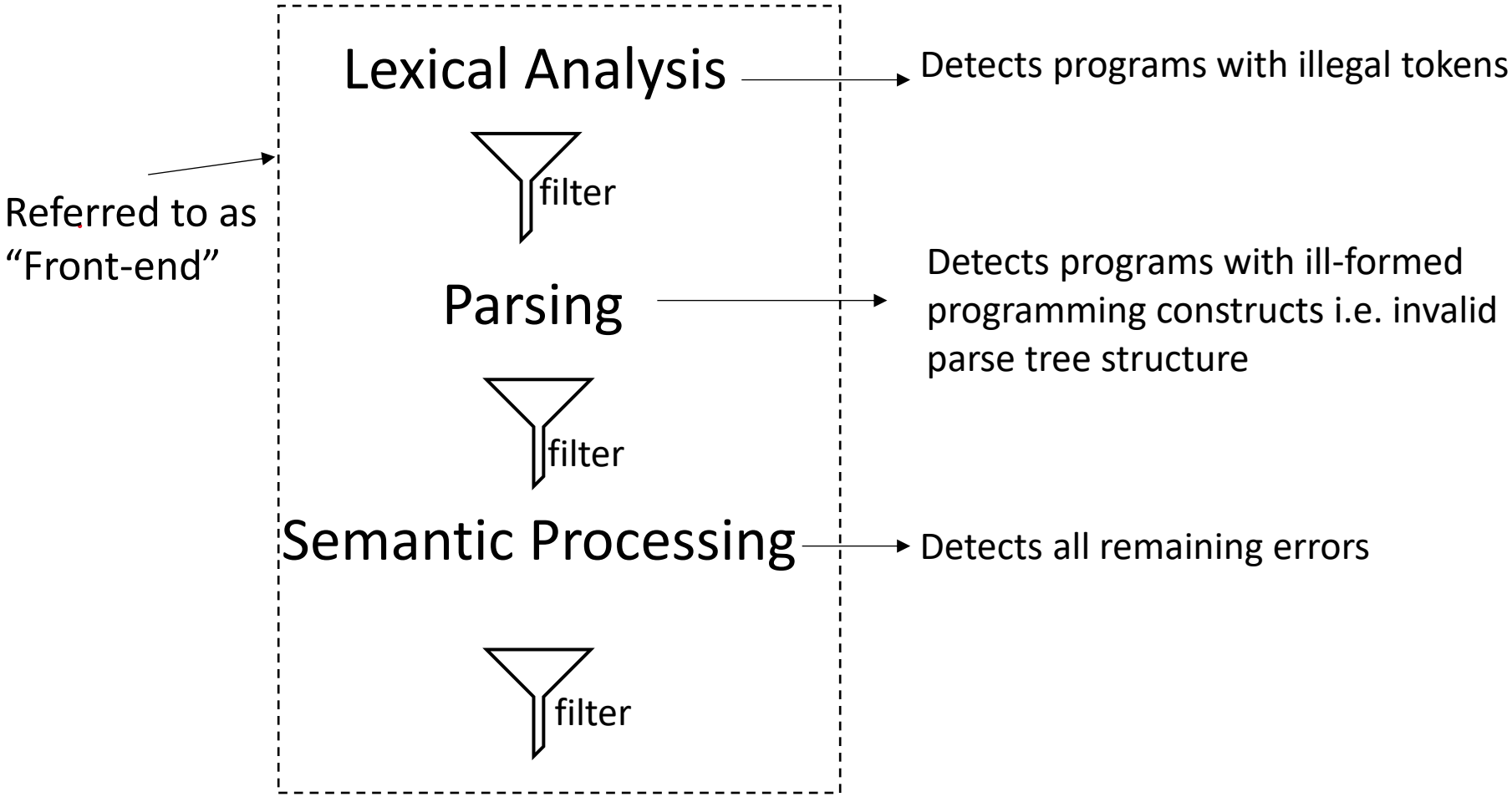
CS620T: Compilers - Principles and Implementation

Autumn 2026

Week 4: Semantic processing

Semantic Processing



Semantic Processing

- Syntax-directed / syntax-driven
  - Routines (called as semantic routines) interpret the meaning of programming constructs based on the syntactic structure
  - Routines play a dual role
    - Analysis – Semantic analysis
      - undefined vars, undefined types, uninitialized variables, type errors that can be caught at compile time, unreachable code, etc.
    - Synthesis – Generation of intermediate code
      - 3 address code
  - Routines create semantic records 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.
  - …

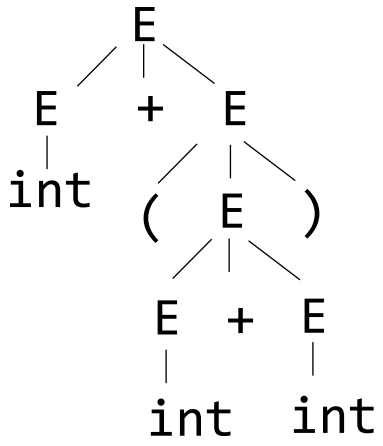
Abstract Syntax Tree

- 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
```

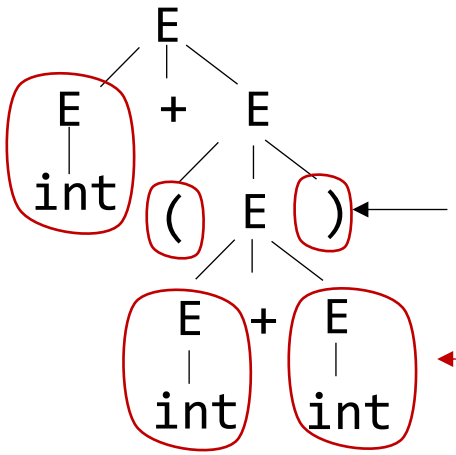
The parse tree for 1+(2+3)



AST - Example

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

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

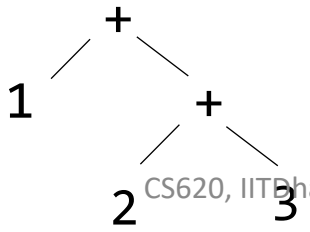


Expresses associativity.
Lower subtree in the hierarchy can express.

Single child.
Can compress.

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

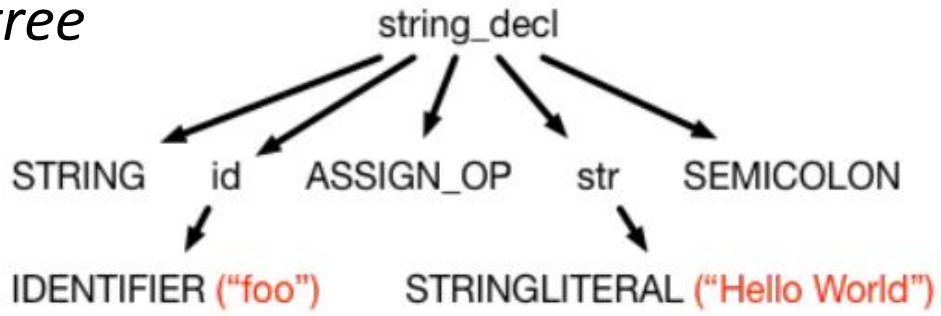
AST for `1+(2+3)` :



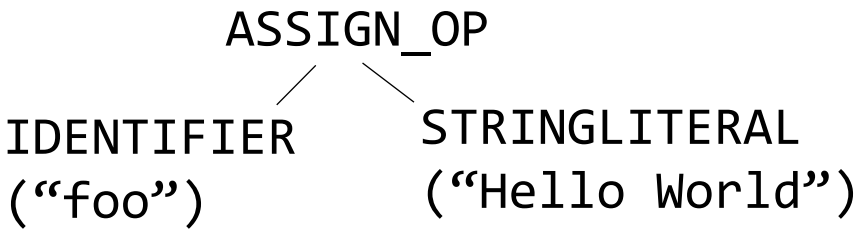
AST - Example

```
STRING foo := “Hello World” ;
```

parse tree ⇨



≡ *AST* ⇩



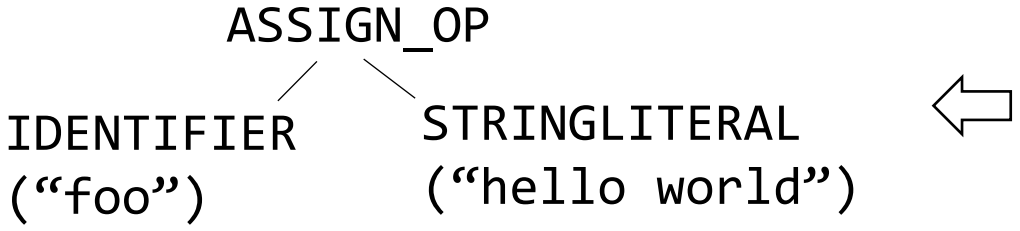
Semantic Analysis – Example

- 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



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 attributes 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

- 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

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

Call stack

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

Parse tree

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

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	Parse tree
E()	E
E1()	|
IsTerm()	INTLITERAL

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

Call stack

E()
E1()
IsTerm()

Parse tree

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	Parse tree
E()	E
E1()	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	Parse tree
E()	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	Parse tree
E()	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
```

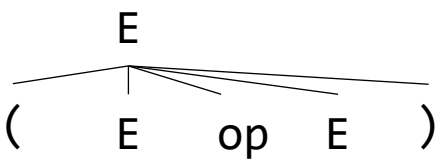
Input string: (2+3) next token

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)

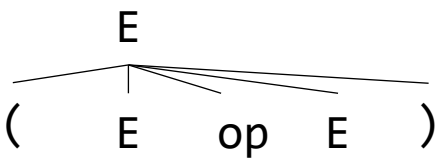
next token

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)

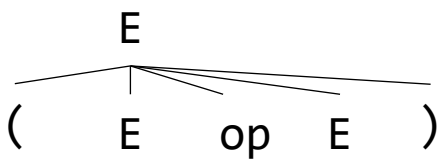
next token

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)

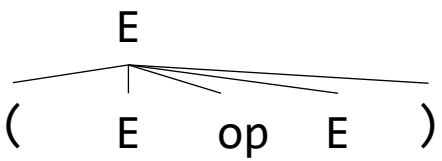
next token

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)

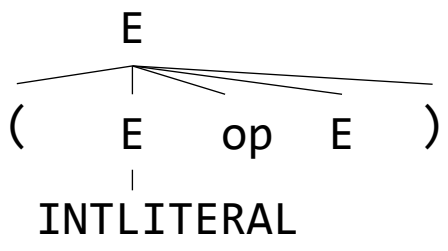
next token ↓

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)

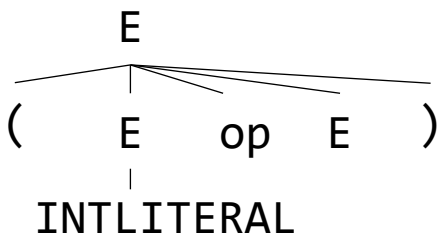
next token

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

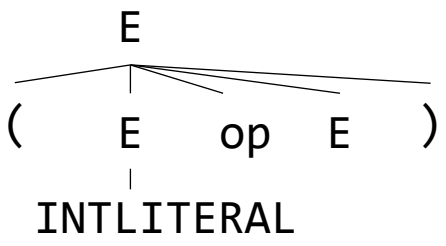
Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

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



"2"

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

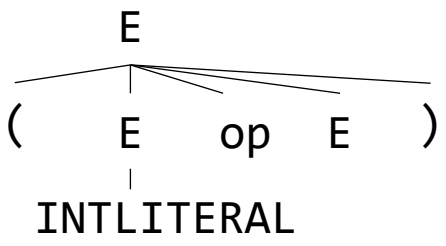
Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

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

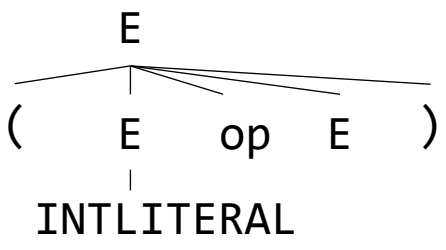
Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

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

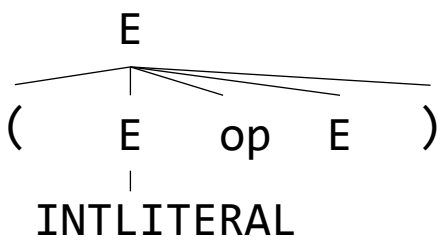
Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

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)

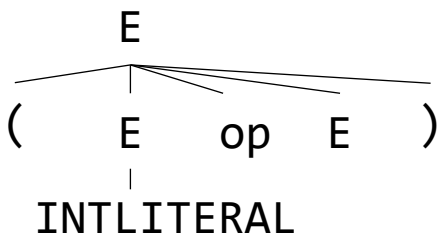
next token

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

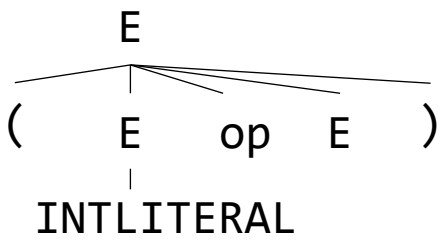
Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

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)

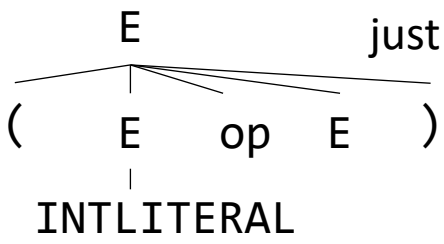
next token

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

Call stack

E()
E2()
Op()

Parse tree



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)

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)

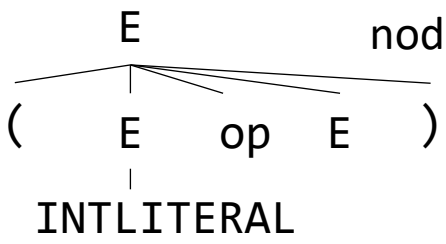
next token

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

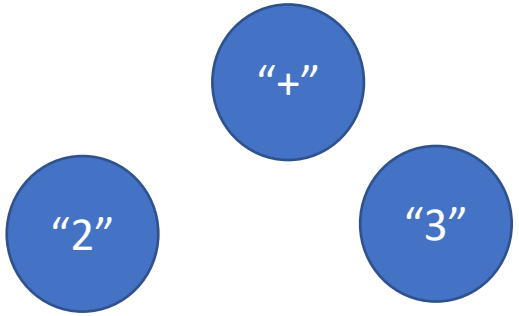
Call stack

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)

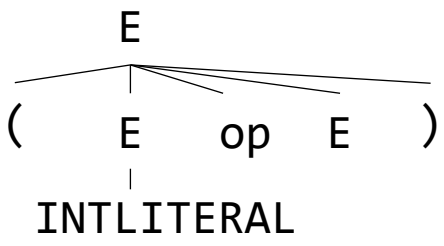
next token

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

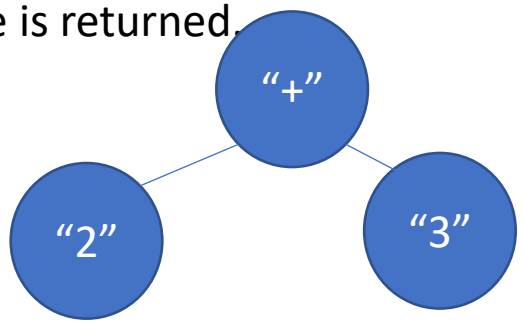
Call stack

E()
E2()

Parse tree



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.



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

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

```
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

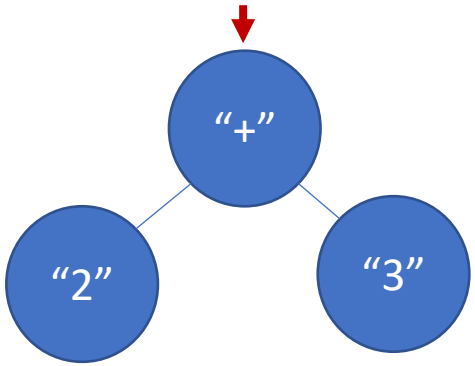
- What did we do when we saw an E (parenthesized expression)?
  - Create an AST node with two children. The node contains the binary operator OP stored as a string. Children point to roots of subtrees representing E.

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, E1.node, E2.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

Demo – AST Construction in Python

- Example input Python program

```
import numpy as np
def my_matmul(A: list[complex], B: list[complex]):
    C=np.matmul(A,B)
    print(C)

A: list[complex]=[[1,2,3],[4,5,6]]
B: list[complex]=[[1,4],[2,5],[3,6]]
my_matmul(A,B)
```

Observe:

- Arguments are type annotated
- Assignment statements are type annotated

Demo – AST Construction in Python

- Types of top-level Python AST nodes

Observe:

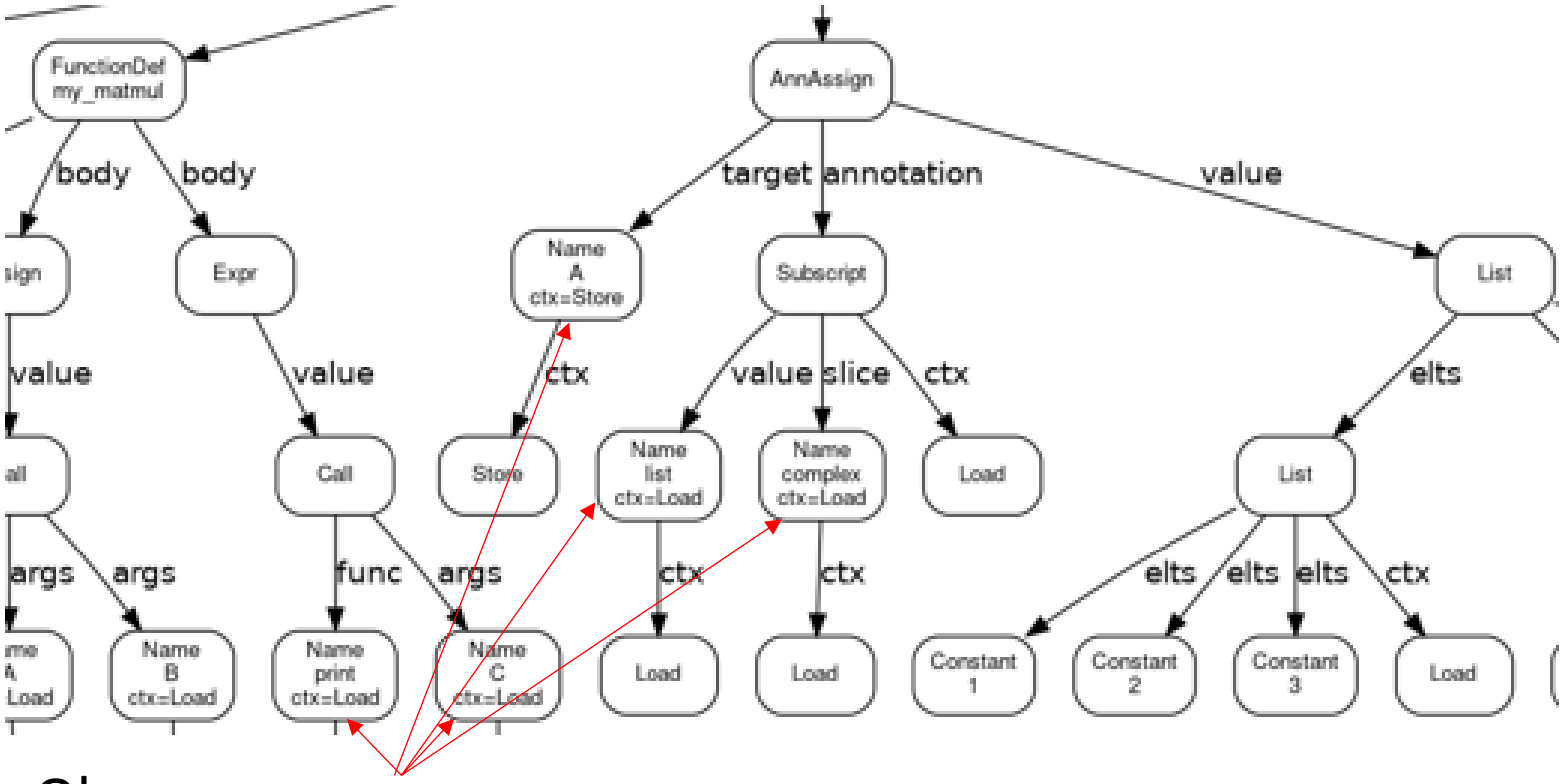
- FuncDef corresponds to function definition
- AnnAssign correspond to annotated assignments
- Expr corresponds to function call, which is an expression
    - To dump the parse tree on the terminal refer to python_parse.py
    - To save the parse tree in a png file, refer to pythonast_to_png.py

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 ← Clang AST nodes follow this approach
  - Note: may want to directly store type information in AST (or could look up in symbol table each time)

Slide courtesy: Milind Kulkarni

Demo – AST Construction in Python



Observe:

- AST nodes of type ‘Name’ have values like “A”, “C”, “B”, “list”, “complex” printed. These are **attributes** of an AST node.
- An additional attribute is context: can be “load” or “store” for identifiers.

L-values and R-values

- Need to distinguish between meaning of identifiers appearing on RHS and LHS of an assignment statement. So additional context is necessary.

```
i := 5;
i := i + 1;
```
} //RHS specifies data that is computed/read.
LHS specifies address where data is stored.

- **L-values**: addresses which can be loaded from or stored into
- **R-values**: data often loaded from address
  - Expressions produce R-values
- Assignment statements: `L-value := R-value;`

```
a := a;
```

a refers to memory location named a. We are storing into that memory location (L-value)

a refers to data stored in the memory location named a. We are loading from that memory location to produce R-value

Address-of operator (in C/C++) and L-value

- Consider:

```
a := &b;  //& is address-of operator. R-value
of a is set to L-value of b.
//expression on the RHS produces data that is
an address of a memory location.

Recall: L-Value = address which can be loaded
from or stored into, R-Value = data (often)
loaded from addresses.
```

Take L-value of b, don't load from it, treat it as an R-value and store the resulting data in a temporary

Dereference operator

- Consider:

```
*a := b;  //* is dereference operator. R-value
of a is set to R-value of b.
//expression on the LHS produces data that is
an address of a memory location.
```

a appearing on LHS is loaded from to produce R-value. That R-value is treated as an address that can be stored into.

Take R-value of a, treat it as an L-value (address of a memory location) and then store RHS data

*Summary: pointer operations & and * mess with meaning of L-value and R-values*

Observations

- Identifiers appearing on LHS are (normally) treated as L-values. Appearing on RHS are treated as R-values.
  - So, when you are visiting an `id` node in an AST, you cannot generate code (load-from or store-into) until you have seen how that identifier is used. => until you visit the parent.
    - So, be conservative and mark it as “store-into” type of node.
- Temporaries are needed to store result of current expression

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