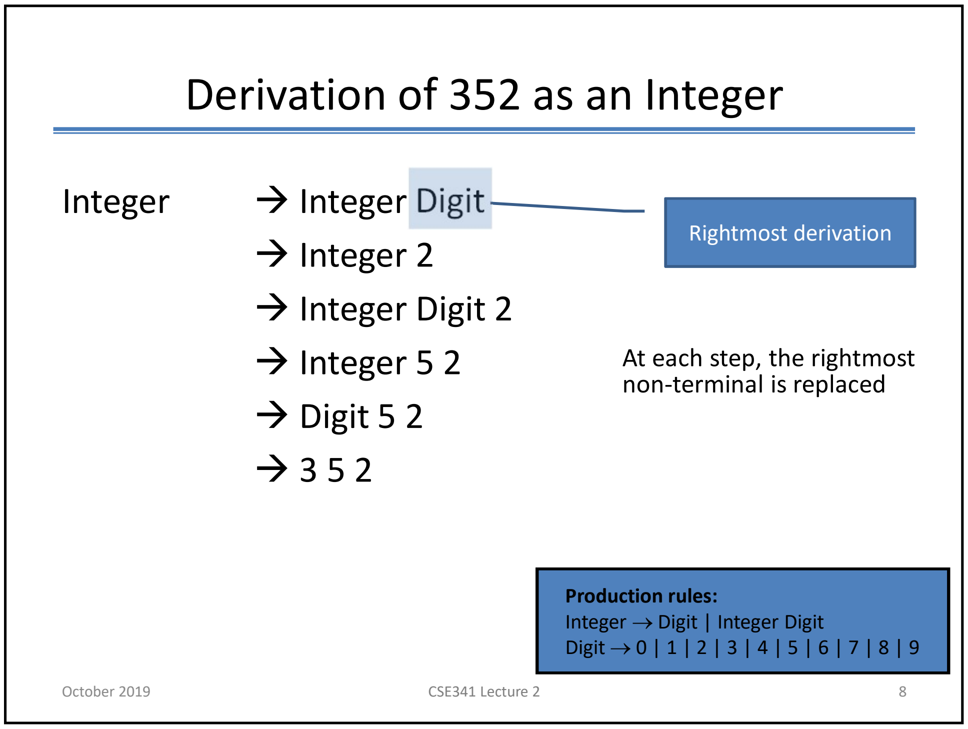
John Backus & Peter Naur Panini



Example: Decimal Numbers

- Grammar for unsigned decimal integers
  - Terminal symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Non-terminal symbols: Digit, Integer
  - Production rules:
    - Integer → Digit | Integer Digit
    - Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  - Start symbol: Integer
- Can derive any unsigned integer using this grammar
  - Language = set of all unsigned decimal integers

October 2019 CSE341 Lecture 2 7



Derivation of 352 as an Integer

```
Integer   → Integer Digit
          → Integer 2
          → Integer Digit 2
          → Integer 5 2
          → Digit 5 2
          → 3 5 2
```

Rightmost derivation

At each step, the rightmost non-terminal is replaced

Production rules:
Integer → Digit | Integer Digit
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

October 2019 CSE341 Lecture 2 8

Derivation of 352 as an Integer

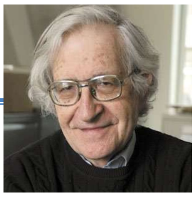
$$
\begin{aligned}
\text{Integer} &\rightarrow \text{Integer Digit}\\
&\rightarrow \text{Integer Digit Digit}\\
&\rightarrow \text{Digit Digit Digit}\\
&\rightarrow \text{3 Digit Digit}\\
&\rightarrow \text{3 5 Digit}\\
&\rightarrow \text{3 5 2}
\end{aligned}
$$

At each step, the leftmost non-terminal is replaced

Leftmost derivation

Production rules:
Integer → Digit | Integer Digit
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Chomsky Hierarchy



- Regular grammars
  - Regular expressions, finite-state automata
  - Used to define lexical structure of the language
- Context-free grammars
  - Non-deterministic pushdown automata
  - Used to define concrete syntax of the language
- Context-sensitive grammars
  - Unrestricted grammars
  - Recursively enumerable languages, Turing machines

Regular Grammars

- Left regular grammar
  - All production rules have the form
    $A \rightarrow \omega$ or $A \rightarrow B\omega$
  - Here A, B are non-terminal symbols, ω is a terminal symbol
- Right regular grammar
  - $A \rightarrow \omega$ or $A \rightarrow \omega B$
- Example: grammar of decimal integers

Lexical Analysis

- Source code = long string of ASCII characters
- Lexical analyzer splits it into **tokens**
  - Token = sequence of characters (symbolic name) representing a single terminal symbol

• Identifiers:	myVariable …
• Literals:	123 5.67 true …
• Keywords:	char sizeof …
• Operators:	+ - * / …
• Punctuation:	; , } { …

- Discards whitespace and comments

Regular Expressions

• x	character x
• \x	escaped character, e.g., \n
• { name }	reference to a name
• M | N	M or N
• M N	M followed by N
• M*	0 or more occurrences of M
• M+	1 or more occurrences of M
• [x_1 … x_n]	One of x_1 … x_n

– Example: [aeiou] – vowels, [0-9] - digits

Examples of Tokens in C

- Lexical analyzer usually represents each token by a unique integer code

```
– "+"      { return(PLUS); }       // PLUS = 401
– "-"      { return(MINUS); }      // MINUS = 402
– "*"      { return(MULT); }       // MULT = 403
– "/"      { return(DIV); }        // DIV = 404
```

- Some tokens require regular expressions

```
– [a-zA-Z_][a-zA-Z0-9_]*   { return (ID); }  // identifier
– [1-9][0-9]*              { return(DECIMALINT); }
– 0[0-7]*                  { return(OCTALINT); }
– (0x|0X)[0-9a-fA-F]+      { return(HEXINT); }
```

Reserved Keywords in C

- *auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, wchar_t, while*
- C++ added a bunch: *bool, catch, class, dynamic_cast, inline, private, protected, public, static_cast, template, this, virtual* and others
- Each keyword is mapped to its own token

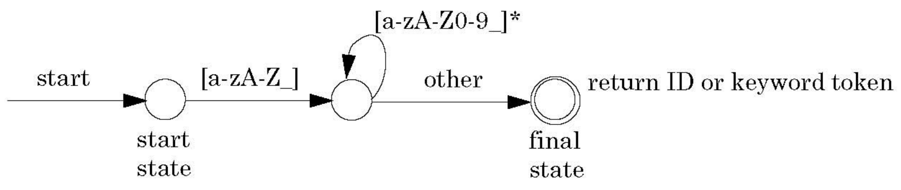
Automatic Scanner Generation

- **Lexer** or **scanner** recognizes and separates lexical tokens
  - Parser usually calls lexer when it's ready to process the next symbol (lexer remembers where it left off)
- Scanner code usually generated automatically
  - Input: lexical definition (e.g., regular expressions)
  - Output: code implementing the scanner
  - Typically, this is a **deterministic finite automaton** (DFA)
  - Examples: Lex, Flex (C and C++), JLex (Java)

Finite State Automata

- Set of states
  - Usually represented as graph nodes
- Input alphabet + unique “end of program” symbol
- State transition function
  - Usually represented as directed graph edges (arcs)
  - Automaton is **deterministic** if, for each state and each input symbol, there is at most one outgoing arc from the state labeled with the input symbol → uniqueness of computation
- Unique start state
- One or more final (accepting) states

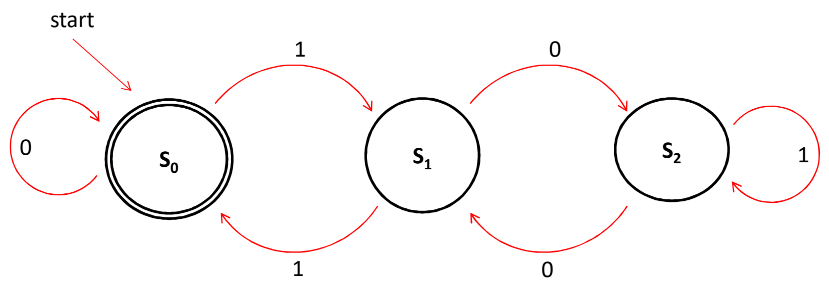
DFA for C Identifiers



Traversing a DFA

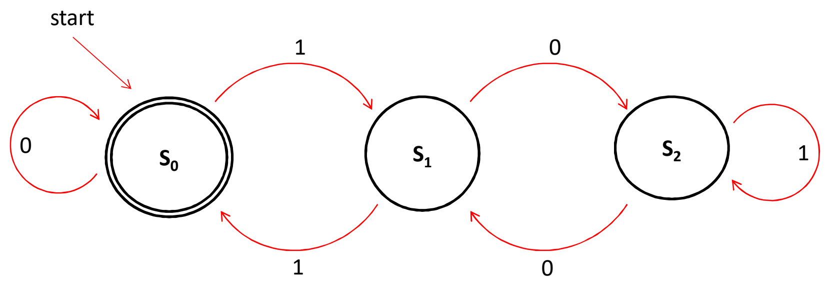
- **Configuration** = state + remaining input
- **Move** = traversing the arc exiting the state that corresponds to the leftmost input symbol, thereby consuming it
- If no such arc, then…
  - If no input and state is final, then accept
  - Otherwise, error
- Input is **accepted** if, starting with the start state, the automaton consumes all the input and halts in a final state

Traversing a DFA – Example



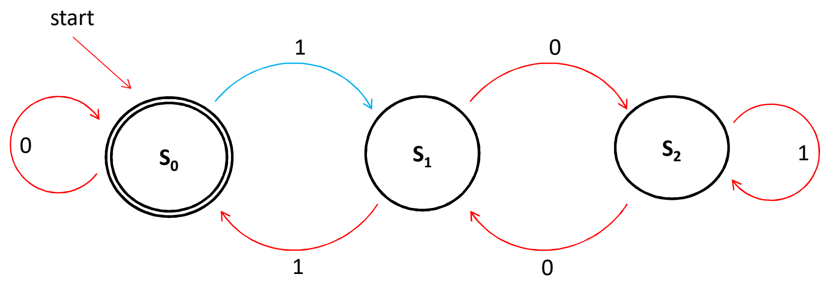
A DFA that accepts only binary numbers that are multiples of 3. The state S_0 is both the start state and an accept state.

Traversing a DFA – Example



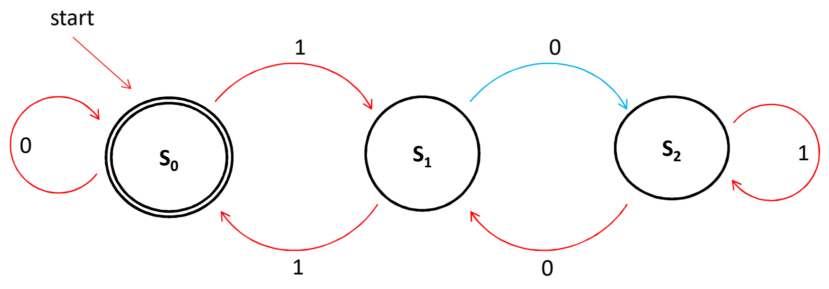
Input: 1001

Traversing a DFA – Example



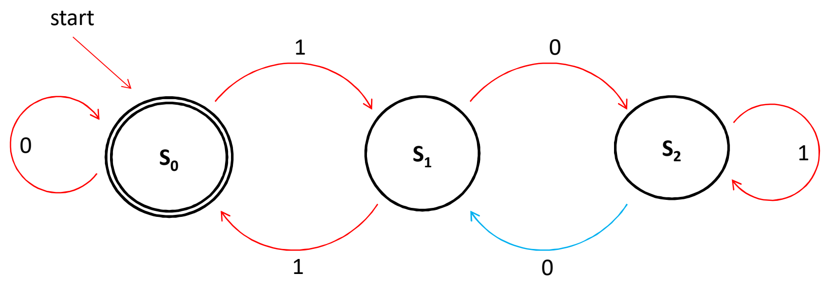
Input: 1001

Traversing a DFA – Example

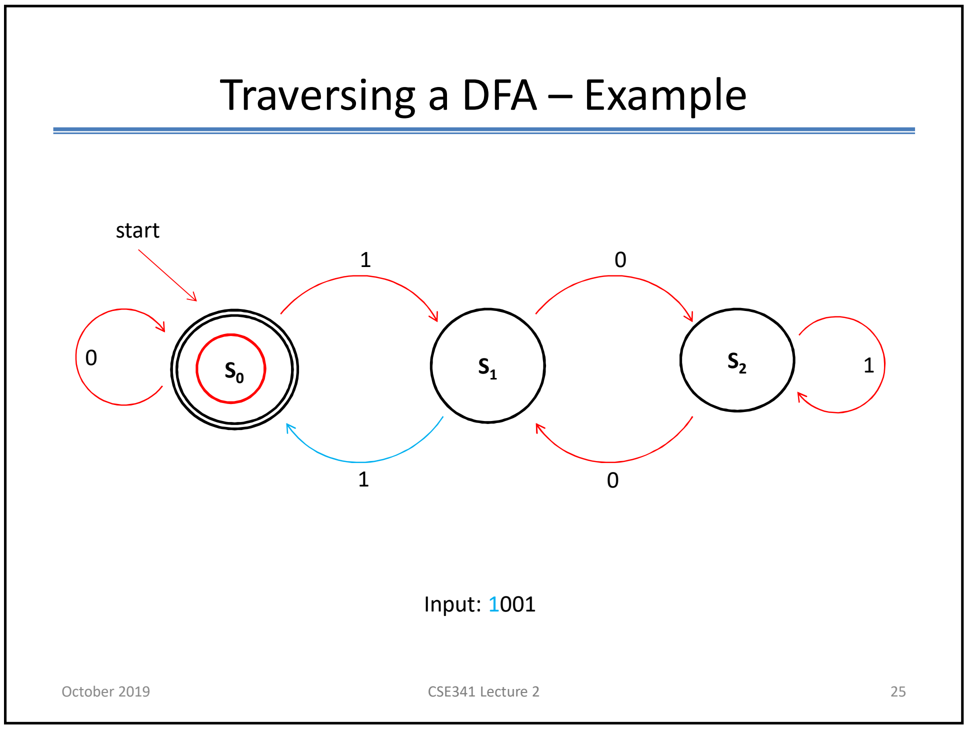


Input: 1001

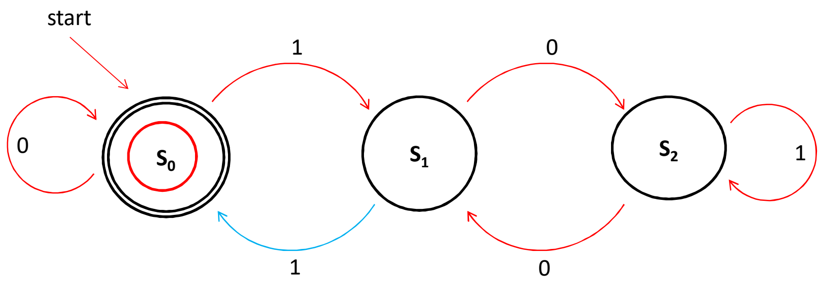
Traversing a DFA – Example



Input: 1001

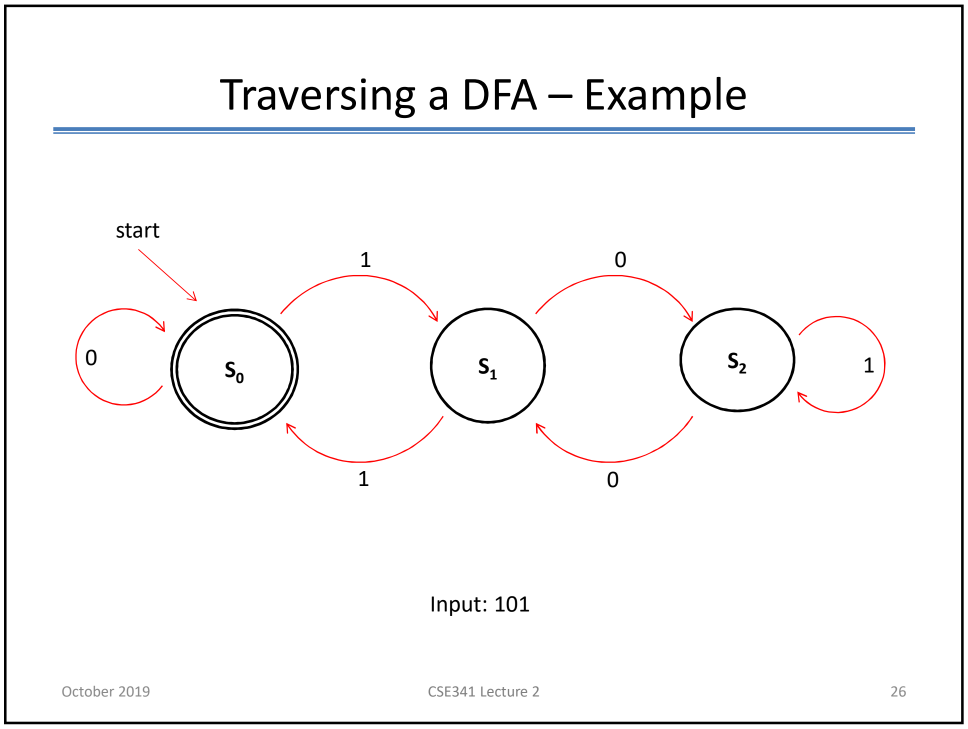


Traversing a DFA – Example

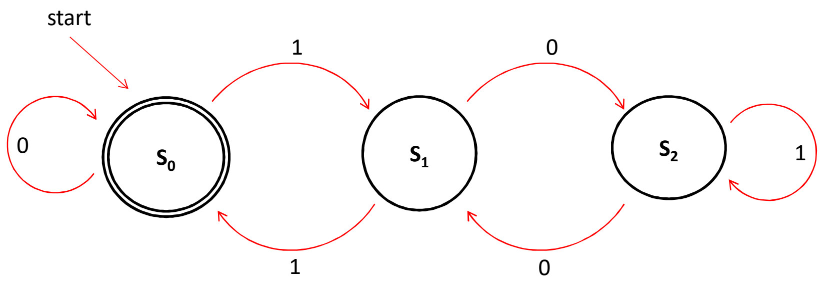


Input: 1001

October 2019 CSE341 Lecture 2 25

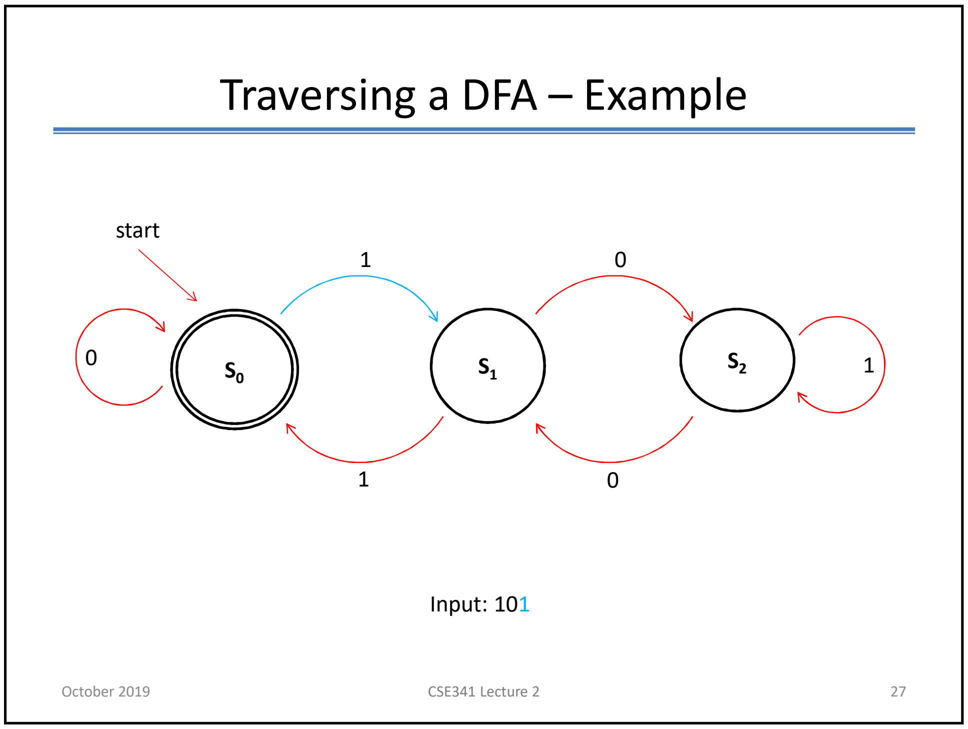


Traversing a DFA – Example

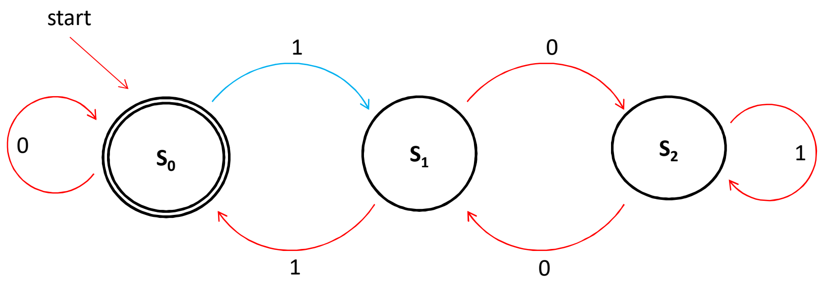


Input: 101

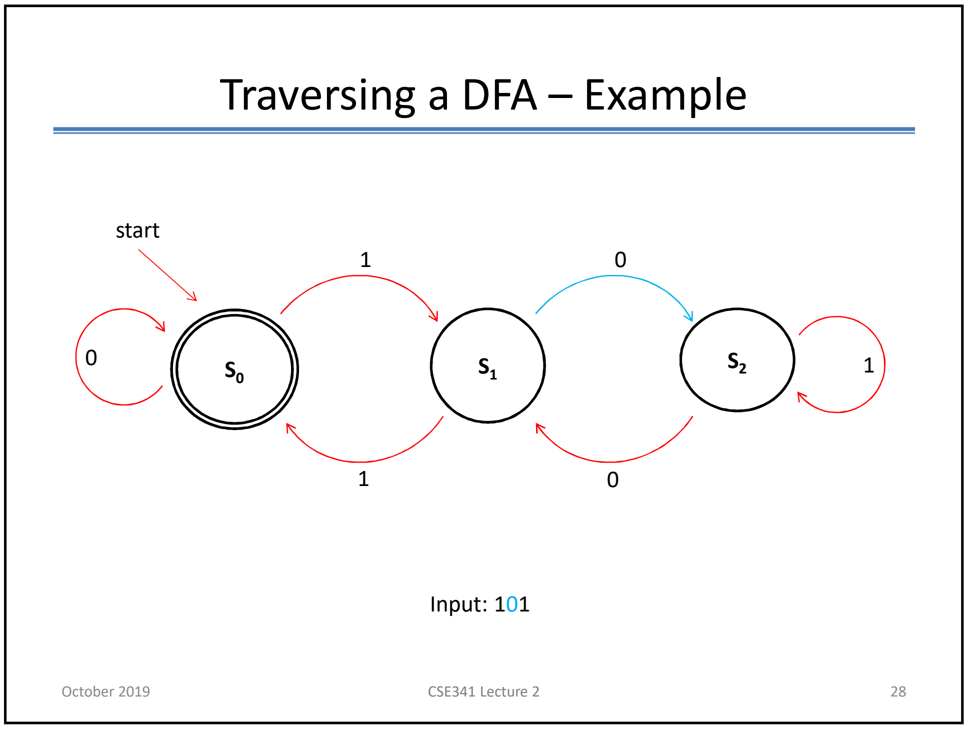
October 2019 CSE341 Lecture 2 26



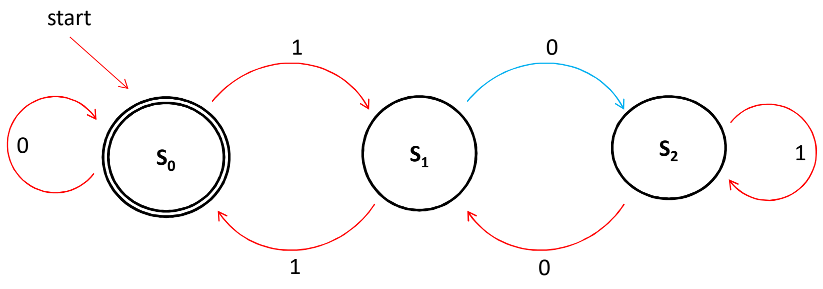
Traversing a DFA – Example



Input: 101

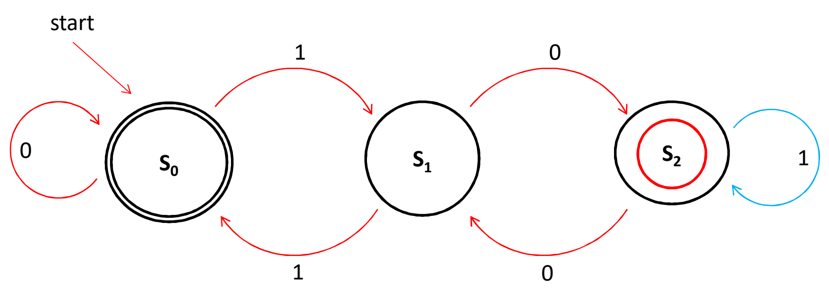


Traversing a DFA – Example



Input: 101

Traversing a DFA – Example



Input: 101

Context-Free Grammars

- Used to describe concrete syntax
  - Typically using BNF notation
- Production rules have the form $A \rightarrow \omega$
  - A is a non-terminal symbol, ω is a string of terminal and non-terminal symbols
- **Parse tree** = graphical representation of derivation
  - Each internal node = LHS of a production rule
    - Internal node must be a non-terminal symbol
  - Children nodes = RHS of this production rule
  - Each leaf node = terminal symbol (token) or "empty"

Regular Grammars

- Left regular grammar
  - All production rules have the form
    A → ω or A → Bω
  - Here A, B are non-terminal symbols, ω is a terminal symbol
- Right regular grammar
  - A → ω or A → ωB
- Example: grammar of decimal integers

Context-Free Grammars

- CFG is a set of recursive productions rules used to generate patterns of strings.
- Consists of:
  - a set of terminal symbols (the characters of the alphabet that appear in the strings generated by the grammar)
  - a set of non-terminal symbols (placeholders for patterns of terminal symbols that can be generated by the non-terminal symbols)
  - a set of productions (rules for replacing non-terminal symbols (on the left side of the production) in a string with other non-terminal or terminal symbols (on the right side of the production))
  - a start symbol (special non-terminal symbol that appears in the initial string generated by the grammar)

Context-Free Grammars

To generate a string of terminal symbols from a CFG:

- Begin with a string consisting of the start symbol;
- Apply one of the productions with the start symbol on the left hand size, replacing the start symbol with the right hand side of the production;
- Repeat the process of selecting non-terminal symbols in the string, and replacing them with the right hand side of some corresponding production, until all non-terminals have been replaced by terminal symbols.

Syntactic Correctness

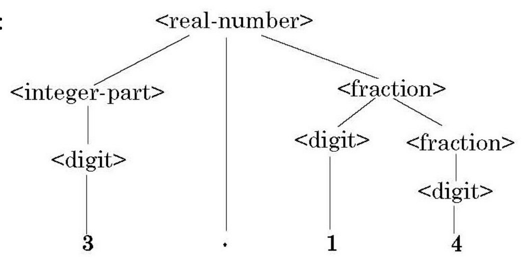
- Lexical analyzer produces a stream of tokens
- **Parser** (syntactic analyzer) verifies that this token stream is syntactically correct by constructing a valid parse tree for the entire program
  - Unique parse tree for each language construct
  - Program = collection of parse trees rooted at the top by a special start symbol
- Parser can be built automatically from the BNF description of the language's CFG
  - Example tools: yacc, Bison

CFG For Floating Point Numbers

```
<real-number> ::= <integer-part> '.' <fraction-part>
<integer-part> ::= <digit> | <integer-part> <digit>
<fraction> ::= <digit> | <digit> <fraction>
<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

::=	stands for production rule
<…>	are non-terminals
|	represents alternatives for the right-hand side of a production rule

Sample parse tree:



Grammars and Derivations

- Application of a sequence of rules…

<program> ::= begin <stmt_list> end
<stmt_list> ::= <stmt> ; <stmt_list>
<stmt> ::= <var> = <expr>
<var> ::= A | B | C
<expr> ::= <var> + <var> | <var>-<var> | <var>

Each string in the derivation including <program> is called a sentential form

<program> $\Rightarrow$ begin <stmt_list> end
$\Rightarrow$ begin <stmt> ; <stmt_list> end
$\Rightarrow$ begin <var> = <expr> ; <stmt_list> end
$\Rightarrow$ begin A = <expr> ; <stmt_list> end
$\Rightarrow$ begin A = <var>+<var> ; <stmt_list> end
$\Rightarrow$ begin A = B + <var>; <stmt_list> end
$\Rightarrow$ begin A = B + C; <stmt_list> end
$\Rightarrow$ begin A = B + C; <stmt> end
$\Rightarrow$ begin A = B + C; <var> = <expr> end

Parse Trees

- Grammars define hierarchical syntactic structure → parse trees

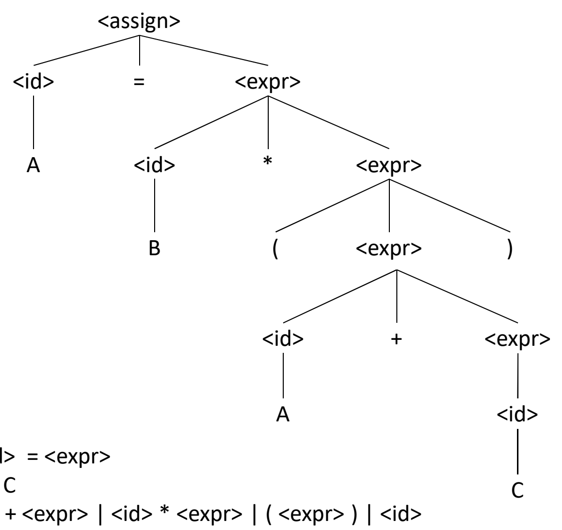
<assign> ::= <id> = <expr>
<id> ::= A | B | C
<expr> ::= <id> + <expr> | <id> * <expr> | (<expr>) | <id>

A = B * (A + C) can be generated by leftmost derivation…

$$
\begin{aligned}
\langle\text{assign}\rangle &\Rightarrow \langle\text{id}\rangle = \langle\text{expr}\rangle \\
&\Rightarrow A = \langle\text{expr}\rangle \\
&\Rightarrow A = \langle\text{id}\rangle * \langle\text{expr}\rangle \\
&\Rightarrow A = B * \langle\text{expr}\rangle \\
&\Rightarrow A = B * (\langle\text{expr}\rangle) \\
&\Rightarrow A = B * (\langle\text{id}\rangle + \langle\text{expr}\rangle) \\
&\Rightarrow A = B * (A + \langle\text{expr}\rangle) \\
&\Rightarrow A = B * (A + \langle\text{id}\rangle) \\
&\Rightarrow A = B * (A + C)
\end{aligned}
$$

Parse Trees

A = B * (A + C)



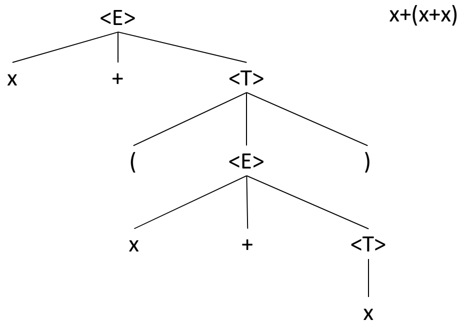
<assign> ::= <id> = <expr>
<id> ::= A | B | C
<expr> ::= <id> + <expr> | <id> * <expr> | (<expr>) | <id>

Parsers

- Parsers:
  - Determine if the input program is syntactically correct
  - Produce a parse tree for the correct input program
- Classify parsers by the order in which they build the parse tree:
  - Top-down
  - Bottom-up

Recursive Descent Parsing

- Perform a depth-first search of the derivation tree for the input being parsed – top-down
- Example:
  - E ::= x + T
  - T ::= (E)
  - T ::= x



LL Algorithms

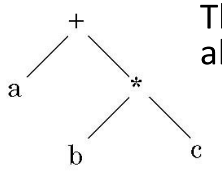
- Recursive descent – coded directly from BNF
- Parsing table – do not implement BNF rules
- Both are versions of **LL algorithms**
  - Works on same subset of all CFGs
  - 1st L: left to right scan of input
  - 2nd L: leftmost derivation is generated

LR Parsing

- Bottom-up parsing
- **LR algorithms**
  - L: left to right scan of input
  - R: rightmost derivation is generated

Concrete vs. Abstract Syntax

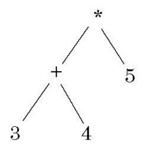
- Different languages have different concrete syntax for representing expressions, but expressions with common meaning have the same abstract syntax
  - C: a+b*c Forth: bc*a+ (reverse Polish notation)

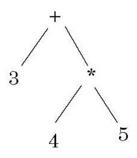


This expression tree represents the abstract "meaning" of expression

Assumes certain operator precedence (why?)
Not the same as parse tree (why?)
Does the value depend on traversal order?

Expression Notation





Inorder traversal (3+4)*5=35 3+(4*5)=23

When constructing expression trees, we want inorder traversal to produce correct arithmetic result based on operator precedence and associativity

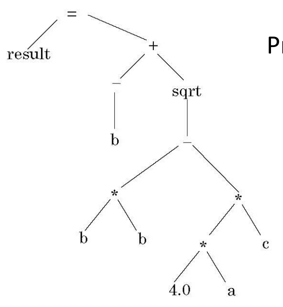
Postorder traversal 3 4 + 5 * =35 3 4 5 * + =23

Easily evaluated using operand stack (example: Forth)
Leaf node: push operand value on the stack
Non-leaf binary or unary operator: pop two (resp. one) values from stack, apply operator, push result back on the stack
End of evaluation: print top of the stack

Mixed Expression Notation

`result = -b + sqrt(b*b - 4.0 * a * c);`

unary prefix operators



Prefix:

= result + $-_1$ b sqrt $-_2$ * b b * * 4.0 a c

Need to indicate arity to distinguish between unary and binary minus

Postfix, Prefix, Mixfix in Java and C

- Increment and decrement: x++, --y

  x = ++x + x++ legal syntax, undefined semantics!
- Ternary conditional

  (conditional-expr) ? (then-expr) : (else-expr);
  - Example:

  int min(int a, int b) { return (a<b) ? a : b; }
  - This is an expression, NOT an if-then-else command
  - What is the type of this expression?

Expression Compilation Example

```
float position, initial, rate;
position = initial + rate * 60;
```

lexical analyzer

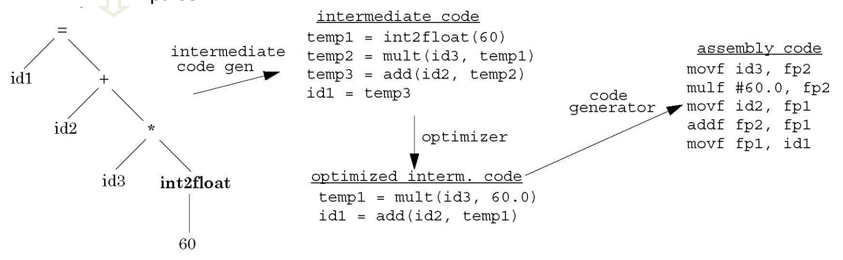
```
[ID, "position"] [ASSIGN, '='] [ID, "initial"] [PLUS, '+'] [ID, "rate"] [MULT, '*'] [NUM, 60] [SEMICOLON,';']
```

tokenized expression:

```
id1 = id2 + id3 * 60
```

implicit type conversion (why?)

parser



```
intermediate code
temp1 = int2float(60)
temp2 = mult(id3, temp1)
temp3 = add(id2, temp2)
id1 = temp3
```

```
optimized interm. code
temp1 = mult(id3, 60.0)
id1 = add(id2, temp1)
```

```
assembly code
movf id3, fp2
mulf #60.0, fp2
movf id2, fp1
addf fp2, fp1
movf fp1, id1
```

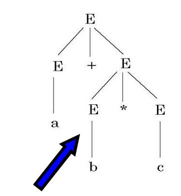
Syntactic Ambiguity

```
<expr> ::= <expr> + <expr> | <expr> * <expr> | a | b | c
```

This grammar is **ambiguous**

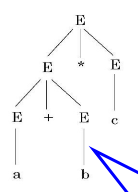
How to parse a+b*c using this grammar?

Parse Tree from a rightmost derivation starting from <expr> + <expr>



Both parse trees are syntactically valid

Parse Tree from a leftmost derivation starting with <expr> * <expr>



Only this tree is semantically correct (operator precedence and associativity are semantic, not syntactic rules)

Problem: this tree is syntactically correct, but semantically incorrect

Sentential Form

- For a grammar G, with start symbol S, any derivation S $\Rightarrow$ α is called a sentential form:
  - If contains only terminal symbols, is a sentence in L(G)
  - If contains one or more non-terminals, it is just a sentential form (not a sentence in L(G))
- A left-sentential form is a sentential form that occurs in the leftmost derivation of some sentence
- A right-sentential form is a sentential form that occurs in the rightmost derivation of some sentence
- E.g.:
  - <P> $\Rightarrow$ begin <stmt_list> end
  - <P> $\Rightarrow$ begin A = <var> + <var>; <stmt_list> end
  - <P> $\Rightarrow$ begin A = B + C; B = C end

Derivation Order

- Leftmost derivation: the replaced non-terminal is always the left-most non-terminal in the previous sentential form
  - <assign> $\Rightarrow$ <id> = <expr>
  - <assign> $\Rightarrow$ A = <expr>
  - <assign> $\Rightarrow$ A = <id> * <expr>
  - <assign> $\Rightarrow$ A = B * <expr>
  - <assign> $\Rightarrow$ A = B * <id>
  - <assign> $\Rightarrow$ A = B * C

```
<assign> := <id> = <expr>
<id> := A | B | C
<expr> := <id> + <expr> | <id> * <expr>
          | ( <expr> ) | <id>
```

- Rightmost derivation: other way around…
  - <assign> $\Rightarrow$ <id> = <expr>
  - <assign> $\Rightarrow$ <id> = <id> * <expr>
  - <assign> $\Rightarrow$ <id> = <id> * <id>
  - <assign> $\Rightarrow$ <id> = <id> * C
  - <assign> $\Rightarrow$ <id> = B * C
  - <assign> $\Rightarrow$ A = B * C
- Derivation order has no effect on the language generated by grammar

Shift-Reduce Parsing

- Stack implementation of shift-reduce parsing
- Four possible action
  - **Shift** – move the next input symbol to top of stack
  - **Reduce** – replace the handle on the top of stack by non-terminal
  - **Accept** – successful completion of parsing
  - **Error** – syntax error discovered
- Initial stack empty…
- End of input is empty…
- Note: LR parsers are more powerful than LL parsers
  - they can see the entire RHS before choosing a production

Shift-Reduce Parsing

- The parser repeatedly matches the right-hand side (RHS) of a production against a substring in the current right-sentential form
- At each match, it applies a reduction to build the parse tree:
  - each reduction replaces the matched substring with the nonterminal on the left-hand side (LHS) of the production
  - each reduction adds an internal node to the current parse tree
  - the result is another right-sentential form
- The final result is a rightmost derivation, in reverse

Handles

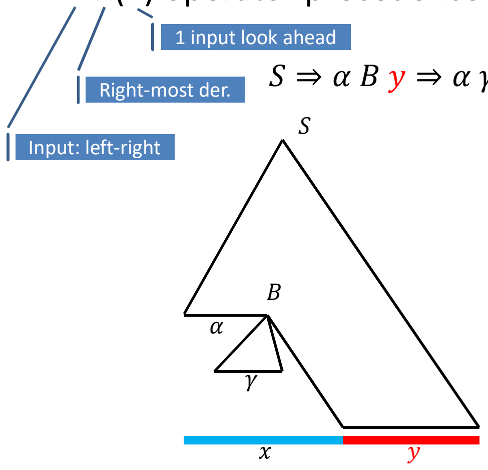
- We are trying to find a substring α of the current right-sentential form where:
  - matches some production A := α
  - reducing α to A is one step in the reverse of a rightmost derivation
- Call such a string a handle

Handle

- A handle of a right-sentential form γ is a pair <$A \Rightarrow \beta$, *k*> where $A \Rightarrow \beta$ is a production rule and *k* is the position in γ of β's rightmost symbol.
- If <$A \Rightarrow \beta$, *k*> is a handle, then replacing β in γ at position *k* with A produces the previous right-sentential form from which γ is derived in a rightmost derivation
  - $S \Rightarrow \alpha A \omega \Rightarrow \alpha \beta \omega$
- Because γ is a right-sentential form, the substring to the right of handle contains only terminal symbols ➔ the parser does not need to scan past the handle (only lookahead)
- If the grammar is **unambiguous**, then every right sentential form of the grammar has exactly one handle

LR(1) Parsing

- LR(1) operator precedence



$$S \Rightarrow \alpha\, B\, y \Rightarrow \alpha\, \gamma\, y \Rightarrow x\, y$$

Because γ is a right-sentential form, the substring to the right of a handle contains only terminal symbols
⇒ the parser doesn't need to scan past the handle (only lookahead)

Shift-Reduce Parsing

E ::= E + T | T, T ::= T * F | F, F := (E) | id

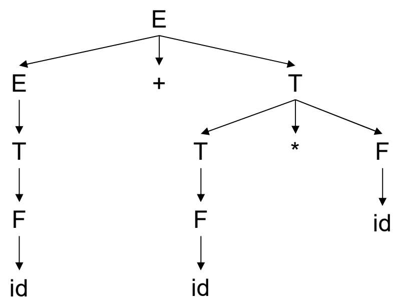
id+id*id

Step	Parse Stack	Right-most Sentential	LookAhead	Unscanned	Parser Action
0	*empty*		*id*	id+id*id	Shift
1	id	id+id*id	+	+id*id	Reduce
2	F	F+id*id	+	+id*id	Reduce
3	T	T+id*id	+	+id*id	Reduce
4	E	E+id*id	+	+id*id	Shift
5	E+	E+id*id	id	id*id	Shift
6	E+id	E+id*id	*	*id	Reduce
7	E+F	E+F*id	*	*id	Reduce
8	E+T	E+T*id Why not E*id?	*	*id	Shift
9	E+T*	E+T*id	id	id	Shift
10	E+T*id	E+T*id	empty	empty	Reduce
11	E+T*F	E+T*F	empty	empty	Reduce
12	E+T	E+T	empty	empty	Reduce
13	E	E	empty	empty	Accept

Parse Tree for this Example

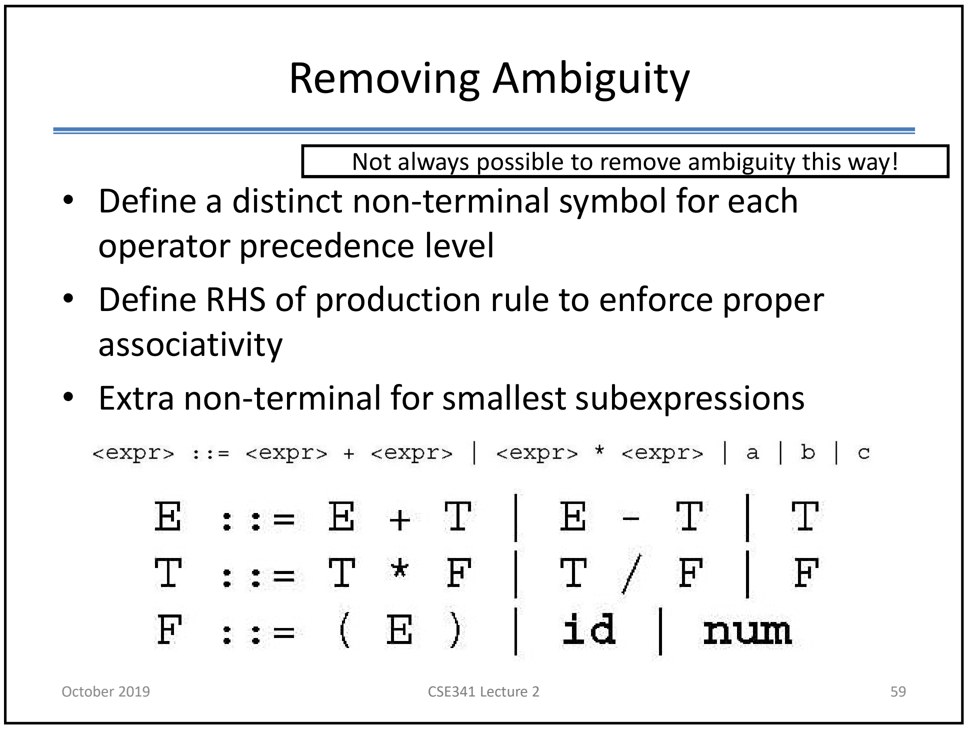
E ::= E + T | T,   T ::= T * F | F,   F := (E) | id

id+id*id



Conflicts

- Generic shift-reduce strategy:
  - If there is a handle on top of the stack, reduce
  - Otherwise, shift
- But what if there is a choice?
  - If it is legal to shift or reduce, there is a
    - shift-reduce conflict
  - If it is legal to reduce by two different
    - productions, there is a reduce-reduce conflict
- Source of conflicts:
  - Ambiguous grammars always cause conflicts
  - So do many non-ambiguous grammars



Removing Ambiguity

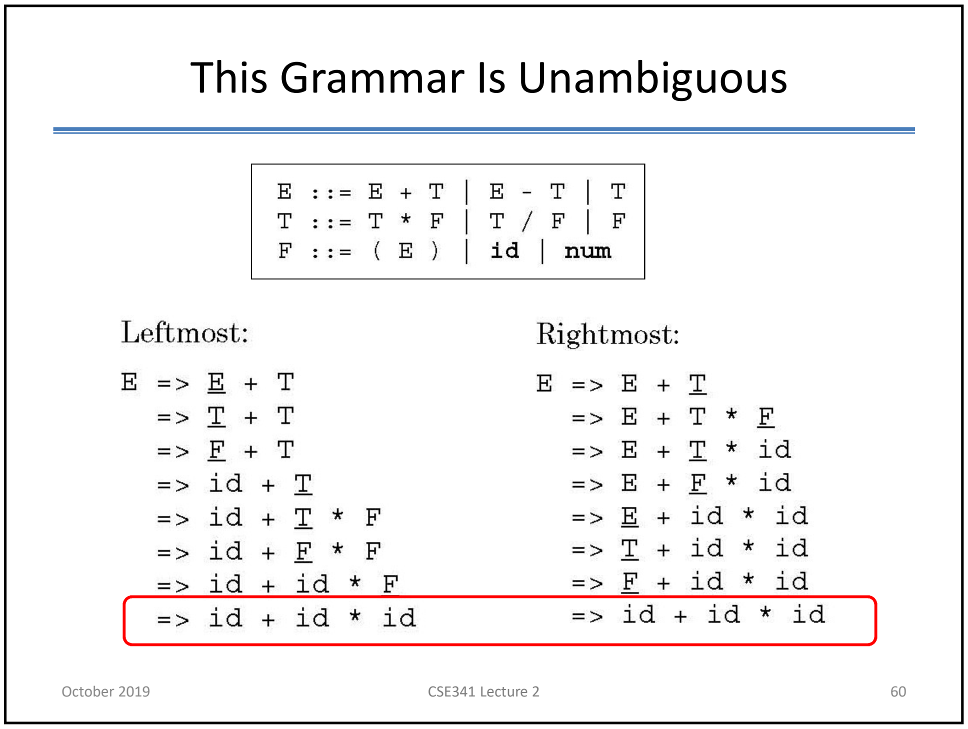
Not always possible to remove ambiguity this way!

- Define a distinct non-terminal symbol for each operator precedence level
- Define RHS of production rule to enforce proper associativity
- Extra non-terminal for smallest subexpressions

```
<expr> ::= <expr> + <expr> | <expr> * <expr> | a | b | c
```

```
E ::= E + T | E - T | T
T ::= T * F | T / F | F
F ::= ( E ) | id | num
```

October 2019 CSE341 Lecture 2 59



This Grammar Is Unambiguous

```
E ::= E + T | E - T | T
T ::= T * F | T / F | F
F ::= ( E ) | id | num
```

Leftmost:

E => $\underline{E}$ + T
=> $\underline{T}$ + T
=> $\underline{F}$ + T
=> id + $\underline{T}$
=> id + $\underline{T}$ * F
=> id + $\underline{F}$ * F
=> id + id * F
=> id + id * id

Rightmost:

E => E + $\underline{T}$
=> E + T * $\underline{F}$
=> E + $\underline{T}$ * id
=> E + $\underline{F}$ * id
=> $\underline{E}$ + id * id
=> $\underline{T}$ + id * id
=> $\underline{F}$ + id * id
=> id + id * id

October 2019 CSE341 Lecture 2 60

Left- and Right-Recursive Grammars

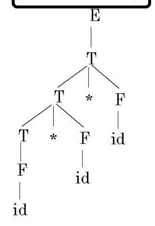
```
E ::= E + T | E - T | T
T ::= T * F | T / F | F
F ::= ( E ) | id | num
```

Leftmost non-terminal on the RHS of production is the same as the LHS

```
E ::= T + E | T - E | T
T ::= F * T | F / T | F
F ::= ( E ) | id | num
```

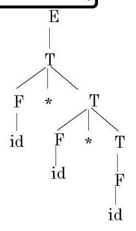
Right-recursive grammar

Left recursive parse tree for id * id * id with left-associativity



Can you think of any operators that are right-associative?

Right recursive parse tree for id * id * id with right associativity



Operator Associativity

- Needed in the absence of parentheses for operators on both sides of an operand, e.g., … ^ a ^ …
- Right associative: operators are grouped from right
  - 5 ^ 4 ^ 3 ^ 2

    $\Rightarrow$ 5^(4^(3^2))
- Left-associative: operators are grouped from left
  - 5 + 4 + 3 + 2 $\Rightarrow$ ((5 + 4) + 3) + 2
- **left-associative**: addition, subtraction, multiplication, and division
- **right-associative**: exponentiation, assignment and conditional

Right-Associative Exponentiation

<factor> ::= <exp> ** <factor> | <exp>
<exp> ::= (<expr>)| <id>

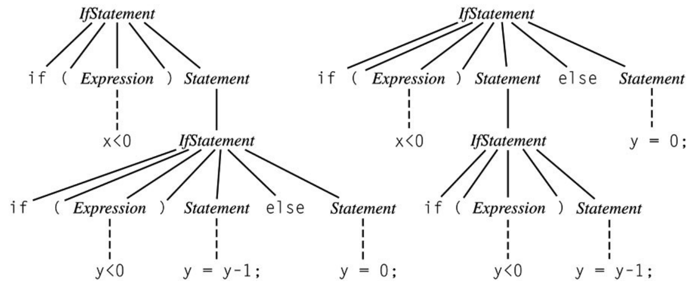
"Dangling Else" Ambiguity

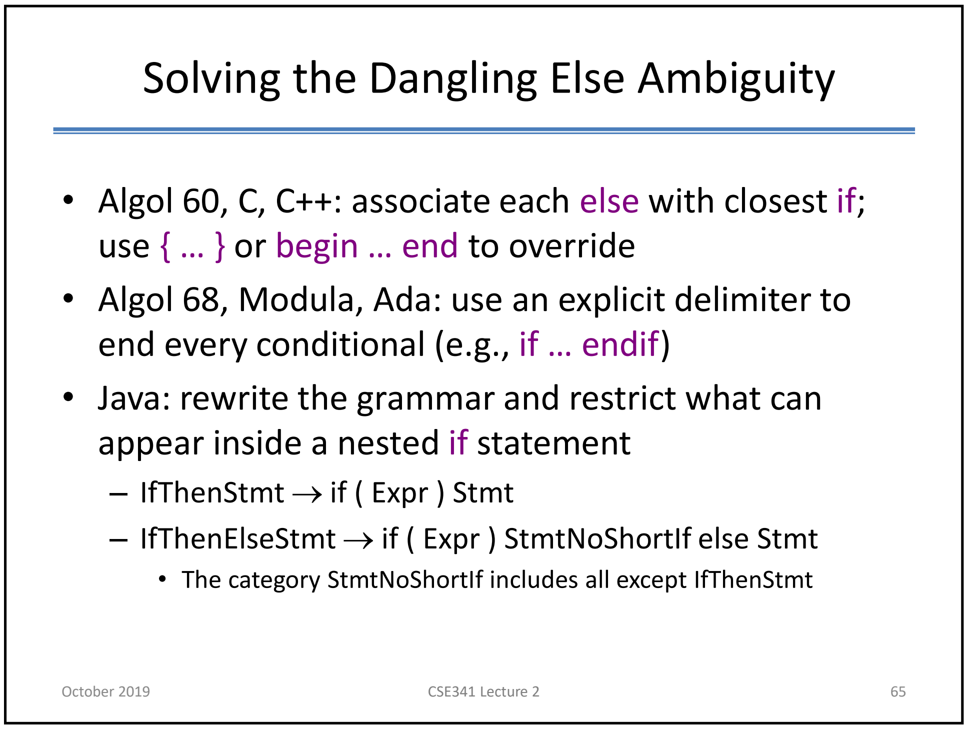
```
stmt ::= if (expr) then stmt | if (expr) then stmt else stmt
```

```
if (x < 0)
  if (y < 0)  y = y - 1;
  else y = 0;
```

With which if does this else associate?

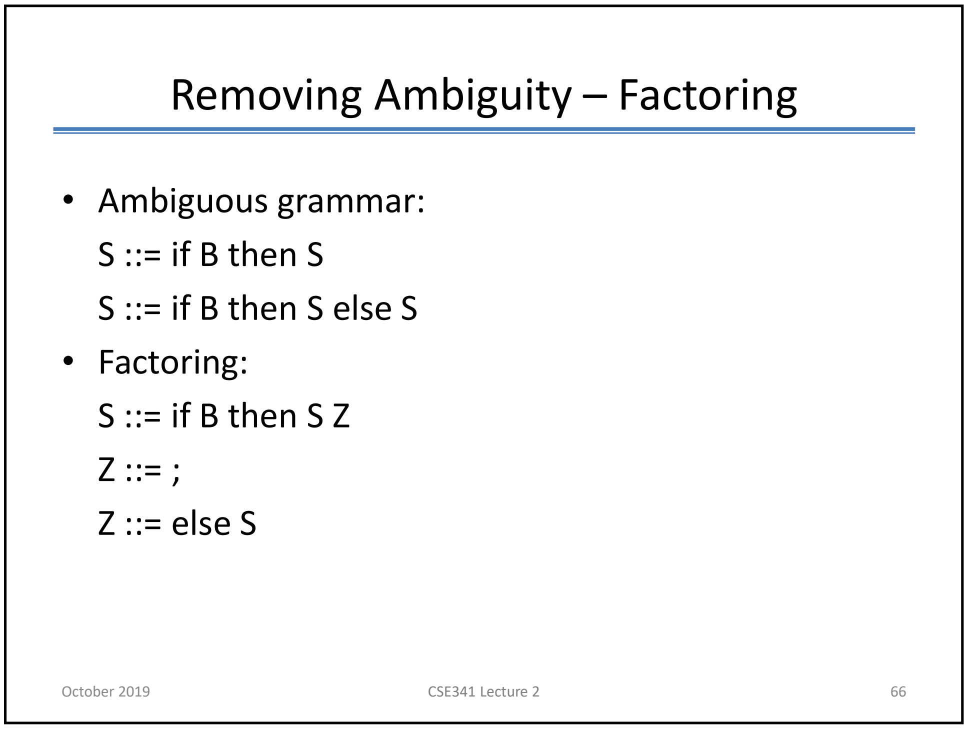
Classic example of a **shift-reduce conflict**


IfStatement
if (Expression) Statement
x<0
IfStatement
if (Expression) Statement else Statement
y<0
y = y-1;
y = 0;
IfStatement
if (Expression) Statement else Statement
x<0
IfStatement
y = 0;
if (Expression) Statement
y<0
y = y-1;



Solving the Dangling Else Ambiguity

- Algol 60, C, C++: associate each else with closest if; use { … } or begin … end to override
- Algol 68, Modula, Ada: use an explicit delimiter to end every conditional (e.g., if … endif)
- Java: rewrite the grammar and restrict what can appear inside a nested if statement
  - IfThenStmt $\rightarrow$ if (Expr) Stmt
  - IfThenElseStmt $\rightarrow$ if (Expr) StmtNoShortIf else Stmt
    - The category StmtNoShortIf includes all except IfThenStmt



Removing Ambiguity – Factoring

- Ambiguous grammar:

  S ::= if B then S

  S ::= if B then S else S
- Factoring:

  S ::= if B then S Z

  Z ::= ;

  Z ::= else S

Removing Ambiguity – Factoring

- Given

  $A ::= \alpha\ \beta_1 \mid \alpha\ \beta_2 \mid \ldots \mid \alpha\ \beta_n$
- Invent a new non-terminal F and replace

  $A ::= \alpha\ F$

  $F ::= \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Removing Ambiguity – Substitution

- Given all B productions on LHS …

  …

  $A ::= B\ \alpha$

  $B ::= \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

  …
- Replace $A ::= B\ \alpha$ with

  $A ::= \beta_1\ \alpha \mid \beta_2\ \alpha \mid \ldots \mid \beta_n\ \alpha$

EBNF

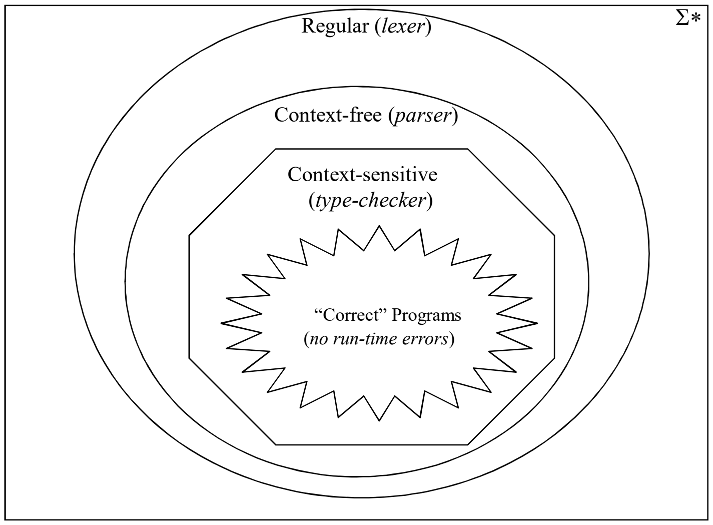
- Extending BNF to improve on minor inconveniences
- Use brackets for optional part on RHS
  - <if_stmt> := if (<expr>) <stmt> [else <stmt>]
- Use braces on RHS for indefinite repeat or none
  - <ident_list> := <identifier> { , <identifier> }
- Multiple choice option
  - <term> := <term> (* | / | %) <factor>
  - <term> := <term> * <factor> | <term> / <factor> | <term> % <factor>
- Meta-symbols: {}, [], ()

More Powerful Grammars

- Context-sensitive: production rules have the form

  $\alpha A \beta \to \alpha \omega \beta$
  - A is a non-terminal symbol, α,β,ω are strings of terminal and non-terminal symbols
  - Deciding whether a string belongs to a language generated by a context-sensitive grammar is PSPACE-complete
  - Emptiness of a language is undecidable
- Unrestricted: equivalent to Turing machine

Grammars


Σ*
Regular (*lexer*)
Context-free (*parser*)
Context-sensitive (*type-checker*)
"Correct" Programs (*no run-time errors*)

Semantic Analysis

- Beyond context free grammar
  - Is x declared before it is used?
  - Is x declared but never used?
  - Is an expression type consistent?
  - Is an array reference in bounds?
  - …
- Choices
  - Use context sensitive grammars
    - Hard to define and costly to use
  - Use attribute grammar
    - Can help to some extend
  - Use ad hoc methods
    - Mostly required

Attribute Grammars

- Augment context free grammar with rules
  - Each grammar symbol has an associated set of attributes
  - The attributes are evaluated by the semantic rules attached to the productions
- Syntax directed translation
  - Use attribute grammar for semantic analysis
    - Type checking
    - Generate intermediate code or representation
- Static Semantics
  - Not for running of a program, but for legal forms of programs

Attribute Grammar

- Example
  - Expression evaluation

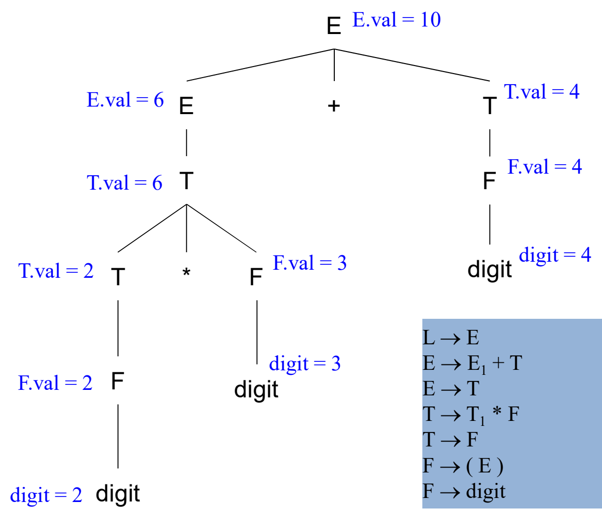
$L \rightarrow E$	print(E.val)
$E \rightarrow E_1 + T$	E.val := E_1.val + T.val
$E \rightarrow T$	E.val := T.val
$T \rightarrow T_1 * F$	T.val := T_1.val * F.val
$T \rightarrow F$	T.val := F.val
$F \rightarrow (E)$	F.val := E.val
$F \rightarrow$ digit	F.val := digit.lexval

Predicate functions: static semantic rule (production rule in BNF)

Attribute computation ~ semantic functions…

Attribute Grammar and Parse Tree

Input: 2 * 3 + 4



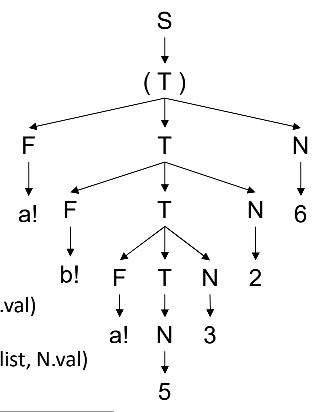
$L \rightarrow E$	print(E.val)
$E \rightarrow E_1 + T$	E.val := E_1.val + T.val
$E \rightarrow T$	E.val := T.val
$T \rightarrow T_1 * F$	T.val := T_1.val * F.val
$T \rightarrow F$	T.val := F.val
$F \rightarrow (E)$	T.val := E.val
$F \rightarrow$ digit	F.val := digit.lexval

Semantics and Attributed Grammar

- Semantics
  - Give the meaning of the language
  - BNF
    - $S \rightarrow (T)$, $T \rightarrow F\ T\ N \mid N$
    - $F \rightarrow$ a! | b!, $N \rightarrow$ num
      - Input: (a! b! a! 5 3 2 6)
  - With the attributed grammar

$S \rightarrow (T)$	print (T.list)
$T \rightarrow N$	T.list := [N.val]
$T \rightarrow F\ T_1\ N$	if (F.op = a) T.list := append (T_1.list, N.val)
	if (F.op = b) T.list := removeLastN (T_1.list, N.val)
$F \rightarrow$ a!	F.op := a
$F \rightarrow$ b!	F.op := b
$N \rightarrow$ num	N.val := num

Remove last N.val elements from T_1.list



Semantics and Attributed Grammar

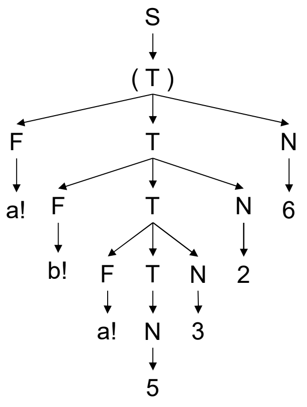
- Semantics
  - Give the meaning of the language
  - BNF

    $S \rightarrow (T)$, $T \rightarrow F\ T\ N \mid N$

    $F \rightarrow a! \mid b!$, $N \rightarrow num$

    - Input: (a! b! a! 5 3 2 6)
  - With the attributed grammar

$S \rightarrow (T)$	print (T.val)
$T \rightarrow F\ T_1\ N$	T.val := T_1.val E.op N.val
$T \rightarrow N$	T.val := N.val
$F \rightarrow a!$	F.op := +
$F \rightarrow b!$	F.op := *
$N \rightarrow num$	N.val := num



Attributed grammar defines the meaning for the production rules which gives the meaning for the input program

Build Parse Tree

- Parse tree construction
  - Can be done with only synthesized attributes:
    - child: first child pointer
    - sib: right sibling pointer
    - addr: parse tree node address

$S \rightarrow AB$	S.addr = new-node(S)
	S.child = A.addr; A.sib = B.addr;
$A \rightarrow CdE$	A.addr = new-node(A); d.addr = new-node(d);
	A.child = C.addr; C.sib = d.addr; d.sib = E.addr;
$A \rightarrow F$	A.addr = new-node(A)
	A.child = F.addr;
$B \rightarrow EF$	B.addr = new-node(B)
	B.child = E.addr; E.sib = F.addr
……	……

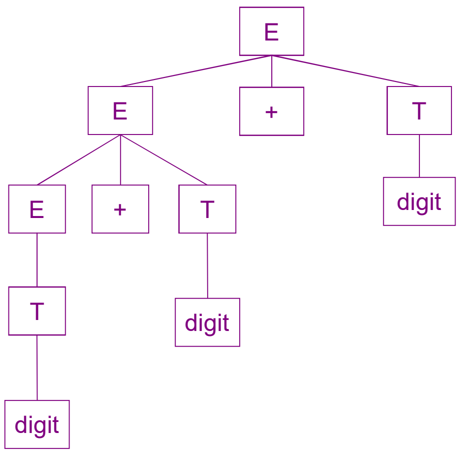
Build Parse Tree

- Parse tree construction example

$L \rightarrow E$	L.addr = new-node(L);
	L.child = E.addr;
$E \rightarrow E_1 + T$	E.addr := new-node(E);
	add.addr = new-node(+);
	E.child = E_1.addr;
	E_1.sib = add.addr;
	add.sib = T.addr;
$E \rightarrow T$	L.addr = new-node(L);
	E.child := T.addr;
$T \rightarrow$ digit	T.addr = new-node(T);
	T.child = new-node(digit);

Build Parse Tree

Input: 1 + 2 + 3

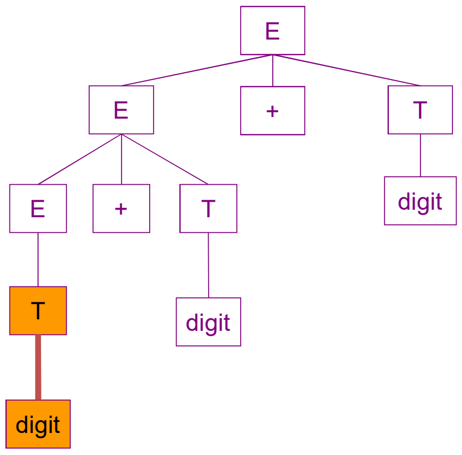


$L \rightarrow E$	L.addr = new-node(L);
	L.child = E.addr;
$E \rightarrow E_1 + T$	E.addr := new-node(E);
	add.addr = new-node(+);
	E.child = E_1.addr;
	E_1.sib = add.addr;
	add.sib = T.addr;
$E \rightarrow T$	L.addr = new-node(L);
	E.child := T.addr;
$T \rightarrow$ digit	T.addr = new-node(T);
	T.child = new-node(digit);

Build Parse Tree

Input: 1 + 2 + 3

T → digit (1)

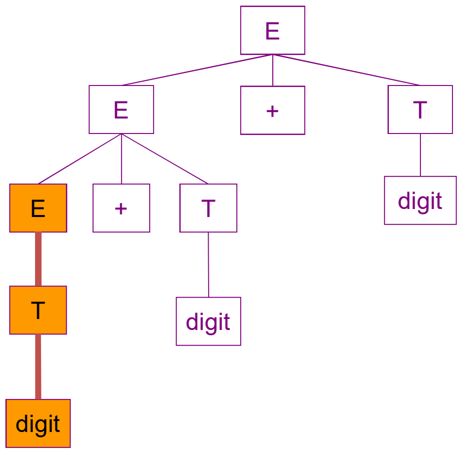


L → E	L.addr = new-node(L);
	L.child = E.addr;
E → E_1 + T	E.addr := new-node(E);
	add.addr = new-node(+);
	E.child = E_1.addr;
	E_1.sib = add.addr;
	add.sib = T.addr;
E → T	L.addr = new-node(L);
	E.child := T.addr;
T → digit	T.addr = new-node(T);
	T.child = new-node(digit);

Build Parse Tree

Input: 1 + 2 + 3

T → digit (1)
E → T

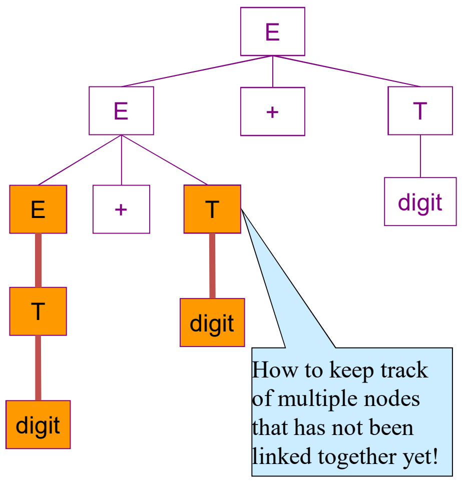


L → E	L.addr = new-node(L);
	L.child = E.addr;
E → E_1 + T	E.addr := new-node(E);
	add.addr = new-node(+);
	E.child = E_1.addr;
	E_1.sib = add.addr;
	add.sib = T.addr;
E → T	L.addr = new-node(L);
	E.child := T.addr;
T → digit	T.addr = new-node(T);
	T.child = new-node(digit);

Build Parse Tree

Input: 1 + 2 + 3

T → digit (1)
E → T
Shift +
T → digit (2)

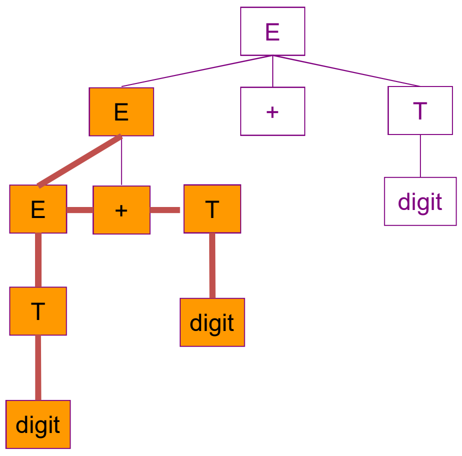

E
E
+
T
E
+
T
digit
T
digit
digit
How to keep track of multiple nodes that has not been linked together yet!

L → E	L.addr = new-node(L); L.child = E.addr;
$E \to E_1 + T$	E.addr := new-node(E); add.addr = new-node(+); $E.child = E_1.addr$; $E_1.sib = add.addr$; add.sib = T.addr;
E → T	L.addr = new-node(L); E.child := T.addr;
T → digit	T.addr = new-node(T); T.child = new-node(digit);

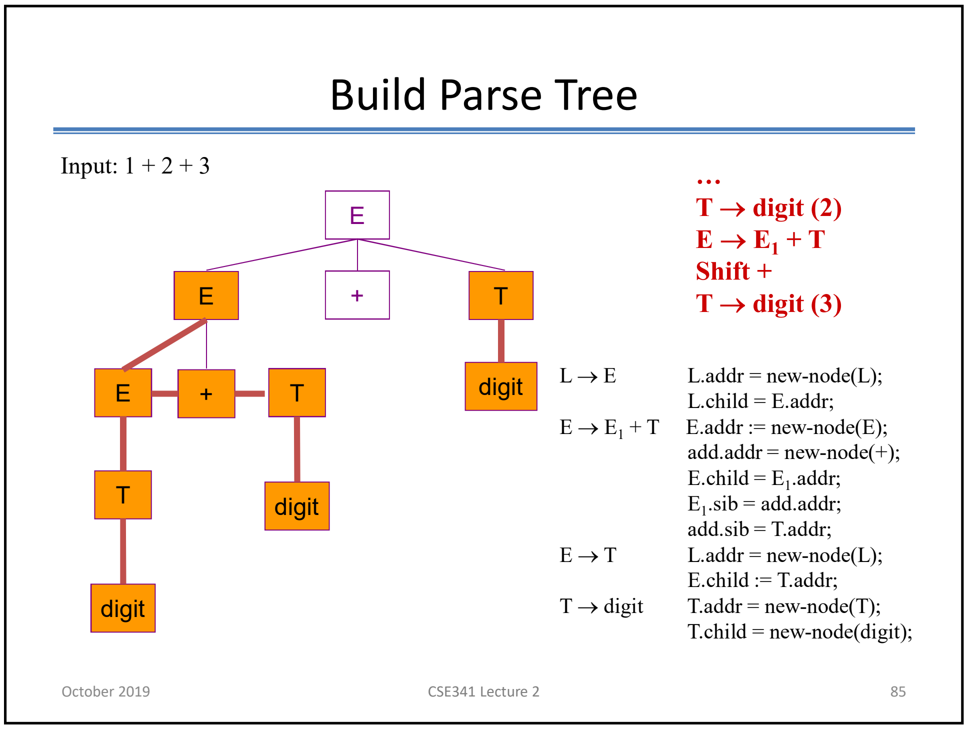
Build Parse Tree

Input: 1 + 2 + 3

T → digit (1)
E → T
Shift +
T → digit (2)
$E \to E_1 + T$

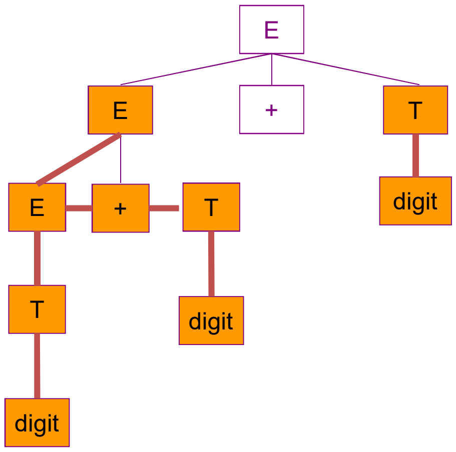


L → E	L.addr = new-node(L); L.child = E.addr;
$E \to E_1 + T$	E.addr := new-node(E); add.addr = new-node(+); $E.child = E_1.addr$; $E_1.sib = add.addr$; add.sib = T.addr;
E → T	L.addr = new-node(L); E.child := T.addr;
T → digit	T.addr = new-node(T); T.child = new-node(digit);



Build Parse Tree

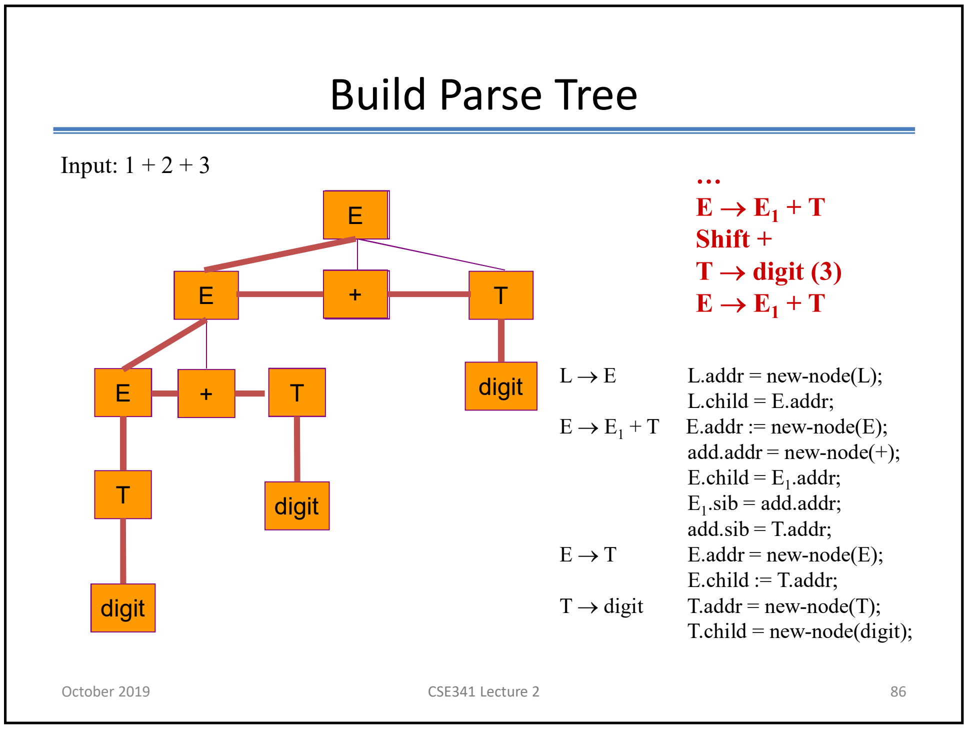
Input: 1 + 2 + 3



…
T → digit (2)
E → E_1 + T
Shift +
T → digit (3)

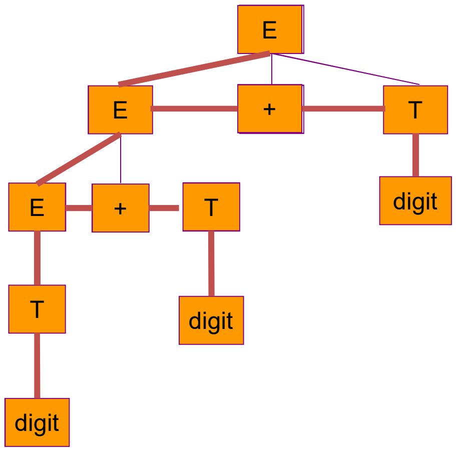
L → E	L.addr = new-node(L); L.child = E.addr;
E → E_1 + T	E.addr := new-node(E); add.addr = new-node(+); E.child = E_1.addr; E_1.sib = add.addr; add.sib = T.addr;
E → T	L.addr = new-node(L); E.child := T.addr;
T → digit	T.addr = new-node(T); T.child = new-node(digit);

October 2019 CSE341 Lecture 2 85



Build Parse Tree

Input: 1 + 2 + 3



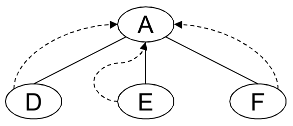
…
E → E_1 + T
Shift +
T → digit (3)
E → E_1 + T

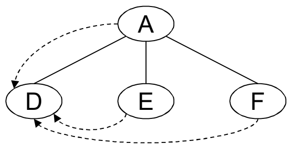
L → E	L.addr = new-node(L); L.child = E.addr;
E → E_1 + T	E.addr := new-node(E); add.addr = new-node(+); E.child = E_1.addr; E_1.sib = add.addr; add.sib = T.addr;
E → T	E.addr = new-node(E); E.child := T.addr;
T → digit	T.addr = new-node(T); T.child = new-node(digit);

October 2019 CSE341 Lecture 2 86

Attribute Grammar

- Two types of attributes
  - Synthesized attributes
    - Attribute values are evaluated from bottom up
    - The value of the parent is defined on the values of the children
  - Inherited attributes
    - Attribute values are evaluated from top down
    - The value of a node is defined on the values of its parent and/or siblings
- Attribute rules
  - Only reference local information
    - only refer to symbols in the corresponding production





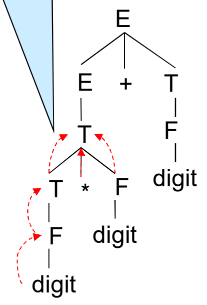
S-Attribute Grammar

- Only has synthesized attributes
- Suitable for shift-reduce parsing
- Example
  - Expression evaluation

$L \rightarrow E$	print(E.val)
$E \rightarrow E_1 + T$	E.val := E_1.val + T.val
$E \rightarrow T$	E.val := T.val
$T \rightarrow T_1 * F$	T.val := T_1.val * F.val
$T \rightarrow F$	T.val := F.val
$F \rightarrow (E)$	F.val := E.val
$F \rightarrow$ digit	F.val := digit.lexval

Derive parent's value from children's values

lexval: value from lex



Inherited Attributes

- Example
  - Adding type to symbol table

L's attribute value "type" depends on left sibling T's "type"

$D \rightarrow TL$	L.type := T.type
$T \rightarrow$ int	T.type := integer
$T \rightarrow$ real	T.type := real
$L \rightarrow L_1$, id	L_1.type := L.type; addtype (id.entry, L.type)
$L \rightarrow$ id	addtype (id.entry, L.type)

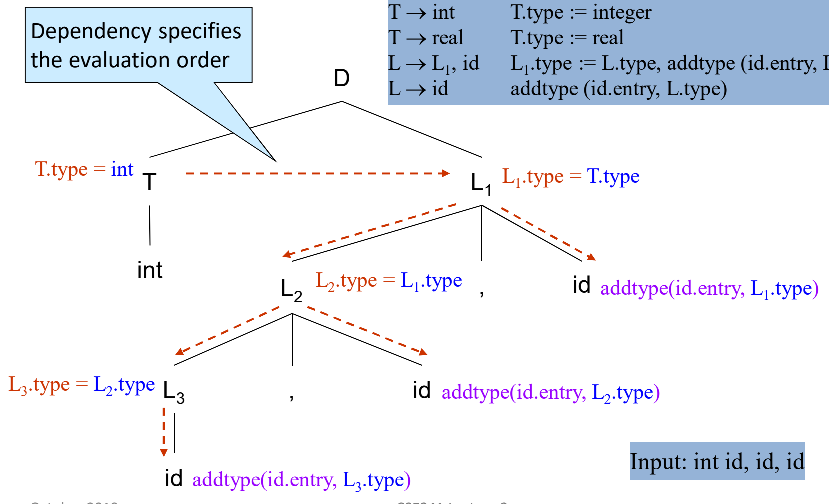
Attribute "type" is still synthesized

- Id.entry is not defined here
- It is the entry to the symbol table

L_1's attribute value "type" depends on its parent "type" value
$\Rightarrow$"type" is an inherited attribute

Dependency Graph

$D \rightarrow TL$	L.type := T.type
$T \rightarrow$ int	T.type := integer
$T \rightarrow$ real	T.type := real
$L \rightarrow L_1$, id	L_1.type := L.type, addtype (id.entry, L,type)
$L \rightarrow$ id	addtype (id.entry, L.type)


Dependency specifies the evaluation order
D
T.type = int
T
int
L_1
L_1.type = T.type
L_2
L_2.type = L_1.type
,
id
addtype(id.entry, L_1.type)
L_3.type = L_2.type
L_3
,
id
addtype(id.entry, L_2.type)
id
addtype(id.entry, L_3.type)

Input: int id, id, id

Dependency Graph

- Evaluation based on the dependencies
- Circularity check
  - Dependency graph should be acyclic
- Build pass tree
  - Fist build the parse tree, then evaluate
  - Cannot be done with parsing, require a separate evaluation pass
- Can we do better?
  - L-attributed grammar
    - Parser stack based technique
    - Marker nonterminals (not covered)
  - Rewrite grammar rules

Issues in Attributed Grammar

- Attribute rules are confined to local production info
  - Excessive copying due to production rules in the grammar

$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$T.val := E.val$
$F \rightarrow digit$	$F.val := id.digit$

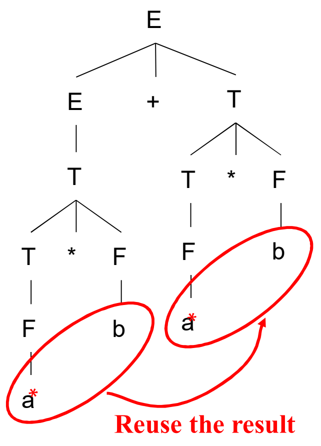
- $E \rightarrow T$, $T \rightarrow F$ causes additional copying of val attribute

Can something be done to save the copying effort?
$\Rightarrow$ Use global table

Issues in Attributed Grammar

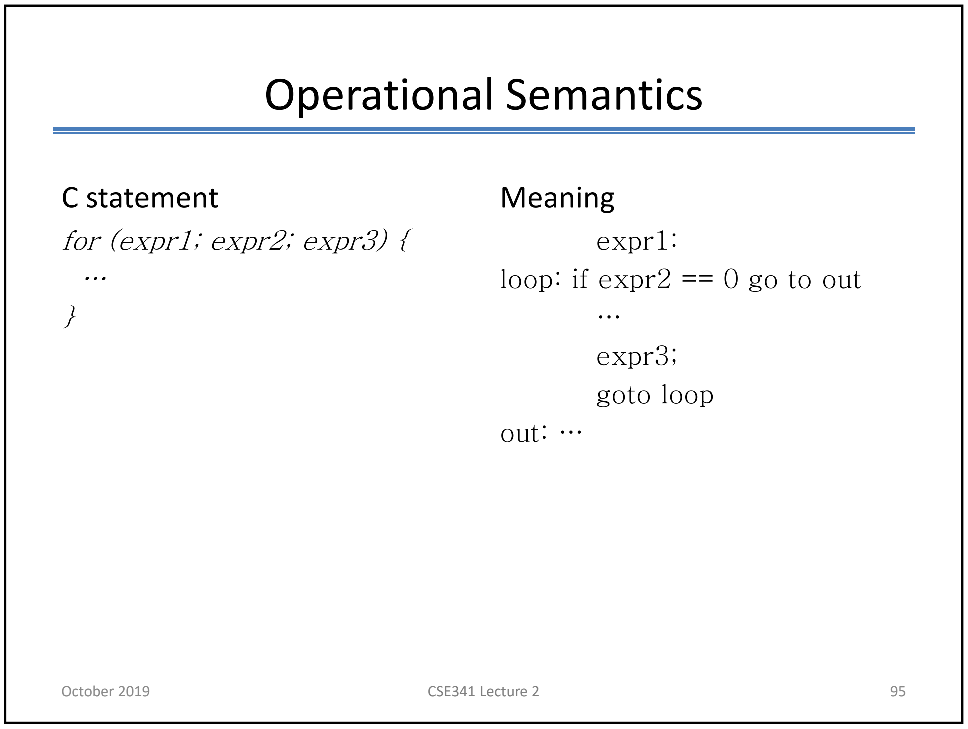
- Attribute rules are confined to local production info
  - Not able to optimize
  - Need to build the parse tree

⇒ Need to have a separate optimization phase
⇒ Requires tree traversal



Dynamic Semantics

- Describing meaning of expressions, statements, and program units of a PL
- **Operational semantics**: describing meaning of a statement or program → specifying the effects of running it on a machine…
  - Natural OS: highest level – what's the final result
  - Structural OS: lowest level – precise meaning of a program by examining the complete sequence of state changes during programs' run
- Do we need another intermediate language to define operational semantics…



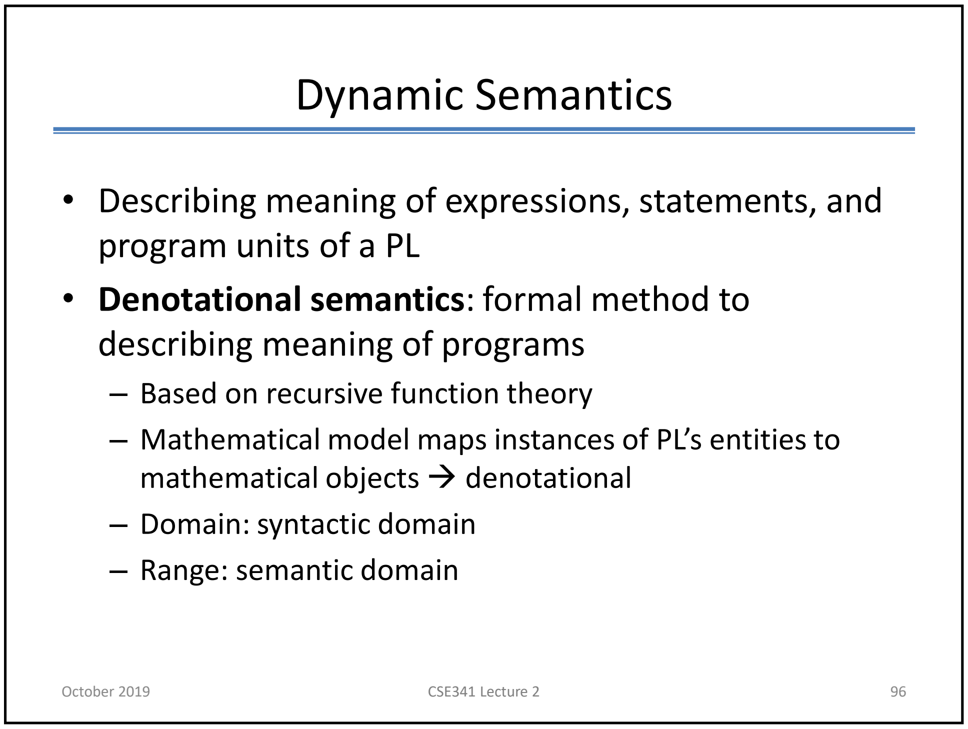
Operational Semantics

C statement

```
for (expr1; expr2; expr3) {
  …
}
```

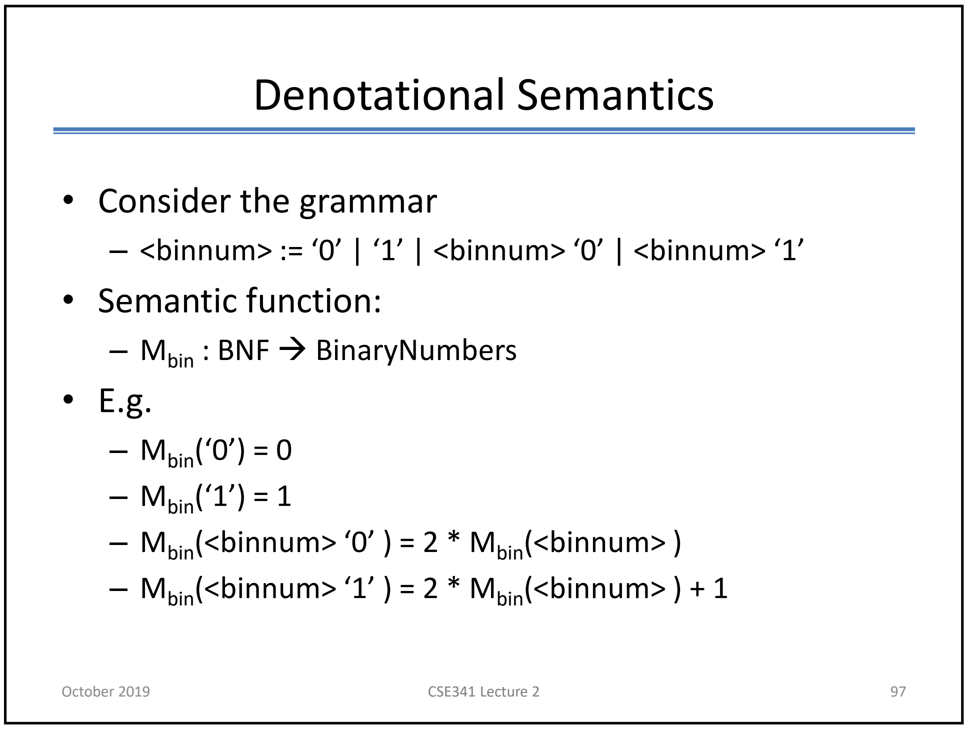
Meaning

```
           expr1;
loop: if expr2 == 0 go to out
           …
           expr3;
           goto loop
out: …
```



Dynamic Semantics

- Describing meaning of expressions, statements, and program units of a PL
- **Denotational semantics**: formal method to describing meaning of programs
  - Based on recursive function theory
  - Mathematical model maps instances of PL's entities to mathematical objects → denotational
  - Domain: syntactic domain
  - Range: semantic domain



Denotational Semantics

- Consider the grammar
  - <binnum> := ‘0’ | ‘1’ | <binnum> ‘0’ | <binnum> ‘1’
- Semantic function:
  - M_{bin} : BNF $\rightarrow$ BinaryNumbers
- E.g.
  - M_{bin}(‘0’) = 0
  - M_{bin}(‘1’) = 1
  - M_{bin}(<binnum> ‘0’) = 2 * M_{bin}(<binnum>)
  - M_{bin}(<binnum> ‘1’) = 2 * M_{bin}(<binnum>) + 1