Lecture 7

- An introduction to bottom-up parsing
- An introduction to the CUP parser generator
- Grammar rules and actions

Today's readings

- Text
  Ch. 4: §4.5
- CUP User's Manual
  http://www.cs.princeton.edu/~appel/modern/java/CUP/

Parsers for programming languages

- In practice, parsers for programming languages are not universal CF parsers (those are too slow)
- Instead we use restricted "top-down" or "bottom-up" type CF parsers
- If you ever have to write a parser completely by hand, you probably will write a *top down recursive descent predictive parser*
  – We will look at how to do that later
- But now we will look at *bottom-up shift-reduce parsers* and tools for building them

Industrial-strength parsing

- The industry standard approach uses **bottom-up shift-reduce** parsers
- These are more powerful (they can handle more grammars) than top down recursive descent predictive parsers
- But they are complex and quite hard to write correctly by hand...
- ...so parser generator tools are essentially always used to create them

Parser generator tools

- We can automate the parser-writing process with the help of a parser generator tool
- This is a good thing; compilers in general are tricky to write by hand
- (The original FORTRAN compiler took *18 person-years* to write, without tools... and XQuery is much more complicated than FORTRAN)
- The best-known parser generator is *yacc* (Yet Another Compiler Compiler), written in 1970's at Bell Labs; CUP is basically a Java-centric *yacc*
- These tools generate LALR parsers, an efficient powerful type of bottom-up shift-reduce parser

Shift reduce parsing

- Shift reduce parsing is *bottom-up* parsing: it traces a parse tree for an input string bottom up, starting with leaves and working up to the root
- Leaves in the parse tree correspond to tokens in the input; the root is the grammar's start symbol
- Think of this as a process of "***reduction***" of an input string to the start symbol of the grammar
  - reduction is the reverse of derivation

Reduction

- Each reduction step:
  - Replaces a substring of a sentential form
  - ... which matches the RHS of some production
  - ...with the LHS of the production

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

$$aAbcde$$

Reduction

- Each reduction step
  - Replaces a substring of a sentential form
  - ... which matches the RHS of some production
  - ...with the LHS of the production

$$
\begin{aligned}
S &\rightarrow aABe \\
B &\rightarrow d \\
A &\rightarrow Abc \mid b
\end{aligned}
$$

$aAbcde$

Reduction

- Each reduction step
  – Replaces a substring of a sentential form
  – ... which matches the RHS of some production
  – ...with the LHS of the production

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

$$aAbcde$$

Reduction

- Each reduction step
  - Replaces a substring of a sentential form
  - ... which matches the RHS of some production
  - ...with the LHS of the production

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

$aAde$

Example

- Consider the following grammar

$$
\begin{aligned}
S &\rightarrow aABe \\
A &\rightarrow Abc \mid b \\
B &\rightarrow d
\end{aligned}
$$

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

Example

- 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: 1

$abbcde$

Example

- 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: 2

$abbcde \quad aAbcde$

Example

- 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: 3

$abbcde \quad aAbcde \quad aAde$

Example

- 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: 4

abbcde *aAbcde* *aAde* *aABe*

Example

- 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: 5

$abbcde \quad aAbcde \quad aAde \quad aABe \quad S$

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 just shown trace out this derivation in reverse:

  $S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$
- So either reduction or derivation shows that a particular string is a sentence in the language defined by the grammar, and has a corresponding parse tree

Implementing shift-reduce parsing

- Issues:
  - Often several reductions seem possible to apply to a given sentential form; how do you pick one to perform next?
  - Can you be sure you've picked the right one, so you won't have to backtrack later?
  - OK, I basically see what reduce is, what about shift?
  - How do you implement this all efficiently?
- See Algorithm 4.7 in the text; we'll cover it later
- For now let's look briefly at how to use a parser generator

Using a parser generator

- Step 1: write down a context-free grammar
- Step 2: give it to a parser generator and have it write a parser for you
- Done! ... Actually there is usually a little more to it than that
- For one thing, you are usually interested in more than just "did the input parse or not"
- You also want the parser to take appropriate actions: do actual computations, create nodes in an abstract syntax tree, print out syntax error messages, or whatever is appropriate
- These actions are specified along with the grammar rules you give to the parser generator

Parsing and other actions

- Consider a simple calculator language
- Sentences are expressions involving `+` and `*`, parenthesis, and terminated by a semicolon, e.g:

  ```
  5 * (1+2);
  ```
- We would like to know if a given expression is syntactically correct; but we also want to know its value, e.g.: `15`
- We can do both at the same time: while reducing subexpressions bottom-up, compute their values with actions attached to grammar rules

Syntax directed translation

- Attaching actions to grammar rules lets you implement the powerful technique of *syntax-directed translation*
- The action code associated with a rule will be executed when the rule is used in the parsing process
- The action code can cause construction of an intermediate representation, such as an abstract syntax tree
- Or the action code can directly execute the intended semantics of the input text, as in the following calculator example

Grammar for the calculator

- Terminals (i.e. tokens): `SEMI, PLUS, TIMES, LPAREN, RPAREN, number`
- Nonterminals: `E_list, E_part, E, T, F`
- Start symbol: `E_list`

```
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
```

Constructing a CUP specification

- All terminals must be declared

```
terminal  SEMI, PLUS, TIMES,
          LPAREN, RPAREN, number;
```

- All nonterminals must also be declared

```
non terminal E_list, E_part;
non terminal E, T, F;
```

- The left-hand side of the first grammar rule is taken to be the start symbol
- Now we can write the grammar in CUP syntax (we'll attach actions to grammar rules later)

A CUP 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
           ;
```

CUP writes a `parser` class

- Give all that to CUP, and it will produce a definition of a class `parser`
- That class defines a constructor that takes an object that implements the Scanner interface:
  `public parser(java_cup.runtime.Scanner s)`
  ... a parser object will use that Scanner object to read tokens
- The class also defines an instance method that performs the parse and returns a resulting `Symbol` object for the root of the parse tree:
  `public java_cup.runtime.Symbol parse()`

CUP writes a `sym` class

- CUP will also produce a definition of a class `sym`
- That class defines integer ID's for all the terminal symbols declared in the CUP spec
- Those integer ID's must be known to the scanner, so it can return Symbol objects that have their `int sym` instance variables set appropriately
- The parser and scanner better use the same `sym` class! Otherwise, the parser won't know what the scanner is talking about

Running a CUP parser

- Now if you have a CUP-compatible scanner defined in a class named, say, `Lexer`, you can parse an input file `f` with (ignoring file encoding issues):

```
(new parser(
        new Lexer(
            new FileReader("f")))).parse();
```

- But we would like also to have the parser perform actions while it is parsing
- This requires attaching action code to grammar productions
- Action code is delimited by `{: :}`

Toward CUP actions

- CUP makes use of `java_cup.runtime.Symbol` objects
- As you know, a CUP-compatible scanner creates and returns `Symbol` objects to the parser to represent tokens it has scanned
- The scanner can make the `value` instance variable of the `Symbol` point to the value of the token
  - since `value` is of type Object, this value can be anything
- The scanner also sets the `sym` instance variable of the `Symbol` to the int ID of the token
- And: CUP also creates one new `Symbol` object itself, whenever it reduces according to a rule

Symbol class

- Recall the Symbol class has members something like this:

```
public class Symbol {

 public Object value;  // value of the token
 public int sym;      // ID of the token
 public int line;    // line # of token in input
 public int col;     // col # of token in input

 // various constructors...
}
```

CUP actions and Symbols

- A CUP parser creates a new `Symbol` whenever it reduces according to a rule
  - this new `Symbol` corresponds to the symbol on the LHS of the rule
  - the symbols on the RHS of the rule already have `Symbol`s that have been created for them: They are either terminals, or already reduced nonterminals
- In a CUP action, the `Symbol.value` fields of all the symbols in the rule are accessible in action code associated with the rule
- This is a very powerful mechanism; you can compute arbitrary things and pass them up the parse tree

Syntax for referring to Symbol values

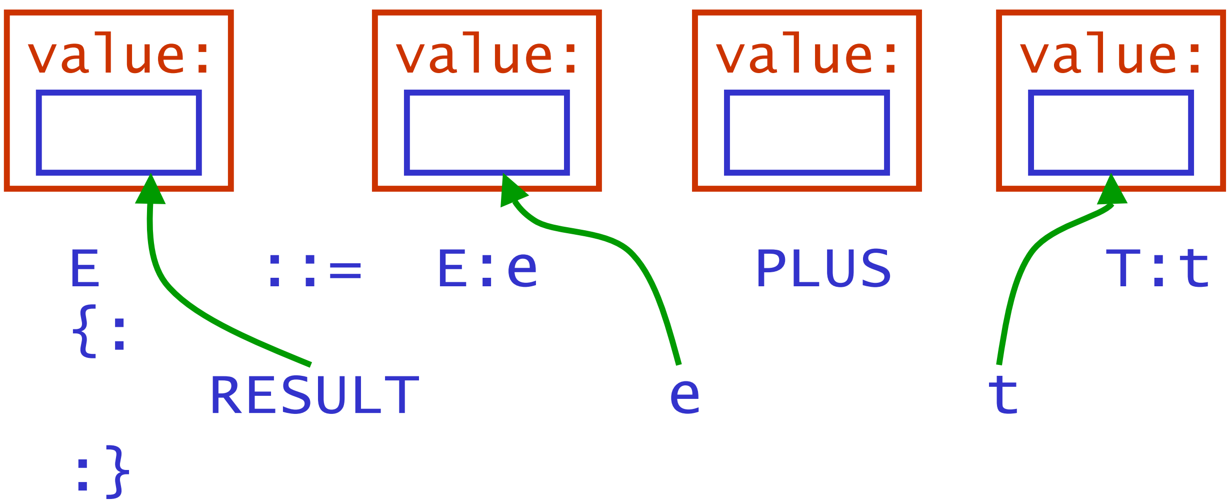
- In action code attached to a rule, you can refer to `Symbol` objects for all the symbols in the rule

- If a symbol on the RHS of the rule is followed by a colon `:` and an identifier, then that identifier refers to the `value` field of that symbol's `Symbol` object

- The predefined identifier `RESULT` refers to the `value` field of the rule's LHS `Symbol` object

Grammar symbols, Symbols, and actions

- In this rule, `E`, `T`, `PLUS` are grammar symbols; `E`, `T` on RHS are declared to have identifiers `e`, `t` associated with them
- When the rule is reduced, `Symbol` objects will exist as shown and their `value` fields can be referred to with the identifiers shown in the attached action code



Symbol value types and references

- The `value` instance variable of a `Symbol` is of type `Object`, so it can point to an object of any type
- In action code, you could access particular features of the value by first casting the corresponding identifier to the value's "true" type
- A better idea: CUP permits declaring the Java types of terminals and nonterminals; then CUP will automatically generate the casts for you

Symbol value type declarations

- Supose when the lexer scans a `number` token, it returns a `Symbol` with `value` field pointing to an `Integer` containing the value of the number:

```
terminal SEMI,PLUS,TIMES,LPAREN,RPAREN;
terminal Integer number;
```

- We will compute and propagate `Integer` values up the parse tree; so declare `E,T,F` as `Integer` also:

```
non terminal E_list, E_part;
non terminal Integer E, T, F;
```

- Now we can write actions that operate on named value fields, including `RESULT`

Adding some actions

- Intuitively what should happen when this rule is used to reduce?:

```
E ::= E:e PLUS T:t
```

- If other actions are written appropriately, the RHS `E` and `T` symbols will already have Integer values computed for them
- We want to add those values, and make the result the value of the LHS `E` symbol. So:

```
E      ::=   E:e   PLUS   T:t
{: RESULT = new Integer(
     t.intValue()+ e.intValue( )  ); :}
```

Adding more actions

- Intuitively what should happen when this rule is reduced?:

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

- `number` is a terminal symbol; we assume the scanner sets the `value` field of the `Symbol` object it returns to be an Integer representing the value of the number
- We want to just pass that value up the parse tree so it can be used when other rules are reduced. So:

```
F ::=  number:n
       {: RESULT = n; :}
       ;
```

Adding yet more actions

- Intuitively what should happen when this rule is used in a reduction?:

```
E_part ::= E:e  SEMI  ;
```

- The `SEMI` token is the semicolon marking end of an expression; when this rule is used to reduce, it means a complete expression has been parsed, and (if actions are written appropriately) the value of the expression is in the `E` symbol
- We want to print out that value. So:

```
E_part ::= E:e  SEMI
            {: System.out.println(e); :}
            ;
```

A CUP calculator grammar w/actions

```
E_list ::= E_list E_part | E_part;
E_part ::= E:e  SEMI
          {: System.out.println(e); :}
          ;
E         ::= E:e PLUS T:t
          {: RESULT = new Integer(
         t.intValue()+ e.intValue( )); :}
          |  T:t
          {: RESULT = t; :}
          ;
```

Grammar with actions

```
T       ::= T:t TIMES F:f
        {: RESULT=new Integer(
  t.intValue() * f.intValue()); :}
        |  F:f {: RESULT = f; :} ;

F       ::= LPAREN E:e RPAREN
        {: RESULT=e; :}
        |  number:n
        {: RESULT=n; :}
        ;
```

The result

- Now whenever the rule `E_part ::= E` is reduced, the value of the symbol is printed; this value is an Integer that has been computed and propagated up the parse tree in the appropriate way
- So for example if a file containing `5 * (1+2); 5 * 1+2; 5*5 *5;` is parsed, the output will be

```
15
7
125
```

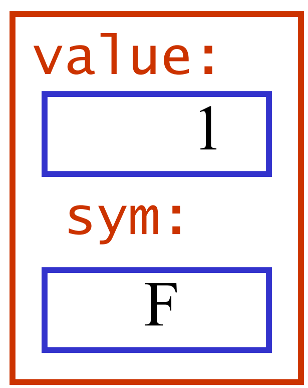
The calculator in action

- Let's look at individual steps in the parsing/action process in more detail, for a simple case
- Suppose the input character stream to the lexer is:
  `1 + 2 ;`
- Then sequence of Symbols from the lexer will be:

`value:`	`value:`	`value:`	`value:`	`value:`
1		2		
`sym:`	`sym:`	`sym:`	`sym:`	`sym:`
number	PLUS	number	SEMI	EOF

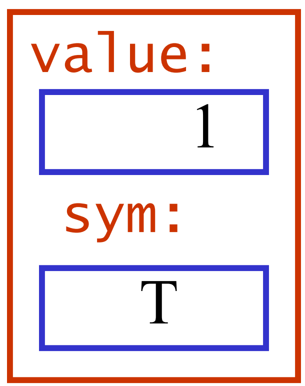
The calculator in action, step 1

- The parser will shift the first token, and reduce using the rule
  `F ::=  number:n   {: RESULT=n; :}    ;`
- The reduction creates a new `Symbol` object for the LHS `F`, filling in the `value` from the token `Symbol`



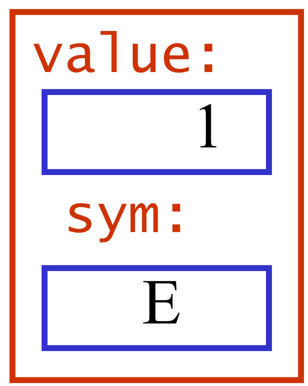
The calculator in action, step 2

- The parser will next reduce using the rule
  `T ::=  F:f    {: RESULT=f; :}    ;`
- The reduction creates a new `Symbol` object for the LHS `T`, filling in the `value` from the `F` `Symbol`



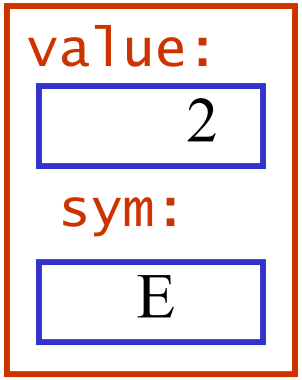
The calculator in action, step 3

- The parser will next reduce using the rule
  `E ::=  T:t    {: RESULT=t; :}    ;`
- The reduction creates a new `Symbol` object for the LHS `E`, filling in the `value` from the `T` `Symbol`



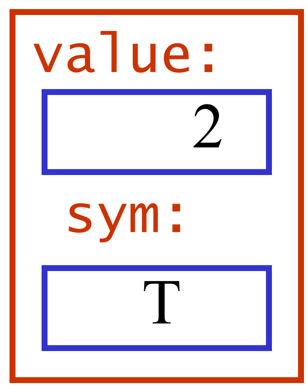
The calculator in action, step 4

- The parser will now shift two more tokens, and then reduce using the rule
  `F ::=  number:n   {: RESULT=n; :}    ;`
- The reduction creates a new `Symbol` object for the LHS `F`, filling in the `value` from the token `Symbol`



The calculator in action, step 5

- The parser will next reduce using the rule
  `T ::=  F:f    {: RESULT=f; :}     ;`
- The reduction creates a new `Symbol` object for the LHS `T`, filling in the `value` from the `F` `Symbol`

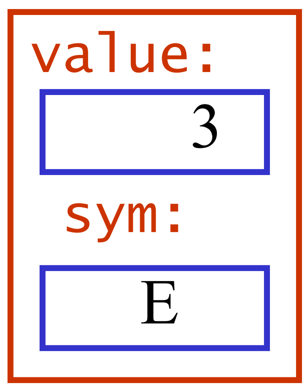


The calculator in action, step 6

- Now the parser can reduce using the rule

```
E      ::= E:e PLUS T:t
          {: RESULT = new Integer(
         t.intValue()+ e.intValue( )); :} ;
```

- The reduction creates a new `Symbol` object for the LHS `E`, filling in the `value` from the computation

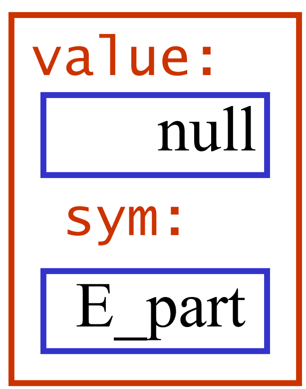


The calculator in action, step 7

- Next the parser will shift the last token, and reduce using the rule

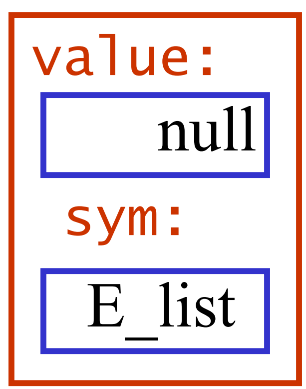
```
E_part ::= E:e  SEMI
     {: System.out.println(e); :} ;
```

- The action here prints the `value` from the `E Symbol`. A `Symbol` is created for the LHS, but its `value` field remains null



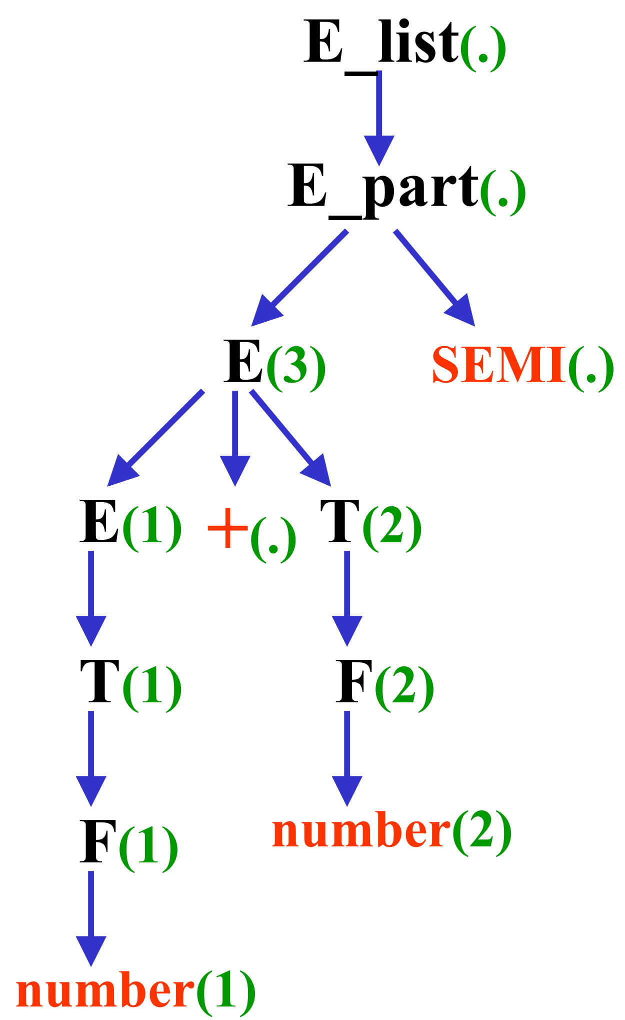
The calculator in action, step 8

- Seeing EOF next, the parser will reduce to the start symbol by the rule
  `E_list ::= E_part ;`
- There is no action specified. A `Symbol` is created for the LHS, but its `value` field remains null. This `Symbol` is returned from the `parse()` method.



The parse tree

- Shown is the parse tree on input
  `1 + 2 ;`
- Next to each grammar symbol is shown the `value` field of the corresponding `Symbol` object, in parens `()`
- `(.)` means null



For next time

- Shift-reduce parsing algorithms and data structures, textbook section 4.5