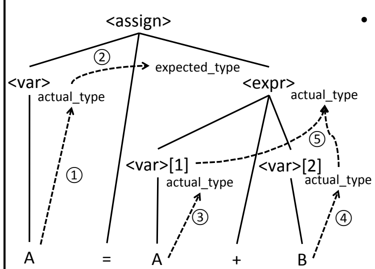
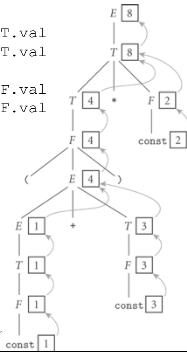
Attribute Evaluation Order



- Determining attribute evaluation order for any attribute grammar is a complex problem, requiring the construction of a dependency graph to show all attribute dependencies

Decoration of a parse tree for the val attribute evaluation of (1 + 3) * 2

```
E1 → E2 + T     E1.val = E2.val + T.val
E1 → E2 - T     E1.val = E2.val - T.val
E  → T          E.val  = T.val
T1 → T2 * F     T1.val = T2.val * F.val
T1 → T2 / F     T1.val = T2.val / F.val
T  → F          T.val  = F.val
F1 → - F2       F1.val = - F2.val
F  → (E)        F.val  = E.val
F  → const      F.val  = C.val
```



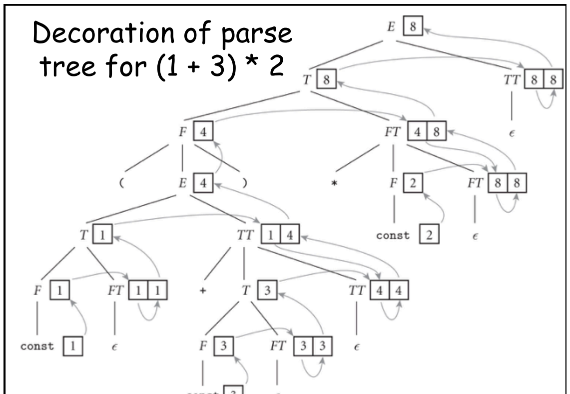
Attribute Grammar for Constant Expressions based on LL(1) CFG

1. $E \longrightarrow T\ TT$
   ▷ TT.st := T.val ▷ E.val := TT.val
2. $TT_1 \longrightarrow +\ T\ TT_2$
   ▷ TT_2.st := TT_1.st + T.val ▷ TT_1.val := TT_2.va
3. $TT_1 \longrightarrow -\ T\ TT_2$
   ▷ TT_2.st := TT_1.st − T.val ▷ TT_1.val := TT_2.va
4. $TT \longrightarrow \epsilon$
   ▷ TT.val := TT.st
5. $T \longrightarrow F\ FT$
   ▷ FT.st := F.val ▷ T.val := FT.val
6. $FT_1 \longrightarrow *\ F\ FT_2$
   ▷ FT_2.st := FT_1.st × F.val ▷ FT_1.val := FT_2.va
7. $FT_1 \longrightarrow /\ F\ FT_2$
   ▷ FT_2.st := FT_1.st ÷ F.val ▷ FT_1.val := FT_2.va
8. $FT \longrightarrow \epsilon$
   ▷ FT.val := FT.st
9. $F_1 \longrightarrow -\ F_2$
   ▷ F_1.val := − F_2.val
10. $F \longrightarrow (\ E\ )$
    ▷ F.val := E.val
11. $F \longrightarrow$ const
    ▷ F.val := const.val

Attribute Grammar for CE LL(1) CFG

- Attributes
  - *st*: subtotal attribute to record intermediate evaluation result so far
  - *val*: value attribute to copy the right-most leaf back up to the root

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



Program Assignment 2
Due date: 11/9 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 → A EE
EE.st = A.val     E.val = EE.val

EE1 → | A EE2
EE2.st = EE1.st | A.val        (  "|" bitwise OR )
EE1.val = EE2.val

EE → ε
EE.val = EE.st

A → B AA
AA.st = B.val     A.val = AA.val

AA1 → ^ B AA2
AA2.st = AA1.st ^ B.val        ( "^" bitwise XOR )
AA1.val = AA2.val

AA → ε
AA.val = AA.st

B → C BB
BB.st = C.val     B.val = BB.val

BB1 → & C BB2
BB2.st = BB1.st & C.val        ( "&" bitwise AND )
BB1.val = BB2.val

BB → ε
BB.val = BB.st

C1 → <C2
C1.val = C2.val << 1           ( "<<" bitwise shift left one )

C1 → >C2
C1.val = C2.val >> 1           ( ">>" bitwise shift right one )

C1 → ~C2
C1.val = ~C2.val               ( "~" bitwise NOT )

C → ( E )
C.val = E.val

C → 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 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	Operational Semantics

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

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:

  $\{\langle i_1, v_1\rangle, \langle i_2, v_2\rangle, \ldots, \langle i_n, v_n\rangle\}$
  - 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:

  $\text{VARMAP}(i_j, s) = v_j$

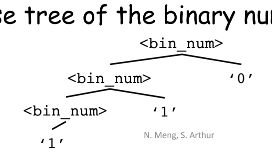
Program State

- 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' <bin_num>->'1' <bin_num>-><bin_num> '0' <bin_num>-><bin_num> '1'	M_{bin}('0')=0 M_{bin}('1')=1 M_{bin}(<bin_num> '0')= 2*M_{bin}(<bin_num>) M_{bin}(<bin_num> '1')= 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)

Statement Basics

- The meaning of a single statement executed in a state s is a new state s', which reflects the effects of the statement

  M_{stmt}(stmt , s) = s'

Assignment Statements

$M_a(x := E, s)\ \Delta=$
$s' = \{<i_1', v_1'>, <i_2', v_2'>, \ldots, <i_n', v_n'>\}$,
where for j = 1, 2, ..., n,
$v_j' = \text{VARMAP}(i_j, s)$ if $i_j \neq x$
$v_j' = M_e(E, s)$ if $i_j = x$

Sequence of Statements

M_{stmt}(stmt1; stmt2 , s) $\Delta=$
M_{stmt}(stmt2 , M_{stmt}(stmt1 , s))
or
M_{stmt}(stmt1; stmt2 , s) = s'' where
s' = M_{stmt}(stmt1 , s)
s'' = M_{stmt}(stmt2 , s')

Sequence of Statements

```
x := 5;
y := x + 1;
write(x * y);
```

$\left.\begin{array}{l} x := 5; \\ \left.\begin{array}{l} y := x + 1; \\ write(x * y); \end{array}\right\} P2 \end{array}\right\} P1 \Big\} P0$

Initial state s_0 = <mem_0, i_0, o_0>

$M_{stmt}(P_0, s_0) = M_{stmt}(P_1, \underbrace{M_a(x := 5, s_0)}_{s_1})$

s_1 = <mem_1, i_1, o_1> where

VARMAP(x, s_1) = 5

VARMAP(z, s_1) = VARMAP(z, s_0) for all $z \neq x$

$i_1 = i_0$, $o_1 = o_0$

Sequence of Statements

$\left.\begin{array}{l} x := 5; \\ \left.\begin{array}{l} y := x + 1; \\ write(x * y); \end{array}\right\} P2 \end{array}\right\} P1 \Big\} P0$

$M_{stmt}(P_1, s_1) = M_{stmt}(P_2, \underbrace{M_a(y := x + 1, s_1)}_{s_2})$

s_2 = <mem_2, i_2, o_2> where

VARMAP(y, s_2) = M_e(x + 1, s_1) = 6

VARMAP(z, s_2) = VARMAP(z, s_1) for all $z \neq y$

$i_2 = i_1$, $o_2 = o_1$

Sequence of Statements

$\left.\begin{array}{l} x := 5; \\ \left.\begin{array}{l} y := x + 1; \\ write(x * y); \end{array}\right\} P2 \end{array}\right\} P1 \Big\} P0$

$M_{stmt}(P_2, s_2) = M_{stmt}(write(x * y), s_2) = s_3$

s_3 = <mem_3, i_3, o_3> where

VARMAP(z, s_3) = VARMAP(z, s_2) for all z

$i_3 = i_2$, $o_3 = o_2 \cdot M_e(x * y, s_2) = o_2 \cdot 30$

Sequence of Statements

Therefore,

$M_{stmt}(P, s_0) = s_3$ = <mem_3, i_3, o_3> where

VARMAP(y, s_3) = 6

VARMAP(x, s_3) = 5

VARMAP(z, s_3) = VARMAP(z, s_0) for all $z \neq x, y$

$i_3 = i_0$

$o_3 = o_0 \cdot 30$

Logical Pretest Loops

- The meaning of the loop is the value of program variables after the loop body has been executed the prescribed number of times, assuming there have been no errors
- The loop is converted from iteration to recursion, where the recursion control is mathematically defined by other recursive state mapping functions
- Recursion is easier to describe with mathematical rigor than iteration

Logical Pretest Loop

- M_l(while B do L, s) Δ=

  if M_b(B, s) = false then

  s

  else

  M_l(while B do L, **M_{stmt}(L, s)**)

Postest Loop ?

- M_{ptl}(do L until not B, s) Δ = ?

Key Points of Denotational Semantics

- Advantages
  - **Compact & precise**, with solid mathematical foundation
  - Provide a **rigorous** way to think about programs
  - Can be used to prove the correctness of programs
  - Can be an aid to language design

Key Points of Denotational Semantics

- Disadvantages
  - **Require mathematical sophistication**
  - Hard for programmer to use
- Uses
  - Semantics for Algol-60, Pascal, etc.
  - Compiler generation and optimization

Summary

- Each form of semantic description has its place
- Operational semantics
  - Informally describe the meaning of language constructs in terms of their effects on an ideal machine
- Denotational semantics
  - Formally define mathematical objects and functions to represent the meanings