CS406: Compilers

Spring 2021

Week 6: Parsers (LR(k)) and Semantic Processing

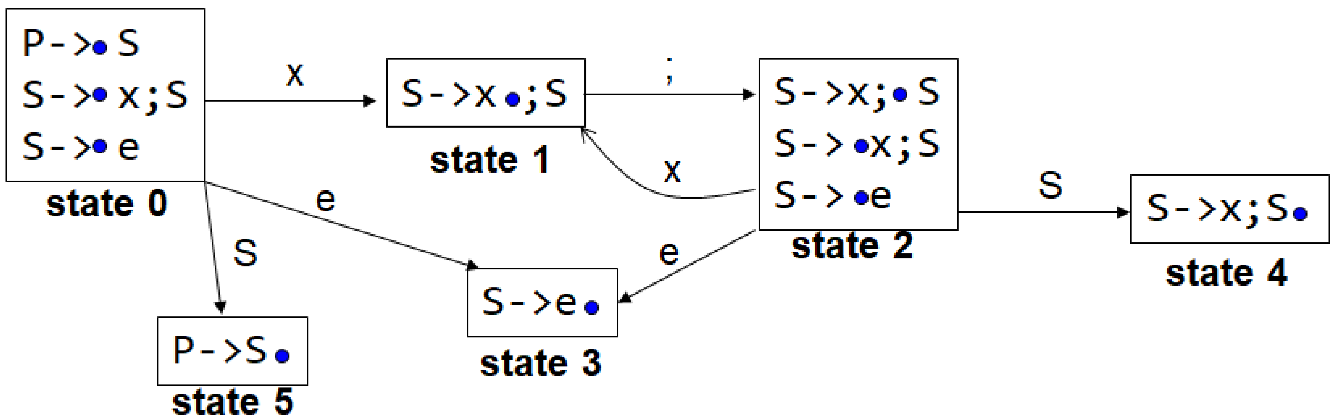
LR(0) Parsing

- Previous Example of LR Parsing was LR(0)
  - No (0) lookahead involved
  - Operate based on the parse stack state and with goto and action tables (How?)

LR(0) Parsing

- Assume: Parse stack contains α == saying that a e.g. prefix of `x;x` is seen in the input string

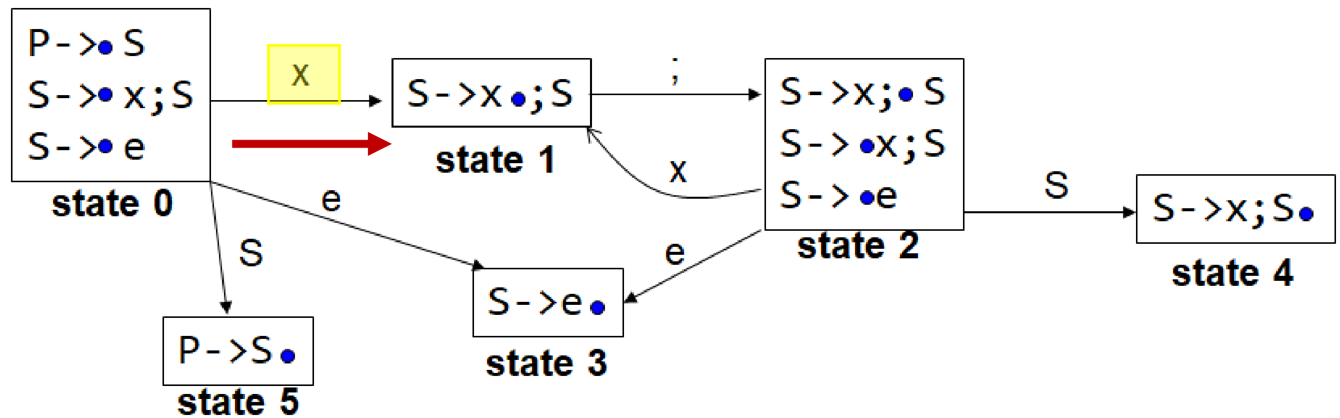
Parse Stack
0
0 1
0 1 2
0 1 2 1



LR(0) Parsing

- Assume: Parse stack contains α == saying that a prefix of `x;x` is seen in the input string

Parse Stack
0
0 1
0 1 2
0 1 2 1

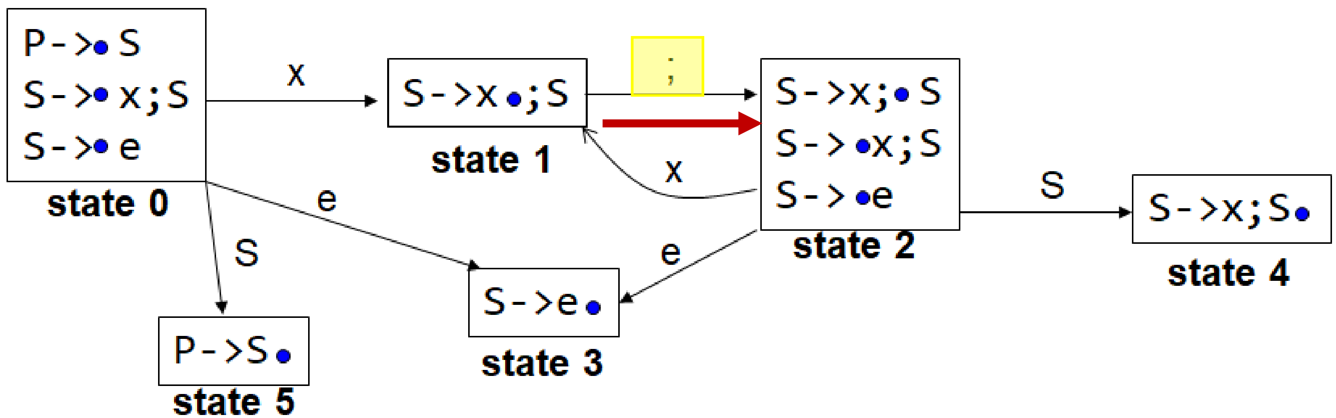


Go from state 0 to state 1 consuming x

LR(0) Parsing

- Assume: Parse stack contains α == saying that a prefix of `x;x` is seen in the input string

Parse Stack
0
0 1
0 1 2
0 1 2 1

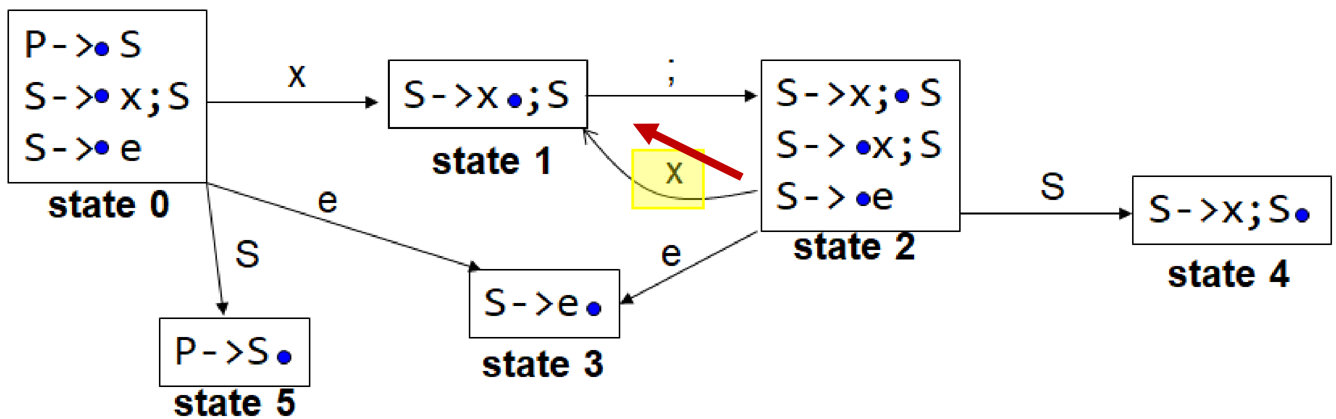


Go from state 1 to state 2 consuming ;

LR(0) Parsing

- Assume: Parse stack contains α == saying that a prefix of `x;x` is seen in the input string

Parse Stack
0
0 1
0 1 2
0 1 2 1



Go from state 2 to state 1 consuming x

LR(0) Parsing

- Assume: Parse stack contains α.

=> we are in some state s

LR(0) Parsing

- Assume: Parse stack contains α.

=> we are in some state s.

We reduce by X->β if state s contains X->β●

- Note: reduction is done based solely on the current state.

LR(0) Parsing

- Assume: Parse stack contains α.

=> we are in some state s.

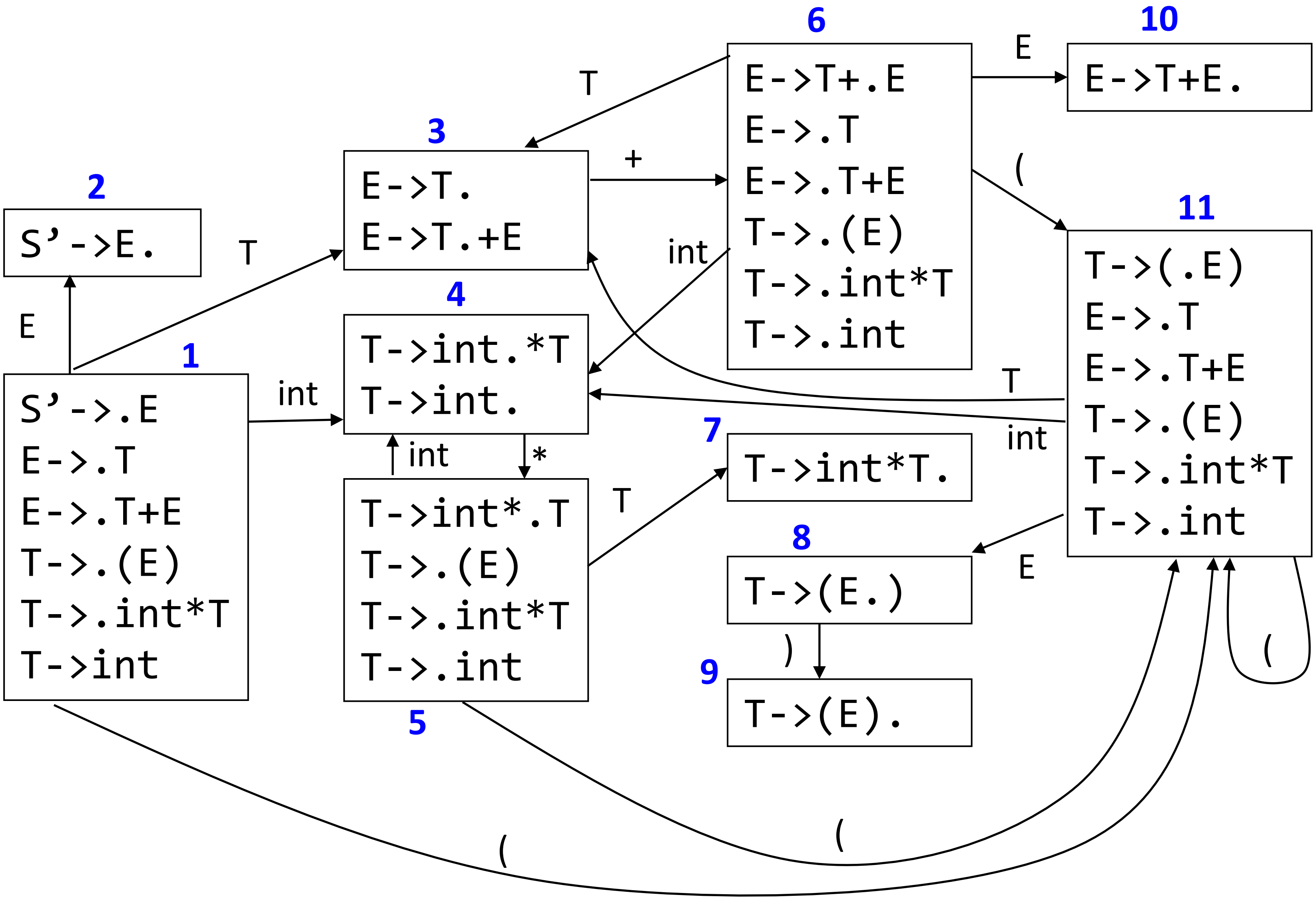
- Assume: Next input is t

We shift if s contains X->β ● tω

== s has a transition labelled t

LR(0) Parsing

- What if s contains `X->β ● tω` and `X->β●` ?



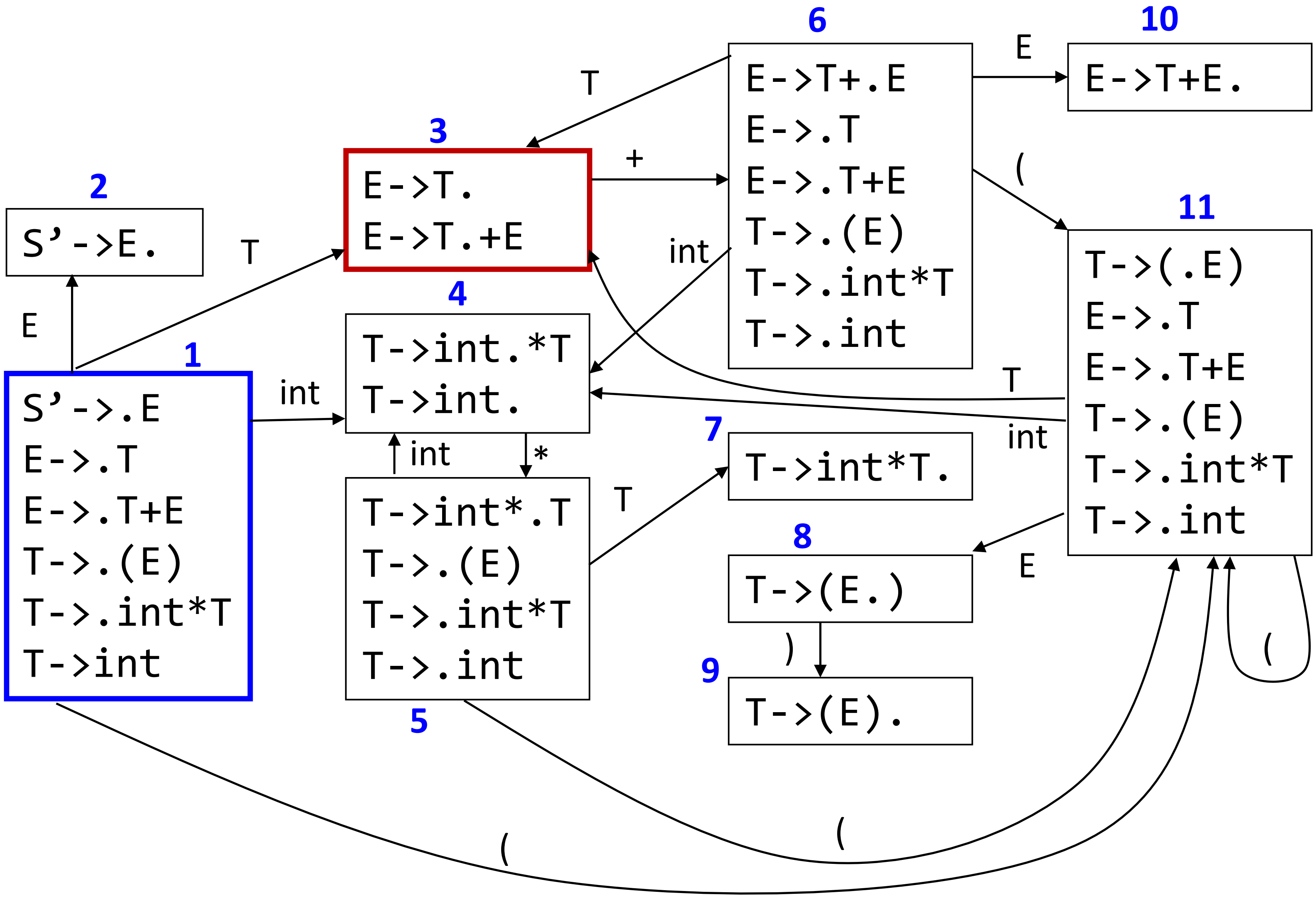
Conflicts or not?

SLR Parsing

- SLR Parsing improves the shift-reduce conflict states of LR(0):

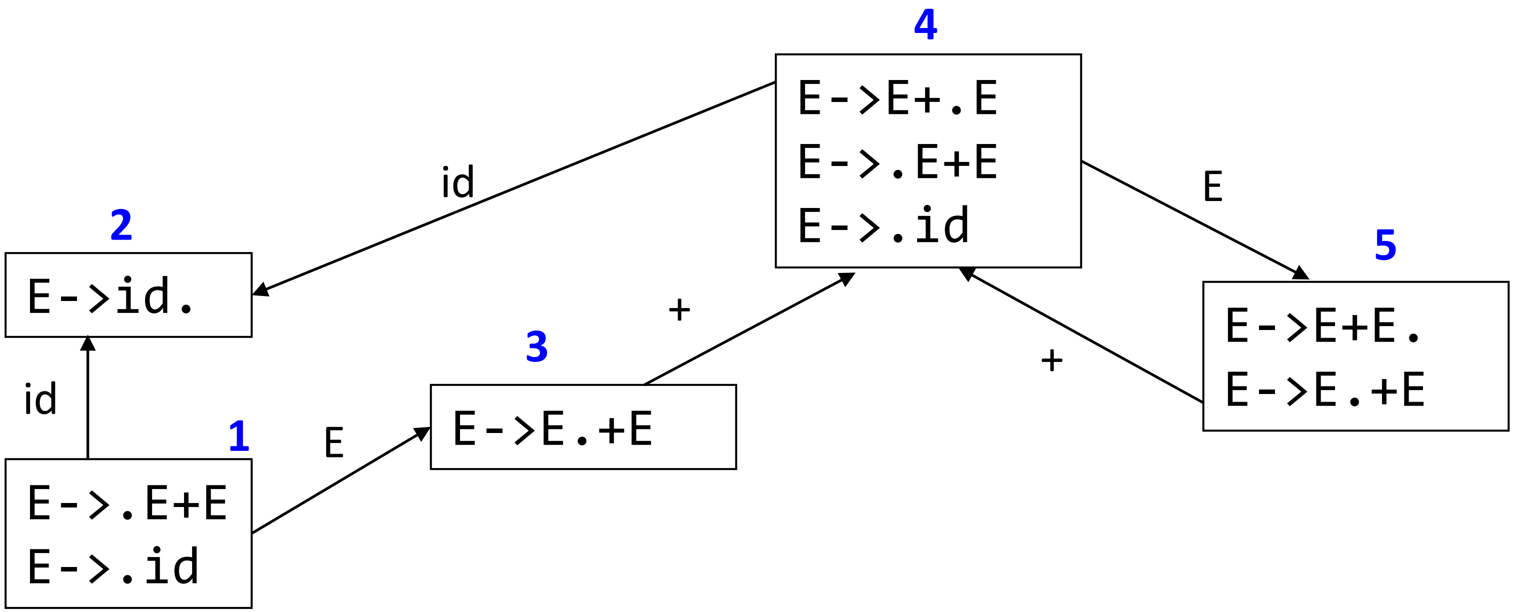
Reduce `X->β●` `only if`

`t` ϵ Follow(X)



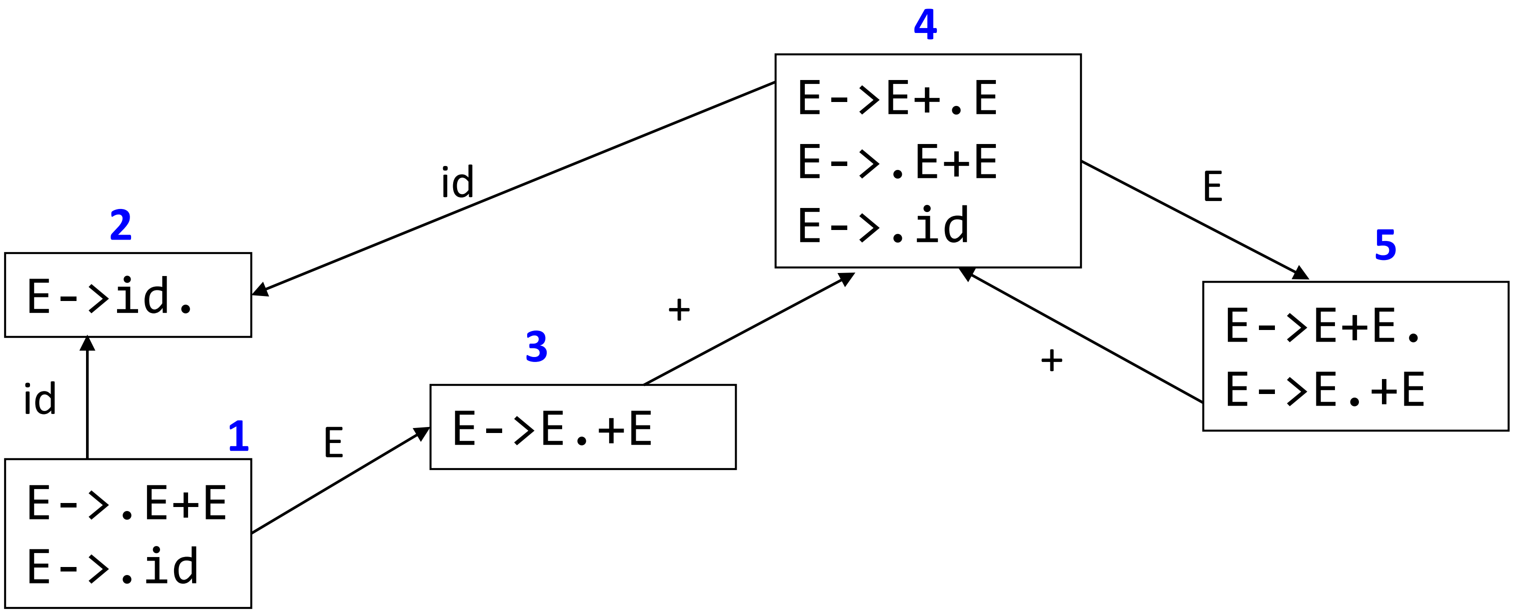
Follow(E) = { $,) }   => reduce by E->T. only if <u>next input</u> is $ or)

lookahead 1



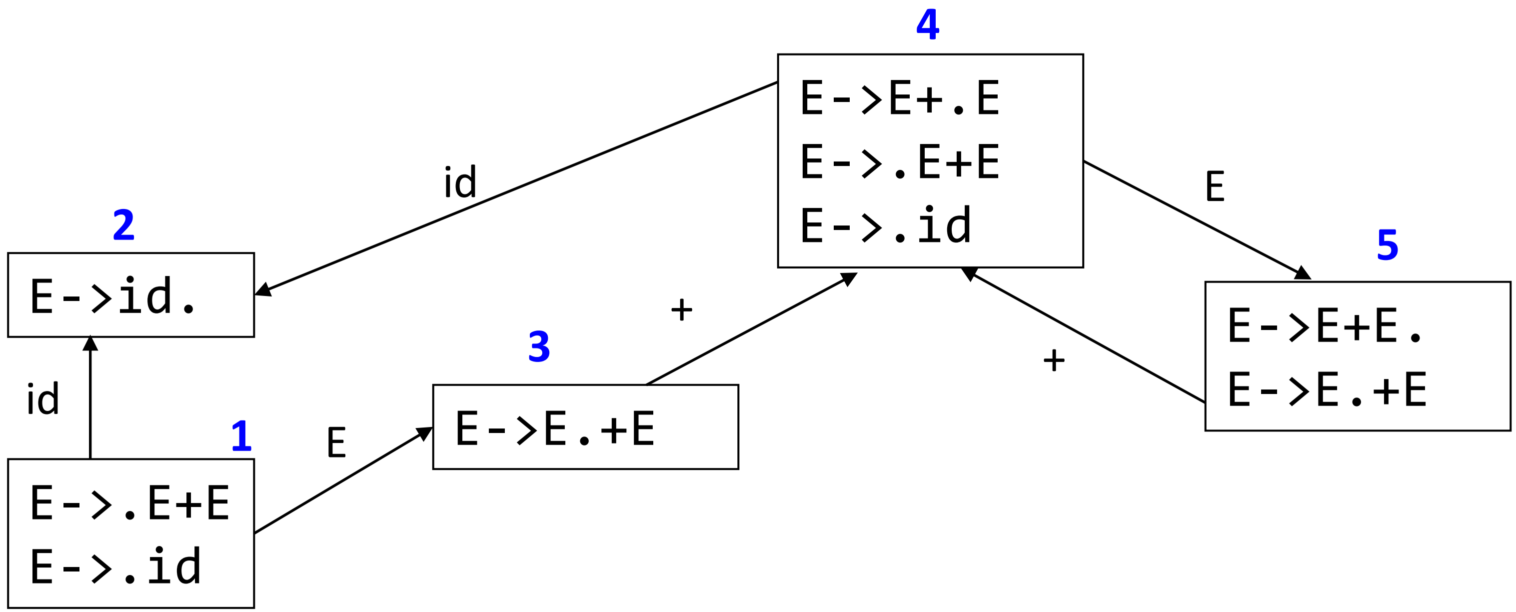
What about the grammar **E-> E + E | id** ?

LR(0)?



What about the grammar **E-> E + E | id** ?

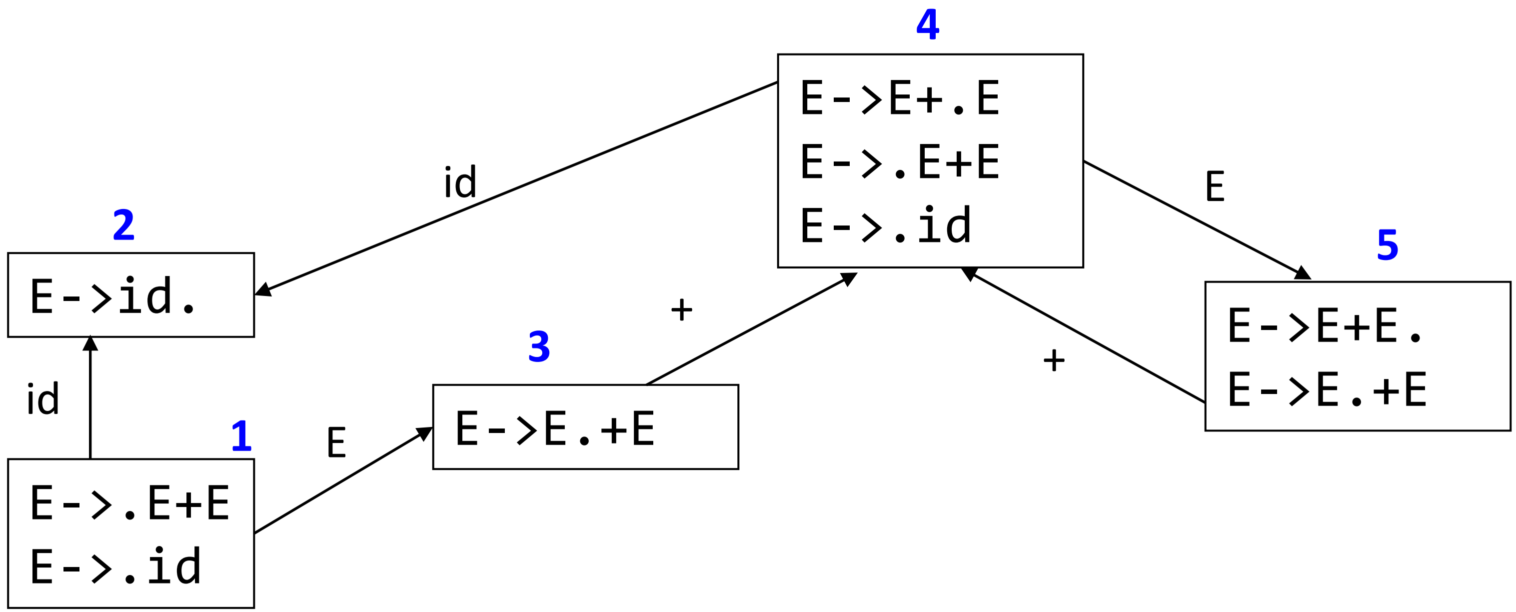
LR(0)? SLR(1)?



What about the grammar **E-> E + E | id** ?

LR(0)? SLR(1)?

Follow(E) = {+,$} => in state 5, reduce by E->T. only if <u>next input</u> is $ or +



What about the grammar **E-> E + E | id** ?

LR(0)? SLR(1)?

Follow(E) = {+,$} => in state 5, <u>reduce by E->T. only if next input is $ or +</u>

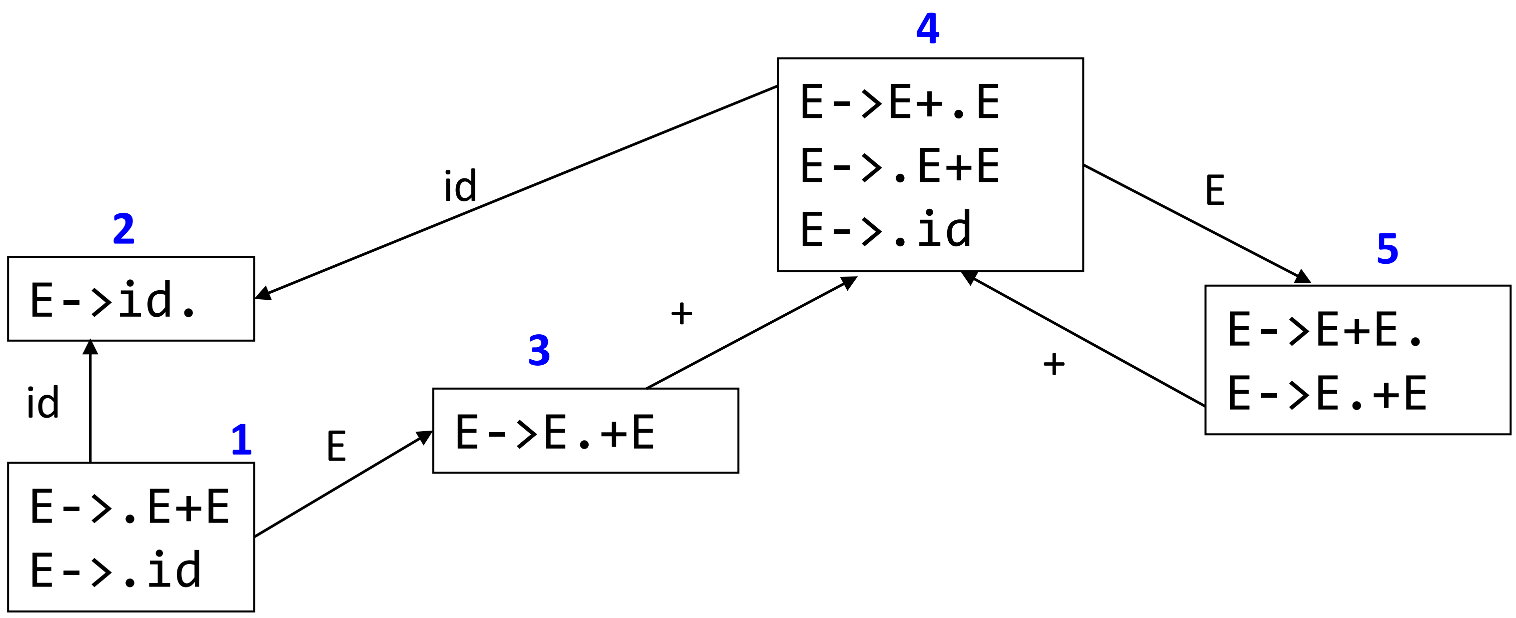
But state 5 has E->E.+E (shift if next input is +)

Shift-reduce conflict!

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



What about the grammar **E-> E + E | id** ?

LR(0)? SLR(1)?

Follow(E) = {+,$} => in state 5, <u>reduce by E->T. only if next input is $ or +</u>

But state 5 has E->E.+E (shift if next input is +)

Shift-reduce conflict!

%left +

says reduce if the next input symbol is + i.e. prioritize rule E+E. over E.+E

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

"Front-end"

Lexical Analysis – Detects programs with illegal tokens

filter

Parsing – Detects programs with ill-formed programming constructs i.e. invalid parse tree structure

filter

Semantic Processing – Detects all remaining errors

filter

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, a method can be overridden.
  - …

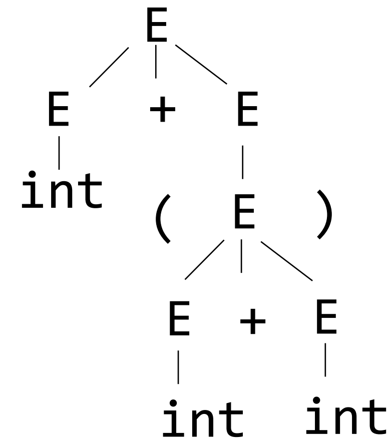
Abstract Syntax Tree

- 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
```

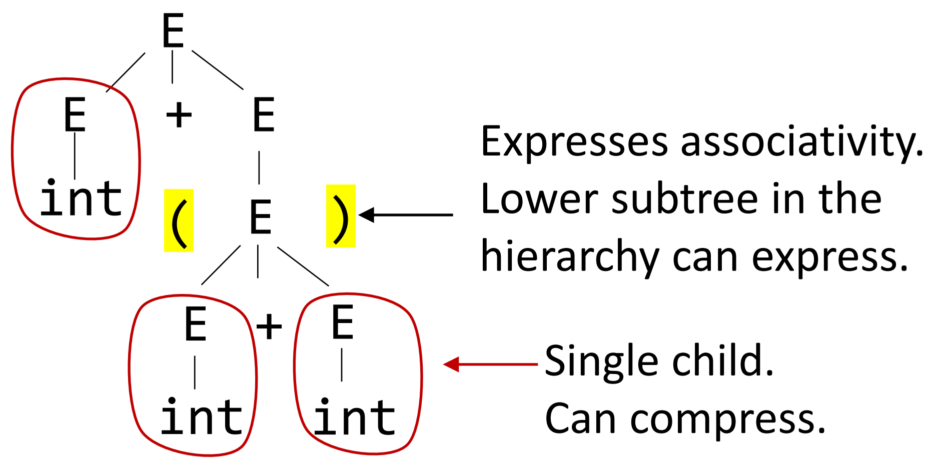
The parse tree for 1+(2+3)



Abstract Syntax Tree - Example

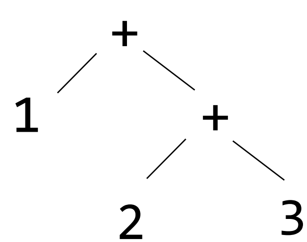
- Not all details (nodes) of the parse tee are helpful for semantic analysis

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



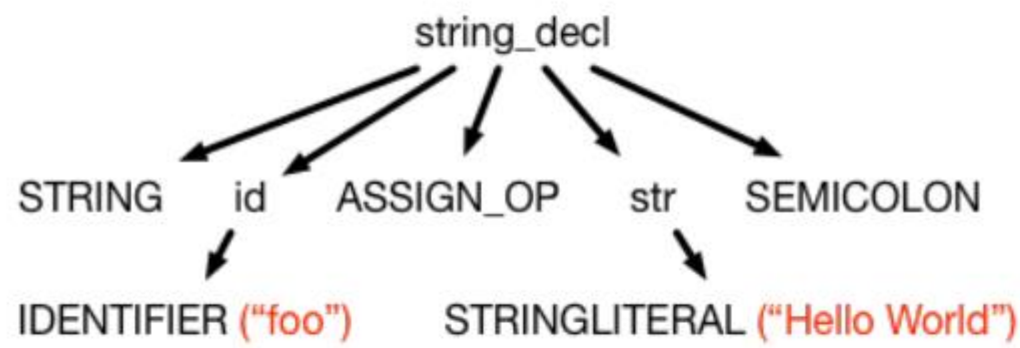
We need to compute the result of the expression. So, a simpler structure is sufficient:

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



AST - Example

STRING foo := “Hello World” ;



≡

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

Semantic Analysis – Example

- Context-free grammars cannot specify all requirements of a language
  - Identifiers <u>declared</u> 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 vs. Dynamic Scope

- Most languages are statically scoped
  - Scope depends 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

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

value of a is 4 here

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

Exercise: Static vs. Dynamic Scope

```
#define a (x+1)

int x = 2;

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

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

int main() { b(); c(); }
```

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
  - Do multiple passes.
- 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

Case study - Semantic Analysis of Expressions

- 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?

Building Abstract Syntax Trees

- Can build while parsing a FPE
  - Via bottom-up building of the tree
- Create subtrees, make those subtrees left- and right- children of a newly created root.
  - Modify 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 Abstract Syntax Trees

- Can build while parsing a FPE
  - Via bottom-up building of the tree
- Create subtrees, make those subtrees left- and right-children of a newly created root.
  - Modify 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 Abstract Syntax Trees

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 Abstract Syntax Trees

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

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

   return IsTerm(s, INTLITERAL);

}
```

Building Abstract Syntax Trees

- Can build while parsing a FPE
  - Via bottom-up building of the tree
- Create subtrees, make those subtrees left- and right-children of a newly created root.
  - Modify 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 Abstract Syntax Trees

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.

```
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 Abstract Syntax Trees

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.

```
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 Abstract Syntax Trees

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

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

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

Building Abstract Syntax Trees

```
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 Abstract Syntax Trees

```
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?

Visualizing Parsing - 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

Visualizing Parsing - 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

Predicting rule 1

Visualizing Parsing - 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)

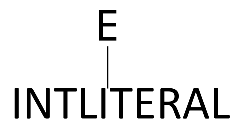
next token

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

Call stack

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

Parse tree



Visualizing Parsing - 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 expects an INTLITERAL but the next token is LPAREN. So, returns NULL.

Visualizing Parsing - 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 also advances pointer in GetNextToken() before returning NULL

Visualizing Parsing - 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

Visualizing Parsing - 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

Predicting rule 1 failed, ret is NULL.

Visualizing Parsing - 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 in SetCurTokenSequence

Visualizing Parsing - 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)

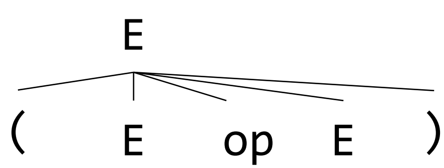
next token

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

Call stack

E()
E2()

Parse tree



Predicts Rule 2

Visualizing Parsing - 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)

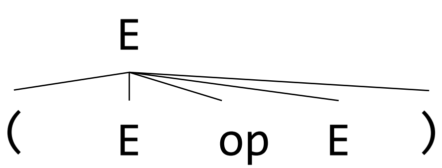
next token

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

Call stack

E()
E2()

Parse tree



E2 check for LPAREN succeeds (next token is moved forward)

Visualizing Parsing - 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)

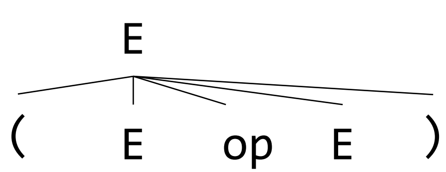
next token

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

Call stack

E()
E2()

Parse tree



E2 calls E()

Visualizing Parsing - 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)

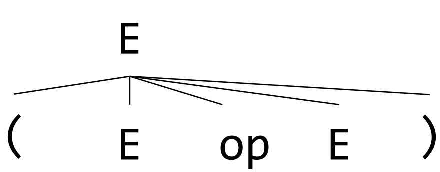
next token

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

Call stack

E()
E2()
E()

Parse tree



E calls E1(), predicts rule 1

Visualizing Parsing - 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)

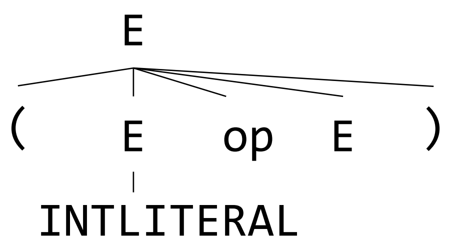
next token

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

Call stack

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

Parse tree



E1 calls IsTerm()

Visualizing Parsing - 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)

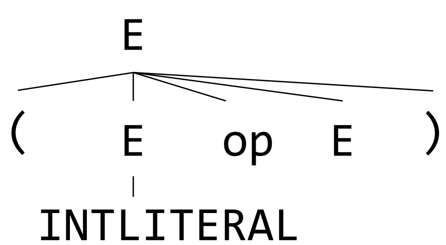
next token

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

Call stack

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

Parse tree



IsTerm() expects INTLITERAL and the next token is INTLITERAL. So, it creates AST Node and stores the INTLITERAL's val

Visualizing Parsing - 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)

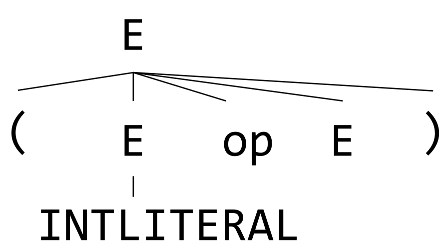
next token

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

Call stack

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

Parse tree



IsTerm() expects INTLITERAL and the next token is INTLITERAL. So, it creates AST Node and stores the INTLITERAL's val

Observations - Hand-written Parser

1. AST node is created bottom-up
2. Value associated with INTLITERAL 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

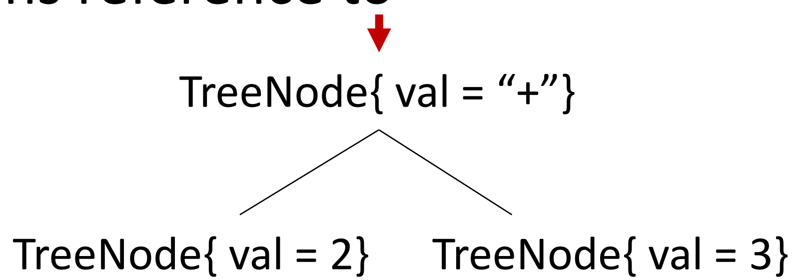
- What did we do when we saw an INTLITERAL?
  - Create a TreeNode
  - Initialize it with a value (string equivalent of INTLITERAL in this case)
  - Return a pointer to TreeNode

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.

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 returns reference to



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`

- Attributes are of two types: Synthesized, Inherited

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

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

Expressions

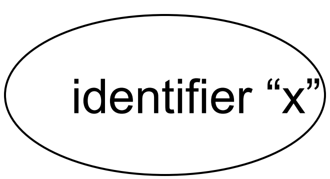
- Three semantic actions needed
  - eval_binary (processes binary expressions)
    - Create AST node with two children, point to AST nodes created for left and right sides
  - eval_unary (processes unary expressions)
    - Create AST node with one child
  - process_op (determines type of operation)
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

Expressions Example

x + y + 5

Expressions Example

x + y + 5



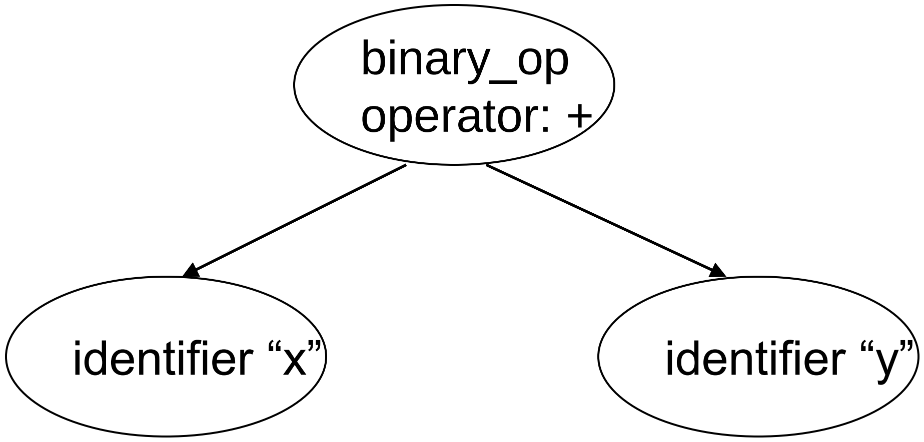
Expressions Example

x + y + 5

Slide courtesy: Milind Kulkarni

Expressions Example

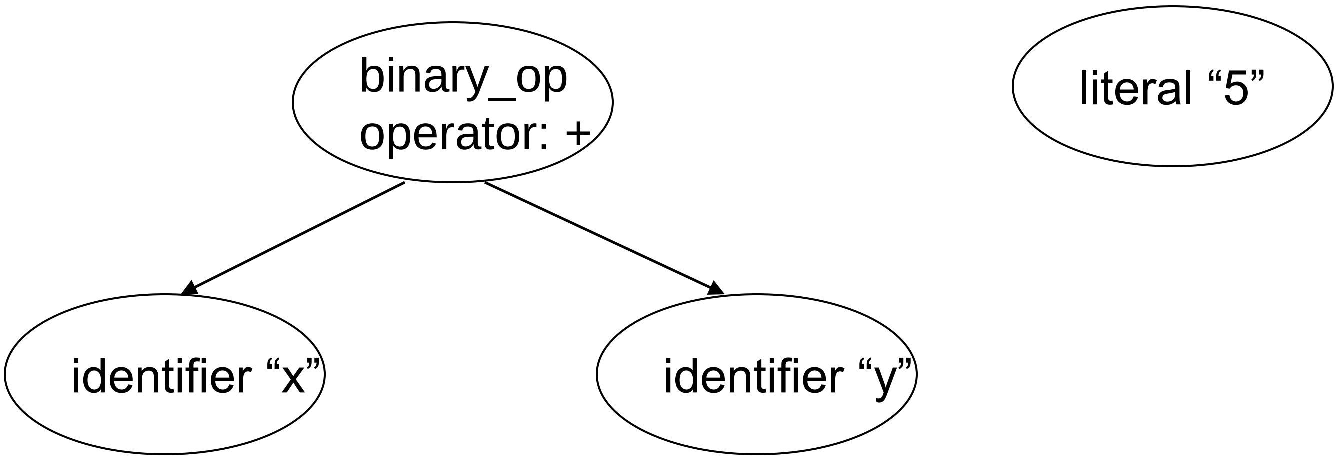
x + y + 5



Slide courtesy: Milind Kulkarni

Expressions Example

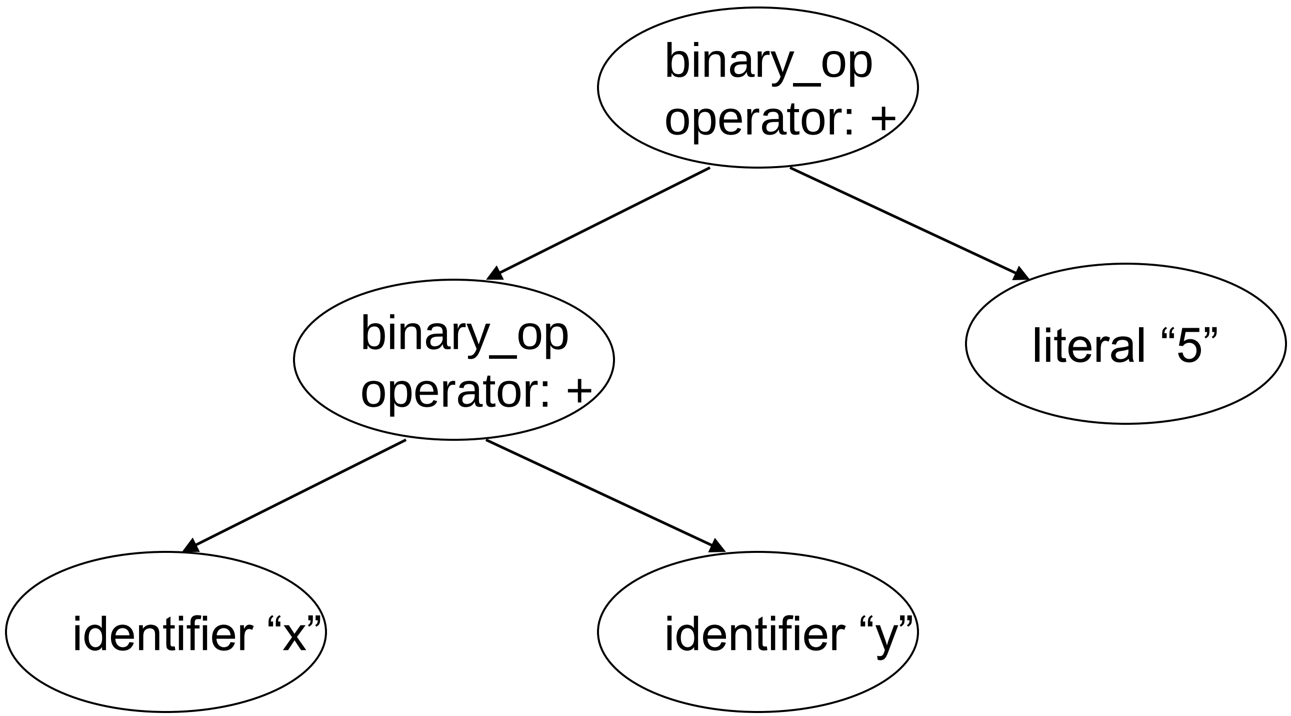
x + y + 5



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.