CSE 131A

Lecture 8

LR Parsers
Parsing Conflicts
Building the AST

Scott B. Baden
Department of Computer Science & Engineering
University of California, San Diego
Winter 2007

Today's lecture

- Inside a bottom up parser
  - Rightmost derivations and right-sentential forms
  - Handles and handle pruning
  - LR(k) grammars
- Conflicts
- Implementing the AST

Bottom up parsing

- Recall that a bottom up parser attempts to construct a parse tree for an input string bottom up, starting with the leaves and working up to the root
- Leaves correspond to tokens in the input; the root is the grammar's start symbol
- We'll next look at how to implement the basic bottom up, shift reduce parsing algorithm

Review: reducing to the start symbol

- Recall that a bottom up parser performs a series of reductions
- Consider the following grammar

$$S \rightarrow aABe$$
$$A \rightarrow Abc \mid b$$
$$B \rightarrow d$$

- We can reduce the terminal string *abbcde* to the start symbol in 5 steps

$$a\underline{b}bcde \rightarrow a\underline{Abc}de \rightarrow aA\underline{d}e \rightarrow \underline{aABe} \rightarrow S$$

Reduction and derivation

- A reduction of a string to the start symbol is exactly the reverse of a derivation of the string from the start symbol
- The reductions shown previously trace out this derivation in reverse:

  S* ⇒ *aABe* ⇒ *aAde* ⇒ *aAbcde* ⇒ *abbcde
- Either a reduction or a derivation shows that a string is a sentence in the language defined by the grammar …
- … and has a corresponding parse tree

Rightmost derivations

- A *rightmost derivation* is one in which at each step the rightmost nonterminal in the sentential form is replaced by the RHS of a rule with the nonterminal on the LHS

  S* ⇒ *aABe* ⇒ *aAde* ⇒ *aAbcde* ⇒ *abbcde

- The grammar

  S → *aABe*
  A → *Abc* | *b*
  B → *d*

- A sentential form that can be obtained by a rightmost derivation from the start symbol is a *right-sentential form* of the grammar (and similarly with leftmost derivations)

Finding the substring to reduce: handles

- In a bottom up parser, we reduce a string to the start symbol following a rightmost derivation in reverse
- At each step, we have a right-sentential form
- Question: which substring of the current sentential form should be reduced during the next step, to continue the rightmost derivation in reverse?
- This substring, together with the rule to use to reduce it, is called the *handle* of the sentential form

Handle defined

- **Definition:**

  A **handle** of a right sentential form γ is

  - a substring β at a particular position within γ and a production $A \rightarrow \beta$ such that
  - ... β was produced from A
  - ... in the previous step in a rightmost derivation of γ

$$S \Rightarrow aA\underline{B}e \Rightarrow a\underline{A}de \Rightarrow a\underline{A}bcde \Rightarrow abbcde$$

$$S \rightarrow aABe$$
$$A \rightarrow Abc \mid b$$
$$B \rightarrow d$$

A handle of a right-sentential form

- Let α, β, w be strings of symbols
- Let $\Rightarrow$ mean rightmost derivation
- Suppose $S \Rightarrow^* \alpha A w \Rightarrow \alpha \beta w$
- A handle of the right-sentential form $\alpha \beta w$ is
  - " β in the position between α and w
  - with the production $A \rightarrow \beta$ "
  - (β was derived from A in the previous step)
- Since this is a rightmost derivation,

  w must be empty or contain what kind of symbols?

$a\underline{A}de \Rightarrow aAbcde$

$A \rightarrow Abc \mid b$

An Example

- Consider the grammar

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow (E)$$
$$E \rightarrow \mathbf{id}$$

- A rightmost derivation ***with handles*** …

$$E \Rightarrow \underline{E + E} \Rightarrow E + \underline{E * E} \Rightarrow$$
$$E + E * \underline{id_3} \Rightarrow$$
$$E + \underline{id_2} * id_3 \Rightarrow \underline{id_1} + id_2 * id_3$$

Identifying the handle

- Substring ***id1*** and the rule $E \rightarrow id_1$ is a handle of the right sentential form

  $$id_1 + id_2 * id_3$$

- Why?
- There is a production $E \rightarrow$ **id, …**
- …and replacing id_1 by E produces the previous right sentential from $E + id_2 * id_3$ in the right most derivation from the start symbol
- The string to the right of a handle is either empty, or contains only terminals (why?)

How many handles?

- If a grammar is unambiguous, every sentence has one unique rightmost derivation
- Therefore with an unambiguous grammar, every right-sentential form has exactly one handle
- However, there may be more than one handle in an ambiguous grammar, because there may be multiple rightmost derivations
- This leaves the question: how do we find a handle?
- We'll return to this later

Non-unique handles in an ambiguous grammar

- The first rightmost derivation we saw was:

$$E \Rightarrow \underline{E + E} \Rightarrow E + \underline{E * E} \Rightarrow$$
$$E + E * \underline{id_3} \Rightarrow$$
$$E + \underline{id_2} * id_3 \Rightarrow \underline{id_1} + id_2 * id_3$$

- But the grammar is ambiguous, so there is another rightmost derivation, but with different handles:

$$E \Rightarrow \underline{E * E} \Rightarrow E * \underline{id_3} \Rightarrow$$
$$\underline{E + E} * id_3 \Rightarrow$$
$$E + \underline{id_2} * id_3 \Rightarrow \underline{id_1} + id_2 * id_3$$

Handle pruning

- We can think of bottom-up parsing as a process of repeatedly finding and "pruning" handles until there is nothing left but ***S***
- Beginning with a string of terminals ***w*** …
- If ***w*** is a sentence of the grammar …
  - then $w = \gamma_n$,
  - where γ_n is the nth right-sentential form of an unknown rightmost derivation that we seek

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$$

Starting the reduction

- Given the unspecified derivation

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$$

- Locate the handle β_n where $\gamma_n = \alpha\, \beta_n x$
- Replace β_n by the LHS of some production $A \rightarrow \beta_n$
- This gives us the $(n\text{-}1)^{st}$ right sentential form γ_{n-1}

Completing the reduction

- The rightmost derivation in reverse

  $S \Leftarrow \gamma_0 \Leftarrow \ldots \Leftarrow \gamma_{n-1} \Leftarrow \gamma_n = w$
- Locate handle β_n in γ_n (precedes terminal x)

  $S \Leftarrow \gamma_0 \Leftarrow \ldots \Leftarrow \gamma_{n-1} \Leftarrow \alpha \beta_n x = w$
- Replace β_n with LHS of some production $A \rightarrow \beta_n$

  $S \Leftarrow \gamma_0 \Leftarrow \ldots \Leftarrow \alpha A x \Leftarrow \alpha \beta_n x = w$
- This gives us the $(n\text{-}1)^{st}$ right sentential form

  $\gamma_{n-1} = \alpha A x$
- Repeat for γ_{n-2} and so on
- The reverse of the sequence of reductions is a rightmost derivation for w

An Example

- Consider our grammar again, with input X+Y∗Z

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow \mathbf{id}$$

Right sentential form	Handle	Production
X + **Y** ∗ **Z**	**X**	$E \rightarrow$ ***id***
E + **Y** ∗ **Z**	**Y**	$E \rightarrow$ ***id***
$E + E$ ∗ **Z**	**Z**	$E \rightarrow$ ***id***
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

Bottom up, left-to-right, shift reduce parsers

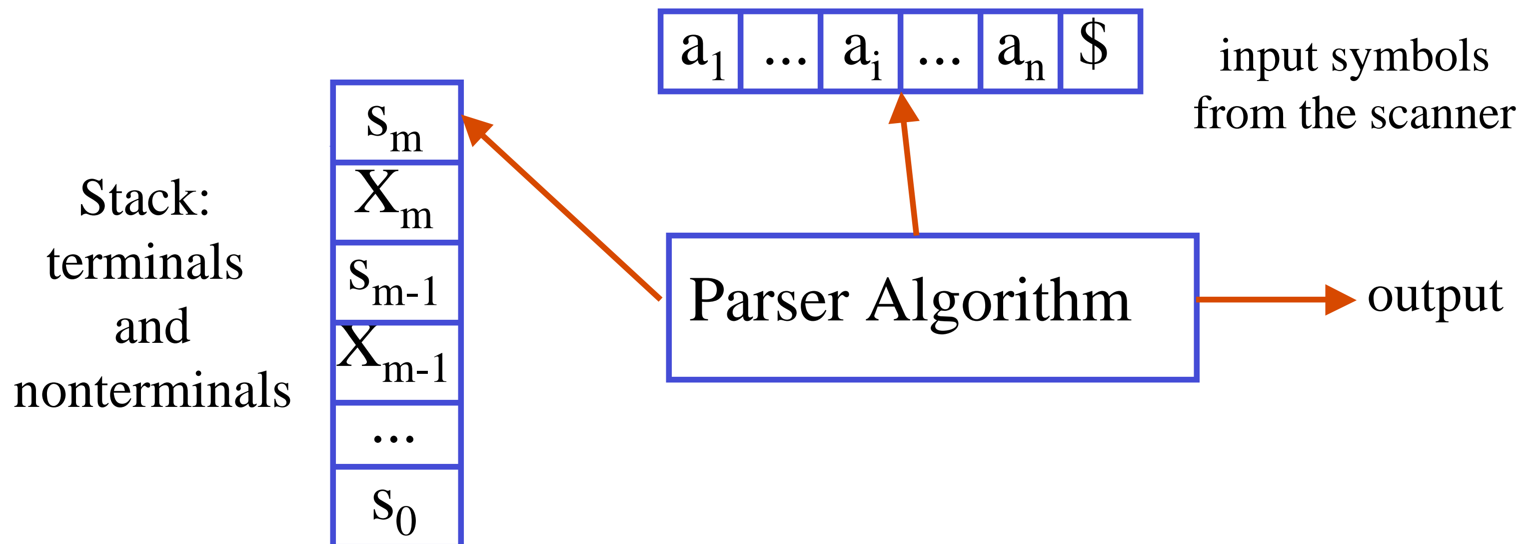
- All bottom up parsers will perform reductions in the process of consuming the token stream
- Our parser reads the input token stream left-to-right
- Reductions generally consume the leftmost part of the input before the rightmost part
- An ***LR*** parser reads input **L**eft to right, following a **R**ightmost derivation in reverse
- An LR parser is also called a ***shift-reduce*** parser

Implementing an LR parser

- The parser will find a rightmost derivation in reverse, or detect an error
- It needs to do these things repeatedly until it reaches the start symbol:
  - Find the handle of the input and reduce
  - Find the handle of the result (why is that?) and reduce

Implementing an LR parser

- All parsers have the same basic structure
  - The ***stack*** holds grammar symbols for reduction; only symbols at the top of the stack (TOS) are accessed
  - The ***input buffer*** holds the string of tokens to be parsed; tokens are read in sequence



Shifting and reducing

- The parser **shifts** input tokens onto the stack until it finds a match of the symbols on the top of stack (TOS) with the RHS of some production
- It then **reduces** those symbols to the LHS of the production, replacing with the LHS on TOS
- The cycle repeats until the stack contains the start symbol, in which case the parser **accepts** the input
- If it cannot find a production it raises an **error**

Implementation

- The right-sentential form we are reducing is distributed across the stack and input buffer
  - The stack holds the left part
  - The unread input contains the remainder
- We use the special symbol $ to mark the bottom of the stack and the right end of the input
- Initial state with empty and stack and input to be parsed:

Stack	Input
$	***w*** $

Actions of the parser

- The main actions of the parser are
- **shift**
  - shift the next input symbol onto TOS
  - The parser must know that the TOS does not already contain a handle
- **reduce**
  - replace the handle on TOS with nonterminal symbol
  - The parser must know that the handle is on the top of the stack, how many symbols it contains, and which production to use

Other actions

- **accept**
  - Parsers signals that it has successfully completed the parse
- **error**
  - Parser detects a syntax error in the input and calls an error reporting/handling routine

The basic algorithm

- We *shift* symbols from the input onto the stack if needed until we find a handle β on top
- The handle may consist of several symbols and there may be more under the handle
- We then *reduce* the handle β to the LHS of the handle's production; the reduction replaces β on the top of the stack with the symbol on the LHS of the production
- The process continues until there is an error, or the stack contains the start symbol and the input is empty:

Stack	Input
$ ***S***	$

Example

Stack	Input	Action
$\$$	$\mathbf{X} + \mathbf{Y} * \mathbf{Z}\$$	Shift
$\$\mathbf{X}$	$+ \mathbf{Y} * \mathbf{Z}\$$	Reduce by $E \rightarrow \boldsymbol{id}$
$\$E$	$+ \mathbf{Y} * \mathbf{Z}\$$	Shift
$\$E +$	$\mathbf{Y} * \mathbf{Z}\$$	Shift
$\$E + \mathbf{Y}$	$* \mathbf{Z}\$$	Reduce by $E \rightarrow \boldsymbol{id}$
$\$E + E$	$* \mathbf{Z}\$$	Shift
$\$E + E *$	$\mathbf{Z}\$$	Shift
$\$E + E * \mathbf{Z}$	$\$$	Reduce by $E \rightarrow \boldsymbol{id}$
$\$E + E * E$	$\$$	Reduce by $E \rightarrow \boldsymbol{E} * E$
$\$E + E$	$\$$	Reduce by $E \rightarrow \boldsymbol{E} + E$
$\$E$	$\$$	**accept**

Stacks for bottom-up parsing

- All handles eventually appear at the top of stack (Theorem)
- The parser doesn't need to search down the stack to find the handle
- This is why a stack is natural for the job
  – corresponds to the ability of pushdown automata to recognize CFLs

Computing the handle, or not

- Handles are the key to bottom-up parsing
- We haven't shown how to efficiently compute the next handle
- We will return to that issue
- When the parser cannot deterministically compute the handle we have a parser **conflict**

Parser Conflicts

- A shift reduce parser isn't able to parse all context free grammars
- With a problematic grammar, the parser can enter a state where it is not able to decide...
  - whether to shift or reduce, called a **shift/reduce conflict**
  - which reduction to make, called a **reduce/reduce conflict**

Non-LR(k) grammars

- A LR parser can use "lookahead" of a certain number k of tokens in the input to attempt to resolve conflicts: this is a LR(k) parser
- A grammar that leads to conflicts in a LR(k) parser is a non-LR(k) grammar
- An ambiguous grammar is non-LR(k) for all k; other grammars may be just non-LR(1) for example
- CUP produces an LALR(1) parser (Lookahead LR, 1 token look ahead), an efficient LR(1) parser

Non-LR(k) grammars

- A non-LR(k) grammar will lead to shift/reduce or reduce/reduce conflicts in an LR(k) parser
- A LR(k) parser such as CUP has conventions to handle non-LR(k) grammars:
  - shift upon encountering a shift/reduce conflict (often useful)
  - reduce first rule in reduce/reduce conflict (dangerous)

Example of an ambiguous grammar

- The dangling–else grammar:

  stmt → **if** *expr* **then** *stmt*
  | **if** *expr* **then** *stmt* **else** *stmt*
  / **other**

- With the parser in the following configuration

  Stack: **if** *expr* **then** *stmt* **Input:** **else** . . .

- We don't know whether the handle is **if** *expr* **then** *stmt*, even if we look inside the stack
- We could look past the **else** to try to decide; but even looking at the entire input may leave reduce-reduce conflicts
- The "shift in shift-reduce conflict" convention solves this dangling-else problem correctly!
- We were lucky this time

Conflicts in CUP

- Consider the following simple CUP spec:

```
terminal        PLUS, id;
non terminal    E;

E ::=  E PLUS E;
E ::=  id;
```

- This grammar produces conflicts in CUP (Consider the input **a + b + c**)
- Let's analyze and eliminate them

Signs of conflict

- CUP reports a Shift/Reduce conflict that can arise when ..
- .. the parser is in a state where TOS contains the symbols
  E PLUS E …
- and the next input token is **PLUS**

***** Shift/Reduce conflict found in state #5**

between E ::= E PLUS E (*)
and E ::= E (*) PLUS E

under symbol PLUS

- CUP resolves the conflict in favor of shifting

Debugging output in CUP

- CUP provides some debug flags:
  -dump_grammar causes CUP to prints one production per line with no actions shown
  -dump_states causes CUP to print a representation of all its states

- Information about **state #5**

```
lalr_state [5]: {
  [E ::= E PLUS E (*) , {EOF PLUS }]
  [E ::= E (*) PLUS E , {EOF PLUS }]
}
transition on PLUS to state [3]
```

  – Shows the rules "active" in this state
  – Possible lookahead symbols (in **{}**'s) can be for each rule to reduce
  – **(*)** shows top of stack

Remedies

- Two alternatives look reasonable
  - reduce **E PLUS E** to **E**
  - shift **PLUS** in expectation of a longer handle
- CUP's convention is to shift
- This is the wrong choice since it breaks the usual interpretation that addition is left-associative

  (1+2)+3 **1+(2+3)**

- Let's fix it

Avoiding shift-reduce conflicts

- Some conflicts can be eliminated by attaching precedence and associativity specifiers to CUP grammar rules
- Another approach is to rewrite the grammar rules themselves
- This may involve introducing new nonterminal symbols
- In the present example, this second approach is easy and works well

No conflicts in CUP

- Consider the following CUP spec:

```
terminal            PLUS, id;
non terminal        E, T;
E ::=  E PLUS T  | T;
T ::=  id;
```

- This grammar defines the same language as the previous one, but unambiguously
- Forces a left-associative grouping of PLUS (look at the parse tree of e.g. **id+id+id** to confirm this)
- There are no conflicts:
  - if **T** is on top of the stack, reduce is the only option
  - if **E** is on top of stack with lookahead **PLUS**, shift is the only option

Another option

- What would happen with this CUP spec:

```
terminal          PLUS, id;
non terminal      E, T;
E ::=  T PLUS E  | T;
T ::=  id;
```

- This grammar defines the same language as the previous one, and has no conflicts
- What about **id+id+id**

Abstract Syntax Trees

Concrete and abstract syntax trees

- The *concrete syntax tree* shows the complete syntactic structure of a sentence, with internal nodes labeled with non-terminals
- An *abstract syntax tree* (AST) is often more useful
- Suppresses unimportant details that aren't need to express the meaning of the corresponding program fragment
  - Some punctuation
  - Some productions
- Disadvantage: the parse must be completed for the AST to be returned

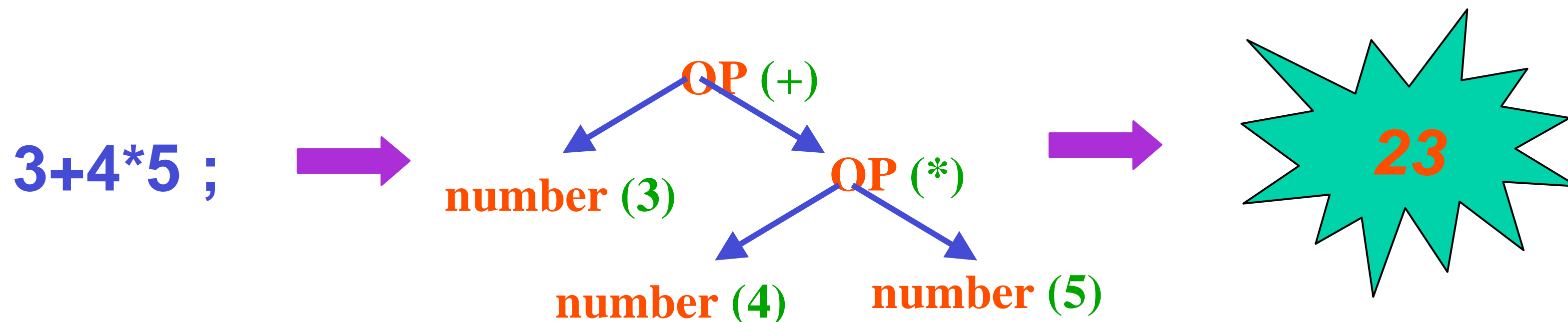
A simple calculator grammar

- Recall the simple calculator grammar:

E_list → E_list E_part | E_part
E_part → E SEMI
E → E PLUS T | T
T → T TIMES F | F
F → LPAREN E RPAREN | number

AST approach

- Rather than evaluate the expression directly…
  - We'll generate an AST
  - Write an interpreter to evaluate the expressions represented by the AST
- We'll next consider AST generation
- We'll return to interpretation of the AST later on



AST Design

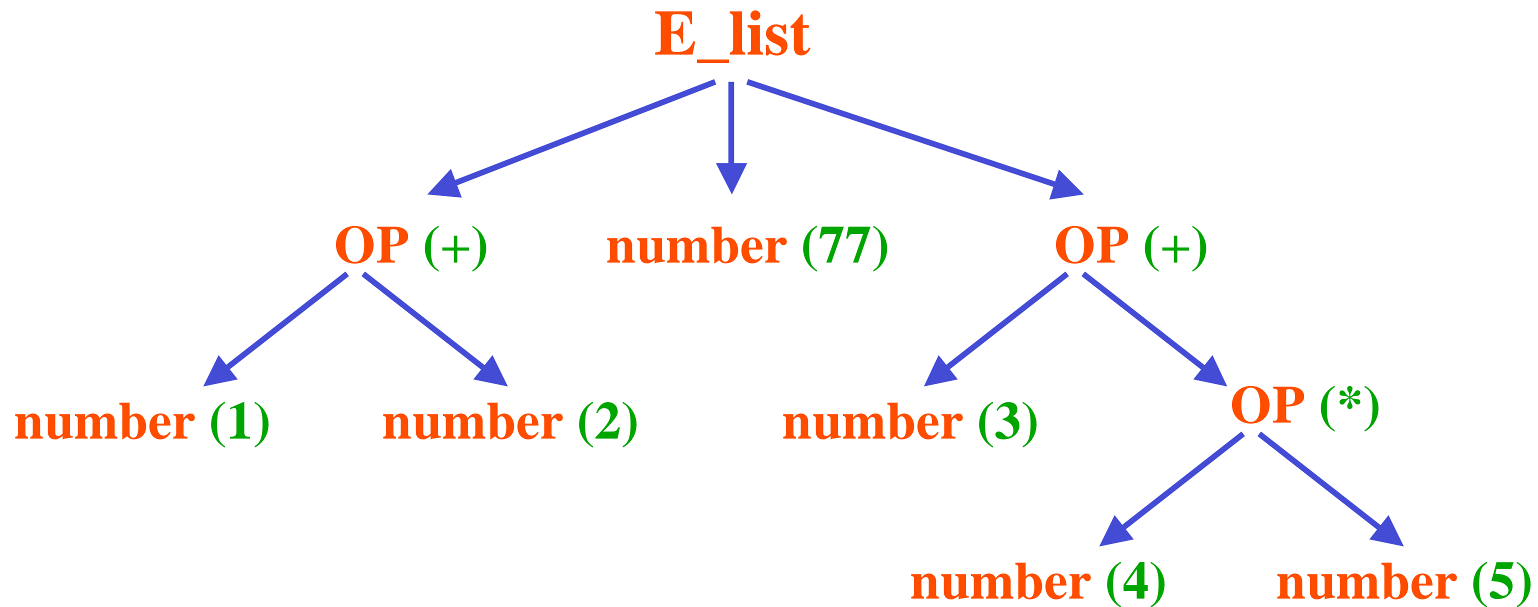
- An AST is organized as a tree of nodes
- In general, an AST is not a binary tree
- The **Symbol** object returned by the parser's **parse()** method will have a **value** field referring to the root node of the AST
- The basic operation is to construct AST node objects, and to pass them up the parse tree using the **value** fields of **Symbol** objects
- Different kinds of AST nodes corresponding to different productions in the grammar
- Nodes correspond to terminals and non-terminals in the grammar

What should be in the AST?

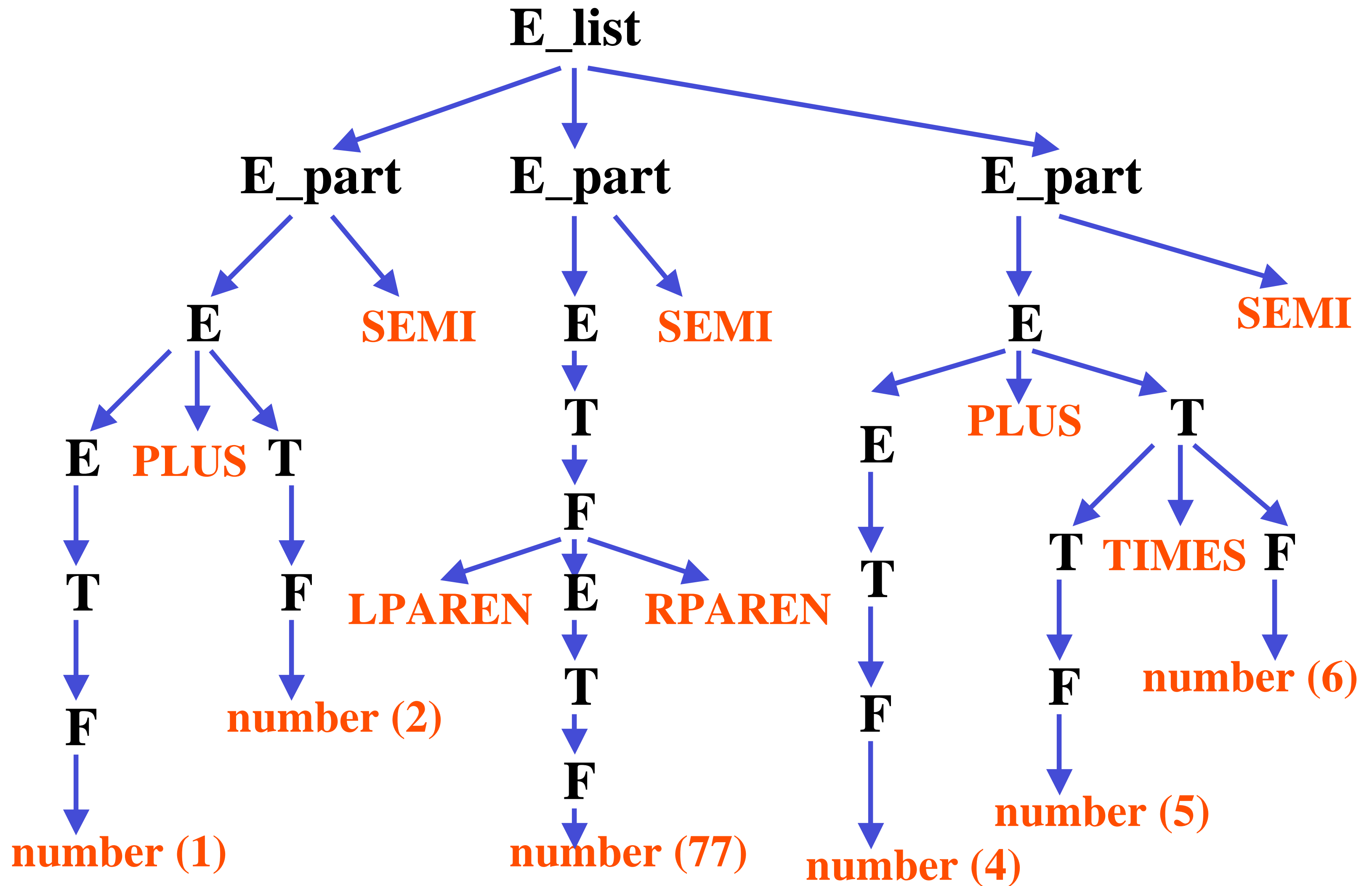
- Consider the parse tree for input, a list of 3 expressions
  1 + 2 ; (77); 4+5*6 ;
- The first is the sum of 2 numbers, **1** and **2**
- The second is just the number **77**
  (the parenthesis don't have any significance)
- The third is the sum of 2 numbers, **4** and the product **5*6**

An AST

- Here is one representation for the AST



The concrete syntax tree



Object oriented design

- An object-oriented approach is natural for AST implementation
- AST Nodes have some common attributes
  - They have some number of children
    (Use a flexible data structure that can store pointers to an arbitrary number of children, e.g. java.util.Vector)
  - They point at a certain position in the source text
- We define a base class ASTNode
  - E.g. an abstract class
  - We try to collect as much of the common behavior into the base class as we can (what is the common behavior?)
- We define one subclass for each AST node type; we might have a hierarchy

ASTNode derived classes

- There should be as many ASTNode subclasses as there are distinct kinds of nodes in the abstract syntax trees you want to construct
- In our example the different kinds we need are: **EList**, **Op**, **NumberNode**
- Each subclass will have additional behavior specialized to its own requirements
- What special things do these classes need?
  - **Op** needs the name of the operator, "*****" or "**+**"
  - **NumberNode** needs a representation of the number
  - Specialized constructors

Constructing the AST with actions

- We build the AST from the bottom up, in the same way as an LR parser traces the parse tree bottom-up
- Recall that action code attached to CUP rules has access to the value fields of the Symbol objects for every grammar symbol in the rule
- The LHS symbol RESULT field needs to be set to refer to an ASTNode
- Then the value field of the Symbol object returned by the parse() method will refer to the root of the AST

Constructing the AST: actions

- We can write actions "bottom up" starting with rules far from the start symbol
- Here we create an AST node for the rule

```
F ::= number:n
```

- We build a container holding information about the number token, and pass it up the parse tree via the RESULT variable

```
{: RESULT = new NumberNode(n); :}
```

Constructing the AST: more actions

- What should the action be for the rule

```
T ::= F
```

- We have previously constructed an ASTNode for the non-terminal(s) on the RHS
- We just pass it up the parse tree:

```
T ::= F:x
{: RESULT = x; :}
```

- Similarly for E ::= T, E_part ::= E SEMI

Yet more actions

- What should the action be for the rule

```
T ::= T TIMES F
```

- This requires a "*" Op node for the LHS, and the RHS symbols T and F are children:

```
T ::= T:t TIMES F:f
{: ASTNode n = new Op("*");
  n.add(t); n.add(f); RESULT=n; :}
```

- Similarly for E ::= E PLUS T

Constructing the AST: still more actions

- What should the action be for the rule

E_list ::= E_part

- Create a new EList node for the LHS, adding the RHS node E_part as a child

```
E_list ::= E_part:e
  {: ASTNode n = new EList();
    n.add(e); RESULT=n; :}
```

Constructing the AST: one more action

- What should the action be for the rule

E_list ::= E_list E_part

- The RHS E_list node already exists
- We simply add the E_part node as the rightmost child

```
E_list ::= E_list:el E_part:ep
  {: el.add(ep);
     RESULT=el; :}
```

- Since E_list is the start symbol, an EList node will be returned from parse()

Doing something with the AST

- parse() returns a reference to the AST's root
- Now the AST can be used for various things:
  - Transform the AST structure for optimization
  - Generate OXML
  - Evaluate the AST in an interpreter
  - Generate code for later execution

Using the AST

- In your assignment, you need to emit a conforming OXML document corresponding to the AST
- We'll use onyx_xml to generate the document

XML representation of the AST

```
<elist>
   <op name="+">
     <number>1</number>
     <number>2</number>
   </op>
   <number>77</number>
   <op name="+">
     <number>3</number>
     <op name="*">
       <number>4</number>
       <number>5</number>
     </op>
   </op>
 </elist>
```

Input: **1+2 ; (77); 3+4*5 ;**

XQueryX

- XQueryX is a standard defining an XML representation for XQuery
- The productions of XQuery syntax are mapped directly onto XML productions
- This representation is not necessarily intended to be convenient for a person to read or write
- But it is easily manipulated electronically
- See the XqueryX document for the details

  http://www.w3.org/2001/06/xqueryx

XQueryX

- Consider the following public test case if.xqy

```
if (true) then 4 else 5
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<onyx.ast.Query>
  <onyx.ast.ExprList>
    <onyx.ast.IfThenElseExpr>
      <onyx.ast.Constant
    datatype="onyx.types.Boolean">true</onyx.ast.Constant>
      <onyx.ast.Constant
    datatype="onyx.types.Integer">4</onyx.ast.Constant>
      <onyx.ast.Constant
    datatype="onyx.types.Integer">5</onyx.ast.Constant>
    </onyx.ast.IfThenElseExpr>
  </onyx.ast.ExprList>
</onyx.ast.Query>
```

Our XQueryX spec

Element	ifThenElseExpr	This is the element type (AST node)
Descendant type	Elements	Tells you the kind(s) of children

Attribute and child element names

any-expression-element
any-expression-element
any-expression-element

Description: The first expression is the conditional clause. This should be a boolean value. The second expression is the then clause and the last expression is the the else clause.

```
<onyx.ast.IfThenElseExpr>
  <onyx.ast.Constant datatype="onyx.types.Boolean">true</onyx.ast.Constant>
  <onyx.ast.Constant datatype="onyx.types.Integer">4</onyx.ast.Constant>
  <onyx.ast.Constant datatype="onyx.types.Integer">5</onyx.ast.Constant>
</onyx.ast.IfThenElseExpr>
```

Attributes

Element	constant
Descendant type	Attributes
Attribute and child element names	datatype

Comment: The value of the datatype attribute should be a valid Onyx type. For constants, the datatype is limited to onyx.types.Integer, onyx.types.Decimal, onyx.types.String, and onyx.types.Boolean.

Descendant type: Text Content
Attribute and descendant element names : constant value
Comment: The constant's value is the content of the element

```
<onyx.ast.IfThenElseExpr>
  <onyx.ast.Constant datatype="onyx.types.Boolean">true</onyx.ast.Constant>
  <onyx.ast.Constant datatype="onyx.types.Integer">4</onyx.ast.Constant>
  <onyx.ast.Constant datatype="onyx.types.Integer">5</onyx.ast.Constant>
</onyx.ast.IfThenElseExpr>
```

Constructing the AST nodes

- 3 possible approaches
- Define a static method e.g. toNode( ) that takes an ASTNode argument
  - method contains logic to figure out the kind of argument node; then...
  - ...construct an element for this node; call toNode( ) recursively with each child; add each child in turn
- Define an instance method e.g. toNode( ) in ASTNode class
  - method can be overridden in subclasses as needed
- Use *visitor pattern*, with double dispatch

Visitor

- A visitor is a design pattern representing "an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates"

  [*Design Patterns, Elements of Reusable Object-Oriented Software,* by Gamma et al., Addison Wesley Publishers]

- See for example

  http://classgen.sourceforge.net/docs/lang6.html (ClassGen)

  http://en.wikipedia.org/wiki/Visitor_pattern

  http://www.javaworld.com/javaworld/javatips/jw-javatip98.html

Visitor

- We create a matrix of functions for each combination of required functionality and AST node type
- These functions are not part of the AST definition
- A visitor class for each type of required functionality
  - Node construction
  - Code generation
  - Interpretation
- There will be more work up front, but the programming will be easier when it comes time to implement the interpreter

Visitor

- The Visitor interface defines a method for each kind of node

```
public interface Visitor {
    public   void   visit( Op o)
    public   void   visit( Elist el  )
    public   void   visit( Num n  )
}
```

- We implement this class for each type of required functionality
- Each kind of node implements the Visitable interface

```
interface Visitable {
    void accept(Visitor visitor);
}
```

The interface

- We add a accept( ) method to the interface of each ASTNode class

```
public class Op extends ASTNode implements Visitable{
    …
    void accept(Visitor visitor)
       { visitor.visit(this); }
```

- The Visitor object is defined elsewhere and it carries out an action like "emit XML," "generate code," etc.
- We create a separate Visitor object for each kind of visitation we wish to perform
- We have factored out the functionality from the ASTNode class and put it in a common place

The visitors

```
public class EmitXMLVisitor implements Visitor{
    …
    public   void   visit( Op o)        { …}
    public   void   visit( Elist el  )   { …}
    public   void   visit( Num n  )    { …}

}
```

- Each of these visitation methods handles one "column" of the matrix
- When we add new functionality, we have to implement a new Visitor class
- When we add a new AST node type, we add a new entry to each Visitor class, and also to the Visitor interface

```
Visitor visitor = new EmitXMLVisitor();
Op.accept(visitor);
```