CS406: Compilers

Spring 2022

Week 6: Semantic Processing: AST construction, Intermediate Code Generation

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

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
|
INTLITERAL

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

<u>Call stack</u>

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

<u>Parse tree</u>

```
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

E

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

```
    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)

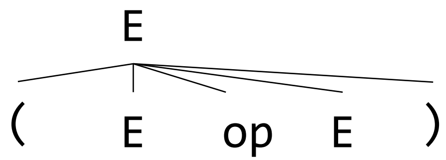
next token

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)

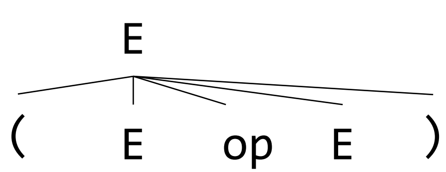
next token

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)

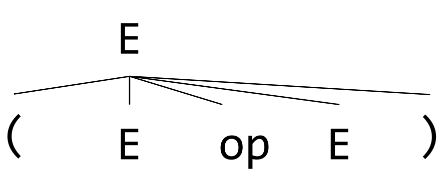
next token

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)

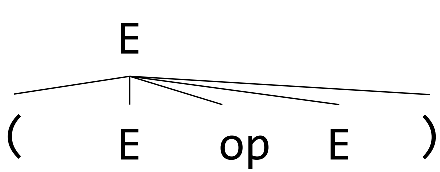
next token

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)

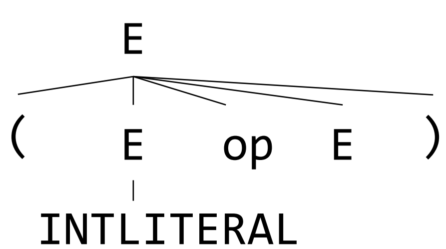
next token

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)

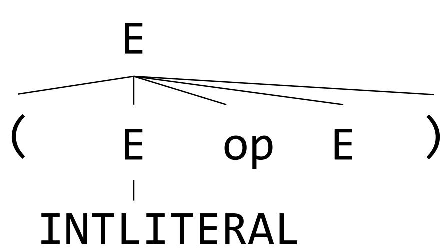
next token

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

<u>Call stack</u>

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

<u>Parse tree</u>



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)

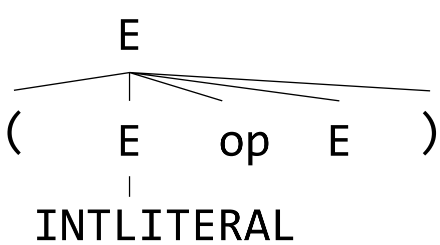
next token

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

Call stack

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

Parse tree



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.

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

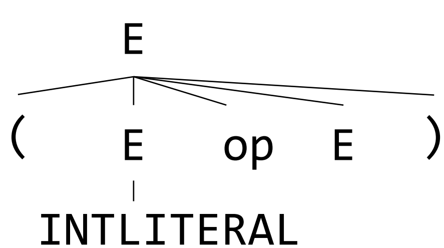
Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

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

Call stack

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

Parse tree



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

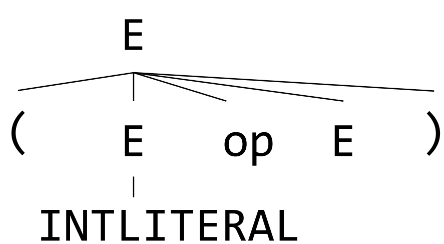
Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

E1 returns the pointer to the tree node.

<u>Call stack</u>

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

<u>Parse tree</u>



“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

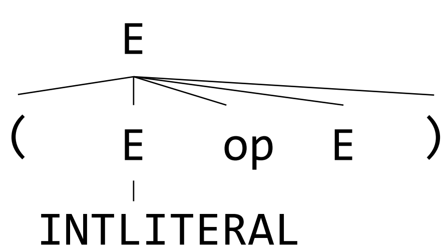
Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

E returns the pointer to the tree node.

Call stack

E()
E2()
E()

Parse tree



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

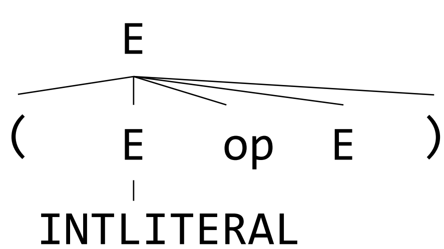
Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

E2() 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.

Call stack

E()
E2()

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

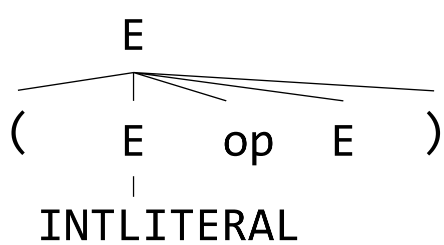
Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

E2() calls Op()

Call stack

E()
E2()

Parse tree



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

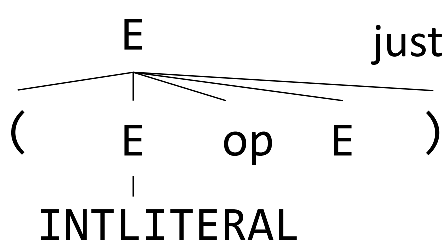
Sequence of tokens given by scanner: LPAREN INTLITERAL ADD INTLITERAL RPAREN

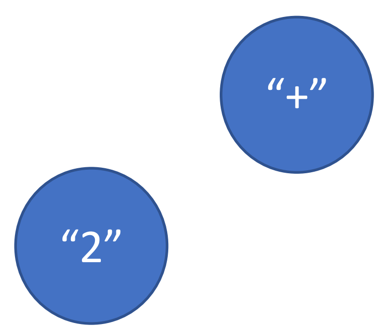
Op() first matches the next token with ADD and creates a node with value '+'. It then returns a pointer to the tree node just created. (note the next token pointer is also advanced)

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)

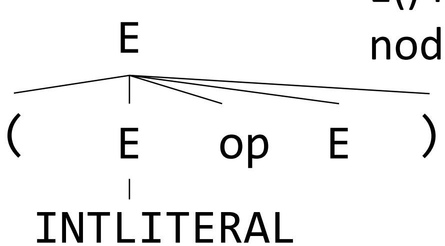
next token

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

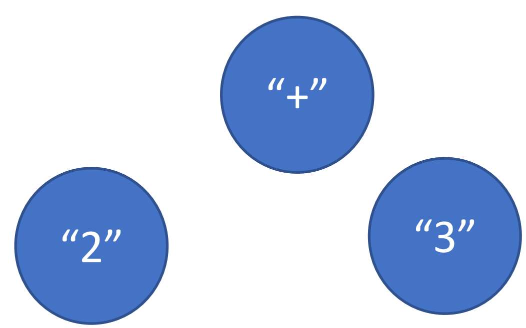
Call stack

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)

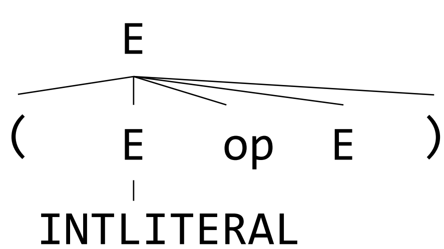
next token

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

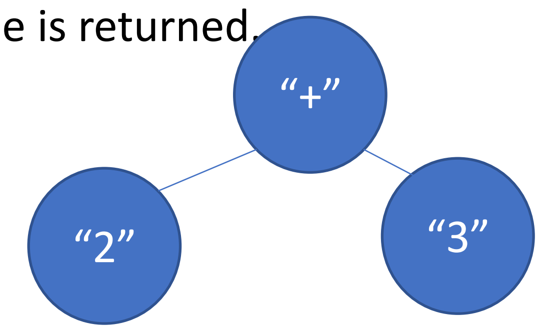
Call stack

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
```

$\xrightarrow{\text{triggers}}$

```
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

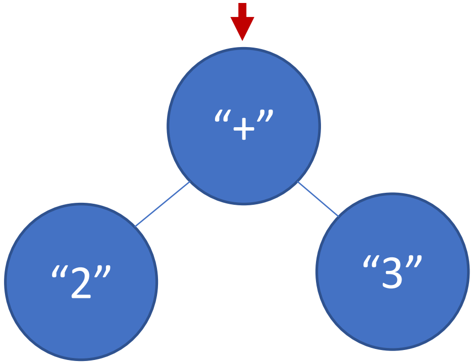
- What did we do when we saw an E (parenthesized expression)?
  - Create an AST node with two children. The node contains the binary operator OP stored as a string. Children point to roots of subtrees representing E.

E -> (E op E) $\xrightarrow{\text{triggers}}$

```
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 <u>notations augmenting grammar rules</u>

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)

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?

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

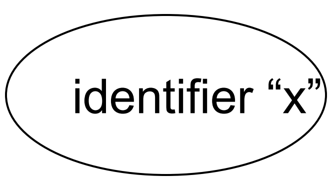
Slide courtesy: Milind Kulkarni

Expressions Example

```
x + y + 5
```

Expressions Example

x + y + 5



Slide courtesy: Milind Kulkarni

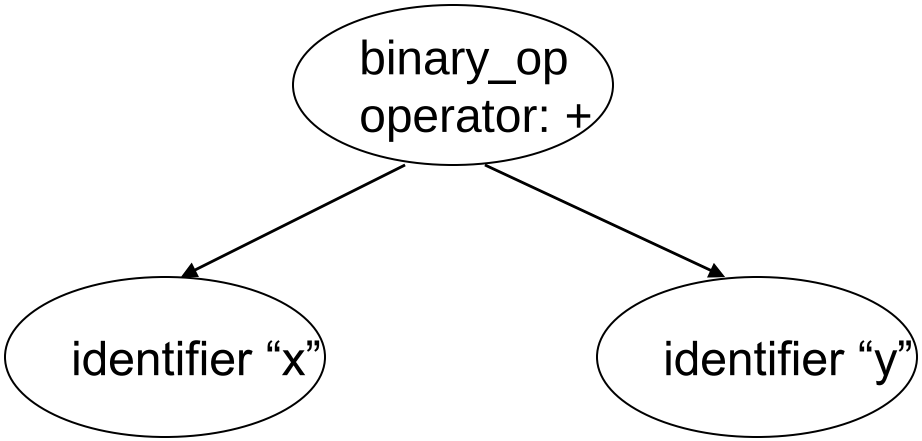
Expressions Example

x + y + 5

Slide courtesy: Milind Kulkarni

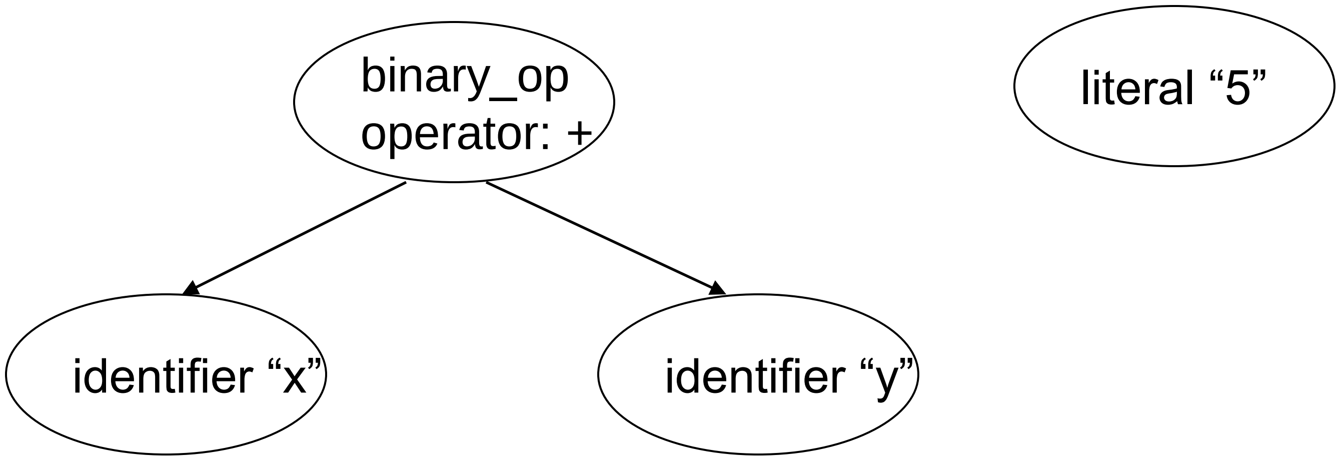
Expressions Example

x + y + 5



Expressions Example

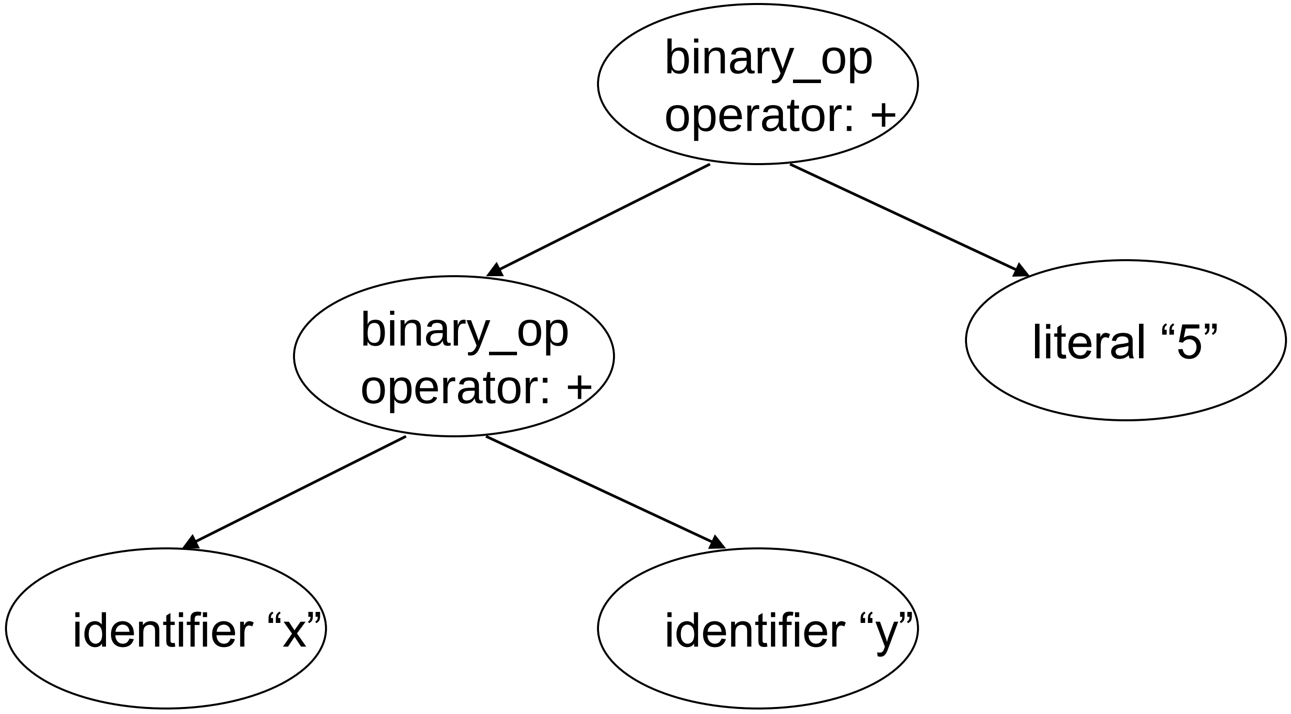
x + y + 5



Slide courtesy: Milind Kulkarni

Expressions Example

x + y + 5



Slide courtesy: Milind Kulkarni

Short Quiz

- https://forms.gle/3BJW3eYeBuwqkBLH7

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;
```

Suggested Reading

- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D.Ullman: Compilers: Principles, Techniques, and Tools, 2/E, AddisonWesley 2007
  - Chapter 2 (2.3, 2.5, 2.7, 2.8), Chapter 4 (4.6), Chapter 5 (5.1, 5.2.3, 5.2.4, 5.4), Chapter 6(6.2)
- Fisher and LeBlanc: Crafting a Compiler with C
  - Chapter 6 (6.2-6.4), Chapter 7 (7.1, 7.3), Chapter 8 (8.2, 8.3)

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

3 Address Code (3AC)

- **How is it generated?** Choices available:
  1. Do a post-order walk of AST
  - Generate/Emit code as a string/data_object when you visit a node
  - Pass the code to the parent node

    Parent generates code for self after the code for children is generated. The generated code is appended to code passed by children and passed up the tree

```
data_object generate_code() {
    //preprocessing code
    data_object lcode=left.generate_code();
    data_object rcode=right.generate_code();
    return generate_self(lcode, rcode);
}
```

  2. Can generate directly in semantic routines or after building AST

3 Address Code (3AC)

- Generating 3AC directly in semantic routines.

```
INT x;
x:=3*4+5+6+7;
```

```
MULI 3 4 T1
ADDI T1 5 T2
ADDI T2 6 T3
ADDI T3 7 T4
STOREI T4 x
```

Comments:

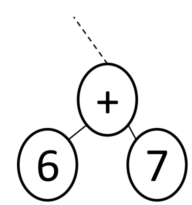
```
x = 3*4+5+6+7 is broken into:
t1 = 3*4;
t2 = 5+t1;
t3 = 6+t2;
t4 = 7+t3;
x = t4
```

- Walk the AST in post-order and infer at an internal node (labelled op) that it computes a constant expression

```
INT x;
x:=3*4+5+6+7;
```

```
STOREI 30 x
```

Comments:



L-values and R-values

- Need to distinguish between meaning of identifiers appearing on RHS and LHS of an assignment statement

```
i := 5;          //RHS specifies data that is computed/read.
i := i + 1;      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

Temporaries

- Earlier saw the use of temporaries e.g.

```
INT x;
FLOAT y, z;
z:=x+y;
```

```
ADDF x y T1
STOREF T1 z
```

- Think of them as unlimited pool of registers with memory to be allocated later
- Optionally declare them in 3AC. Name should be unique and should not appear in program text

```
INT x
FLOAT y z T1
ADDF x y T1
STOREF T1 z
```

- Temporary can hold L-value or R-value

Temporaries and L-value

- Yes, a temporary can hold 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.
- Temporaries are needed to store result of current expression
- a `data_object` should store:
  - Code
  - L-value or R-Value or constant
  - Temporary storing the result of the expression

Simple cases

- Generating code for constants/literals
    - Store constant in temporary
    - Optional: pass up flag specifying this is a constant
- Generating code for identifiers
    - Generated code depends on whether identifier is used as L-value or R-value
        - Is this an address? Or data?
    - One solution: just pass identifier up to next level
        - Mark it as an L-value (it's not yet data!)
        - Generate code once we see how variable is used

Slide courtesy: Milind Kulkarni

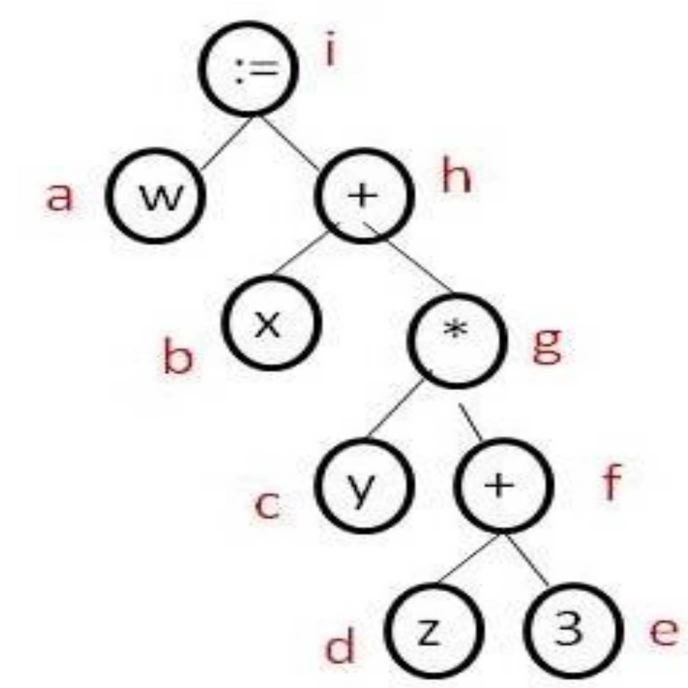
Generating code for expressions

- Create a new temporary for result of expression
- Examine data-objects from subtrees
- If temporaries are L-values, load data from them into new temporaries
  - Generate code to perform operation
  - In project, no need to explicitly load (variables can be operands)
- If temporaries are constant, can perform operation immediately
  - No need to perform code generation!
- Store result in new temporary
  - Is this an L-value or an R-value?
- Return code for entire expression

Example - assignment statement

AST for

`w:=x+y*(z+3);`



Visit Node a:

Temp: w

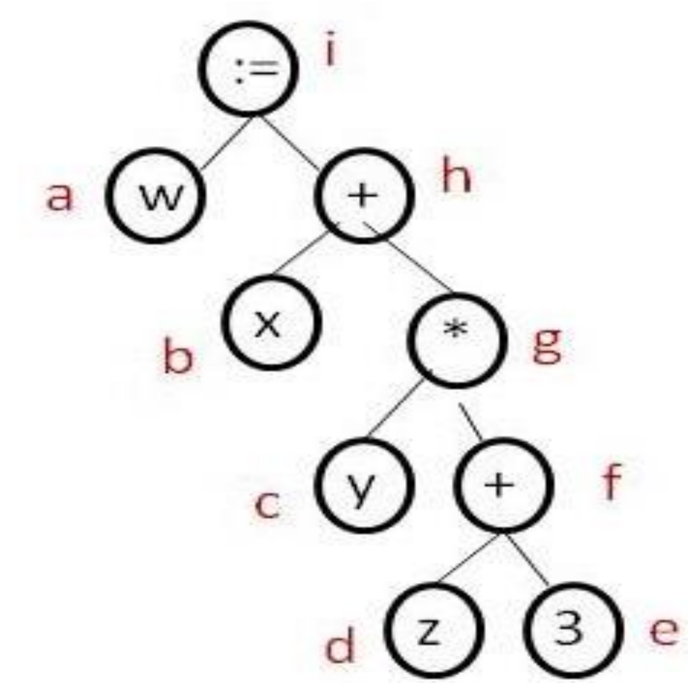
Type: l-value

Code: --

Example - assignment statement

AST for

`w:=x+y*(z+3);`



Visit Node b:

Temp: x

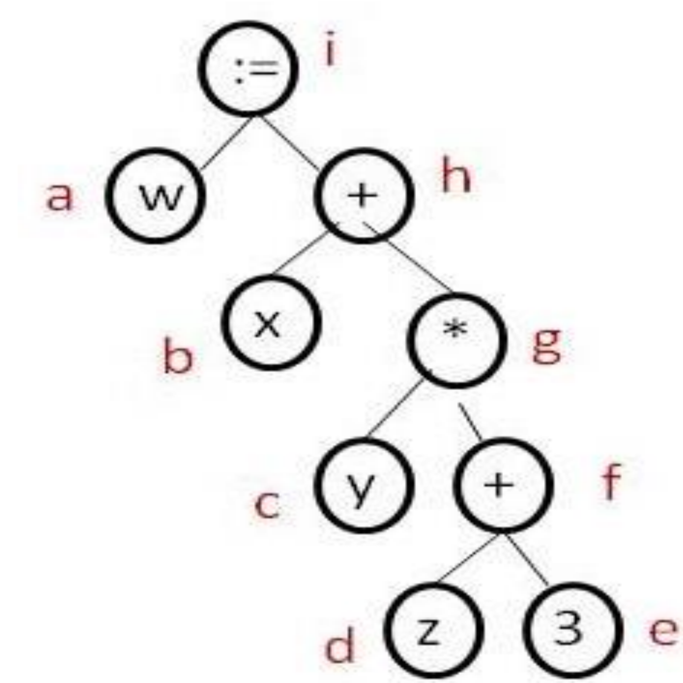
Type: l-value

Code: --

Example - assignment statement

AST for

`w:=x+y*(z+3);`



Visit Node c:

Temp: y

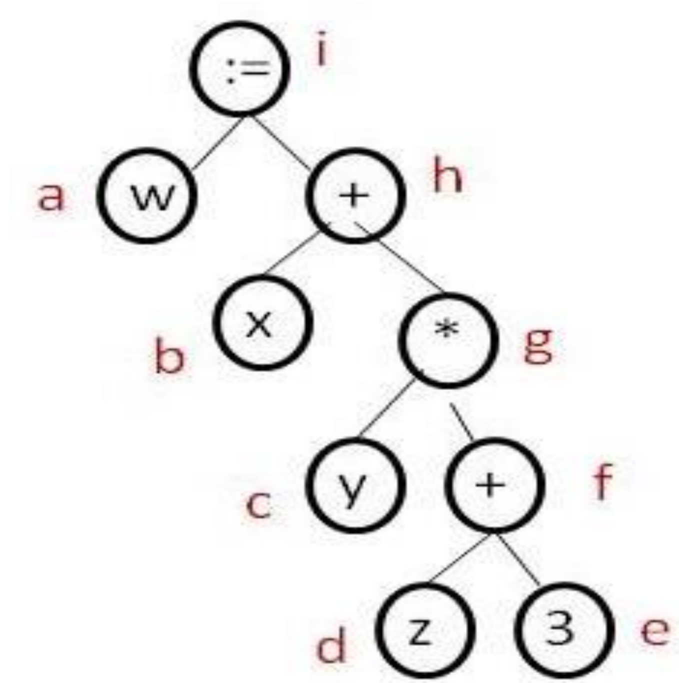
Type: l-value

Code: --

Example - assignment statement

AST for

`w:=x+y*(z+3);`



Visit Node d:

Temp: z

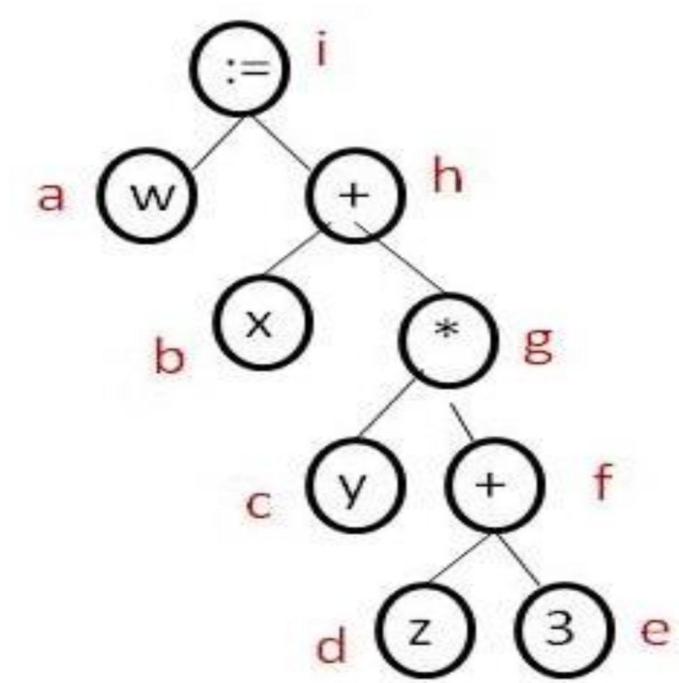
Type: l-value

Code: --

Example - assignment statement

AST for

`w:=x+y*(z+3);`



Visit Node e:

Temp: 3

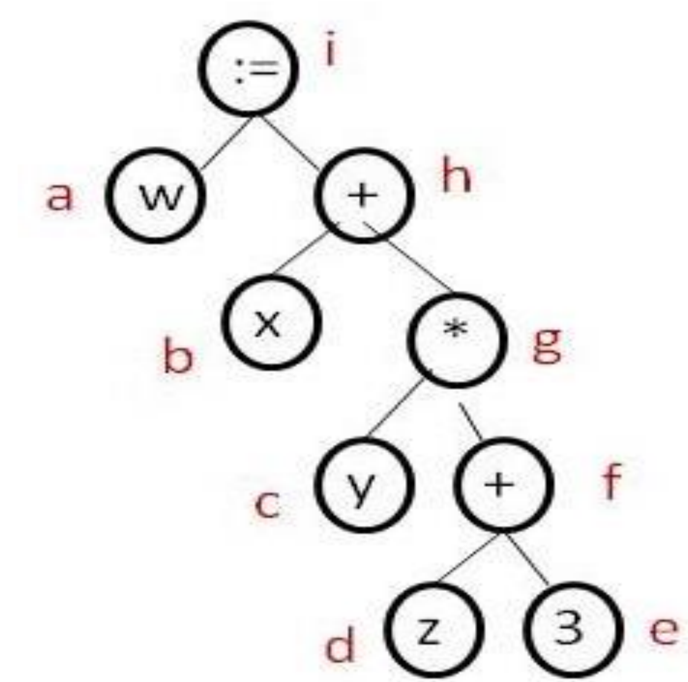
Type: constant

Code: --

Example - assignment statement

AST for

`w:=x+y*(z+3);`



Visit Node f:

Temp: T1

Type: R-value

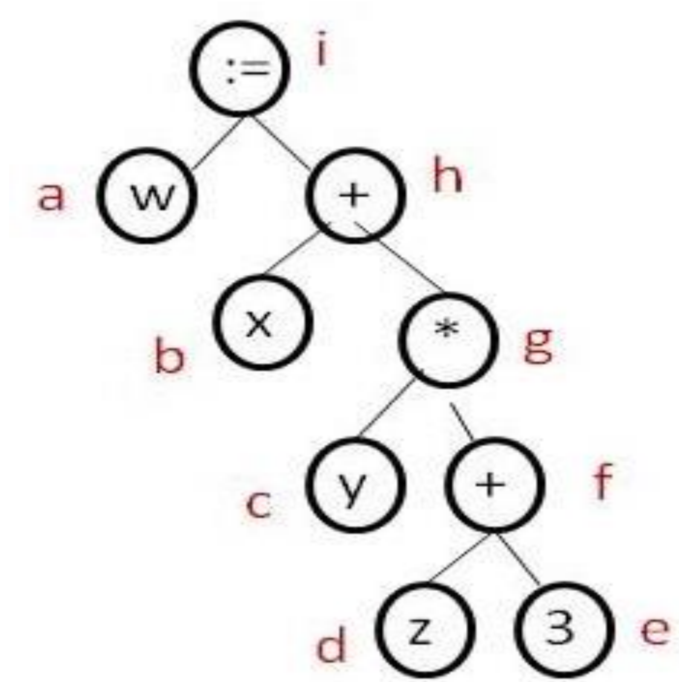
Code:

```
LD z T2

ADD T2 3 T1
```

Example - assignment statement

AST for

`w:=x+y*(z+3);`



Visit Node g:

Temp: T3

Type: R-value

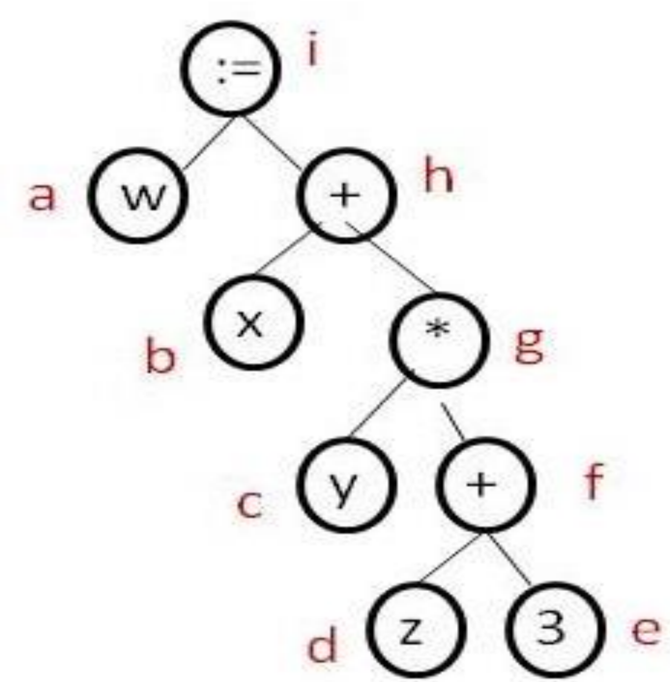
Code:

```
LD y T4
LD z T2
ADD T2 3 T1
MUL T4 T1 T3
```

Example - assignment statement

AST for

`w:=x+y*(z+3);`



Visit Node h:

Temp: T5

Type: R-value

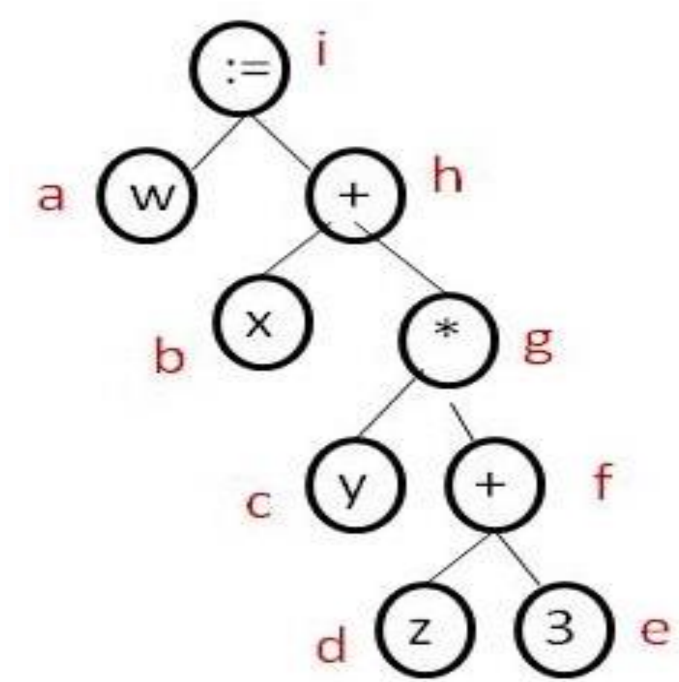
Code:

```
LD x T6
LD y T4
LD z T2

ADD T2 3 T1

MUL T4 T1 T3
ADD T6 T4 T5
```

Example - assignment statement

AST for

`w:=x+y*(z+3);`



Visit Node i:

Temp: NA

Type: NA

Code:

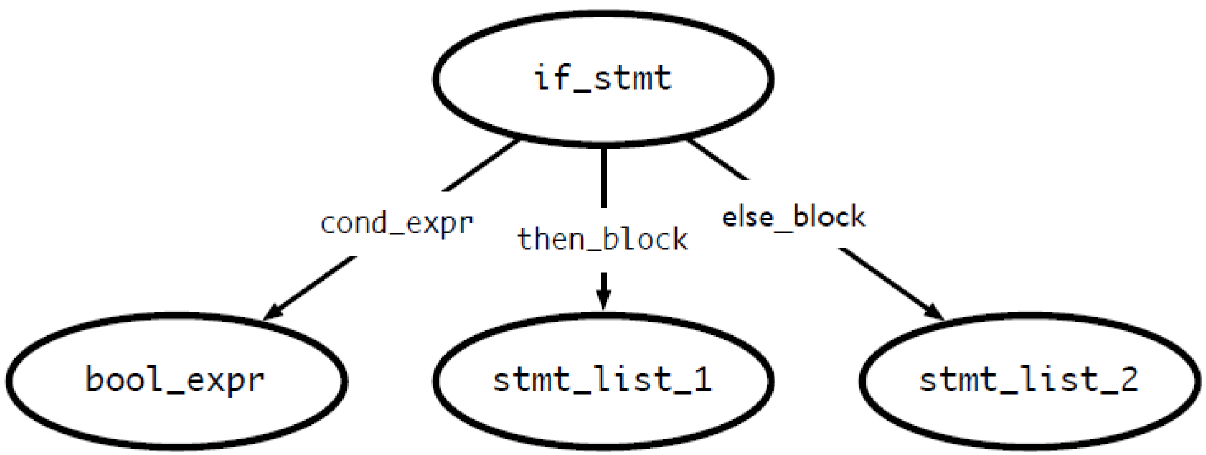
```
LD x T6
LD y T4
LD z T2

ADD T2 3 T1

MUL T4 T1 T3
ADD T6 T4 T5
ST T5 w
```

If statements

```
if <bool_expr_1>
  <stmt_list_1>
else
  <stmt_list_2>
endif
```


Slide courtesy: Milind Kulkarni

If statements



Slide courtesy: Milind Kulkarni

Generating code for ifs

```
if <bool_expr_1>
  <stmt_list_1>
else
  <stmt_list_2>
endif
```

```
  <code for bool_expr_1>
  j<!op> ELSE_1
  <code for stmt_list_1>
  jmp OUT_1
ELSE_1:
  <code for stmt_list_2>
OUT_1:
```

Notes on code generation

- The <op> in j<!op> is dependent on the type of comparison you are doing in <bool_expr>
- When you generate JUMP instructions, you should also generate the appropriate LABELs
- Remember: labels have to be unique!

Slide courtesy: Milind Kulkarni

Code-generation – if-statement

Program text	**3AC**
`INT a, b;`	`STOREI 2 T1    //a := 2`
`a := 2;`	`STOREI T1 a`
`IF (a = 1)`	`STOREI 1 T2    //a = 1?`
`    b := 1;`	`NE a T2 label1`
`ELSIF (TRUE)`	`STOREI 1 T3    //b := 1`
`    b := 2;`	`STOREI T3 b`
`ENDIF`	`JUMP label2   //to out label`
	`LABEL label1  //elsif label`
	`STOREI 1 T4  //TRUE can be handled by checking 1 = 1?`
	`STOREI 1 T5`
	`NE T4 T5 label3  //jump to the next elsif label`
	`STOREI 2 T6  //b := 2`
	`STOREI T6 b`
	`JUMP label2  //jump to out label`
	`LABEL label3  //out label`
	`LABEL label2  //out label`

Suggested Reading

- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D.Ullman: Compilers: Principles, Techniques, and Tools, 2/E, AddisonWesley 2007
  - Chapter 2 (2.8), Chapter 6(6.2, 6.3, 6.4)
- Fisher and LeBlanc: Crafting a Compiler with C
  - Chapter 7 (7.1, 7.3), Chapter 11 (11.2)