CS323: Compilers

Spring 2023

Week 5: Parsers (discussion and conclusion), Semantic Processing

Discussion: LR and LL Parsers

- LR Parsers:
  - For the next token, `t,` in input sequence, LR parsers try to answer: i) should I put this token on stack? or ii) should I replace a set of tokens that are at the top of a stack?

    In shift states (case i), if there is no transition out of that state for `t`, it is a syntax error.
- LL Parsers:
  - LL parsers ask the question: which rule should I use next based on the next input token `t`?. Only after expanding all non-terminals of the rule considered, they move on to consume the subsequent input tokens

Discussion: LR and LL Parsers

Parse Table (Top-Down)


```
Grammar:
1: S -> F
2: S -> (S + F)
3: F -> a

input:
(a+)

Accepted or Not
accepted?
```

Discussion: LR and LL Parsers

Goto and Action Table?


```
Grammar:
1: S -> F
2: S -> (S + F)
3: F -> a

input:
(a+)

Accepted or Not
accepted?
```

Hand-Written Parser - FPE

- Fully parenthesized expression (FPE)
  - Expressions (algebraic notation) are the normal way we are used to seeing them. E.g. 2 + 3
  - *Fully-parenthesized* expressions are simpler versions: every binary operation is enclosed in parenthesis
    - E.g. 2 + 3 is written as (2+3)
    - E.g. (2 + (3 * 7))
    - We can ignore order-of-operations (PEMDAS rule) in FPEs.

FPE – definition

- Either a:
  1. A number (integer in our example) OR
  2. *Open parenthesis* ‘(‘ followed by
     FPE followed by
     an operator (‘+’, ‘-’, ‘*’, ‘/’) followed by
     FPE followed by
     closed parenthesis ‘)’

FPE – Notation

```
1. E -> INTLITERAL
2. E -> (E op E)
3. op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

1. One function defined for every non-terminal
   - `E, op`
2. One function defined for every production
   - `E1, E2`
3. One function defined for all terminals
   - `IsTerm`

```
1.E -> INTLITERAL
2.E -> (E op E)
3.op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

1. One function defined for every non-terminal
   - `E, op`
2. One function defined for every production
   - `E1, E2`
3. One function defined for all terminals
   - `IsTerm`

```
1.E -> INTLITERAL
2.E -> (E op E)
3.op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

This function checks if the next token returned by the scanner matches the expected token. Returns `true` *if match.* `false` *if no match.*

Assume that a scanner module has been provided. The scanner has one function, `GetNextToken`, that returns the next token in the sequence.

Can be any one of: INTLITERAL, LPAREN, RPAREN, ADD, SUB, MUL, DIV

```
bool IsTerm(Scanner* s, TOKEN tok) {
    return s->GetNextToken() == tok;
}
```

Implementing a parser for FPE

1. One function defined for every non-terminal
   - `E, op`
2. One function defined for every production
   - `E1, E2`
3. One function defined for all terminals
   - `IsTerm`

```
1.E -> INTLITERAL
2.E -> (E op E)
3.op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

This function implements production #1: E->INTLITERAL
Returns `true` *if the next token returned by the scanner is an INTLITERAL.* `false` *otherwise.*

```
bool E1(Scanner* s) {

   return IsTerm(s, INTLITERAL);

}
```

Implementing a parser for FPE

1. One function defined for every non-terminal
   - `E, op`
2. One function defined for every production
   - `E1, E2`
3. One function defined for all terminals
   - `IsTerm`

```
1.E -> INTLITERAL
2.E -> (E op E)
3.op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

This function implements production #2: `E->(E op E)`
Returns `true` *if the Boolean expression on line 2 returns* `true`*.* `false` *otherwise.*

```
1: bool E2(Scanner* s) {

2:  return IsTerm(s, LPAREN) &&
           E(s)               &&
           OP(s)              &&
           E(s)               &&
           IsTerm(s, RPAREN);

3: }
```

Implementing a parser for FPE

1. One function defined for every non-terminal
   - `E`, `op`
2. One function defined for every production
   - `E1`, `E2`
3. One function defined for all terminals
   - `IsTerm`

```
1.E -> INTLITERAL
2.E -> (E op E)
3.op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

This function implements production #3: `op->ADD|SUB|MUL|DIV`
Returns `true` *if the next token returned by the scanner is any one from* `ADD, SUB, MUL, DIV`*.* `false` *otherwise.*

```
bool OP(Scanner* s) {

   TOKEN tok = s->GetNextToken();

   if((tok == ADD) || (tok == SUB) || (tok == 
   MUL) || (tok == DIV))
      return true;

   return false;

}
```

Implementing a parser for FPE

1. One function defined for every non-terminal
   - `E, op`
2. One function defined for every production
   - `E1, E2`
3. One function defined for all terminals
   - `IsTerm`

```
1.E -> INTLITERAL
2.E -> (E op E)
3.op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

This function implements the routine for matching non-terminal E

Assume that GetCurTokenSequence returns a reference to the first token in a sequence of tokens maintained by the scanner

```
bool E(Scanner* s) {

   TOKEN* prevToken = s->GetCurTokenSequence();
   if(!E1(s)) {
      s->SetCurTokenSequence(prevToken);
      return E2(s);
   }
   return true;

}
```

Implementing a parser for FPE

This function implements the routine for matching non-terminal E

```
bool E(Scanner* s) {

   TOKEN* prevToken = s->GetCurTokenSequence();
   if(!E1(s)) {
      s->SetCurTokenSequence(prevToken);
      return E2(s);
   }
   return true;

}
```

//This line implements the check to see if the sequence of tokens match production #1: E->INTLITERAL.

Implementing a parser for FPE

This function implements the routine for matching non-terminal E

```
bool E(Scanner* s) {

   TOKEN* prevToken = s->GetCurTokenSequence();
   if(!E1(s)) {
       s->SetCurTokenSequence(prevToken);
       return E2(s);
   }
   return true;
```

//because E1(s) calls s->GetNextToken() internally, the reference to the sequence of tokens would have moved forward. This line restores the reference back to the first node in the sequence so that the scanner provides the correct sequence to the call E2 in next line

Implementing a parser for FPE

This function implements the routine for matching non-terminal E

```
bool E(Scanner* s) {

   TOKEN* prevToken = s->GetCurTokenSequence();
   if(!E1(s)) {
      s->SetCurTokenSequence(prevToken);
      return E2(s);
   }
   return true;
```

//This line implements the check to see if the sequence of tokens match production #2: E->(E op E)

Implementing a parser for FPE

```
IsTerm(Scanner* s, TOKEN tok) { return s->GetNextToken() == tok;}

bool E1(Scanner* s) {
     return IsTerm(s, INTLITERAL);
}

bool E2(Scanner* s) { return IsTerm(s, LPAREN) && E(s) && OP(s) && E(s) && IsTerm(s, RPAREN); }

bool OP(Scanner* s) {
     TOKEN tok = s->GetNextToken();
     if((tok == ADD) || (tok == SUB) || (tok == MUL) || (tok == DIV))
          return true;
     return false;
}

bool E(Scanner* s) {
     TOKEN* prevToken = s->GetCurTokenSequence();
     if(!E1(s)) {
          s->SetCurTokenSequence(prevToken);
          return E2(s);
     }
     return true;
}
```

Start the parser by invoking E().
Value returned tells if the expression is FPE or not.

Exercise

- What parsing technique does this parser use?

LR(k) parsers

- LR(0) parsers
  - No lookahead
  - Predict which action to take by looking only at the symbols currently on the stack
- LR(k) parsers
  - Can look ahead *k* symbols
  - Most powerful class of deterministic bottom-up parsers
  - LR(1) and variants are the most common parsers

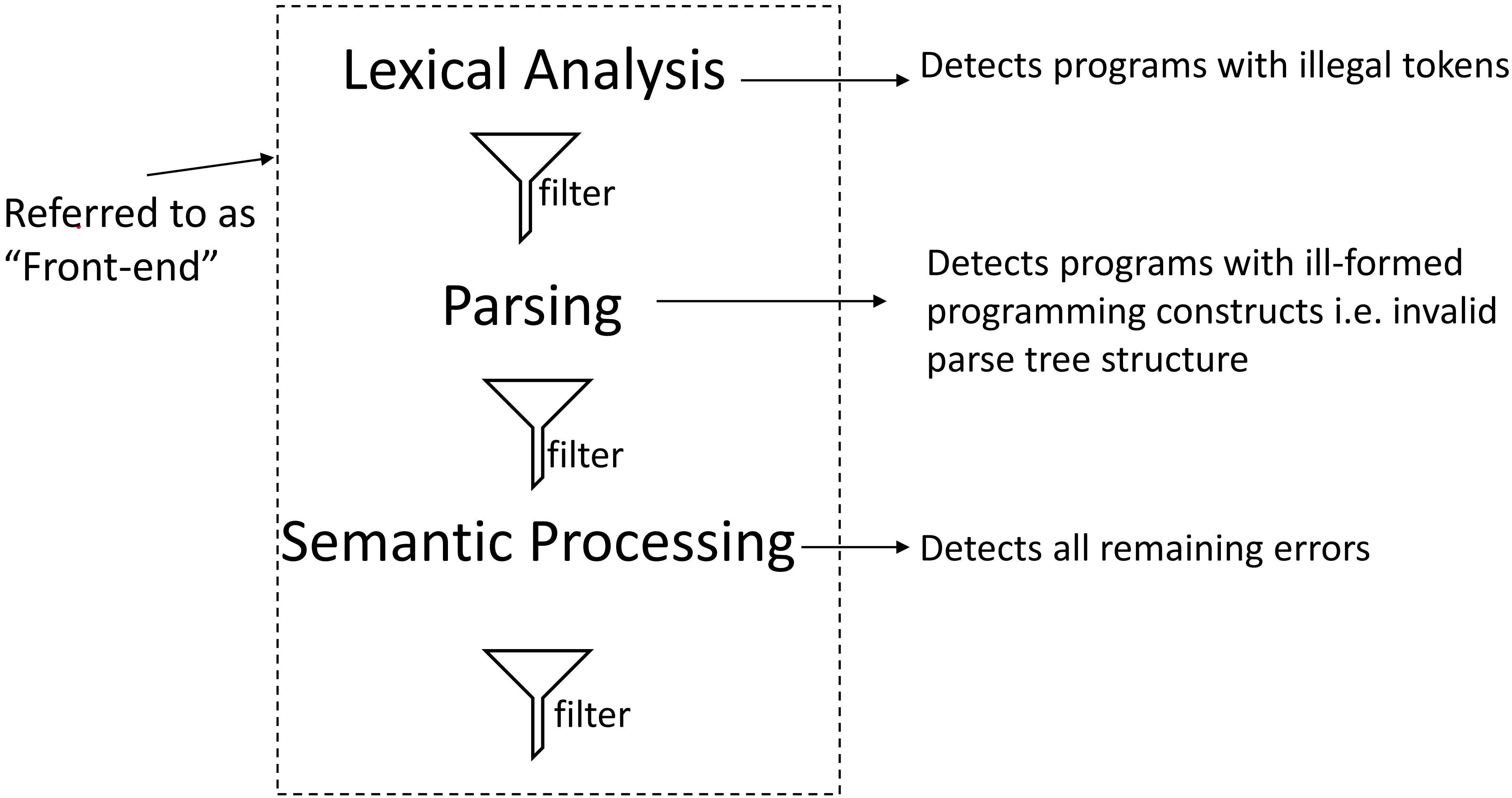
slide courtesy: Milind Kulkarni

Top-down vs. Bottom-up parsers

- Top-down parsers expand the parse tree in *pre-order*
  - Identify parent nodes before the children
- Bottom-up parsers expand the parse tree in *post-order*
  - Identify children before the parents
- Notation:
  - LL(1): Top-down derivation with 1 symbol lookahead
  - LL(k): Top-down derivation with k symbols lookahead
  - LR(1): Bottom-up derivation with 1 symbol lookahead

slide courtesy: Milind Kulkarni

Semantic Processing



Semantic Processing

- Syntax-directed / syntax-driven
  - Routines (called as <u>semantic routines</u>) interpret the meaning of programming constructs based on the syntactic structure

  - Routines play a dual role
    - <u>Analysis</u> – Semantic analysis
      - undefined vars, undefined types, uninitialized variables, type errors that can be caught at compile time, unreachable code, etc.
    - <u>Synthesis</u> – Generation of intermediate code
      - 3 address code

  - Routines create <u>semantic records</u> to aid the analysis and synthesis

Semantic Processing

- Syntax-directed translation: notation for *attaching* program fragments to grammar productions.
  - Program fragments are executed when productions are matched
  - The combined execution of all program fragments produces the translation of the program

    e.g. `E->E+T   { print('+') }`

`Output:` program fragments may create AST and 3 Address Codes

- Attributes: any 'quality' associated with a terminal and non-terminal e.g. type, number of lines of a code, first line of the code block etc.

Why Semantic Analysis?

- Context-free grammars cannot specify all requirements of a language
  - Identifiers declared before their use (scope)
  - Types in an expression must be consistent

    STRING str:= “Hello”;

    str := str + 2;
  - Number of formal and actual parameters of a function must match
  - Reserved keywords cannot be used as identifiers
  - A Class is declared only once in a OO language program, a method of a class can be overridden.
  - …

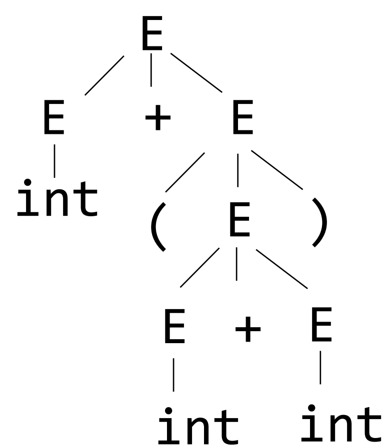
Abstract Syntax Tree

- Abstract Syntax Tree (AST) or Syntax Tree can be the input for semantic analysis.
  - What is Concrete Syntax Tree? – the parse tree
- ASTs are like parse trees but ignore certain details:

E.g. Consider the grammar:

```
E - > E + E
    | ( E )
    | int
```

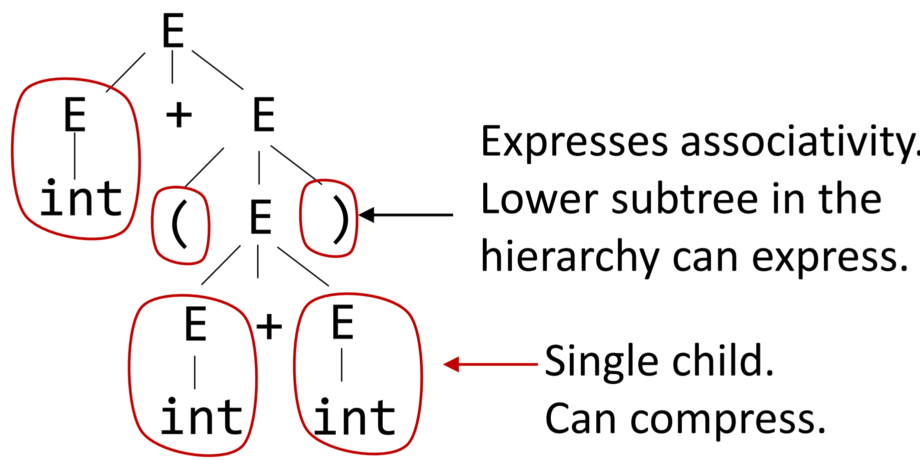
The parse tree for `1+(2+3)`



AST - Example

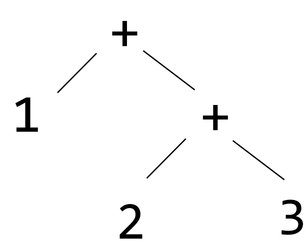
- Not all details (nodes) of the parse tee are helpful for semantic analysis

The *parse tree* for `1+(2+3)` :



We need to compute the result of the expression. So, a simpler structure is sufficient:

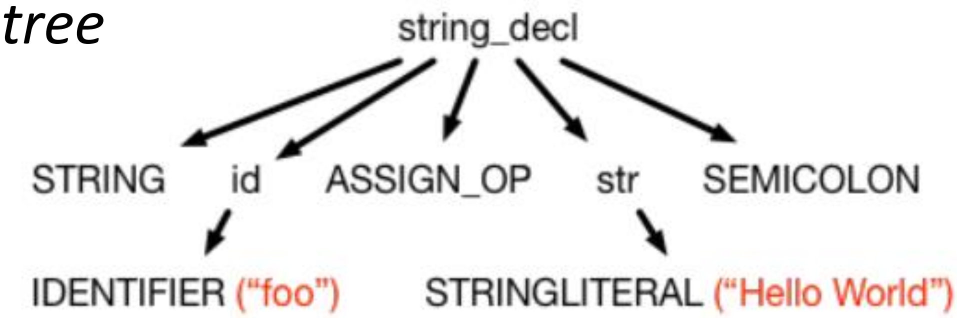
AST for `1+(2+3)`:



AST - Example

`STRING foo := “Hello World” ;`

parse tree ⇨



≡ *AST* ⇩

```
            ASSIGN_OP
           /         \
IDENTIFIER            STRINGLITERAL
(“foo”)               (“Hello World”)
```

Semantic Analysis – Example

- Context-free grammars cannot specify all requirements of a language
  - Identifiers declared before their use (scope)
  - Types in an expression must be consistent

    Type checks
    STRING str:= "Hello";
    str := str + 2;
  - Number of formal and actual parameters of a function must match
  - Reserved keywords cannot be used as identifiers
  - A Class is declared only once in a OO language, a method can be overridden.
  - …

Scope

- **Goal:** matching identifier declarations with uses
- Most languages require this!
- Scope confines the activity of an identifier

```
        ASSIGN_OP
       /         \
IDENTIFIER     STRINGLITERAL
(“foo”)        (“hello world”)
```

What if `foo` is declared as a STRING in an enclosing scope but is an INT in the current scope?

in different parts of the program:

- Same identifier may refer to different things
- Same identifier may not be accessible

Static Scope

- Most languages are statically scoped
  - Scope depends on only the program text (not runtime behavior)
  - A variable refers to the closest defined instance

```
INT w, x;
{
    FLOAT x, z;
    f(x, w, z);
}
g(x)
```

x is a FLOAT here

x is an INT here

Dynamic Scope

- In dynamically scoped languages
  - Scope depends on the execution context
  - A variable refers to the closest enclosing binding in the execution of the program

```
f(){
    a=4; g();
}
g() { print(a); }
```

value of a is 4 here

Exercise: Static vs. Dynamic Scope

```
#define a (x+1) //macro definition

int x = 2; //global var definition

//function b definition
void b() {
    int x = 1;
    printf("%d\n",a);
}

//function c definition
void c() {
    printf("%d\n",a);
}

//the main function
int main() { b(); c(); }
```

Is x statically scoped or dynamically scoped?

Symbol Table

- Data structure that tracks the bindings of identifiers. Specifically, returns the current binding.
  - E.g., stores a mapping of names to types
  - Should provide for efficient retrieval and frequent insertion and deletion of names.
  - Should consider scopes

```
{
    int x = 0;
    //accessing y here should be illegal
    {
        int y = 1;
    }
}
```

- Can use stacks, binary trees, hash maps for implementation

Symbol Table and Classes in OO Language

- Class names may be used before their definition
- Can’t use symbol table (to check class definition)
  - Gather all class names first.
  - Check bindings next.

Implies going over the program text multiple times

- Semantic analysis is done in multiple passes
- One of the goals of semantic analysis is to create/update data structures that help the next round of analysis

Semantic Analysis – How?

- Recursive descent of AST
  - Process a node, n
  - Recurse into children of n and process them
  - Finish processing the node, n

$\Rightarrow$Do a postorder processing of the AST

- As you visit a node, you will add information depending upon the analysis performed
  - The information is referred to as <u>attributes</u> of the node

Building AST - Example

- Fully-Parenthesized Expressions (FPE)
  - Can build while parsing via bottom-up building of the tree
  - Create subtrees, make those subtrees left- and right-children of a newly created root.
  - Need to modify the hand-written recursive parser:

    if:

    token == INTLITERAL, return a reference to newly created node containing a number

    else:

    store references to nodes that are left- and right- expression subtrees
    Create a new node with value = ‘OP’

Building AST - Example

This function creates an AST node and adds information that stores the value of an INTLITERAL in the node. A reference to the AST node is returned.

```
TreeNode* IsTerm(Scanner* s, TOKEN tok) {
    TreeNode* ret = NULL;
    TOKEN nxtToken = s->GetNextToken();
    if(nxtToken == tok)
        ret = CreateTreeNode(nxtToken.val);
    return ret;
}
```

Building AST

E1 needs to change because IsTerm returns a TreeNode.*
E1 returns a TreeNode now.*

Recall: E1 is the function that gets called when predicting using the production: `E -> INTLITERAL`

```
TreeNode* E1(Scanner* s) {

   return IsTerm(s, INTLITERAL);

}
```

Building AST - Example

- <u>Fully-Parenthesized Expressions (FPE)</u>
  - Can build while parsing via bottom-up building of the tree
  - Create subtrees, make those subtrees left- and right-children of a newly created root.
  - Need to modify the hand-written recursive parser:

    if:

    token == INTLITERAL, return a reference to newly created node containing a number

    else:

    store references to nodes that are left- and right- expression subtrees
    Create a new node with value = ‘OP’

Building AST

This function creates an AST node and adds information that stores the value of an op in the node. A reference to the AST node is returned.

Recall: op is the function that gets called when predicting using the production:
`op -> ADD | SUB | MUL | DIV`

```
TreeNode* OP(Scanner* s, TOKEN tok) {
    TreeNode* ret = NULL;
    TOKEN tok = s->GetNextToken();
    if((tok == ADD) || (tok == SUB) || (tok ==
MUL) || (tok == DIV))
        ret = CreateTreeNode(tok.val);
    return ret;
}
```

Building AST

This function sets the references to left- and right- expression subtrees if those subtrees are valid FPEs. Returns reference to the AST node corresponding to the op value, NULL otherwise.

Recall: E2 is the function that gets called when predicting using the production: `E -> (E op E)`

```
TreeNode* E2(Scanner* s, TOKEN tok) {
   TOKEN nxtTok = s->GetNextToken();
   if(nxtTok == LPAREN) {
      TreeNode* left = E(s); if(!left) return NULL;
      TreeNode* root  = OP(s); if(!root) return NULL;
      TreeNode* right = E(s); if(!right) return NULL;
      nxtTok = s->GetNextToken();
      if(nxtTok != RPAREN); return NULL;
         //set left and right as children of root.
      return root;
}
```

Building AST

E needs to change because E1, E2, and OP return a TreeNode.*
E returns a TreeNode now.*

Recall: E is the higher-level function for a non-terminal that gets called when predicting using either of the productions for `E`:
`E -> (E op E) | INTLITERAL`

```
TreeNode* E(Scanner* s) {

    TOKEN* prevToken = s->GetCurTokenSequence();
    TreeNode* ret = E1(s);
    if(!ret) {
        s->SetCurTokenSequence(prevToken);
        ret = E2(s);
    }
    return ret;
}
```

Building AST

```
TreeNode* IsTerm(Scanner* s, TOKEN tok) {
      TreeNode* ret = NULL;
      TOKEN nxtToken = s->GetNextToken();
      if(nxtToken == tok)
            ret = CreateTreeNode(nxtToken.val);
      return ret;
}

TreeNode* E1(Scanner* s) {
      return IsTerm(s, INTLITERAL);
}

TreeNode* E2(Scanner* s) {
      TOKEN nxtTok = s->GetNextToken();
      if(nxtTok == LPAREN) {
            TreeNode* left = E(s);
            if(!left) return NULL;
            TreeNode* root  = OP(s);
            if(!root) return NULL;
            TreeNode* right = E(s)
            if(!right) return NULL;
            nxtTok = s->GetNextToken();
            if(nxtTok != RPAREN); return NULL;
                  //set left and right as children of root.
            return root;
      }
```

Building AST

```
TreeNode* OP(Scanner* s) {
      TreeNode* ret = NULL;
      TOKEN tok = s->GetNextToken();
      if((tok == ADD) || (tok == SUB) || (tok == MUL) || (tok == DIV))
            ret = CreateTreeNode(tok.val);
      return ret;
}

TreeNode* E(Scanner* s) {
      TOKEN* prevToken = s->GetCurTokenSequence();
      TreeNode* ret = E1(s);
      if(!ret) {
            s->SetCurTokenSequence(prevToken);
            ret = E2(s);
      }
      return ret;
}
```

Start the parser by invoking E().
Value returned is the root of the AST.

Exercise

- Did we build the AST bottom-up or top-down?

AST Construction with Hand-written Parser

```
TreeNode* E(Scanner* s) {
TOKEN* prevToken = s->GetCurTokenSequence();
     TreeNode* ret = E1(s);
     if(!ret) {
          s->SetCurTokenSequence(prevToken);
          ret = E2(s);
     }
     return ret;
}
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

      | MUL | DIV
```

Input string: (2+3)

next token

Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

Call stack	Parse tree
E()	E

Start by calling parser function E. Note the call stack contains E(). The parse tree is not constructed. This is a visualization aid.

AST Construction with Hand-written Parser

```
TreeNode* E(Scanner* s) {
TOKEN* prevToken = s->GetCurTokenSequence();
    TreeNode* ret = E1(s);
    if(!ret) {
        s->SetCurTokenSequence(prevToken);
        ret = E2(s);
    }
    return ret;
}
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

      | MUL | DIV
```

Input string: (2+3)

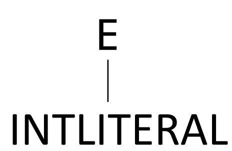
next token

Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

Call stack

E( )
E1( )

Parse tree



E() calls E1(). This is like predicting rule 1.

AST Construction with Hand-written Parser

```
TreeNode* E1(Scanner* s) {
     return IsTerm(s, INTLITERAL);
}
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

      | MUL | DIV
```

Input string: (2+3)

next token

Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

<u>Call stack</u>

```
E()
E1()
IsTerm()
```

<u>Parse tree</u>

E

INTLITERAL

E1() calls IsTerm() with an expectation that INTLITERAL is the next token.

AST Construction with Hand-written Parser

```
TreeNode* IsTerm(Scanner* s, TOKEN tok) {
     TreeNode* ret = NULL;
     TOKEN nxtToken = s->GetNextToken();
     if(nxtToken == tok)
          ret = CreateTreeNode(nxtToken.val);
     return ret;
}
```

```
1.E -> INTLITERAL
2.E -> (E op E)
3.op -> ADD | SUB
      | MUL | DIV
```

Input string: (2+3)

next token

Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

Call stack

```
E()
E1()
IsTerm()
```

Parse tree

E
|
INTLITERAL

IsTerm() calls GetNextToken() which returns LPAREN.

AST Construction with Hand-written Parser

```
TreeNode* IsTerm(Scanner* s, TOKEN tok) {
    TreeNode* ret = NULL;
    TOKEN nxtToken = s->GetNextToken();
    if(nxtToken == tok)
        ret = CreateTreeNode(nxtToken.val);
    return ret;
}
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

       | MUL | DIV
```

Input string: (2+3)

next token

Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

Call stack

```
E()
E1()
IsTerm()
```

Parse tree

IsTerm() calls GetNextToken() which returns LPAREN. In addition, GetNextToken() advances the 'next token' pointer.

AST Construction with Hand-written Parser

```
TreeNode* IsTerm(Scanner* s, TOKEN tok) {
     TreeNode* ret = NULL;
     TOKEN nxtToken = s->GetNextToken();
     if(nxtToken == tok)
          ret = CreateTreeNode(nxtToken.val);
     return ret;
}
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

       | MUL | DIV
```

Input string: (2+3)

next token

Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

<u>Call stack</u>

```
E()
E1()
IsTerm()
```

<u>Parse tree</u>

E

INTLITERAL

IsTerm() calls GetNextToken() which returns LPAREN. In addition, GetNextToken() advances the 'next token' pointer. There is a mismatch (IsTerm expects INTLITERAL (tok=INTLITERAL) but nextToken is LPAREN. So returns NULL.

AST Construction with Hand-written Parser

```
TreeNode* E1(Scanner* s) {
    return IsTerm(s, INTLITERAL);
}
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

      | MUL | DIV
```

Input string: (2+3)

next token

Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

Call stack

E()
E1()

Parse tree

E
INTLITERAL

E1 returns NULL because IsTerm returned NULL
(note that an entry from call stack is popped off)

AST Construction with Hand-written Parser

```
TreeNode* E(Scanner* s) {
TOKEN* prevToken = s->GetCurTokenSequence();
     TreeNode* ret = E1(s);
     if(!ret) {
          s->SetCurTokenSequence(prevToken);
          ret = E2(s);
     }
     return ret;
}
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

      | MUL | DIV
```

Input string: (2+3)

next token

Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

Call stack

E( )

Parse tree

E

E1 returning NULL implies that predicting rule 1 failed. ret is NULL (note that an entry from call stack is popped off).

AST Construction with Hand-written Parser

```
TreeNode* E(Scanner* s) {
TOKEN* prevToken = s->GetCurTokenSequence();
     TreeNode* ret = E1(s);
     if(!ret) {
          s->SetCurTokenSequence(prevToken);
          ret = E2(s);
     }
     return ret;
}
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

      | MUL | DIV
```

Input string: (2+3)

next token

Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

Call stack

E()

Parse tree

E

E restores 'next token' pointer back to the saved pointer prevToken (using SetCurTokenSequence())

AST Construction with Hand-written Parser

```
TreeNode* E(Scanner* s) {
TOKEN* prevToken = s->GetCurTokenSequence();
     TreeNode* ret = E1(s);
     if(!ret) {
          s->SetCurTokenSequence(prevToken);
          ret = E2(s);
     }
     return ret;
}
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

      | MUL | DIV
```

Input string: (2+3)

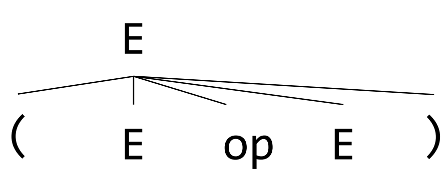
next token

Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

Call stack

E()
E2()

Parse tree



Calls E2. This is like predicting Rule 2. Note the parse tree. Again, the tree is not constructed and is used only to visualize the parsing

AST Construction with Hand-written Parser

```
TreeNode* E2(Scanner* s) {
    TOKEN nxtTok = s->GetNextToken();
    if(nxtTok == LPAREN) {
        TreeNode* left = E(s);
        if(!left) return NULL;
        TreeNode* root  = OP(s);
        ...
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

      | MUL | DIV
```

Input string: (2+3)

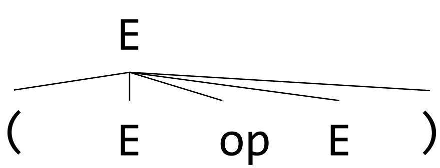
next token

Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

Call stack

E()
E2()

Parse tree



E2 check for LPAREN succeeds (note 'next token' pointer is moved forward after the call to GetNextToken().)

AST Construction with Hand-written Parser

```
TreeNode* E2(Scanner* s) {
    TOKEN nxtTok = s->GetNextToken();
    if(nxtTok == LPAREN) {
        TreeNode* left = E(s);
        if(!left) return NULL;
        TreeNode* root  = OP(s);
        ...
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

      | MUL | DIV
```

Input string: (2+3)

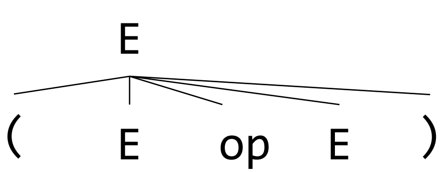
next token

Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

Call stack

E()
E2()

Parse tree



Calls E()

AST Construction with Hand-written Parser

```
TreeNode* E(Scanner* s) {
TOKEN* prevToken = s->GetCurTokenSequence();
     TreeNode* ret = E1(s);
     if(!ret) {
          s->SetCurTokenSequence(prevToken);
          ret = E2(s);
     }
     return ret;
}
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

        | MUL | DIV
```

Input string: (2+3)

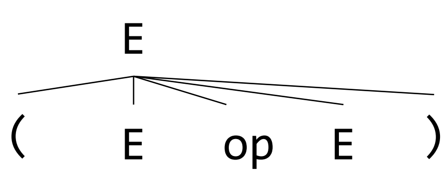
next token

Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

Call stack

E()
E2()
E()

Parse tree



E calls E1() to predict rule 1 to match the E following (in the parse tree

AST Construction with Hand-written Parser

```
TreeNode* E1(Scanner* s) {
     return IsTerm(s, INTLITERAL);
}
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

       | MUL | DIV
```

Input string: (2+3)

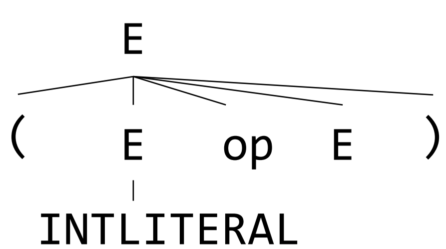
next token

Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

Call stack

E()
E2()
E()
E1()

Parse tree



E1 calls IsTerm() and expects INTLITERAL

AST Construction with Hand-written Parser

```
TreeNode* IsTerm(Scanner* s, TOKEN tok) {
     TreeNode* ret = NULL;
     TOKEN nxtToken = s->GetNextToken();
     if(nxtToken == tok)
          ret = CreateTreeNode(nxtToken.val);
     return ret;
}
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

      | MUL | DIV
```

Input string: (2+3)

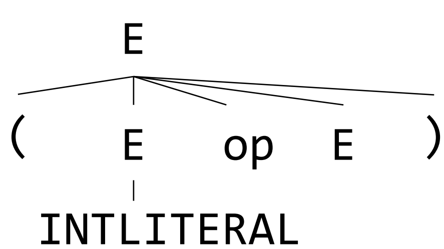
next token

Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

Call stack

```
E()
E2()
E()
E1()
IsTerm()
```

Parse tree



Call to GetNextToken() in IsTerm() now returns INTLITERAL and advances the pointer. The if condition is true.

AST Construction with Hand-written Parser

```
TreeNode* IsTerm(Scanner* s, TOKEN tok) {
      TreeNode* ret = NULL;
      TOKEN nxtToken = s->GetNextToken();
      if(nxtToken == tok)
            ret = CreateTreeNode(nxtToken.val);
      return ret;
}
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

        | MUL | DIV
```

Input string: (2+3)

next token

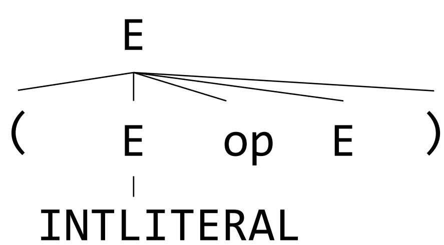
Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

AST Node is created and stores the INTLITERAL's value returned by the scanner (via s->GetNextToken()). Note that in this example we are storing the string corresponding to the integer val.

Call stack

```
E()
E2()
E()
E1()
IsTerm()
```

Parse tree



AST Construction with Hand-written Parser

```
TreeNode* IsTerm(Scanner* s, TOKEN tok) {
     TreeNode* ret = NULL;
     TOKEN nxtToken = s->GetNextToken();
     if(nxtToken == tok)
          ret = CreateTreeNode(nxtToken.val);
     return ret;
}
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

       | MUL | DIV
```

Input string: (2+3)

next token

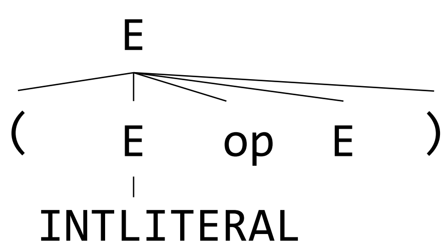
Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

IsTerm() returns the pointer to the tree node created.

Call stack

```
E()
E2()
E()
E1()
IsTerm()
```

Parse tree



“2”

AST Construction with Hand-written Parser

```
TreeNode* E1(Scanner* s) {
    return IsTerm(s, INTLITERAL);
}
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

       | MUL | DIV
```

Input string: (2+3)

next token

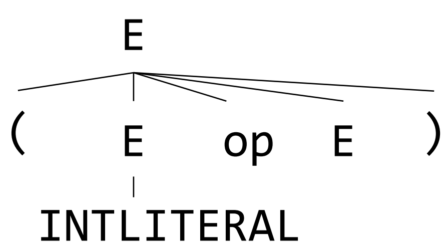
Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

E1 returns the pointer to the tree node.

Call stack

E()
E2()
E()
E1()

Parse tree



"2"

AST Construction with Hand-written Parser

```
TreeNode* E(Scanner* s) {
TOKEN* prevToken = s->GetCurTokenSequence();
     TreeNode* ret = E1(s);
     if(!ret) {
          s->SetCurTokenSequence(prevToken);
          ret = E2(s);
     }
     return ret;
}
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

      | MUL | DIV
```

Input string: (2+3)

next token

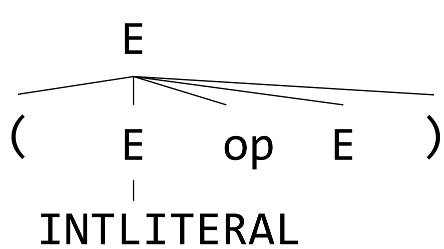
Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

E returns the pointer to the tree node.

Call stack

E()
E2()
E()

Parse tree



“2”

AST Construction with Hand-written Parser

```
TreeNode* E2(Scanner* s) {
    TOKEN nxtTok = s->GetNextToken();
    if(nxtTok == LPAREN) {
        TreeNode* left = E(s);
        if(!left) return NULL;
        TreeNode* root  = OP(s);
        ...
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

      | MUL | DIV
```

Input string: (2+3)

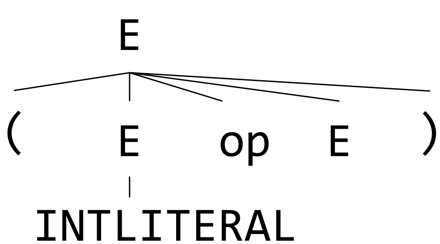
next token

Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

Call stack

E()
E2()

Parse tree



E2() now has a non-null value set for left (left is a pointer to the root of the left subtree). The if condition is false.

AST Construction with Hand-written Parser

```
TreeNode* E2(Scanner* s) {
    TOKEN nxtTok = s->GetNextToken();
    if(nxtTok == LPAREN) {
        TreeNode* left = E(s);
        if(!left) return NULL;
        TreeNode* root  = OP(s);
        ...
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

      | MUL | DIV
```

Input string: (2+3)

next token

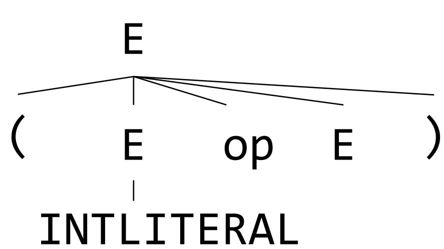
Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

E2() calls Op()

Call stack

E()
E2()

Parse tree



"2"

AST Construction with Hand-written Parser

```
TreeNode* OP(Scanner* s) {
     TreeNode* ret = NULL;
     TOKEN tok = s->GetNextToken();
     if((tok == ADD) || (tok == SUB) || (tok == MUL) ||
     (tok == DIV))
           ret = CreateTreeNode(tok.val);
     return ret;
}
```

```
1.E -> INTLITERAL
2.E -> (E op E)
3.op -> ADD | SUB
      | MUL | DIV
```

Input string: (2+3)

next token

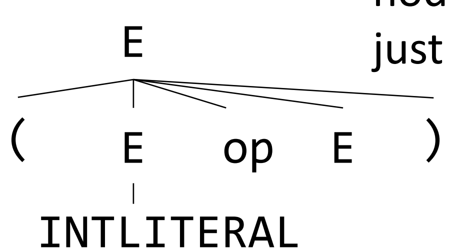
Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

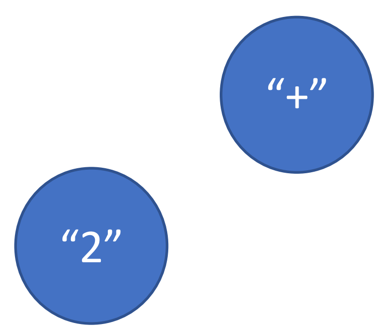
Op() first matches the next token with ADD and creates a node with value ‘+’. It then returns a pointer to the tree node just created. (note the next token pointer is also advanced)

Call stack

E()
E2()
Op()

Parse tree





AST Construction with Hand-written Parser

```
E2(){
...
      TreeNode* root  = OP(s);
      if(!root) return NULL;
      TreeNode* right = E(s)
      if(!right) return NULL;
      nxtTok = s->GetNextToken();
      if(nxtTok != RPAREN); return NULL;
          //set left and right as children of root.
      return root; }
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

        | MUL | DIV
```

Input string: (2+3)

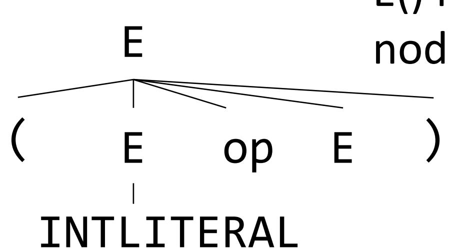
next token

Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

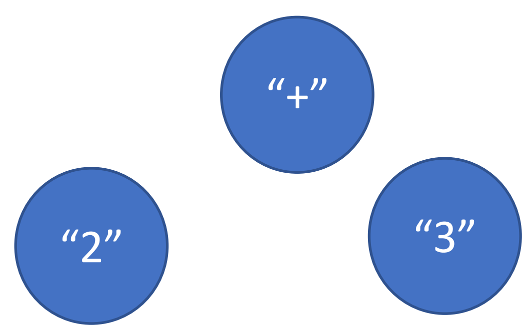
Call stack

E()
E2()

Parse tree



Now ‘root’ in E2 is set to a non-null value. E() is called next. E() in turn calls E1(), which calls IsTerm() that creates a tree node with value “3” and returns a pointer to it.



AST Construction with Hand-written Parser

```
E2(){
...
TreeNode* root  = OP(s);
if(!root) return NULL;
TreeNode* right = E(s)
if(!right) return NULL;
nxtTok = s->GetNextToken();
if(nxtTok != RPAREN); return NULL;
    //set left and right as children of root.
return root; }
```

```
1.E -> INTLITERAL

2.E -> (E op E)

3.op -> ADD | SUB

       | MUL | DIV
```

Input string: (2+3)

next token

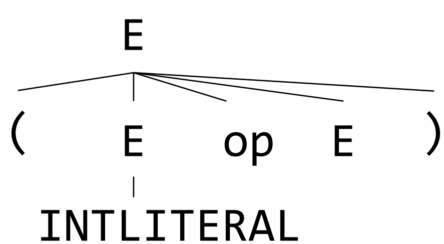
Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

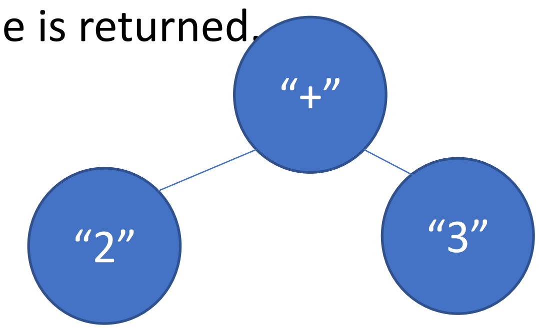
Lastly, the call to GetNextToken() in E2() returns RPAREN and the following if condition fails. Following this failure, the left and right child pointers of the 'root' node are set and the root node is returned.

Call stack

E()
E2()

Parse tree





Observations - AST Construction with Hand-written Parser

1. The AST is created bottom-up
2. Value associated with INTLITERAL/OP is added as information to the AST node
3. Pointer/reference to AST node is returned / passed up the parse tree

Identifying Semantic Actions for FPE Grammar

- What did we do when we saw an INTLITERAL?
  - Create a TreeNode
  - Initialize it with a value (string equivalent of INTLITERAL in this case)
  - Return a pointer to TreeNode

```
E -> INTLITERAL
```

triggers $\longrightarrow$

```
TreeNode* E1(Scanner* s) {
    return IsTerm(s, INTLITERAL);
}
```

$\downarrow$

```
TreeNode* IsTerm(Scanner* s, TOKEN tok) {
    TreeNode* ret = NULL;
    TOKEN nxtToken = s->GetNextToken();
    if(nxtToken == tok)
        ret = CreateTreeNode(nxtToken.val);
    return ret;
}
```

Identifying Semantic Actions for FPE Grammar

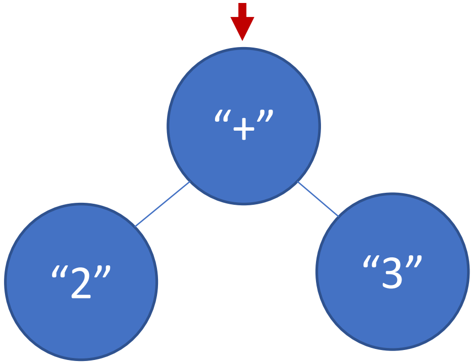
- What did we do when we saw an E (parenthesized expression)?
  - Create an AST node with two children. The node contains the binary operator OP stored as a string. Children point to roots of subtrees representing E.

`E -> (E op E)` triggers →

```
TreeNode* E2(Scanner* s) {
    TOKEN nxtTok = s->GetNextToken();
    if(nxtTok == LPAREN) {
        TreeNode* left = E(s);
        if(!left) return NULL;
        TreeNode* root  = OP(s);
        if(!root) return NULL;
        TreeNode* right = E(s)
        if(!right) return NULL;
        nxtTok = s->GetNextToken();
        if(nxtTok != RPAREN); return NULL;
            //set left and right as children of root.
        return root;
    }
```

Identifying Semantic Actions for FPE Grammar

- What did we do when we saw an E (parenthesized expression)?
    - Create an AST node with two children. The node contains the binary operator OP stored as a string. Children point to roots of subtrees representing E.
    - Returned reference to root



Identifying Semantic Actions for FPE Grammar

- We can capture the semantic actions identified in the previous slides for INTLITERAL and parenthesized E with the help of notations augmenting grammar rules

Syntax Directed Definition

- Notation containing CFG augmented with attributes and rules
- E.g.

`E -> INTLITERAL`	`E.val = INTLITERAL.val`
`E -> (E op E)`	$E.val = E_1.val\ op\ E_2.val$
`op -> ADD`	`op.val = ADD.val`
`| SUB`	`op.val = SUB.val`
`| MUL`	`op.val = MUL.val`
`| DIV`	`op.val = DIV.val`

Syntax Directed Definition

- Being more precise (w.r.t. our example)
- E.g.

`E -> INTLITERAL`	E.node = **new** TreeNode(INTLITERAL.val)
`E -> (E op E)`	E.node = TreeNode(op.node, E_1.node, E_2.node)
`op -> ADD`	op.node = new TreeNode("+")
`     | SUB`	op.node = new TreeNode("-");
`     | MUL`	op.node = new TreeNode("*");
`     | DIV`	op.node = new TreeNode("/");

- Attributes are of two types: Synthesized, Inherited

Syntax Directed Translation

- Complementary notation to SDDs containing CFG augmented with program fragments
- E.g.

`E -> INTLITERAL`	`{E.yylval = INTLITERAL.yylval;}`
`E -> (E op E)`	`{E.yylval = eval_binary(`E_1`.yylval, op, `E_2`.yylval)}`
`op -> ADD`	`{op.yylval = ADD.yylval}`
`     | SUB`	`{op.yylval = SUB.yylval}`
`     | MUL`	`{op.yylval = MUL.yylval }`
`     | DIV`	`{op.yylval = DIV.yylval}`

- Less readable than SDD. However, more efficient for optimizing

Referencing identifiers

- What do we return when we see an identifier?
  - Check if it is symbol table
  - Create new AST node with pointer to symbol table entry
  - Note: may want to directly store type information in AST (or could look up in symbol table each time)

Slide courtesy: Milind Kulkarni

Referencing Literals

- What about if we see a literal?

  primary → INTLITERAL | FLOATLITERAL

- Create AST node for literal
- Store string representation of literal
  - "155", "2.45" etc.
- At some point, this will be converted into actual representation of literal
  - For integers, may want to convert early (to do *constant folding*)
  - For floats, may want to wait (for compilation to different machines). Why?

Slide courtesy: Milind Kulkarni

Symbol Table Implementation

Beyond Syntactic Analysis

- Till parsing phase:
  - **`INT x:=2;`** ≡ **`INT x:=5;`**
  - **`x:=y+100`** ≡ **`x:=y+100000000000000;`**
  - **`INT x, y;`**

    `y := (2.3 * 6);`

    `x := ( main );`

- Beyond parsing phase

  ...toward meaningful, executable programs

Example Micro Program

- Refer to the grammar in PA2 to know the programming constructs fully.

  - MicroProgram

Symbol Table

- A *symbol table* maintains
  - Symbolic names
  - Attributes of a name
    - E.g. type, scope, accessibility
- Used to manage declarations of symbols and their correct usage

Symbol Table – Names

For the sample program shown below identify all names (note: this is not a valid micro program)

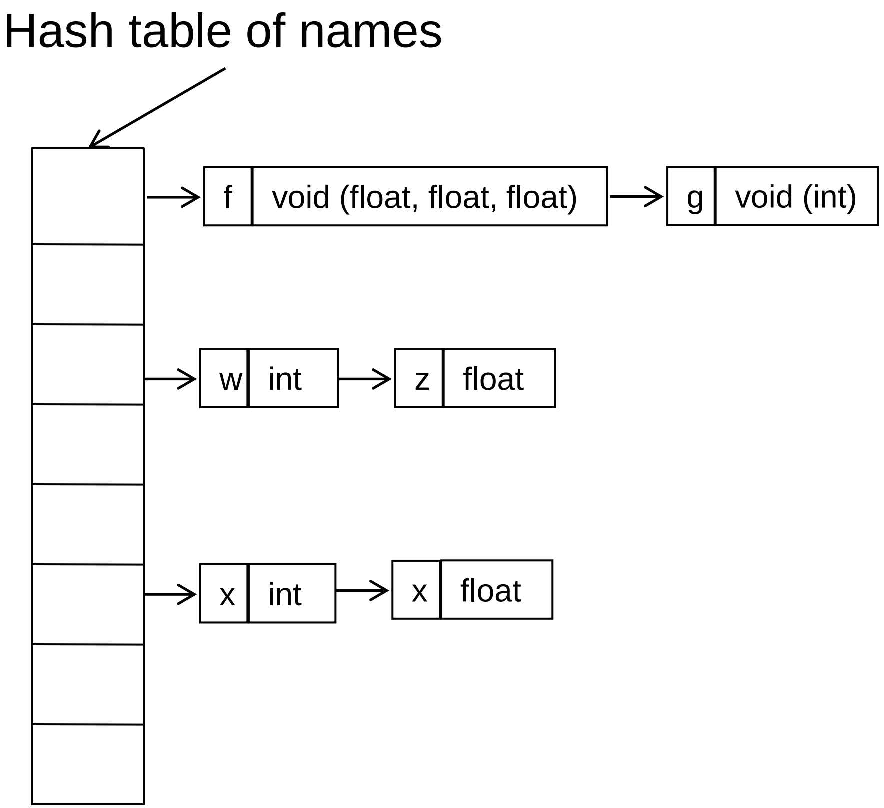
```
PROGRAM scope_test
BEGIN
#global declarations
FUNCTION void f(float, float, float)
FUNCTION void g(int)
{
    INT w, x;
    {
       FLOAT x, z;
       f(x, w, z);
    }
    g(x);
}

END
```

Symbol Table Implementation – High-level Requirements

- Should accommodate:
  - Efficient retrieval of names
  - Frequent insertion and deletion of names
- Should consider *scopes*

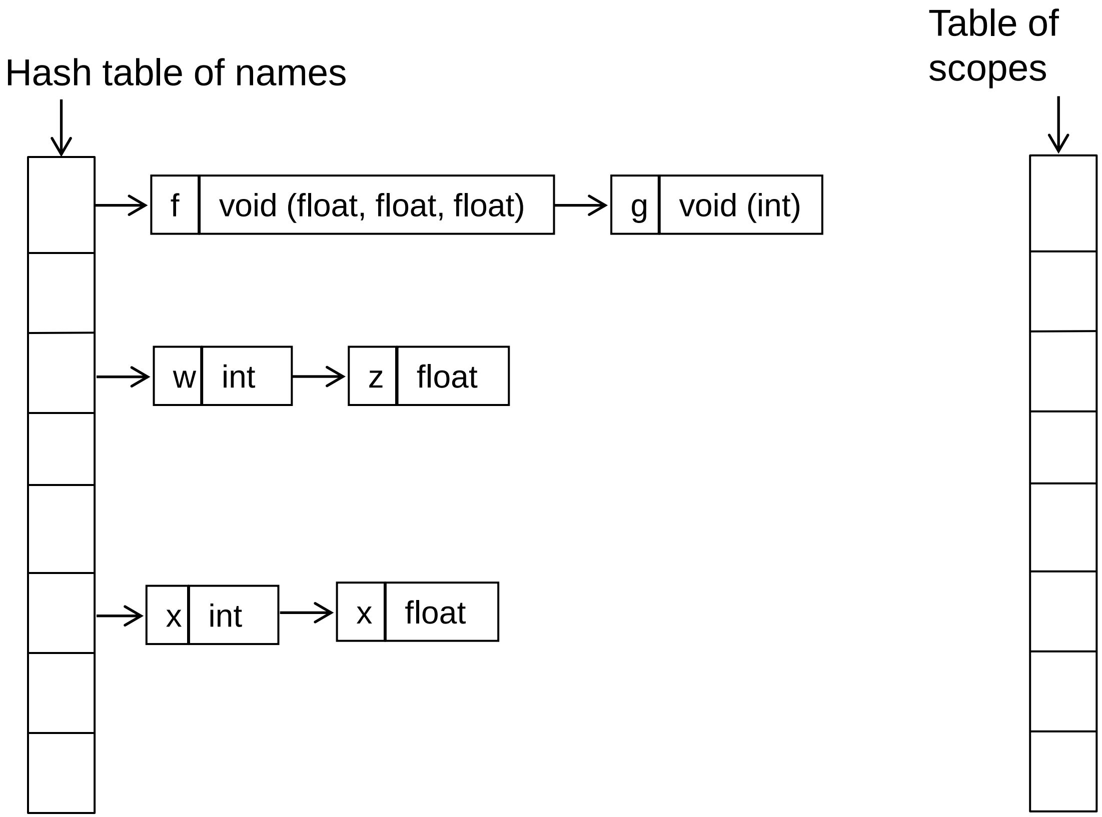
Symbol Table – an implementation




```
PROGRAM scope_test
BEGIN
#global declarations
FUNCTION void f(float, float, float)
FUNCTION void g(int)
{
     INT w, x;
     {
         FLOAT x, z;
         f(x, w, z);
     }
     g(x);
}

END
```

Symbol Table – an implementation

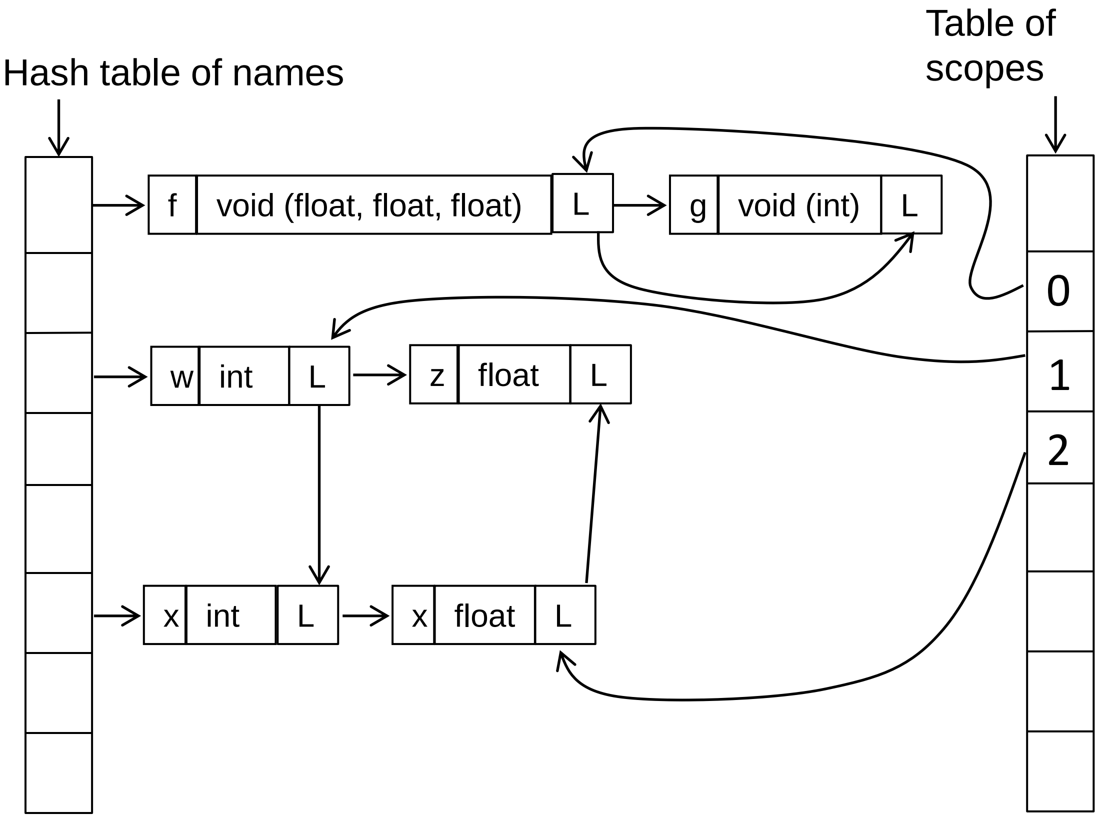



```
PROGRAM scope_test
BEGIN
#global declarations
FUNCTION void f(float, float, float)
FUNCTION void g(int)
{
     INT w, x;
     {
         FLOAT x, z;
         f(x, w, z);
     }
     g(x);
}

END
```

- be aware of current scope
- Be aware of all active scopes
- Chain names by their scope-levels

Symbol Table – an implementation

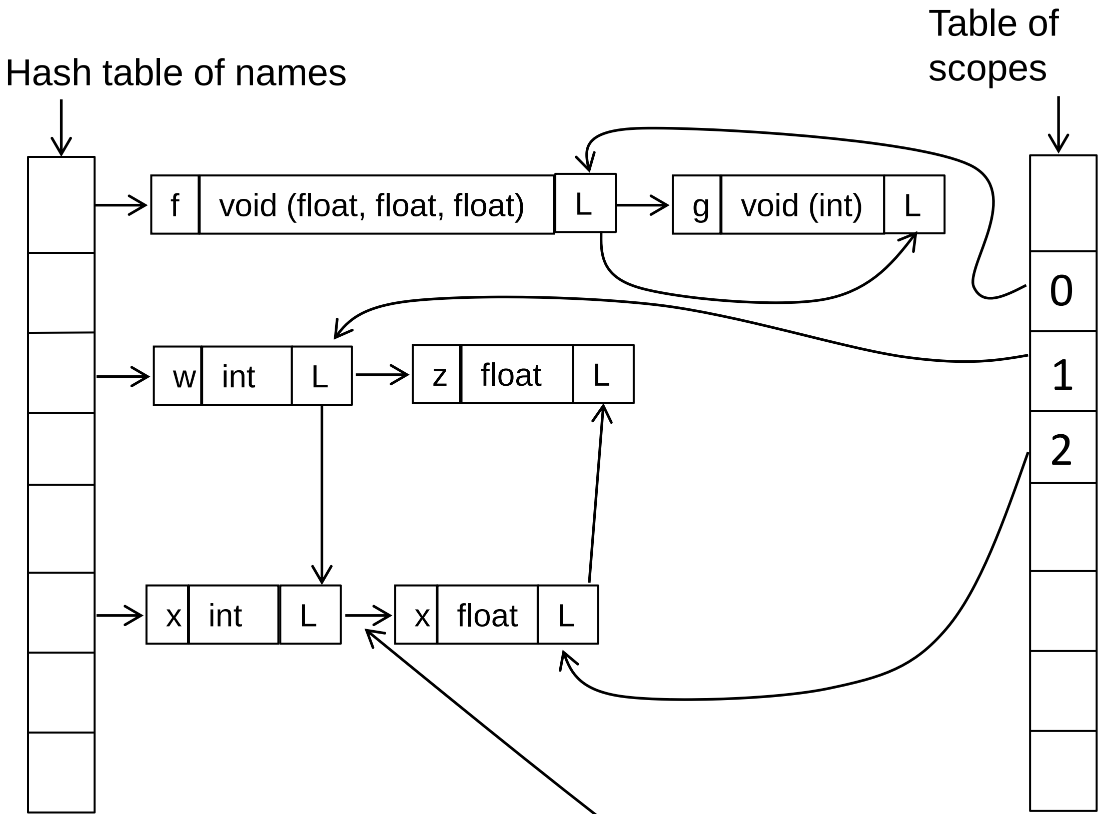



```
PROGRAM scope_test
BEGIN
#global declarations
FUNCTION void f(float, float, float)
FUNCTION void g(int)
{
        INT w, x;
        {
            FLOAT x, z;
            f(x, w, z);
        }
        g(x);
}

END
```

- Chain names by their scope-levels

Symbol Table – an implementation

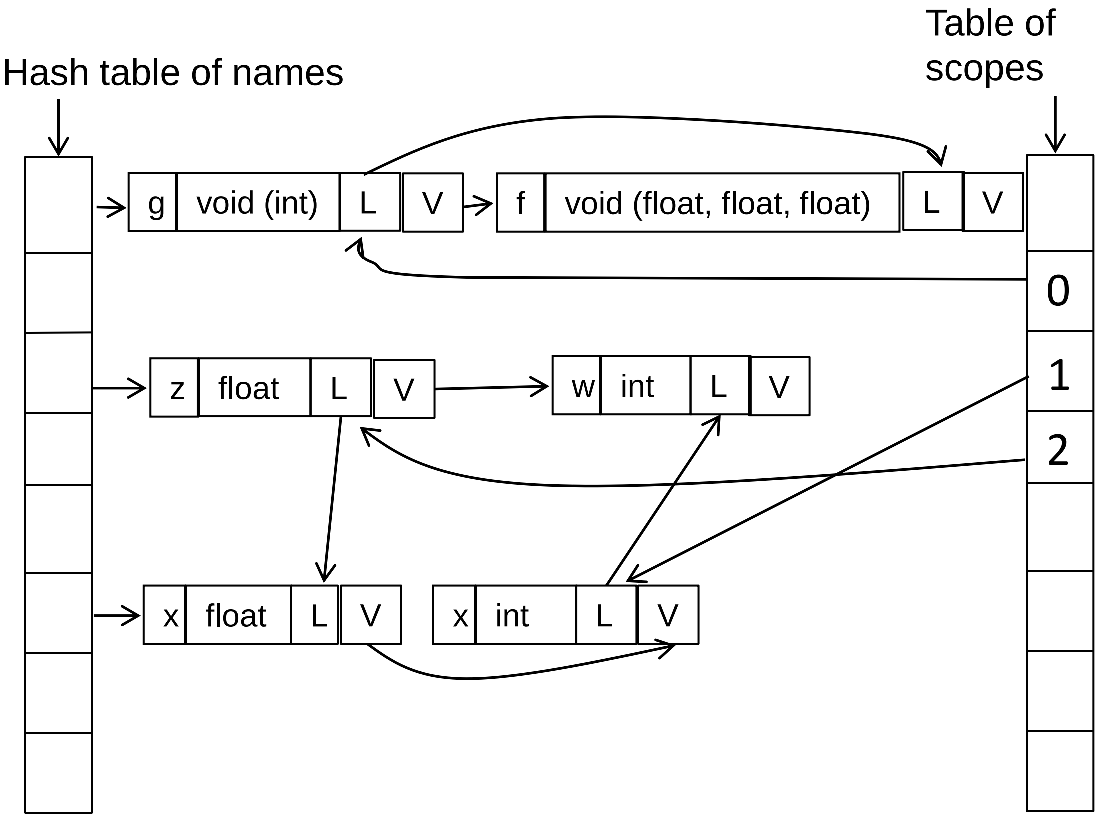



```
PROGRAM scope_test
BEGIN
#global declarations
FUNCTION void f(float, float, float)
FUNCTION void g(int)
{
     INT w, x;
     {
         FLOAT x, z;
         f(x, w, z);
     }
     g(x);
}

END
```

Abandon scope: remove references to all entries that exists at the current level.
Tip: make the hash chain a <u>doubly linked list</u>.

Symbol Table – an implementation



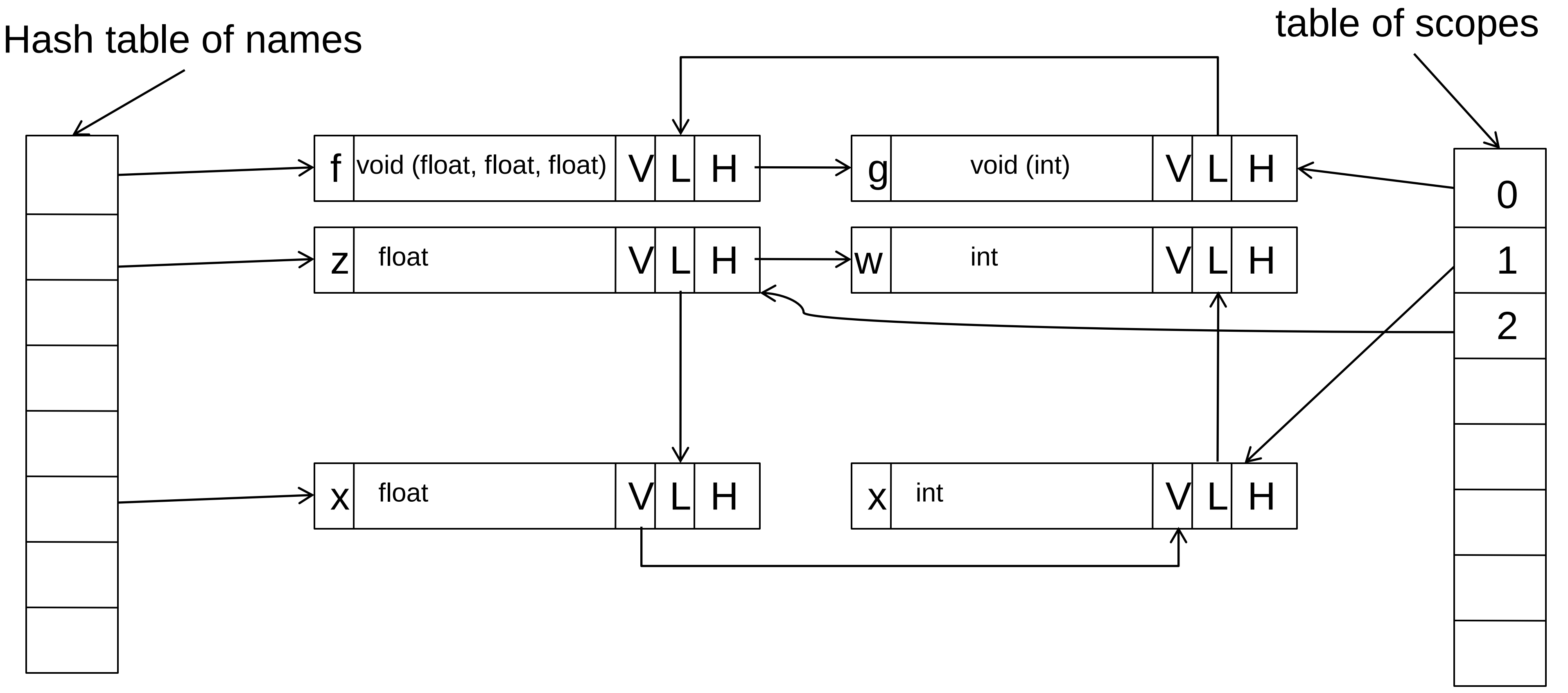

```
PROGRAM scope_test
BEGIN
#global declarations
FUNCTION void f(float, float, float)
FUNCTION void g(int)
{
     INT w, x;
     {
          FLOAT x, z;
          f(x, w, z);
     }
     g(x);
}

END
```

Notice the order of objects: “insert at the front of the list”

What if I want to access the integer x here?
Tip: maintain an ordered stack for each symbol name appearing in the current scope.

Symbol Table – an implementation



Expressions

- Three semantic actions needed
  - eval_binary (processes binary expressions)
    - Create AST node with two children, point to AST nodes created for left and right sides
  - eval_unary (processes unary expressions)
    - Create AST node with one child
  - process_op (determines type of operation)
    - Store operator in AST node

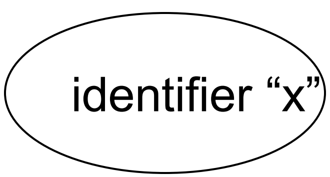
Slide courtesy: Milind Kulkarni

Expressions Example

x + y + 5

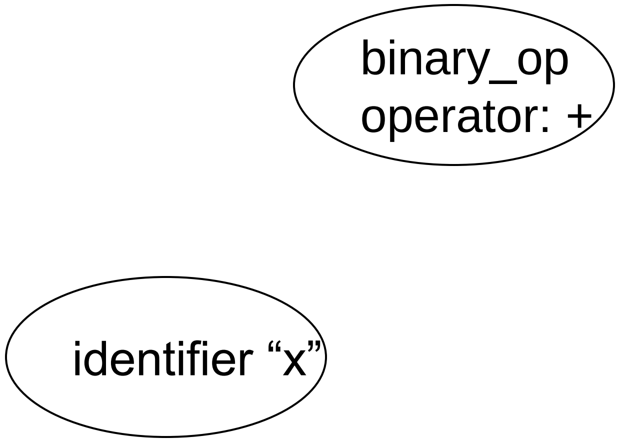
Expressions Example

$x + y + 5$



Expressions Example

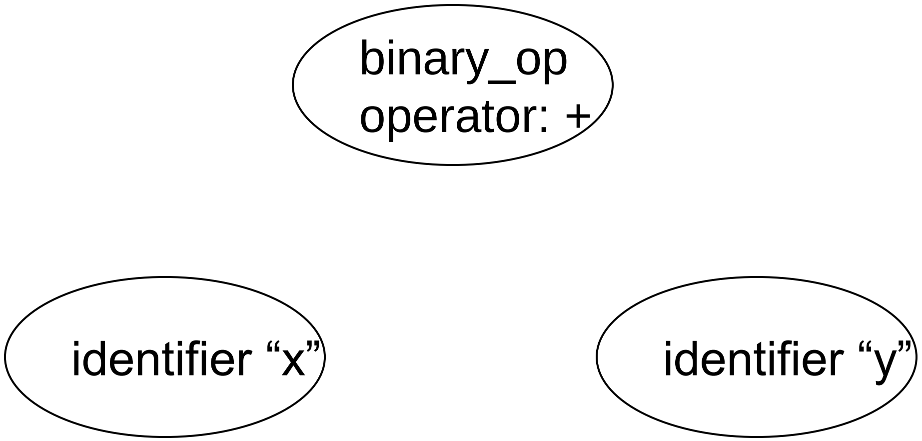
x + y + 5



Slide courtesy: Milind Kulkarni

Expressions Example

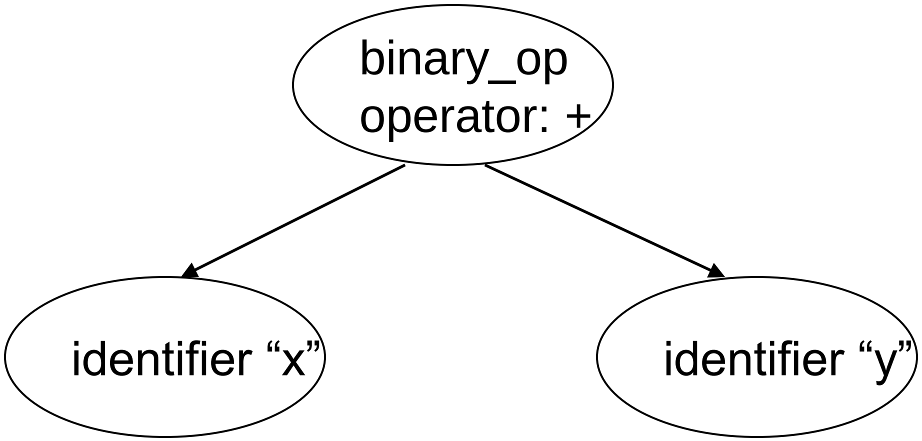
$x + y + 5$



Slide courtesy: Milind Kulkarni

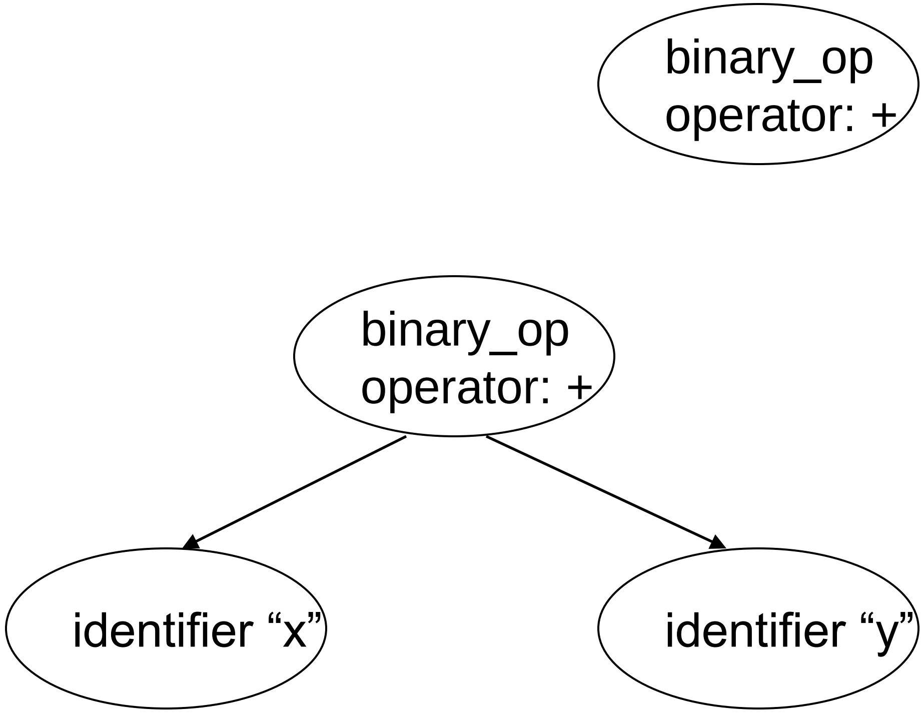
Expressions Example

x + y + 5



Expressions Example

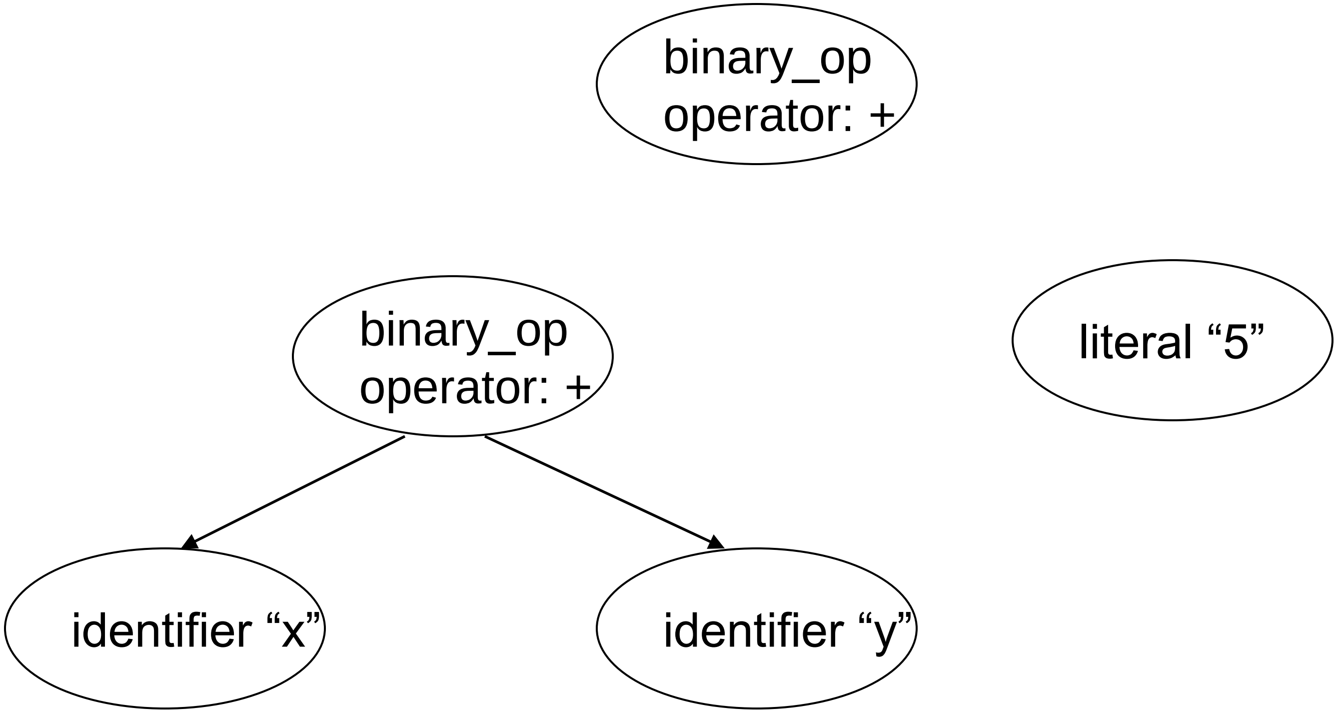
x + y + 5



Slide courtesy: Milind Kulkarni

Expressions Example

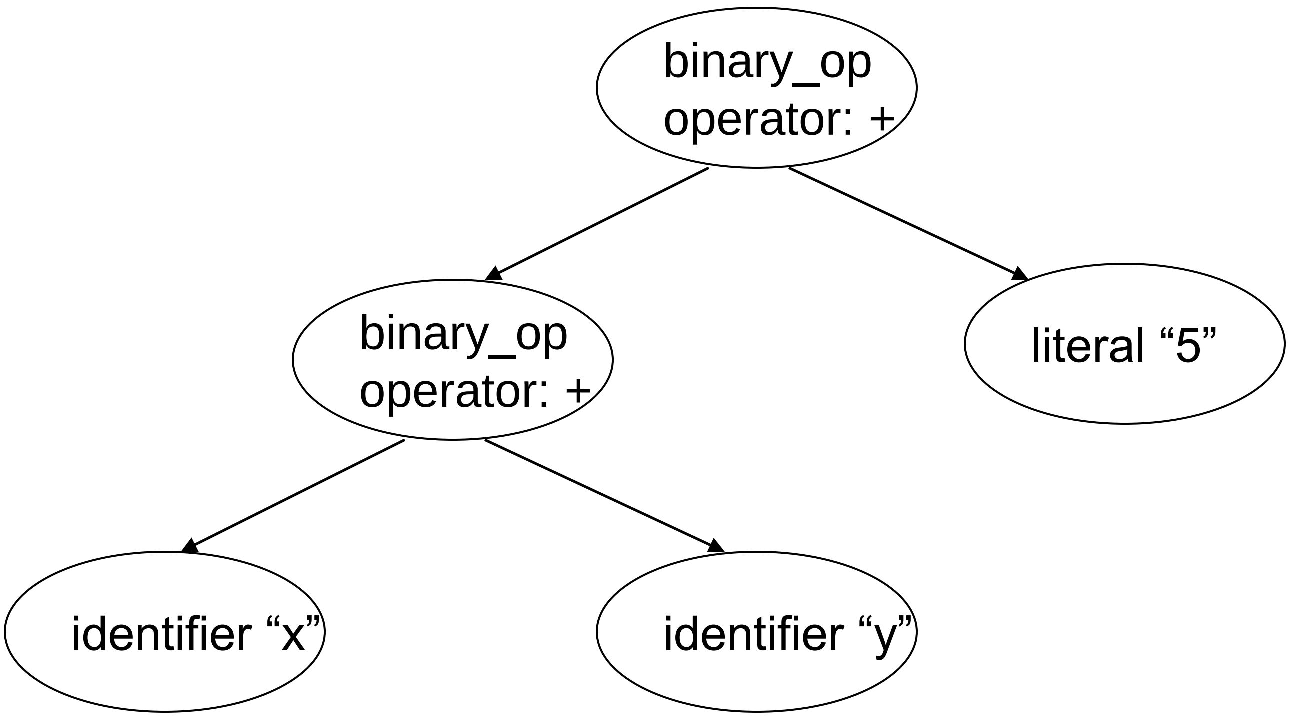
x + y + 5



Slide courtesy: Milind Kulkarni

Expressions Example

x + y + 5



Slide courtesy: Milind Kulkarni

Intermediate Representation

- Compilers need to synthesize code based on the ‘interpretation’ of the syntactic structure
- Code can be generated with the help of AST or can directly do it in semantic actions (recall: SDTs augment grammar rules with program fragments. Program fragments contain semantic actions.)
- Generated code can be directly executed on the machine or an intermediate form such as 3-address code can be produced.

3 Address Code (3AC)

- **What is it?** sequence of elementary program instructions
  - Linear in structure (no hierarchy) unlike AST
  - Format:

```
op A, B, C    //means C = A op B.
//op: ADDI, MULI, SUBF, DIVF, GOTO, STOREF etc.
```

  - E.g.

program text	**3-address code**

`INT x;`
`FLOAT y, z;`
`z:=x+y;`
	`ADDF x y T1`
	`STOREF T1 z`

`INT a, b, c, d;`
`d = a-b/c;`
	`DIVI b c T1`
	`SUBI a T1 T2`
	`STOREI T2 d`

```
Comments:
d = a-b/c; is broken into:
t1 = b/c;
t2 = a-t1;
d = t2;
```

3 Address Code (3AC)

- **Why is it needed?** To perform *significant* optimizations such as:
  - common sub-expression elimination
  - statically analyze possible values that a variable can take etc.

  How?

  Break the long sequence of instructions into “basic blocks” and operate on/analyze a graph of basic blocks