

CS321 F'04 Lecture Notes
Lecture 11

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"); : }
  ;

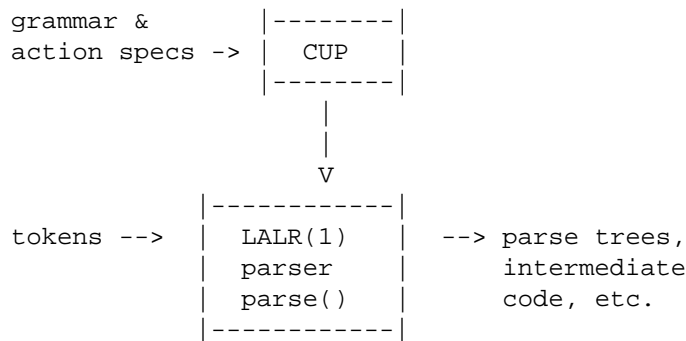
t ::= t TIMES f
  { : System.out.println("reduce t:t*f"); : }
  | f
  { : System.out.println("reduce t:f"); : }
  ;

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

CUP Parser Generator

CUP = Constructor for Useful Parsers

(Java variant of yacc parser generator for C.)



Grammar: BNF rules

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

General input format:

- 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)

What CUP Generates

To run CUP on a spec file foo.cup:

```
java -classpath /u/cs321-01/CUP \
     java_cup.Main < foo.cup
```

(where you can substitute the root of your CUP installation in the classpath).

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 PCAT project, you should use the framework provided on the web page.

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 `ID` 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.”

To get non-default behavior you can give CUP explicit precedence and associativity info.

Expressing (E)BNF in CUP

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

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

Constructing lists, e.g., `idlist → 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 Precedence & Associativity

Explicit precedence and associativity can be given for each token and/or each grammar rule.

For tokens, associativity is specified by `precedence left` or `precedence 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
    ;
```

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.

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/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 right ASGN;
precedence left PLUS, MINUS;
precedence left TIMES, SLASH;
precedence left UNARYMINUS;
precedence right UPARROW;
```

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

Attribute Evaluation

Attribute grammars can be used with parse tree (real or virtual) or 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.

Example: 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 **descendants** 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.

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**.

Synthesized attributes on Parse Trees

Attribute values at 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 → E          print (E.val)
E → E1 + E2  E.val := E1.val + E2.val
E → E1 * E2  E.val := E1.val * E2.val
E → (E1)     E.val := E1.val
E → I        E.val := I.val
I → I1 digit  I.val := 10 * I1.val
                    + digit.lexval - '0'
I → digit     I.val := digit.lexval - '0'
```

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

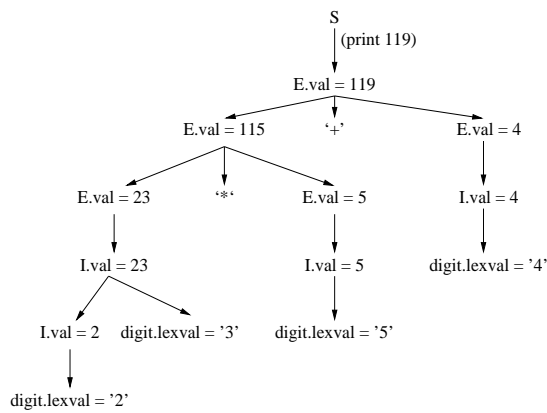
Evaluation can be done while parsing (either top-down or bottom-up).

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

Syntax-directed definitions that use **only** synthesized attributes are called **S-attributed definitions**.

Example

“Decorated” parse tree for input 23*5+4.



Note: Same parse tree and attribute evaluation pattern would hold for **static** attributes, such as expression type, code sequence, etc.

Example (Desk Calculator)

Parse Stack	Semantic Stack	Input	Action
		23*5+4\$	shift
digit	'2'	3*5+4\$	reduce $I \rightarrow \text{digit}$
I	2	3*5+4\$	shift
I digit	2 '3'	*5+4\$	reduce $I \rightarrow I\text{digit}$
I	23	*5+4\$	reduce $E \rightarrow I$
E	23	*5+4\$	shift
E *	23 _	5+4\$	shift
E * digit	23 _ '5'	+4\$	reduce $I \rightarrow \text{digit}$
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 $I \rightarrow \text{digit}$
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

Note that parse stack and semantic stack always have equal depths.

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

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

CUP Calculator

```
terminal Character digit;
    (this token has Character attributes)
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 = new Integer(e1.intValue() +
                        e2.intValue()); :}
| E:e1 TIMES E:e2
{: RESULT = new Integer(e1.intValue() *
                        e2.intValue()); :}
| LPAREN E:e RPAREN {: RESULT = e; :}
| I:i      {: RESULT = i; :}
;
I ::= I:i digit:d
{: RESULT = new Integer(
    10*i.intValue() +
    Character.digit(d.charValue(),10)); :}
| digit
{: RESULT = new Integer(
    Character.digit(d.charValue(),10)); :}
;
```

The types of the value fields can be different for each terminal and non-terminal, and must be declared in the CUP specification.

This example is awkward because the items on the semantic stack must be **objects** (i.e., they can't be bare integers or characters), so we have to use “wrapper classes.”

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.

Example from desk calculator

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

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

<code>E:e1</code>	<code>PLUS</code>	<code>E:e2</code>
<code>v₁</code>	<code>v₂</code>	<code>v₃</code>

Then `e1` is bound to `v1`, `e2` is bound to `v3`, 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;
```

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