CS321 Languages and Compiler Design I

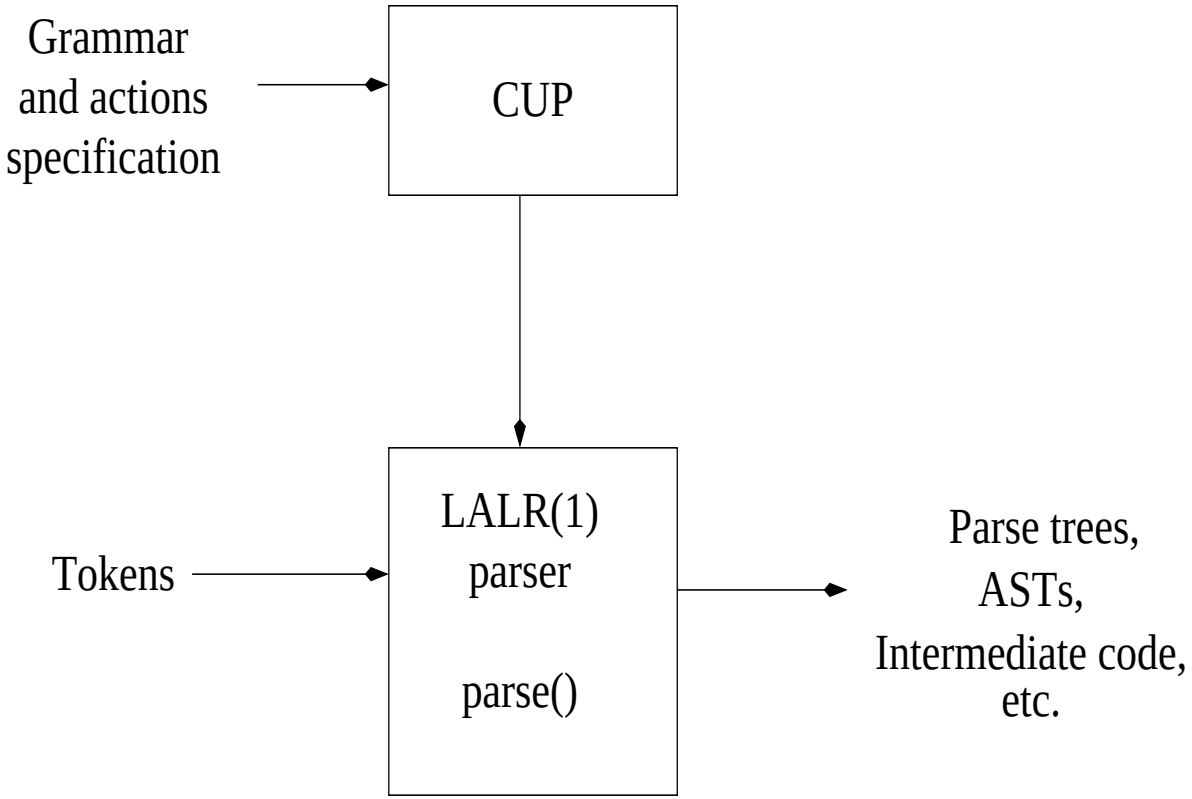
Fall 2010

Lecture 11

CUP Parser Generator

`CUP` = **C**onstructor for **U**seful **P**arsers

(Java variant of `yacc` parser generator for C.)



Grammar: BNF rules

Actions: Java program fragments executed when reduction involving production is made.

CUP File components

- Package specification (optional)
- Import list (optional)
- User Java code fragments (optional)
- Terminal and non-terminal declarations
- Precedence information (optional)
- Start symbol declaration (optional)
- Production rules (BNF) and associated actions (Java code)

CUP EXAMPLE

```
import java_cup.runtime.*;

terminal String ID;
terminal        PLUS,TIMES,LPAREN,RPAREN;
non terminal    e,t,f;

START WITH e;    (declares start symbol)

e ::= e PLUS t
  {: System.out.println("reduce e:e+t"); :}
  | t
  {: System.out.println("reduce e:t"); :}
  ;
t ::= t TIMES f
  {: System.out.println("reduce t:t*f"); :}
  | f
  {: System.out.println("reduce t:f"); :}
  ;
f ::= LPAREN e RPAREN
  {: System.out.println("reduce f:(e)"); :}
  | ID:s
  {: System.out.println("reduce f:ID " + s); :}
  ;
```

What CUP Generates

To run CUP on a spec file `foo.cup`:

```
java -classpath java-cup-11a.jar java_cup.Main < foo.cup
```

or:

```
java -jar java-cup-11a.jar < foo.cup
```

(where you can substitute the location of your CUP jar file in the -classpath or -jar).

By default, CUP generates:

- a file `parser.java` containing a class `parser` with constructor

```
public Parser(java_cup.runtime.Scanner s);
```

and method

```
public java_cup.runtime.Symbol parse()
    throws java.lang.Exception;
```

- a file `symbol.java` containing a class `symbol` that defines enumeration values for each of the terminals.

USING CUP (CONTINUED)

The `Parser` constructor should be passed an object of some lexical analyzer class that implements the interface:

```
interface java_cup.runtime.Scanner {
  public java_cup.runtime.Symbol next_token()
         throws java.lang.Exception;
}
```

Note that the set of terminal tokens is derived from the parser spec, but must be used correctly by the scanner.

The parser and lexer need to be linked with the library package `java_cup.runtime`.

All of this glue mechanism can be customized; for the **fab** project, you should use the framework provided with the assignment.

EXPRESSING (E)BNF IN CUP

BNF production $A \rightarrow \alpha | \beta$ is written:

```
A ::= α     {: action for A → α :}
  | β       {: action for A → β :}
  ;
```

Constructing lists, e.g., `idlist` $\rightarrow$`ID{,ID}` :

- Left-recursion is most efficient:

```
idlist ::= ID
       |   idlist COMMA ID
       ;
```

- Right-recursion also works:

```
idlist ::= ID
       |   ID COMMA idlist
       ;
```

- Lists with 0 or more items are easy:

```
list ::=
     | list item
     ;
```

CUP CONFLICTS

Recall that ambiguous grammar can have shift-reduce and reduce-reduce conflicts, e.g., input `ID + ID + ID` with grammar

```
e ::= e PLUS e
  |  ID
  ;
```

When parser has seen `ID + ID`, it can either:

- **shift** next `+`, reaching `ID + ID + ID`, and then reduce rightmost `ID + ID`, producing final result `ID + (ID + ID)`; or
- **reduce** `ID + ID` to `e` before reading next `+`, producing final result `(ID + ID) + ID`.

By default, `CUP` handle shift/reduce conflicts by **shifting**. This often gives the desired effect, so having shift/reduce conflicts in grammar is considered "ok."

`CUP` handles reduce-reduce conflicts by reducing with rule listed **first** in grammar. This is seldom what you want, so having reduce/reduce conflicts in grammar is considered "bad style."

CUP Precedence & Associativity

To get non-default behavior you can give CUP explicit precedence and associativity info for any token and/or any grammar rule.

For tokens, associativity is specified by `precedence left` or `right` declarations, and precedence is specified by the order of the declarations (highest precedence last). E.g.:

```
precedence left PLUS, MINUS;
precedence left TIMES, SLASH;
precedence right UPARROW;
```

Precedence/associativity of **rules** is normally given by that of rightmost terminal:

```
e ::= e PLUS e      rule has prec/assoc of  PLUS
  |   e TIMES e     rule has prec/assoc of  TIMES
  ;
```

How declarations work

On shift/reduce conflicts, CUP **shifts** if the input symbol has higher precedence than the reduction rule, **reduces** if symbol has lower precedence, and uses rule associativity to choose if precedences are equal.

Examples:

Parse Stack	Input	Action
`e PLUS e`	`TIMES`	SHIFT
`e TIMES e`	`PLUS`	REDUCE `e` $\rightarrow$ `e TIMES e`
`e PLUS e`	`PLUS`	REDUCE `e` $\rightarrow$ `e PLUS e`
`e UPARROW e`	`UPARROW e`	SHIFT

With above declarations, can use ambiguous grammar directly:

```
e ::= e PLUS e | e MINUS e | e TIMES e
  | e SLASH e  | e UPARROW e
  | LPAREN e RPAREN | ID ;
```

CUP UNARY OPERATORS

Sometimes want a single operator to have different associativity or precedence in different rules. E.g., want minus symbol (`MINUS`) to have higher precedence when used as a unary operator than when used as a binary operator.

`CUP` allows you to set the precedence of a rule directly by adding a `%prec` qualifier to it. Unary minus is then handled by defining a "pseudo-terminal" for it, with appropriate precedence.

```
terminal UNARYMINUS;   (pseudo-token declaration)
precedence left PLUS, MINUS;
precedence left TIMES, SLASH;
precedence left UNARYMINUS;
precedence right UPARROW;

e ::=  e PLUS e | e MINUS e | e TIMES e | e SLASH e
  | MINUS e  %prec UNARYMINUS
       (give rule prec/assoc of  UNARYMINUS rather than of '-')
  | e UPARROW e | LPAREN e RPAREN | ID
  ;
```

SYNTAX-DIRECTED TRANSLATION

Use **grammatical structure** of language to guide translation into lower-level form.

Traverse **parse tree** (constructed or virtual) evaluating **semantic rules**.

Semantic rules ("attribute equations"):

- Assign **values** to **attributes** attached to nodes of parser tree.

Examples: type or value of expression; code for statement block.

- Perform side-effects on global state.

Examples: make entries in symbol table; issue errors; generate code to output file.

Attributes are pieces of information (any kind!) attached to **nodes** of a grammar-induced tree.

Semantic rules are associated with grammar **productions**, because each tree node is "built" by a production. (Terminal nodes are assumed to have their attributes "at the beginning.")

Collectively, semantic rules make up an **attribute grammar**.

ATTRIBUTE EVALUATION

Attribute grammars can be used with a parse tree (real or virtual) or an abstract syntax tree.

Evaluation order of semantic rules may or may not follow **reduction** order during parser: depends on form of rules.

Computing attribute values is called **annotating** or **decorating** the tree.

If used with parse tree, often try to compute attribute values **while parsing**. Sometimes, attributes are more important than parse tree itself, e.g., can use attribute grammar on **parse** trees to compute **AST** as an attribute!

More complicated attribute equations may require whole tree to exist first, before attribute evaluation begins.

An attribute is:

- "**synthesized**" if its value at a node depends only on values of attributes of **descendents** of that node; or
- "**inherited**" if its value at a node depends only on the values of attributes of **ancestors** and/or **siblings** of that node.

SYNTHESIZED ATTRIBUTES ON PARSE TREES

Synthesized attribute values at a non-terminal node depend only on values at node's **children**. Values at terminal nodes are provided by lexical analyzer.

Example: desk calculator ("run-time" actions)

$S \rightarrow E$	`print (`E`.val)`
$E \rightarrow E_1$ `+` E_2	E`.val := `E_1`.val +` E_2`.val`
$E \rightarrow E_1$ `*` E_2	E`.val := `E_1`.val *` E_2`.val`
$E \rightarrow$ `(`E_1`)`	E`.val := `E_1`.val`
$E \rightarrow I$	E`.val := `I`.val`
$I \rightarrow I_1$ `digit`	I`.val := 10 *` I_1`.val + digit.lexval - '0'`
$I \rightarrow$ `digit`	I`.val := digit.lexval - '0'`

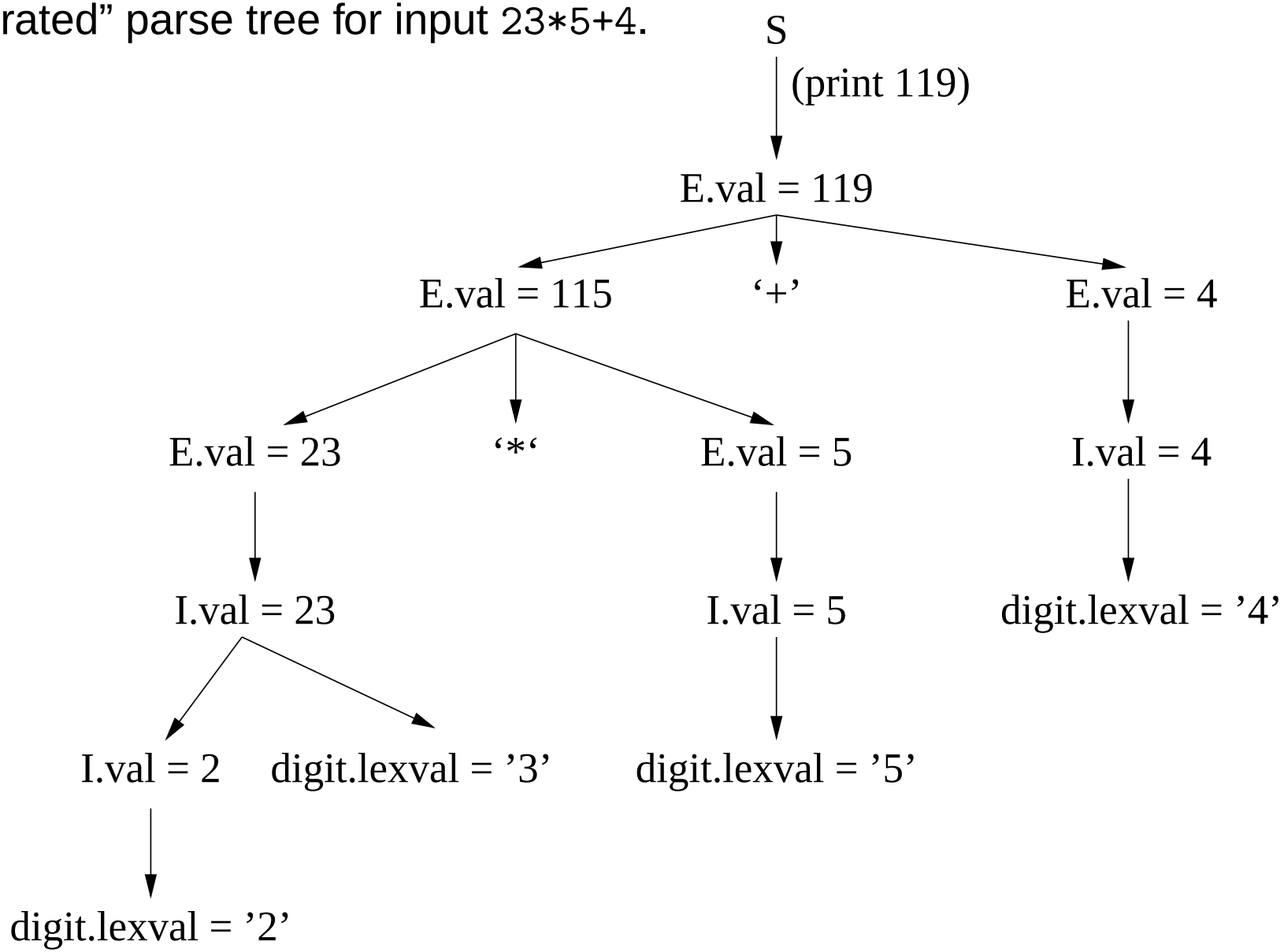
Attributes can be evaluated **bottom-up**.

Evaluation can be done while parsing (either top-down or bottom-up).

When parsing bottom-up, at time of a reduction all attribute values on RHS are known, so LHS can be computed.

EXAMPLE

"Decorated" parse tree for input `23*5+4`.



SEMANTIC STACK METHOD

Implements **synthesized** attribute evaluation in **bottom-up** shift-reduce parser.

Semantic stack is manipulated in parallel with parser stack.

When a terminal is **shifted** onto parser stack, its attributes are pushed onto semantic stack.

Before a **reduction** $A \rightarrow \alpha_1 \alpha_2 \ldots \alpha_k$, the top k values on semantic stack are attributes for RHS.

After the reduction, the top value is the synthesized attribute for LHS non-terminal.

Parse stack and semantic stack always have equal depths. Even when a grammar symbol has no useful attribute, there is a placeholder for it on the semantic stack.

Example (Desk Calculator)

Parse Stack	Semantic Stack	Input	Action
		23*5+4$	shift
digit	'2'	3*5+4$	reduce $I \rightarrow$ digit
I	2	3*5+4$	shift
I digit	2 '3'	*5+4$	reduce $I \rightarrow I$ digit
I	23	*5+4$	reduce $E \rightarrow I$
E	23	*5+4$	shift
E *	23 _	5+4$	shift
E * digit	23 _ '5'	+4$	reduce $I \rightarrow$ digit
E * I	23 _ 5	+4$	reduce $E \rightarrow I$
E * E	23 _ 5	+4$	reduce $E \rightarrow E$*E
E	115	+4$	shift
E +	115 _	4$	shift
E + digit	115 _ '4'	$	reduce $I \rightarrow$ digit
E + I	115 _ 4	$	reduce $E \rightarrow I$
E + E	115 _ 4	$	reduce $E \rightarrow E$+E
E	119	$	reduce $S \rightarrow E$
S	_	$	accept

CUP CALCULATOR

```
terminal PLUS,TIMES,LPAREN,RPAREN;
terminal Character digit;       (this token has a Character attribute)
non terminal Integer S,E,I;     (these non-terminals have Integer attributes)

S ::= E:e                  {: System.out.println(e); :}
  ;
E ::= E:e1 PLUS E:e2       {: RESULT = e1 + e2; :}
  | E:e1 TIMES E:e2        {: RESULT = e1 + e2; :}
  | LPAREN E:e RPAREN      {: RESULT = e; :}
  | I:i                    {: RESULT = i; :}
  ;
I ::= I:i digit:d          {: RESULT = 10 * i + Character.digit(d,10); :}
  | digit:d                {: RESULT = Character.digit(d,10); :}
  ;
```

The type of the attribute can be different for each terminal and non-terminal, and must be declared in the `CUP` specification.

Items on the semantic stack must be **objects**, e.g. instances of class `Integer` or `Character`; they can't be bare `int`s or `char`s. The action code here is relying on automatic wrapping and unwrapping of these types.

CUP Value (Semantic) Stack

`CUP`-generated parser automatically maintains **values** as well as parser states on its parsing stack `stack`, with top-of-stack pointer `top`.

The parsing stack contains `Symbol` objects, each of which has a field

```
Object value;
```

`Symbol` objects can represent either terminals or non-terminals.

For `Symbol` objects representing terminals (obtained from the lexical analyzer) the `value` field is the "attribute" set by the lexer, e.g., the string associated with an ID or the value of an INTEGER literal.

On a **shift**: a lexer-generated `Symbol` is pushed on the parse stack.

On a **reduce**: user **action** code is executed with the labels bound to the `value` fields of symbols in the handle (near the top of the stack). The symbols in the handle are then popped from the stack, and a new `Symbol` (representing the LHS non-terminal) is pushed, with its `value` field is set to the `RESULT` specified in the action.

Note: `Symbol` objects also carry source-file position information (`left` and `right` fields) which is automatically propagated during reduction steps.

EXAMPLE FROM DESK CALCULATOR

```
E ::=  E:e1 PLUS E:e2
  {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
```

(Here the wrapping and unwrapping between `Integer` and `int` as been made explicit.)

Suppose this rule is reduced when the `value` fields of the symbols near the top of stack look like this:

`E:e1`	`PLUS`	`E:e2`
v_1	v_2	v_3

Then `e1` is bound to v_1, `e2` is bound to v_3, and the action is executed, producing a new symbol with `value = RESULT`. So the action is roughly equivalent to:

```
Symbol s = new Symbol();
s.value = new Integer(
              stack.elementAt(top-2).intValue() +
              stack.elementAt(top-0).intValue());
stack.pop(3); stack.push(s); top -= 2;
```