

CSE 131A
“Compiler Construction I”

Lecture 7

Bottom up parsing

Announcements

- In need of a partner?
- Groups have been set up for users listed in [~/../public/Tools/groups/groups_\[1-3\]](#)
- The reference lexer is available via the web portal at:
<http://ieng6.ucsd.edu:6606/lexer.jsp>
- More public tests coming (errors)

A1 Grading

- Grade reports will be sent by email
- If the autograder fails to run any of your tests, contact us
- If there were systematic errors, contact us; you won't be penalized for every occurrence of the same error
- 10% penalty (against highest possible score) for not following instructions
- Be sure to complete the TeamEval with meaningful information

Bug fixing

- How do I figure out how to fix the bugs in my code?
- The feedback you get as part of your grade will tell you the parts of the language that weren't scanned correctly
- With the help of the reference lexer, construct test cases that demonstrate the modes of failure (as well as success) in your program
- In the “real world,” there may not be an oracle telling you whether or not your implementation is correct

Undefined behavior

- This is an important real world issue
- When a certain behavior is left unspecified, the implementer is given free license to determine how to handle it
- Therefore, you will not be tested on undefined behavior
- This generally pertains to special cases

Handling character entities

- You are responsible for handling the 5 character entities in your parser `&`; `>`; ...
- The scanner converts character entities to the corresponding symbol: `&` `>`
"`&`; `&` `<`; `<` `""` `"`; `>`; `>`"
- Onyx_xml will convert all the special symbols back to entities

`<token ...>& & < < " " > ></token>`

Back to parsing

The Parser

- Performs syntactic analysis
- Builds a hierarchical structure (**parse tree** or **abstract syntax tree**) from the tokens, according to the specification of a context free grammar
- Works in concert with the scanner
- The parser also interacts with other modules including **error recovery**, **symbol table manager**, and **code generation**

Derivations

- **Terminals** and **nonterminals**
- **Derivation:** starting with a sequence of grammar symbols, replace one nonterminal with its definition
- **Rightmost** derivation
- **Sentences** and **sentential forms**

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E \text{ OP } E) \Rightarrow -(E \text{ OP } \text{id}) \Rightarrow -(E + \text{id}) \Rightarrow -(\text{id} + \text{id})$

$E \rightarrow E \text{ OP } E \mid (E) \mid - E \mid \text{id}$

$\text{OP} \rightarrow + \mid - \mid * \mid / \mid ^$

$\text{expr} \rightarrow \text{expr op expr}$

$\text{expr} \rightarrow (\text{expr})$

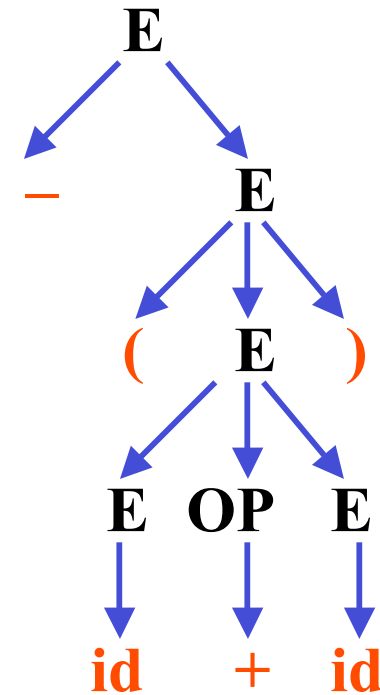
$\text{expr} \rightarrow - \text{expr}$

$\text{expr} \rightarrow \text{id}$

$\text{op} \rightarrow + \mid - \mid * \mid / \mid ^$

Parse trees

- We may view a parse tree as a representation of a derivation
- The root of a parse tree is the start symbol
- Any internal node is a nonterminal symbol; the children are the symbols on the right-hand-side of a production for the parent
- Sentential forms have parse trees whose leaves contain terminals and non-terminals
- What can we say about the leaves of a parse tree for a sentence in the grammar's language?

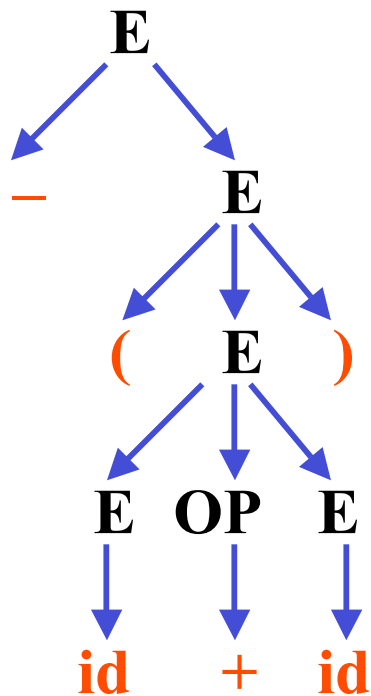


$$E \rightarrow E \text{ OP } E \mid (E) \mid - E \mid \text{id}$$
$$\text{OP} \rightarrow + \mid - \mid * \mid / \mid ^$$

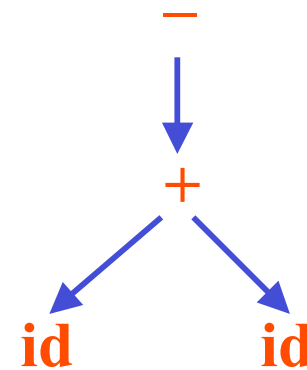
Parse tree vs AST

input: $-(\text{id} + \text{id})$

parse tree



abstract syntax tree



An example for home

- Consider the parse tree for the following grammar

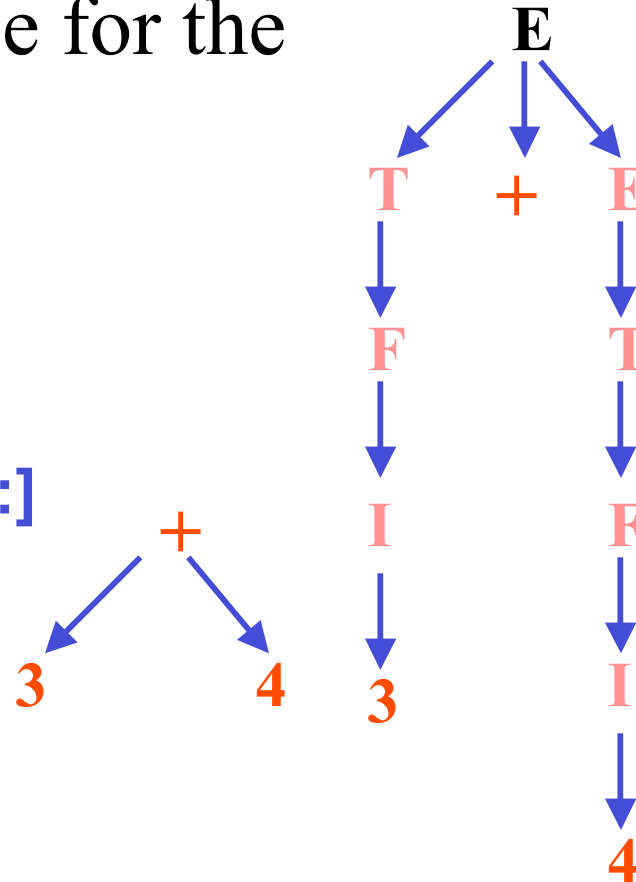
$E \rightarrow T \mid T + E$

$T \rightarrow F \mid F * T$

$F \rightarrow \text{tknInteger}$

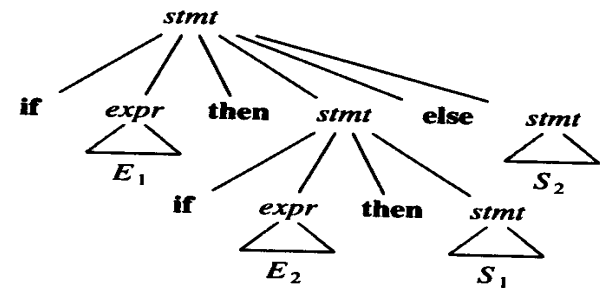
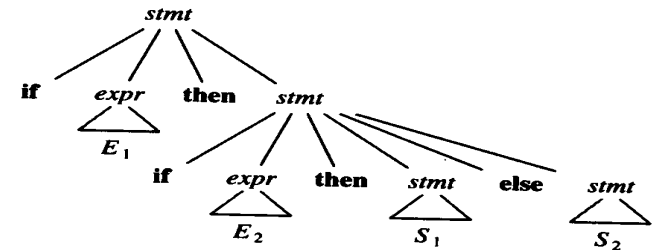
$\text{tknInteger} \rightarrow [:\text{number}:]$

- Derive the AST



Ambiguity

- A grammar that produces more than one parse tree from a given sentence is said to be **ambiguous**
- Equivalent definition: there is more than one distinct leftmost (or rightmost) derivation of the same sentence
- Problem: different parse trees assign different structure to the input, leading to different interpretations of the same program text
- Grammars for programming languages should be unambiguous



“Dangling Else” Ambiguity

- Example: The infamous dangling else problem
statement → **if** "(" *expr* ")" **then** *statement* **else** *statement*
statement → **if** "(" *expr* ")" **then** *statement*
- Given those productions, is the following an **if** with a contained **if-else**, or an **if-else** with a contained **if**? The semantics could be very different!

if (x<0) **then** **if** (y<0) z=0; **else** z=100000;

Expression Ambiguity

- Another example of ambiguity arises in our example expression grammar
- There are two ways of parsing **id + id * id** corresponding to two different groupings

id + (id * id) or **(id + id) * id**

- Can lead to different results

Eliminating Ambiguity

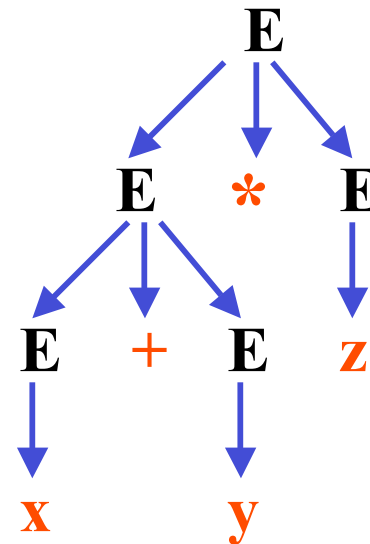
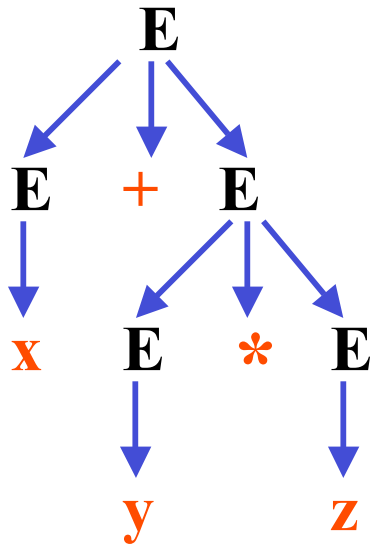
- Ambiguity cannot always be eliminated
 - some CFLs are inherently ambiguous
 - Not possible to determine if a CFG is ambiguous or not!
- But it is sometimes possible to eliminate ambiguity, and construct an equivalent grammar

Ambiguity in expression grammars

- Consider the simple expression grammar

$$E \rightarrow E + E \mid E * E \mid x \mid y \mid z$$

- This grammar is ambiguous; for example the string $x + y * z$ has two parse trees:



Writing expression grammars

- By convention, we want $*$ to apply to its operands first: $*$ has higher precedence than $+$
- Some grammars (“precedence grammars”) permit us to attach information about precedence to productions to fix ambiguities like this
- Alternatively...
 - Rewrite the grammar, introducing new nonterminals ...
 - placing high-precedence operation productions to be ...
 - farther from the start symbol and nearer to terminal symbols
- Avoids changing the language

Ambiguity in expression grammars

- Consider this improved expression grammar

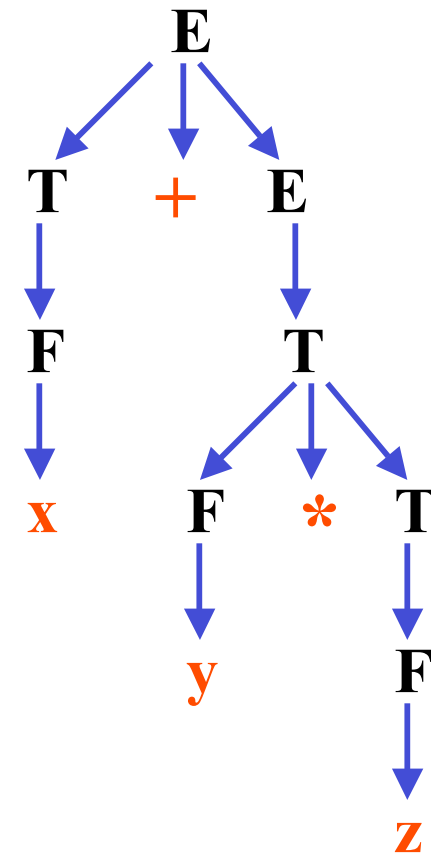
$$E \rightarrow T \mid T + E$$

$$T \rightarrow F \mid F * T$$

$$F \rightarrow x \mid y \mid z$$

$$E \rightarrow E + E \mid E * E \mid x \mid y \mid z$$

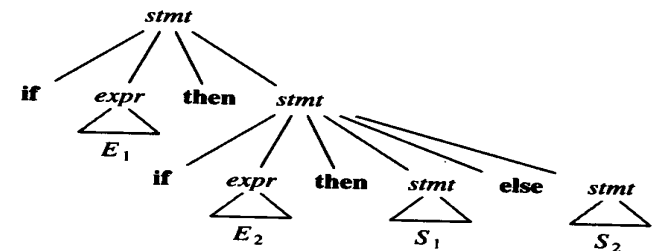
- Defines the same language
- It is unambiguous;
the string $x + y * z$
has only one parse tree, as shown



Fixing the “dangling else” ambiguity

- Consider the “dangling else” grammar

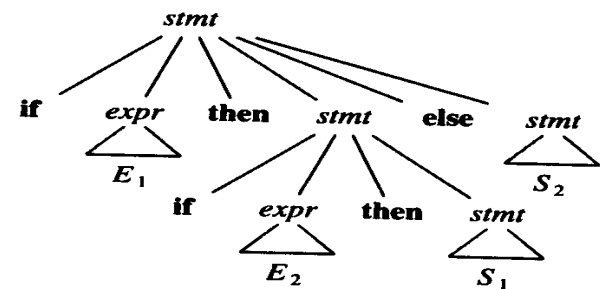
$stmt \rightarrow$ **if** *expr* **then** *stmt*
 | **if** *expr* **then** *stmt* **else** *stmt*
 | **other**



- And this statement:

if E_1 **then** **if** E_2 **then** S_1 **else** S_2

- There are two different parse trees corresponding to this statement



Eliminating ambiguity

- We usually prefer the first interpretation:
“Match each **else** with the closest previous unmatched **if**”
- Let’s write a new grammar
- Idea: We want a statement appearing between a **then** and an **else** to already have all its **if**'s matched with **else**'s (Otherwise the outside **else** would match an unmatched contained **if**)
- We call such a statement that already has all its **if**'s matched with **else**'s a “matched statement”
- We can introduce a new non-terminal called *matched_stmt*

A new grammar to avoid ambiguity

- A matched statement has all **ifs** matched with **elses**

$$\begin{aligned} \textit{matched_stmt} \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{matched_stmt} \\ & \textbf{else } \textit{matched_stmt} \\ & | \textit{other} \end{aligned}$$

- Unmatched statements don't have all their **ifs** matched with **elses**

$$\begin{aligned} \textit{unmatched_stmt} \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\ & | \textbf{if } \textit{expr} \textbf{ then } \textit{matched_stmt} \\ & \textbf{else } \textit{unmatched_stmt} \end{aligned}$$

- And a statement is either matched or unmatched

$$\begin{aligned} \textit{stmt} \rightarrow & \textit{matched_stmt} \\ & | \textit{unmatched_stmt} \end{aligned}$$

- The grammar is now unambiguous –and defines the same language!

A unique parse

- The statement **if** E_1 **then** **if** E_2 **then** S_1 **else** S_2 now has a unique parse tree, corresponding to this unique leftmost derivation:

$stmt \rightarrow unmatched_stmt \rightarrow \mathbf{if\ expr\ then\ stmt}$

$\rightarrow \mathbf{if\ } E_1 \mathbf{\ then\ } stmt \rightarrow \mathbf{if\ } E_1 \mathbf{\ then\ } matched_stmt$

$\rightarrow \mathbf{if\ } E_1 \mathbf{\ then\ if\ } expr \mathbf{\ then\ } matched_stmt \mathbf{\ else\ } matched_stmt$

$\rightarrow \mathbf{if\ } E_1 \mathbf{\ then\ if\ } E_2 \mathbf{\ then\ } matched_stmt \mathbf{\ else\ } matched_stmt$

$\rightarrow \mathbf{if\ } E_1 \mathbf{\ then\ if\ } E_2 \mathbf{\ then\ } other \mathbf{\ else\ } matched_stmt$

$\rightarrow \mathbf{if\ } E_1 \mathbf{\ then\ if\ } E_2 \mathbf{\ then\ } S_1 \mathbf{\ else\ } matched_stmt$

$\rightarrow \mathbf{if\ } E_1 \mathbf{\ then\ if\ } E_2 \mathbf{\ then\ } S_1 \mathbf{\ else\ } other$

$\rightarrow \mathbf{if\ } E_1 \mathbf{\ then\ if\ } E_2 \mathbf{\ then\ } S_1 \mathbf{\ else\ } S_2$

Parsing in practice

Parsing in practice

- In practice we use **bottom-up shift-reduce** parsing
- More powerful than top-down recursive descent parsers (more on these later), which are written by hand
- Difficult to write by hand
- We can automate the parsing process with the help of a parser generator tool
- The best known parser generator is YACC (Yet Another Compiler Compiler), written in 1975 at Bell Labs; CUP is basically a Java-centric YACC
- These tools generate LALR parsers, an efficient and powerful type of bottom-up shift-reduce parser

Shift reduce parsing

- A shift reduce parser constructs a parse tree for an input string from the 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 may think of this as a process of “reducing” an input string to the start symbol of the grammar
- Contrast with a top down parser, which begins with the start symbol and ends with the input

Reduction

- Each reduction step
 - Replaces a substring of the input
 - Matching 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 the input**
 - Matching 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 the input**
 - Matching the **RHS of some production**
 - With the LHS of the production

$$S \rightarrow aABe$$

$$B \rightarrow d$$

$$A \rightarrow \mathbf{Abc} \mid b$$

$$a\mathbf{Abc}de$$


Reduction

- Each reduction step
 - Replaces a **substring of the input**
 - Matching 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

$$S \rightarrow aABe$$

$$A \rightarrow Abc \mid b$$

$$B \rightarrow d$$

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

Example

- The grammar

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

- Step: 1

abbcde

Example

- The grammar

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

- Step: 2

$abbcde \quad aAbcde$

Example

- The grammar

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

- Step: 3

$abbcde \quad aAbcde \quad aAde$

Example

- The grammar

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

- Step: 4

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

Example

- The grammar

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

- Step: 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 string is a sentence in the language defined by the grammar, and has a corresponding parse tree

Implementation

- How do we pick the appropriate reduction (out of many possible) to apply to a sentential form?
- How do we know it's the right one (avoid backtracking)
- How do we shift
- How do we implement this?
- See Algorithm 4.7 in text (more coming later)

Using a parser generator

- CUP will generate code that implements the parser
- The parser not only tells us whether the input parsed correctly..
- It also generates the parse tree
- CUP doesn't take care of all the details
 - We have to decide how to represent the parse tree
 - We have to write the parser actions to generate the tree
 - Syntax error messages
- These actions are specified along with the grammar rules

Actions with parsing

- Recall the simple calculator language
- Inputs are expressions involving $+$ and $*$, parenthesis, and terminated by a semicolon

$5 * (1+2);$

- We would like to know more than just whether or not an expression in this language is syntactically correct; we also want to know its value: **15**
- We can do both at the same time: while parsing subexpressions bottom-up, computing their values with actions attached to reductions

Syntax directed translation

- Attaching actions to grammar rules enables us to implement a powerful technique known as *syntax-directed translation*
 - The action code associated with a rule runs when the rule is used in the parsing process
 - It can lead to construction of an intermediate representation, such as an AST
 - Or it can directly execute the intended semantics of the input text, as in the following calculator example

Grammar for the calculator

- All symbols must be declared
 - SEMI, PLUS, TIMES, LPAREN, RPAREN, number are terminal symbols corresponding to tokens returned by a lexer
 - E_list, E_part, E, T, F are nonterminals
 - E_list is the start symbol

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;
- By convention the LHS of the first grammar rules is taken to be the start symbol
- Now we can write the grammar in CUP syntax
- We'll return to actions a little later

Under the hood of CUP

- CUP generates a definition of class `parser`
`public class parser extends java_cup.runtime.lr_parser`
- See `public/Tools/CUP/src/java_cup/runtime` for the definition
`public abstract class lr_parser`
- `parser` defines a constructor that takes an object implementing the `Scanner` interface
`public parser(java_cup.runtime.Scanner s)`
... a `parser` object will use that to read tokens
- The class 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()`

The `sym` class

- CUP also generates a definition of class `sym`
- `sym` defines integer IDs for all the terminal symbols declared in the CUP specification
- The parser and scanner must agree on the IDs
- Otherwise, they won't understand each other

Actions

- If you have a CUP-compatible scanner defined in a class named, say, `Lexer`, you can parse an input file `f` simply with

```
(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 delimited by `{: :}` to grammar rules

Toward CUP actions

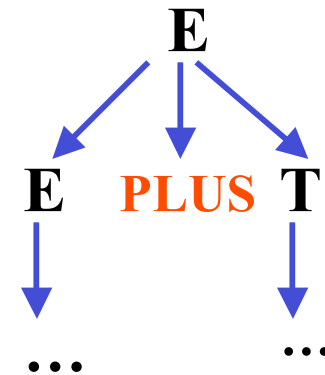
- CUP uses `java_cup.runtime.Symbol` objects
- The scanner returns `Symbols` to the parser, which contain information about tokens it has found
- The scanner stores the value of the token in the `value` instance variable of `Symbol`
- Since `value` is of type `Object`, it can be anything
- The scanner also sets the `sym` instance variable to the ID of the token

```
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  
}
```

CUP actions and Symbol values

- Conceptually, a bottom up parser creates a new parse tree parent node whenever it performs a reduction
 - this new node corresponds to a **Symbol** on the LHS of the rule
 - the (already existing) children of this node correspond to symbols on the RHS of the rule: either terminals, or previously reduced nonterminals

E ::= E:e PLUS T:t
{: RESULT=e + t; :}



CUP actions and Symbol values

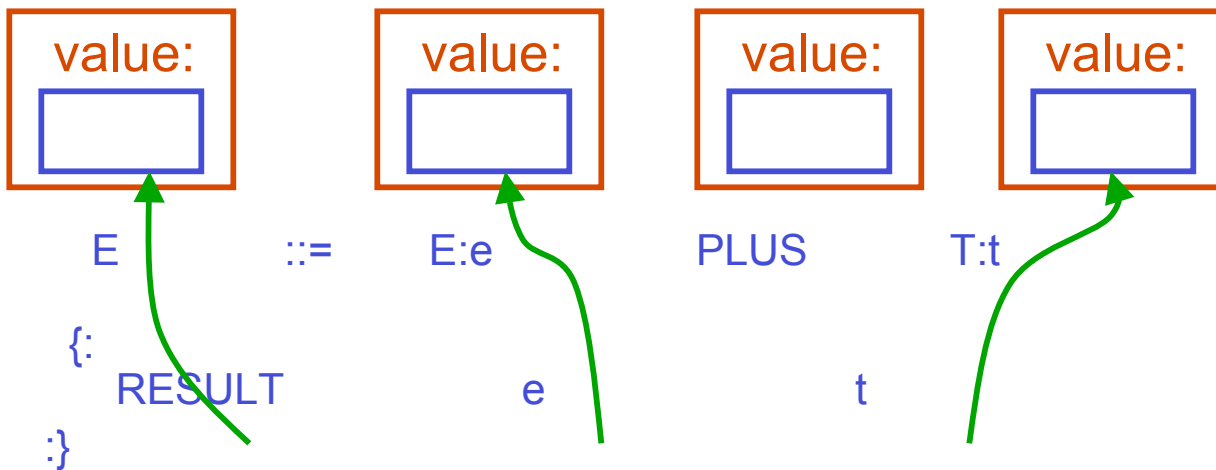
- In an action, the `Symbol.value` fields of all the symbols in the rule are accessible in action code associated with the rule
- This is a very powerful; we can compute arbitrary things and pass them up the parse tree
- We declare identifiers to refer to `value` fields of `Symbols` in the RHS of a rule, using the notation `$\alpha:v$`
- The predefined identifier `RESULT` refers to the `value` field of the rule's LHS

E ::= E:e PLUS T:t

{: RESULT= e + t; :}

Symbols, values, and actions

- In this rule, **E**, **T**, **PLUS** are grammar symbols; **E**, **T** on RHS 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

- The value instance variable of a **Symbol** is of type **Object**, so it can point to an object of any type
- Suppose the lexer scans a **number** token, returning a **Symbol** with **value** field holding to an **Integer**
terminal **Integer number;**
- We will compute and propagate **Integer** values up the parse tree; so declare **E, T, F** as **Integer** also
non terminal **Integer E, T, F;**
- We can now write actions that operate on named value fields, including **RESULT**
- Why aren't all variables typed?
non terminal **E_list, E_part;**
terminal **SEMI, PLUS, TIMES, LPAREN, RPAREN;**

Calculator grammar w/actions

- What should happen when we want to reduce according to the rule?

$E ::= E:e \text{ PLUS } T:t$

- E and T already have associated integer values
- We add those values
- Assignment to **RESULT** corresponds to assigning value assigned to F 's value field
- Allows us to pass values up the tree

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

Another action

F ::= number:n ;

- **number** is a terminal symbol
- We assume that the scanner has associated an Integer value with the corresponding **Symbol** object
- Assignment to **RESULT** corresponds to assigning value assigned to **F**'s value field
- This result is passed up the parse tree so that it can be used in other reductions

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

The calculator in action

- Whenever we find a semicolon, a complete expression has been parsed
- We have reached the end of an expression
`E_part ::= E:e SEMI ;`
- The value is the `E` symbol
- The results in the reduction `E_part ::= E`
- The value of the symbol `E` is printed: an `Integer` that has been computed and propagated up the parse tree in the appropriate way

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

Results

- So for example if a file containing

$5 * (1+2);$

$5 * 1+2; 5 * 5 * 5;$ is parsed..

the output will be

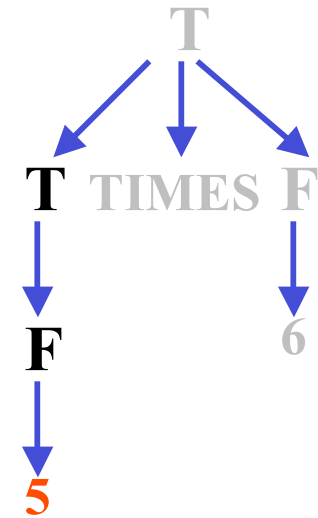
15

7

125

How did it work?

- Let's parse the expression **5 * 6**;
- Read in the first token **5**
- Reduce the **5** to a **Factor** with the rule:
 $F ::= \text{NUMBER}:n$
 $\{ : \text{RESULT} = n; : \};$
- Create a new **Symbol** object **F** with value field **5**
- Reduce to a **Term** using using the rule:
 $T ::= F:f \{ : \text{RESULT} = f; : \};$
- Create a new **Symbol** object **T** copying **F**'s **value** field

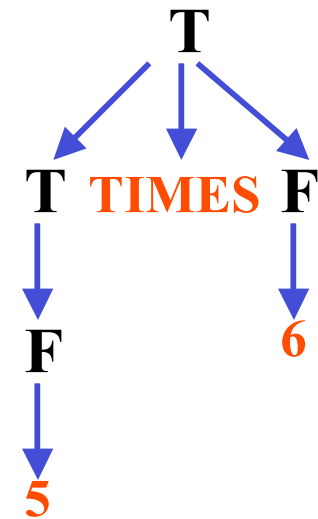


How did it work?

- Shift the token `*` (no action)
- Shift the token `6`
- Reduce the `6` to a factor with the rule:

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

- Create a new **Symbol** object **F** with value field `6`

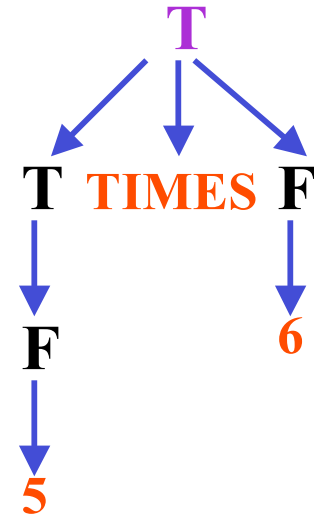


How did it work?

- Reduce the product of a **F**actor and a **T**erm

```
T ::= T:t TIMES F:f  
{: RESULT = t * f;  
:}
```

- Create a new **S**ymbol object **T** with **value** field containing the product



Reaching the end

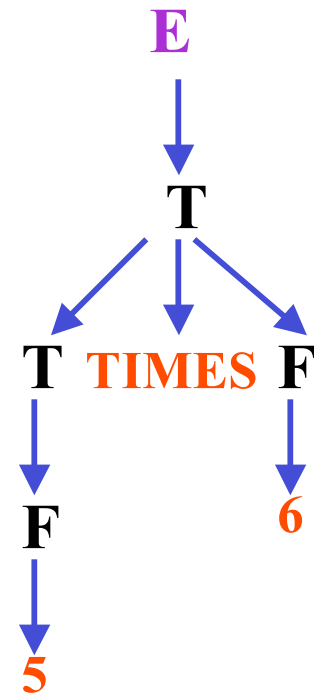
- Reduce the term to an expression

$E ::= T:t$
{: RESULT = t; :} ;

- Create a new **Symbol** object **E** copying **T**'s **value** field

- We next encounter a semicolon

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



How did it work?

- We finally encounter **EOF**
- We reduce to the start symbol

E_list ::= E_part ;

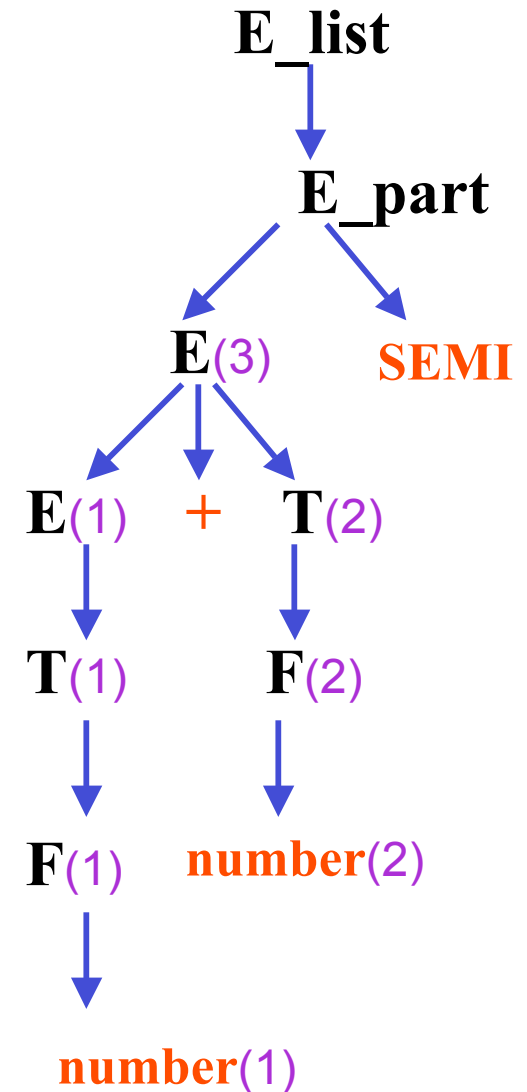
- There is no action
- The returned **Symbol** field is empty

A parse tree

- Parse tree for input

1+2 ;

- Each grammar symbol is labeled by the **value** field in ()
- Nulls aren't shown



A few words about notation

- We don't have the $*$, $+$, and $?$ operators
- How do we express a list with 1 or more elements?

$list ::= NUM \mid list \text{ COMMA } NUM;$

- How do we express a list with 0 or more elements?

$list ::=$
 $\quad \quad \quad \mid list \text{ COMMA } NUM;$

- How about $?$

An example

- Look in `~cs131w/./public/Examples/DC`
- The main directory has the basic calculator
- Unary shows you how to define a unary minus operator
- WithConflict shows an example of shift/reduce conflicts (more on this next time)
- You can tolerate conflicts (in order to find other bugs) with the `-expect` flag .. see the CUP documentation
- Modify the DC example code `TestParser.java` to build a parse tree