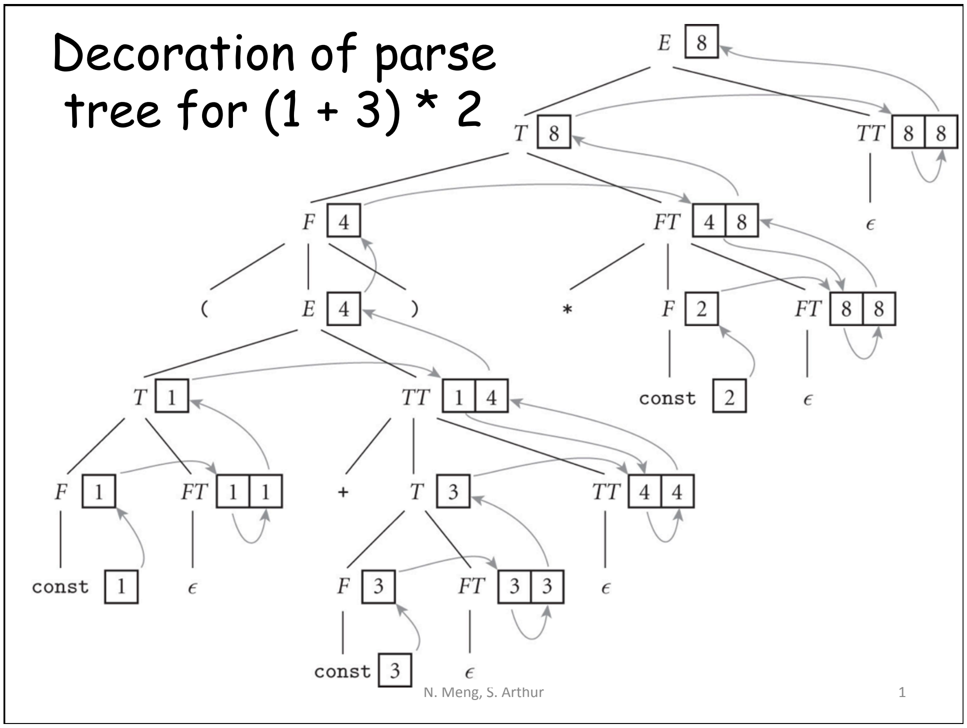
## Decoration of parse tree for (1 + 3) * 2

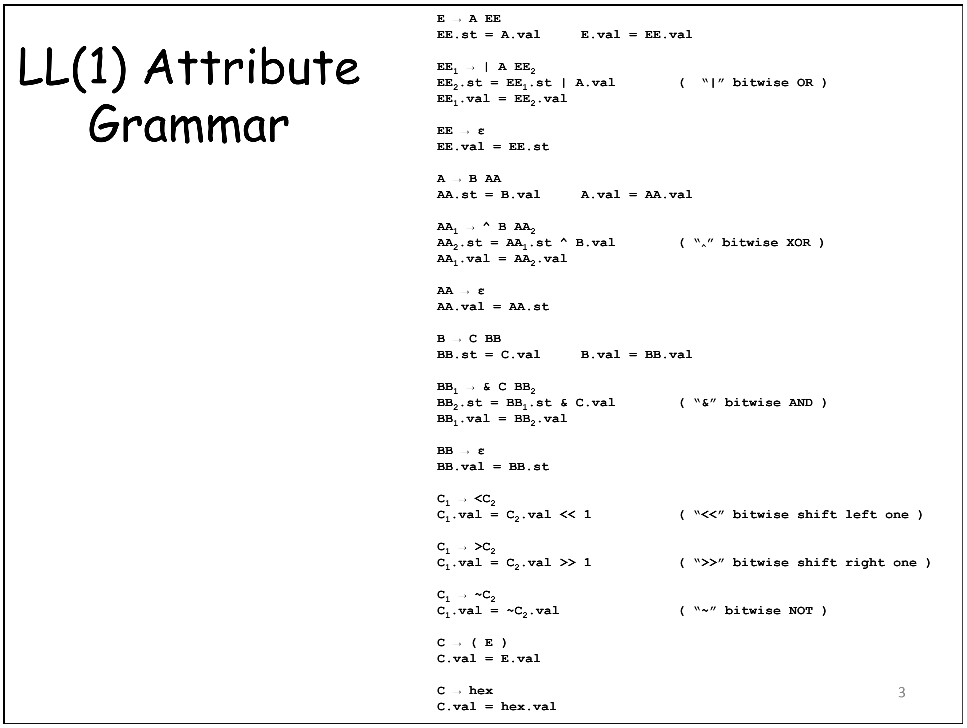


---

## Program Assignment 2
## Due date: 10/20 12:30pm

- Bitwise Manipulation of Hexidecimal Numbers
- CFG

```
E → E “|” A    bitwise OR
E → A
A → A “^” B    bitwise XOR
A → B
B → B “&” C    bitwise AND
B → C
C → “<” C   bitwise shift left 1
C → “>” C   bitwise shift right 1
C → “~” C   bitwise NOT
C → “(”  E  “)”
C → hex
```



## LL(1) Attribute Grammar

$E \rightarrow A\ EE$
$EE.st = A.val \quad E.val = EE.val$

$EE_1 \rightarrow |\ A\ EE_2$
$EE_2.st = EE_1.st\ |\ A.val$ ( "|" bitwise OR )
$EE_1.val = EE_2.val$

$EE \rightarrow \varepsilon$
$EE.val = EE.st$

$A \rightarrow B\ AA$
$AA.st = B.val \quad A.val = AA.val$

$AA_1 \rightarrow \hat{}\ B\ AA_2$
$AA_2.st = AA_1.st \mathbin{\hat{}} B.val$ ( "^" bitwise XOR )
$AA_1.val = AA_2.val$

$AA \rightarrow \varepsilon$
$AA.val = AA.st$

$B \rightarrow C\ BB$
$BB.st = C.val \quad B.val = BB.val$

$BB_1 \rightarrow \&\ C\ BB_2$
$BB_2.st = BB_1.st\ \&\ C.val$ ( "&" bitwise AND )
$BB_1.val = BB_2.val$

$BB \rightarrow \varepsilon$
$BB.val = BB.st$

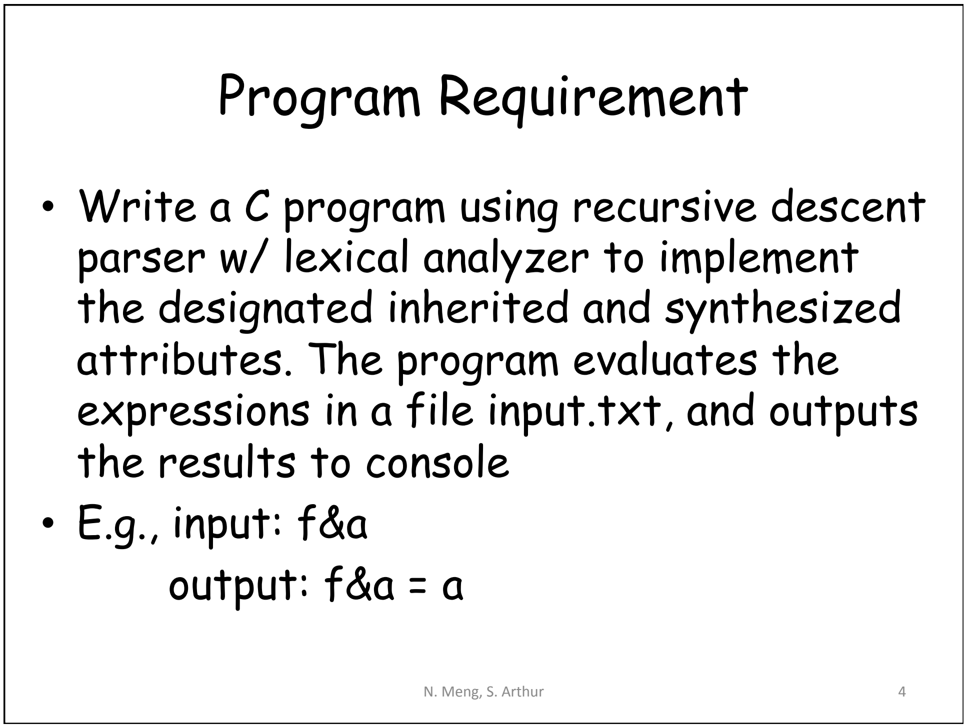
$C_1 \rightarrow \lt C_2$
$C_1.val = C_2.val \ll 1$ ( "<<" bitwise shift left one )

$C_1 \rightarrow \gt C_2$
$C_1.val = C_2.val \gg 1$ ( ">>" bitwise shift right one )

$C_1 \rightarrow \sim C_2$
$C_1.val = \sim C_2.val$ ( "~" bitwise NOT )

$C \rightarrow (\ E\ )$
$C.val = E.val$

$C \rightarrow hex$
$C.val = hex.val$



# Program Requirement

- Write a C program using recursive descent parser w/ lexical analyzer to implement the designated inherited and synthesized attributes. The program evaluates the expressions in a file input.txt, and outputs the results to console
- E.g., input: f&a
  output: f&a = a

# Program Requirements

- You cannot use more than 2 global/non-local variables, and they should be to hold the Operator and HexNumber as detected by the lexical analyzer

# Hints

- To solve the problems, you should take the following steps:
  - Write a lexical analyzer
  - Write a recursive-descent parser
  - Attributes are processed as either pass-in parameters or return value of functions

# Hints

- Write a lexical analyzer
  - You may need to define an enum type for all possible tokens your scanner can generate
  - E.g., when reading hexadecimial numbers 0-9 or a-f, the recognized token is HEX, and the value is saved in HexNumber

# Hints

- Write a recursive-descent parser
  - Parse the program by defining and invoking functions
  - E.g., **E → A EE**

```
     EE.st = A.val E.val = EE.val
int E() {
   int val = A();
   return EE(val);
}
```

# Hints

- There are parameters passed in or returned when invoking functions. When invoking a function, the synthesized attribute is the return value, while the inherited attribute is the passing-in parameter

# Hints

- Sample code of main()

```
int main() {
    int val;
    symbol = getNextToken();
    while (symbol != EOF_) {
        if (symbol != NEW_LINE) {
            val = E();
             printf(" = %x\n", val & 0xf);
        }
        if (symbol == EOF_) break;
        symbol = getNextToken();
    }
    return 1;
}
```

# Submission Requirements

- Pack the following files into a .tar file:
  - Source file: parser.c
  - Executable file: parser
  - Input file: input.txt
  - Output file: output.txt (copy all your console outputs to this file)
  - README file (optional, used if you have any additional comments/explanations about the files)

# DYNAMIC SEMANTICS

# Dynamic Semantics

- Describe the meaning of expressions, statements, and program units
- No single widely acceptable notation or formalism for describing semantics
- Two common approaches:
  - Operational
  - Denotational

# Operational Semantics

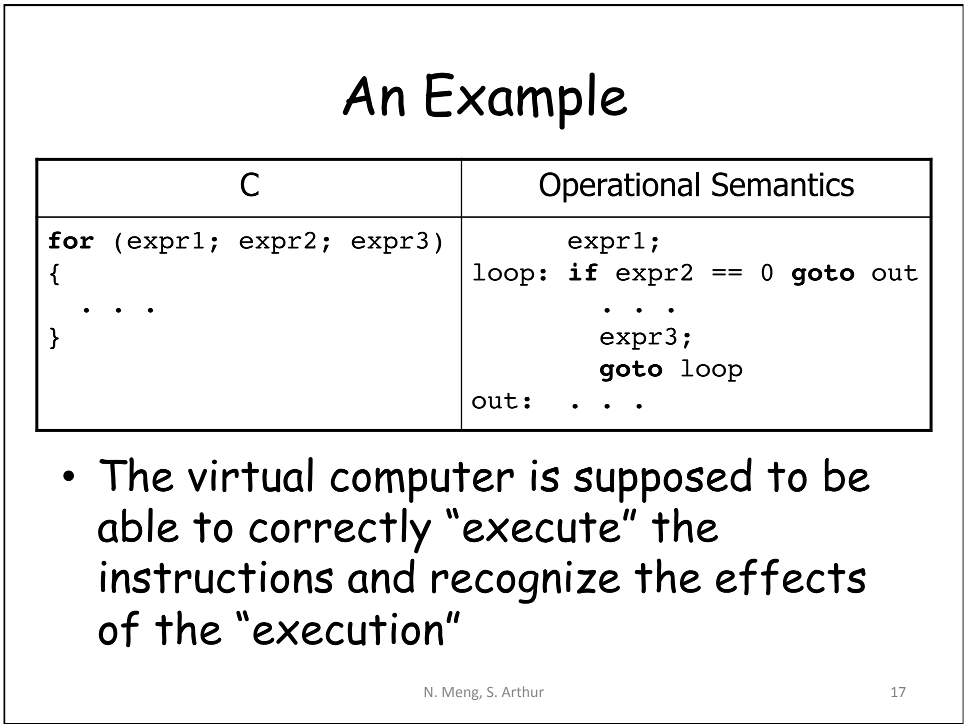
- Gives a program's meaning in terms of its implementation on a **real or virtual machine**
- Change in the state of the machine (memory, registers, etc.) defines the meaning of the statement

# Operational Semantics

- There are different levels of operational semantics
  - **Big-Step Semantics**: At the highest level, **natural operational semantics** are used to describe the final execution result of a complete program
  - **Small-Step Semantics**: At the lowest level, **structural operational semantics** are used to determine the precise meaning of a single statement
    - How does the statement change the state of a real/virtual machine, such as memory and registers ?

# Operational Semantics Definition Process

1. Design an appropriate intermediate language. Each construct of the intermediate language must have an obvious and unambiguous meaning
2. Construct a virtual machine (an interpreter) for the intermediate language. The virtual machine can be used to execute either single statements, code segments, or whole programs



# An Example

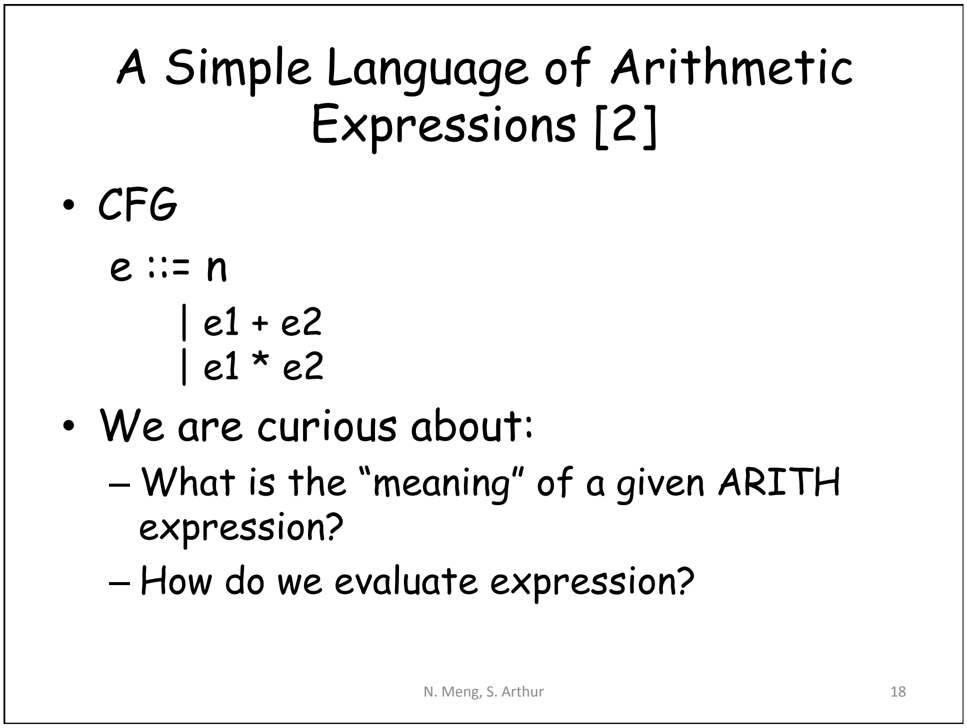
C

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

Operational Semantics

```
      expr1;
loop: if expr2 == 0 goto out
        . . .
        expr3;
        goto loop
out:  . . .
```

- The virtual computer is supposed to be able to correctly “execute” the instructions and recognize the effects of the “execution”



# A Simple Language of Arithmetic Expressions [2]

- CFG

  e ::= n
  | e1 + e2
  | e1 * e2

- We are curious about:
  - What is the “meaning” of a given ARITH expression?
  - How do we evaluate expression?

# Operational Semantics of Arithmetic Expressions [2]

- Specify how expressions should be evaluated
- Defined by cases on the form of expressions
  - – n evaluates to n
    - n is a normal form, an expression that cannot be reduced further
  - – $e_1$ + $e_2$ evaluates to n if
    - $e_1$ evaluates to $n_1$,
    - $e_2$ evaluates to $n_2$, and
    - n is the sum of $n_1$ and $n_2$

# Operational Semantics of Arithmetic Expressions [2]

- – $e_1$ * $e_2$ evaluates to n if
  - $e_1$ evaluates to $n_1$,
  - $e_2$ evaluates to $n_2$, and
  - n is the product of $n_1$ and $n_2$

## Big-Step Operational Semantics [2]

$$\frac{}{n \Downarrow n}$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n \text{ is the sum of } n_1 \text{ and } n_2}{e_1 + e_2 \Downarrow n}$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n \text{ is the product of } n_1 \text{ and } n_2}{e_1 * e_2 \Downarrow n}$$

## Small-Step Operational Semantics [3]

- Describe a single step in the evaluation
- Show intermediate results and how to calculate each result
- Many steps may be needed to get a result

# Small-Step Operational Semantics [3]

$$\frac{n \text{ is the sum of } n_1 \text{ and } n_2}{n_1 + n_2 \rightarrow n} \qquad \frac{n \text{ is the product of } n_1 \text{ and } n_2}{n_1 * n_2 \rightarrow n}$$

$$\frac{e_1 \rightarrow e_1'}{e_1 + e_2 \rightarrow e_1' + e_2} \qquad \frac{e_2 \rightarrow e_2'}{n_1 + e_2 \rightarrow n_1 + e_2'}$$

$$\frac{e_1 \rightarrow e_1'}{e_1 * e_2 \rightarrow e_1' * e_2} \qquad \frac{e_2 \rightarrow e_2'}{n_1 * e_2 \rightarrow n_1 * e_2'}$$

- The semantic rules tell not only the operator meanings, but also evaluation orders
- E.g., (1+2) + (3+4) = 3 + (3+4)

# Key Points of Operational Semantics

- Advantages
  - May be simple and intuitive for small examples
  - Good if used informally
  - Useful for implementation
- Disadvantages
  - Very complex for large programs
  - Lacks mathematical rigor

# Typical Usage of Operational Semantics

- Vienna Definition Language (VDL) used to define PL/I (Wegner 1972)
- Unfortunately, VDL is so complex that it serves no practical purpose

# Denotational Semantics

- The most rigorous, widely known method for describing the meaning of programs
- Solely based on recursive function theory
- Originally developed by Scott and Strachey (1970)

# Denotational Semantics

- Key Idea
  - Define for each language entity both a mathematical object, and a function that maps instances of that entity onto instances of the mathematical object
- The basic idea
  - There are rigorous ways of manipulating mathematical objects but not programming language constructs

# Denotational Semantics

- Difficulty
  - How to create the objects and the mapping functions?
- The method is named *denotational*, because the mathematical objects denote the meaning of their corresponding syntactic entities

# Denotational vs. Operational

- Both denotational semantics and operational semantics are defined in terms of state changes in a virtual machine
- In operational semantics, the state changes are defined by **coded algorithms** in the machine
- In denotational semantics, the state change is defined by **rigorous mathematical functions**

# Program State

- Let the state s of a program be a set of pairs as follows:

  $\{<i_1, v_1>, <i_2, v_2>, \ldots, <i_n, v_n>\}$

  - Each i is the name of a variable
  - The associated v is the current value of the variable
  - Any v can have the special value **undef**, indicating that the associated variable is undefined
- Let VARMAP be a function as follows:

  $VARMAP(i_j, s) = v_j$

# Program State

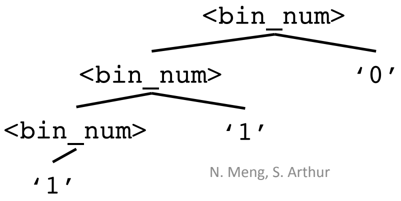
- Most semantics mapping functions for programs and program constructs map from states to states
- These state changes are used to define the meanings of programs and program constructs
- Some language constructs, such as expressions, are mapped to values, not state changes

# An Example

- CFG for binary numbers

  <bin_num> -> '0'
  <bin_num> -> '1'
  <bin_num> -> <bin_num> '0'
  <bin_num> -> <bin_num> '1'

- Parse tree of the binary number 110



# Example Semantic Rule Design

- Mathematical objects
  - Decimal number equivalence for each binary number
- Functions
  - Map binary numbers to decimal numbers
  - Rules with terminals as RHS are translated as direct mappings from terminals to mathematical objects
  - Rules with nonterminals as RHS are translated as manipulations on mathematical objects

# Example Semantic Rules

| Syntax Rules | Semantic Rules |
|---|---|
| <bin_num>->‘0’ | $M_{bin}$(‘0’)=0 |
| <bin_num>->‘1’ | $M_{bin}$(‘1’)=1 |
| <bin_num>-><bin_num> ‘0’ | $M_{bin}$(<bin_num> ‘0’)=<br>2*$M_{bin}$(<bin_num>) |
| <bin_num>-><bin_num> ‘1’ | $M_{bin}$(<bin_num> ‘1’)=<br>2*$M_{bin}$(<bin_num>)+1 |

# Expressions

- CFG for expressions

<expr> -> <dec_num> | <var> | <binary_expr>
<binary_expr> -> <l_expr> <op> <r_expr>
<l_expr> -> <dec_num> | <var>
<r_expr> -> <dec_num> | <var>
<op> -> + | *

# Expressions

```
$M_e$(<expr>, s) $\Delta$=
  case <expr> of
    <dec_num> $\Rightarrow$ $M_{dec}$(<dec_num>)
    <var> $\Rightarrow$ VARMAP(<var>, s)
    <binary_expr> $\Rightarrow$
      if (<binary_expr>.<op> = '+') then
        $M_e$(<binary_expr>.<l_expr>, s) +
        $M_e$(<binary_expr>.<r_expr>, s)
      else
        $M_e$(<binary_expr>.<l_expr>, s) $\times$
        $M_e$(<binary_expr>.<r_expr>, s)
```