CSI4104 Compiler Design
Fall 2018

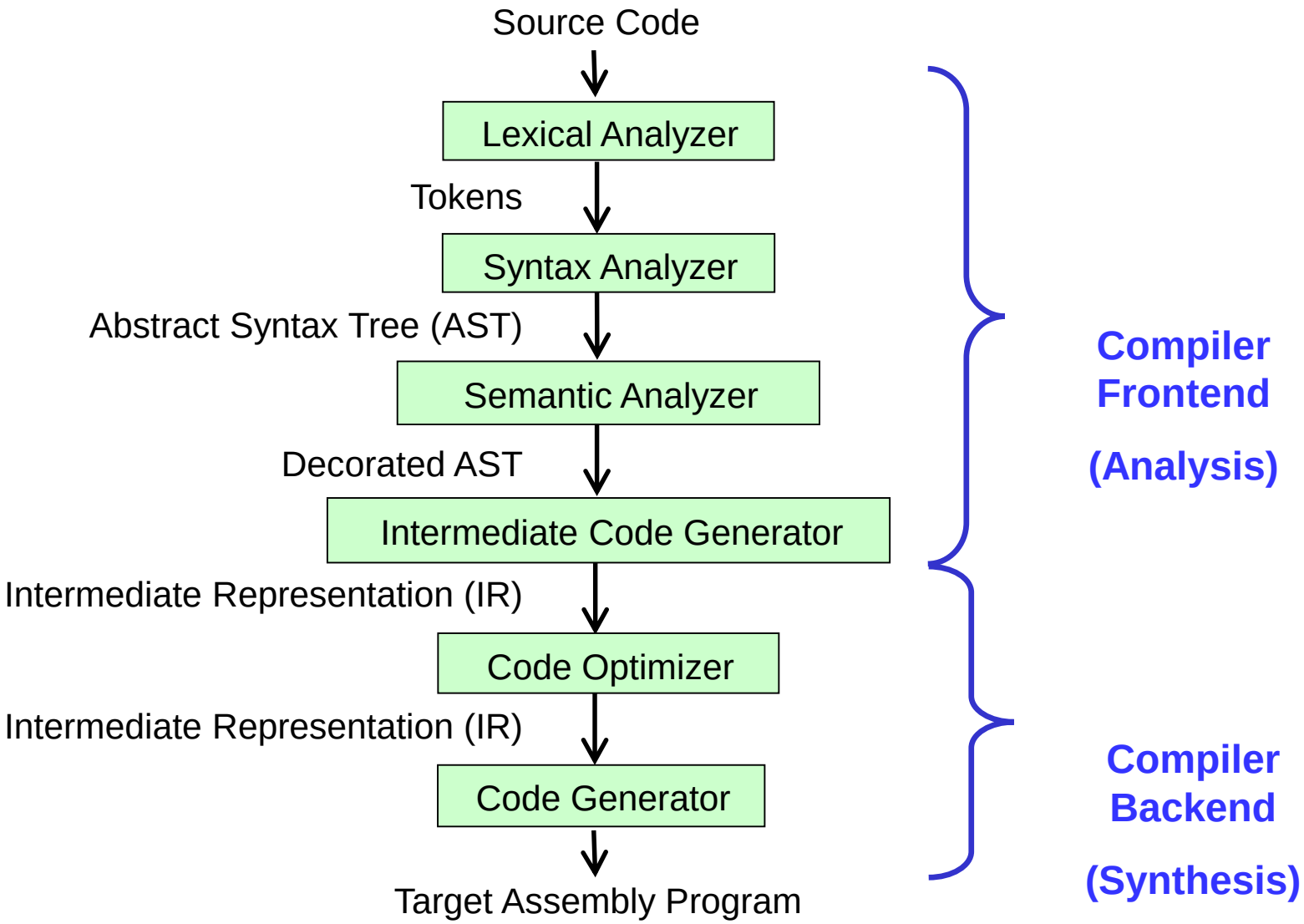
Syntax Analysis

Bernd Burgstaller
Yonsei University

Outlook

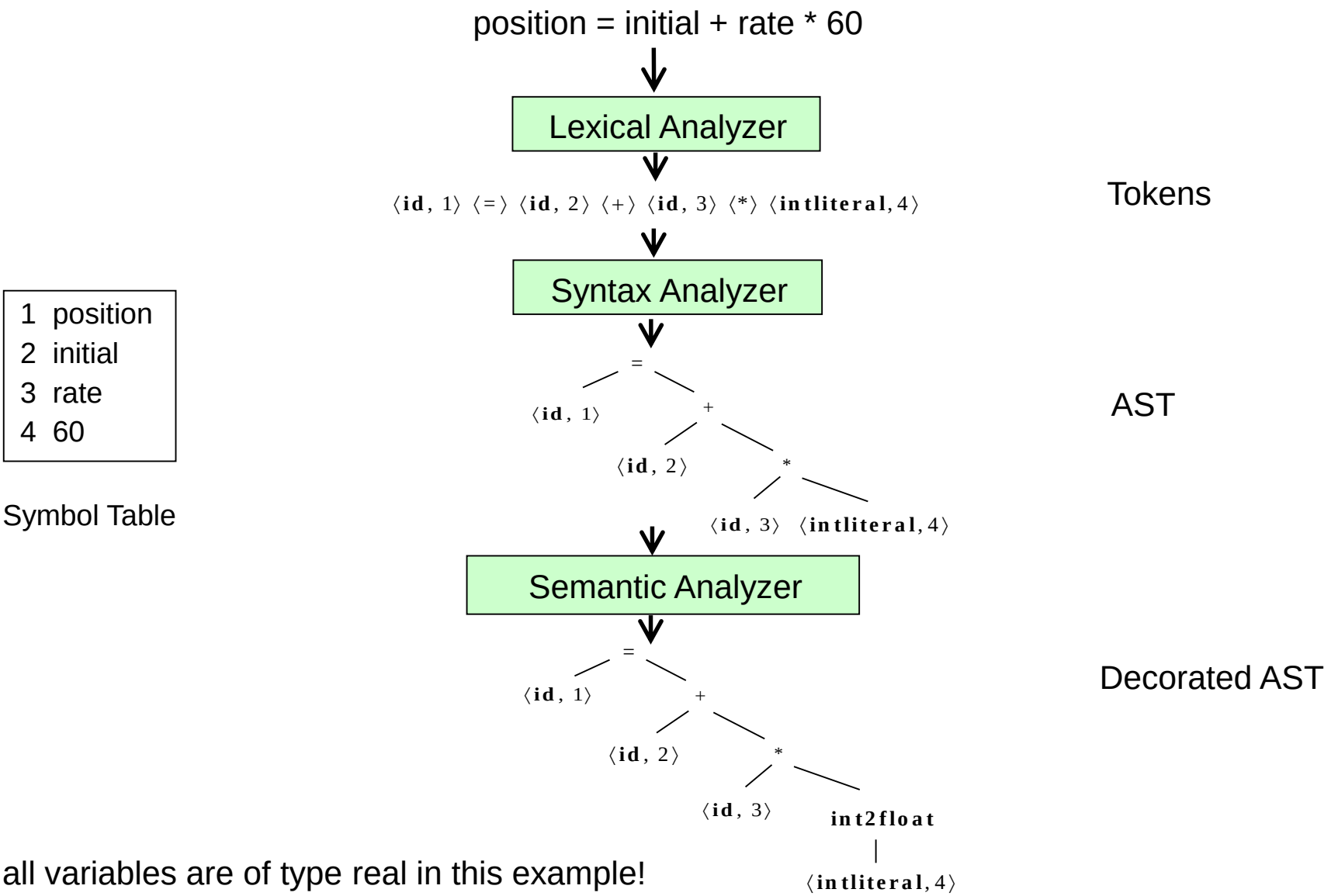
- Syntax and Semantics of Programming Languages
- Specifying the Syntax of a programming language:
  - CFGs, BNF and EBNF
  - Grammar transformations
- Parsing
  - Top-down parsing (LL) vs. bottom-up parsing (LR)
  - Recursive descent (LL) parser construction
  - LL Grammars
- Parser Generators
- AST Construction
  - Parse trees vs. ASTs
- Chomsky's Hierarchy

Recapitulate: the structure of a compiler...



…each phase transforms the program from one representation to the next…

Recapitulate: the structure of a compiler...



Note: all variables are of type real in this example!

English Language: Syntax and Semantics

- John eats apples.
- Apples eat John.

- Syntax:
  - The form or structure of English sentences
  - Not concerned with the meaning of English sentences
  - Specified by the English grammar.

- Semantics:
  - The meaning of English sentences
  - Semantic definition?

Programming Languages: Syntax and Semantics

- `b = b + 1;`
- `if (b==2) printf(“too small\n”);`

- Syntax:
  - The form or structure of programs
  - Not concerned with the meaning of programs
  - Specified by a context-free grammar (CFG)

- Semantics:
  - The meaning of programs.
  - Specified by
    - operational, denotational or axiomatic semantics
    - attribute grammars (will be covered by this course)
    - informal English descriptions (as with C, Java, MiniC)

Context-free Grammars (CFGs)

Example: CFG for micro-English:

Sentence	::=	Subject Verb Object .
Subject	::=	I | a Noun | the Noun
Object	::=	me | a Noun | the Noun
Noun	::=	cat | mat | rat | ε
Verb	::=	like | is | see | sees

Context-free grammars consist of productions of the form

<nonterminal> ::= sequence of (non) terminals

where

- a terminal of a grammar is a token,
- the symbol "|" denotes alternative forms in a production,
- the symbol ε denotes the empty string.

Context-free Grammars

Example: CFG for micro-English:

Sentence	::=	Subject Verb Object .
Subject	::=	I | A Noun | The Noun
Object	::=	me | a Noun | the Noun
Noun	::=	cat | mat | rat | ε
Verb	::=	like | is | see | sees

A derivation shows how to generate a syntactically valid string.

Example: Sentence → Subject Verb Object .
→ I Verb Object .
→ I see Object .
→ I see the Noun .
→ I see the cat .

The non-terminal to be replaced next is underlined in this example.

Context-free Grammars

Example: CFG for micro-English:

Sentence	::=	Subject Verb Object .
Subject	::=	I | A Noun | The Noun
Object	::=	me | a Noun | the Noun
Noun	::=	cat | mat | rat | ε
Verb	::=	like | is | see | sees

How many sentences can be derived in micro-English? Just a few, i.e., finitely many.

Other sentences in micro-English:

I like the cat.
The cat like the mat. // unfortunately not good English...
The mat is the rat. // syntactically valid string.

Not included in the micro-English language: (this grammar doesn't have this adjectives)
I like the black cat. // micro-English does not provide adjectives.

Derivations

- From a grammar we can derive strings (a.k.a. sentences) by generating sequences of tokes.
- In each derivation step, a nonterminal is replaced by a right-hand side of a production for that nonterminal.
  - The strings of terminals and nonterminals appearing in the various derivation steps are called sentential forms.
  - A sentence is a sentential form with terminals only.
- The final sentential form consists of terminals (tokens) only.
- A context-free grammar is a generator of a context-free language: the language defined by the grammar is the set of all strings (or sentential forms, programs) that can be derived.
  - A sentence that cannot be derived is syntactically illegal!

Context-free Grammars

- A context free grammar specifies **all valid strings (a.k.a. programs)** of a given programming language!
- → Formal description of the syntax of a programming language.

- Elements of CFGs:
  - Terminal Symbols (Terminals)
    - Example: '<=', 'while', 'if', '>', '==', ID, INTLITERAL
  - Nonterminal Symbols (Nonterminals)
    - Particular class of phrases in the language
    - Example: Program, Command, Expression, Declaration, …
  - Start Symbol
    - One of the nonterminals, usually the lefthand-side of the first production.
    - Example from micro-English: sentence → subject verb object
  - Productions
    - Example: sentence ::= noun verb

Context-free Grammars

CFGs can be expressed in BNF (Bachus-Naur Form)

To recognize P. Naur's contribution as editor of the Algol60 report and J. W. Backus for applying the notation to the first FORTRAN compiler.

Example in BNF:

```
Program ::= Command
Command ::= single-Command
          |  Command ; single-Command
single-Command
        ::= V-name := Expression
          |   begin Command end
          |   ...
```

Btw EBNF no regex

Context-free Grammars

For our convenience, we will use EBNF or "Extended BNF" rather than simple BNF

EBNF = BNF + **regular expressions**

* means 0 or more occurrences of ...

command → command, single- | command, single-j single-

Example in EBNF

```
Program ::= Command
Command ::= single-Command (; single-Command)*
single-Command
        ::= V-name := Expression
          | begin Command end
          | ...
```

Regular Expressions and CFGs

- The "languages" that can be defined by REs and CFGs have been extensively studied by theoretical computer scientists. These are some important conclusions / terminologies:
  - RE are a "weaker" formalism than CFGs: Any language expressible by a RE can be expressed by a CFG **but not the other way around!** 반대일까
  - The languages expressible as RE are called regular languages
  - Generally: a language that exhibits "self embedding" cannot be expressed by a RE.
  - Programming languages exhibit self embedding.
    Example: an expression can contain another expression

```
expr ::=  id | integer | - expr | ( expr ) | expr op expr

op   ::=  +  |  -  |  *  |  /
```

How many expressions can be derived? Infinitely many.

A CFG for extended micro-English

```
sentence   ::= subject predicate          (P1)
subject    ::= NOUN                       (P2)
             | ARTICLE NOUN               (P3)
predicate  ::= VERB object                (P4)
object     ::= NOUN                       (P5)
             | ARTICLE NOUN               (P6)
```

alteration sign

정관사 들은 : definite article이라고 불림

Verify if

"Peter passed the test"

is a sentence?

Two derivations of "Peter passed the test"

sentence → subject predicate by P1
→ **NOUN** predicate by P2
→ **NOUN VERB** object by P4
→ **NOUN VERB ARTICLE NOUN** by P6

sentence → subject predicate by P1
→ subject **VERB** object by P4
→ subject **VERB ARTICLE NOUN** by P6
→ **NOUN VERB ARTICLE NOUN** by P2

Sentence: **NOUN VERB ARTICLE NOUN**

Sentential forms: above + all the others

Leftmost and rightmost derivations

At each step in the derivation, two choices are made:

- Which nonterminal to replace?
- Which alternative to use for that nonterminal?

Two types of useful derivations:

- Leftmost derivation: always replace the leftmost nonterminal.
- Rightmost derivation: always replace the rightmost noterminal.

Conventions for writing CFGs

- Start symbol:
  - The left-hand side of the first production
  - The letter S, whenever it appears
- Nonterminals:
  - lower-case names such as “sentence” and “expr”
  - capital letters like A, B, C
- Terminals:
  - **boldface** names such as **ID** and **INTLITERAL**
  - digits, operators, punctuation characters (1, +, []. Sometimes in double quotes, e.g., “[“ “]”

Example Grammar for Pascal

program ::= **PROGRAM** *id* **(** *id more_ids* **)** **;** *block* **.**
block ::= *variables* **BEGIN** *stmt more_stmts* **END**
more_ids ::= **,** *id more_ids*
| ε
variables ::= **VAR** *id more_ids* **:** *type* **;** *more_variables*
| ε
more_variables ::= *id more_ids* **:** *type* **;** *more_variables*
| ε
stmt ::= *id* **:=** *exp*
| **READ (** *id more_ids* **)**
| **IF** *exp* **THEN** *stmt* **ELSE** *stmt*
| **WHILE** *exp* **DO** *stmt*
| **BEGIN** *stmt more_stmts* **END**
more_stmts ::= **;** *stmt more_stmts* | ε
exp ::= *num*
| *id*
| *exp* **+** *exp*
| *exp* **–** *exp*
type ::= **integer** | **boolean** | **char**

Note: The productions for 'num' and 'id' have been omitted due to space considerations.

Example Grammar for Pascal (cont.)

- Does the following program adhere to the Pascal syntax?
  - If you cannot find a derivation, then it does not!

```
PROGRAM foo (input, output);
BEGIN
  IF a THEN a:= a+1; b := 22 ENDIF;
END;
```

Extended BNF

- Extended BNF combines BNF with RE
- A production in EBNF looks like

```
LHS ::= RHS
```

where LHS is a nonterminal symbol and RHS is an **extended regular expression**

- An extended RE is just like a regular expression except it is composed of terminals and nonterminals of the grammar.
- Simply put... EBNF adds to BNF the notation of
  - "**(**...**)**" for the purpose of grouping and
  - "*****" to denote "*0 or more repetitions of …* "
  - "**+**" to denote "*1 or more repetitions of ...* "
  - "**?**" to denote "*0 or 1 repetition of ...* "

The miniC grammar is given in EBNF.

Extended BNF Example

A simple expression language:

```
Expression ::=
  PrimaryExp (Operator PrimaryExp)*
PrimaryExp
   ::= Literal | Identifier | ( Expression )
Identifier ::= Letter (Letter|Digit)*
Literal ::= Digit Digit*
Letter ::= a | b | c | ... |z
Digit ::= 0 | 1 | 2 | 3 | 4 | ... | 9
```

An EBNF example from miniC: Kleene Closure

- A miniC program that consists of a sequence of zero or more declarations:
- The BNF productions:

```
program   ::=  decl-list
decl-list ::=  decl-list func-decl
            |  decl-list var-decl
            |  ε
```

- The EBNF productions:

```
program ::=  (func-decl | var-decl)*
```

An EBNF example from miniC: Positive Closure

- A miniC program consists of a sequence of one or more declarations.
- The BNF productions:

```
program   ::=  decl-list
decl-list ::=  decl-list func-decl
            |  decl-list var-decl
            |  func-decl
            |  var-decl
```

- The EBNF productions:

```
program ::=  (func-decl | var-decl)+
```

An EBNF example from miniC: Option ?

- The if-statement with an optional else-part.
- The BNF productions:

```
stmt      ::=  if "(" expr ")" stmt
           |   if "(" expr ")" stmt else stmt
           |   other
```

- The EBNF productions:

```
stmt ::=  if "(" expr ")" stmt (else stmt)?
       |   other
```

A little bit of useful theory

- We will now look at a very few useful bits of theory.
- These will be necessary later when we implement parsers.
  - Grammar transformations
    - A grammar can be transformed in a number of ways without changing the meaning (i.e., the set of strings that it generates)
  - The definition and computation of “starter sets”

1) Grammar Transformations

Left factorization:

$$N ::= X\ Y \mid X\ Z \quad \Rightarrow \quad N ::= X(\ Y \mid Z\ )$$

Y= ε

X

Z

Example:

```
single-Command
   ::= V-name := Expression
     | if Expression then single-Command
     | if Expression then single-Command
                     else single-Command
```

```
single-Command
   ::= V-name := Expression
     |  if Expression then single-Command
                  ( ε | else single-Command)
```

1) Grammar Transformations (cont.)

Elimination of Left Recursion:

$$N ::= X \mid N\ Y \quad \Rightarrow \quad N ::= X\ Y^*$$

Example:

```
Identifier ::= Letter
             | Identifier Letter
             | Identifier Digit
```

left factorization

```
Identifier ::= Letter
             | Identifier (Letter|Digit)
```

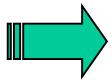
eliminate left recursion

```
Identifier ::= Letter (Letter|Digit)*
```

1) Grammar Transformations (cont.)

Substitution of non-terminal symbols

$$N ::= X \qquad M ::= \alpha\ N\beta$$



$$N ::= X \qquad M ::= \alpha\ X\ \beta$$

Example:

```
single-Command
    ::= for contrVar := Expression
        to-or-dt Expression do single-Command
to-or-dt ::= to | downto
```

```
single-Command ::=
    for contrVar := Expression
    (to|downto) Expression do single-Command
```

Outlook

- Syntax and Semantics of Programming Languages ✓
- Specifying the Syntax of a programming language: ✓
  - CFGs, BNF and EBNF ✓
  - Grammar transformations ✓
- Parsing
  - Top-down parsing (LL) vs. bottom-up parsing (LR)
  - Recursive descent (LL) parser construction
  - LL Grammars
- Parser Generators
- AST Construction
  - Parse trees vs. ASTs
- Chomsky's Hierarchy

Parsing

We will now look at parsing.

Topics:

- Some terminology
- Different types of parsing strategies
  - bottom up
  - top down
- Recursive descent parsing
  - What it is.
  - How to implement it given an EBNF specification.
- Bottom up parsing algorithms

Parse Trees

Correspondence between a derivation and a parse tree:

```
expr ::=  id | int | - expr | ( expr ) | expr op expr
op   ::=  +  |  -  |  *  |  /
```

generate string “slope * x + intercept”:

expr ::= expr op expr

→ expr op id

→ expr + id

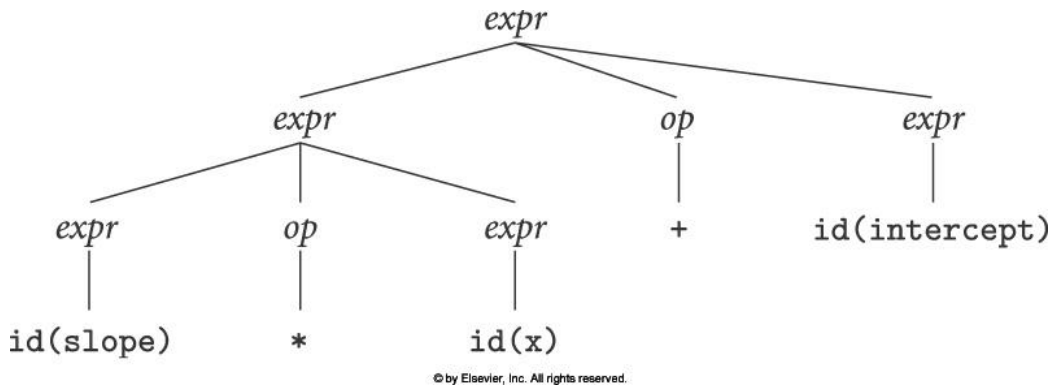
→ expr op expr + id

→ expr op id + id

→ expr * id + id

→ id * id + id

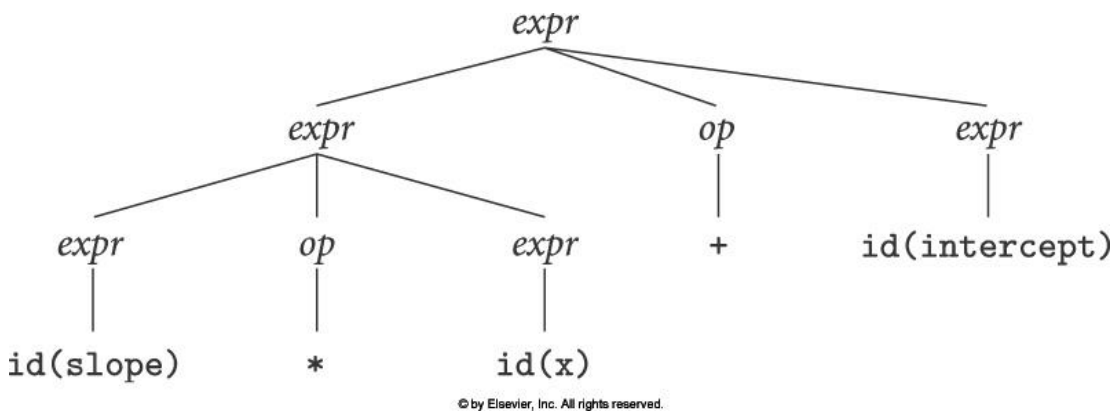
(slope) (x) (intercept)



Parse Trees (cont).

Correspondence between a derivation and a parse tree:

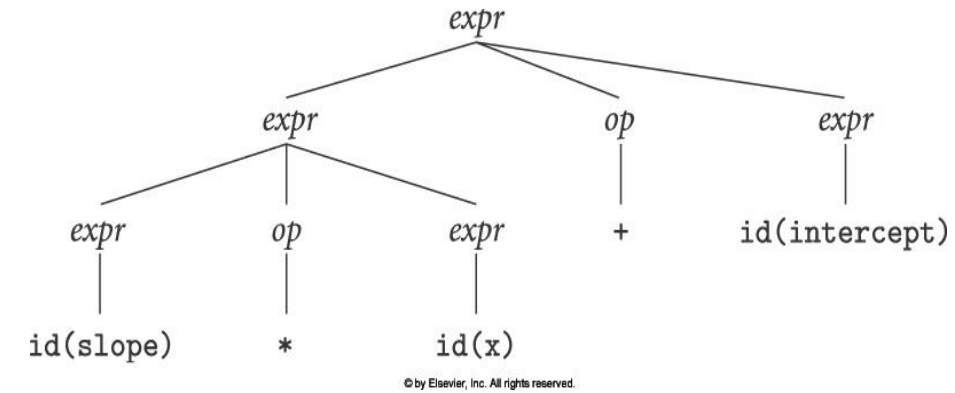
- The parse tree's internal nodes are nonterminals in the productions.
- The children of a node are the terminals and nonterminals on the right-hand side of a production.
- The leaves of the parse tree are the terminals (tokens).
- When read from left to right, the leaves make up the sentence of the derivation.



Ambiguous Grammars

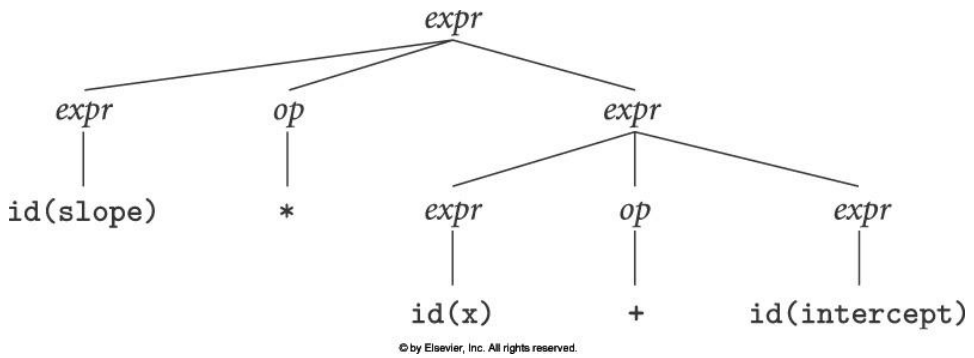
Ambiguity: one sentential form has several distinct parse trees.

Example: slope * x + intercept



Meaning implied by parse tree:

(slope * x) + intercept



slope * (x + intercept)

Problem:
Operator precedence! In the 2nd tree, "+" has precedence over "*" !

Ambiguous Grammars (cont.)

- When more than one distinct derivation of a sentence exists (which means there exist several distinct parse trees), the grammar is ambiguous.
- A programming language construct should have only one parse tree to avoid misinterpretation by a programmer/compiler.
- For expression grammars, associativity and precedence of operators are used to disambiguate the productions.
  - We rewrite the grammar to make it un-ambiguous:

```
expr     ::=  term | expr add_op term
term     ::=  factor | term mult_op factor
factor   ::=  id | number | - factor | ( expr )
add_op   ::=  +  |  -
mult_op  ::=  *   |  /
```

Ambiguous if-then-else

- A well-known example of an ambiguous grammar are the following productions for if-then-else (a.k.a. tangling else):

```
stmt ::=   IF expr THEN stmt
         |   IF expr THEN stmt ELSE stmt
```

- Example: **IF** a **THEN IF** b **THEN** x=false; **ELSE** x=true;

- This grammar can be repaired, but the above problem indicates a programming language design problem.

- Ada uses a different syntax to avoid this problem:

```
stmt ::=    IF expr THEN stmt END IF
         |  IF expr THEN stmt ELSE stmt END IF
```

Some Terminology

- Recognition

  To answer the question “Does the input conform to the syntax of the language ?”.

  - A recognizer uses a CFG to check the syntax of the program. It answers ‘Yes’ if the program corresponds to the CFG, and ‘No’ otherwise (Assignment 2).

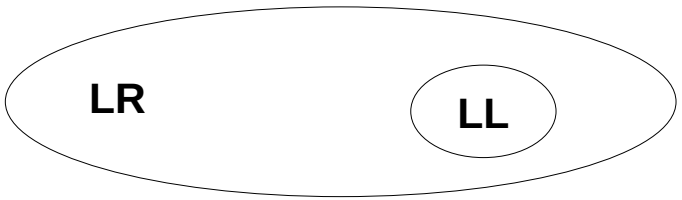
- Parsing

  1) Recognize the input program.
  2) Determine the phrase structure of the input program (for example by generating AST data structures).

  - A parser uses a CFG to parse a sentence or a program.
    - It constructs the leftmost or rightmost derivation, or
    - builds the parse-tree of the program (Assignment 3).

Context-Free Grammar Classes

- For an arbitrary Context-Free Grammar, parsing can take as much as $O(n^3)$ time, where n is the size of the input.
  - too slow for practical applications
- For several classes of grammars, a parser that takes O(n) time can be constructed.
  - Top-down LL parsers for LL grammars (LL = Left-to-right scanning of input, Left-most derivation).
  - Bottom-up parsers for LR grammars (LR = Left-to-right scanning of input, Right-most derivation).
- Grammar classes
  - A grammar is in LL, if it can be parsed by a LL parsing algorithm.
  - A grammar is in LR, if it can be parsed by an LR parsing algorithm.
  - The class of LR grammars is a proper *superset* of the class of LL grammars.



Top-Down Parsers and LL Grammars

- *Top-down parser* is a parser for LL class of grammars
  - Also called a **predictive parser.**
  - LL class is a strict subset of the larger LR class of grammars
  - LL grammars cannot contain **left-recursive productions** (but LR can), for example:

    X ::= X Y …

    and

    X ::= Y Z …
    Y ::= X …
  - LL(*k*) where *k* is lookahead depth: if *k*=1, alternatives productions with **common prefixes** of length 1 cannot be handled:
    X ::= a b … | a c …

- A top-down parser constructs a parse tree from the root down.

- Not too difficult to implement a predictive parser for an unambiguous LL(1) grammar in BNF by hand using *recursive descent.*

Different kinds of Parsing Algorithms

- Two big groups of algorithms can be distinguished:
  - bottom up strategies
  - top down strategies
- Example parsing of "Micro-English"

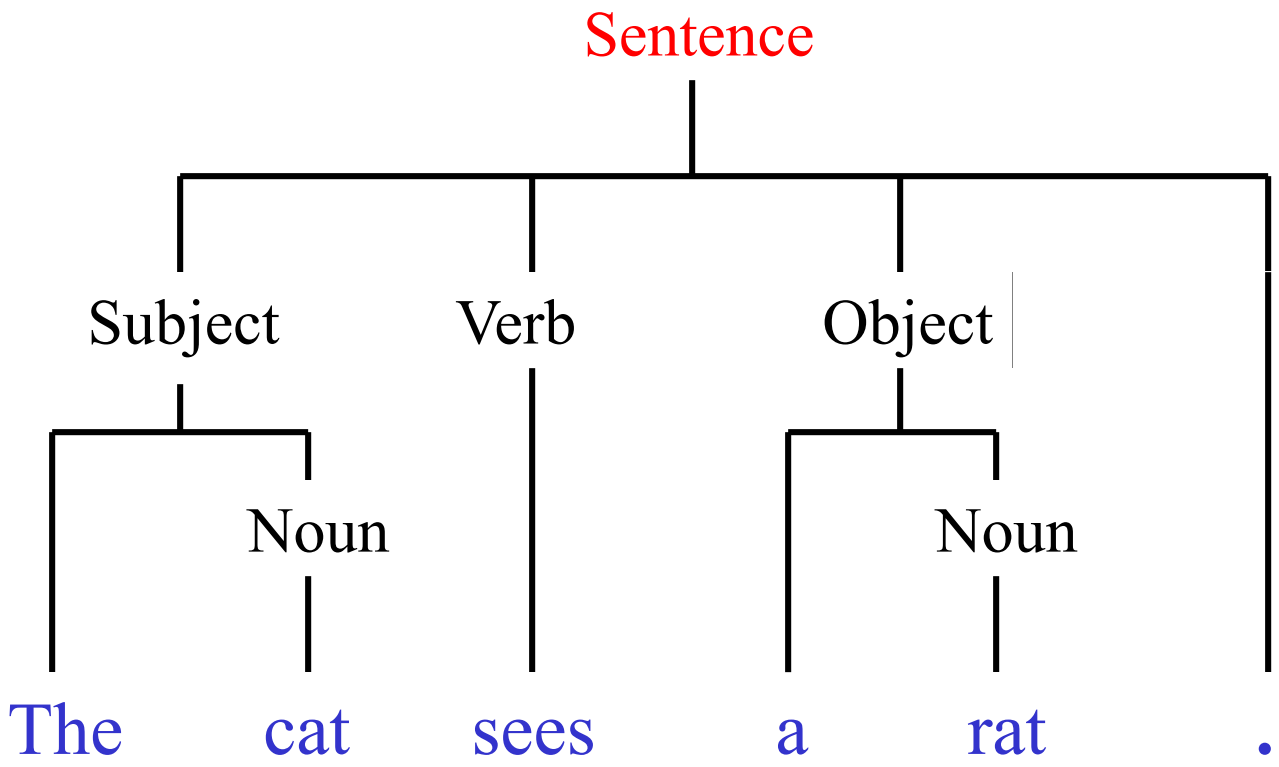
```
Sentence  ::= Subject Verb Object .
Subject   ::= I | a Noun | the Noun
Object    ::= me | a Noun | the Noun
Noun      ::= cat | mat | rat
Verb      ::= like | is | see | sees
```

The cat sees the rat.
The rat sees me.
I like a cat

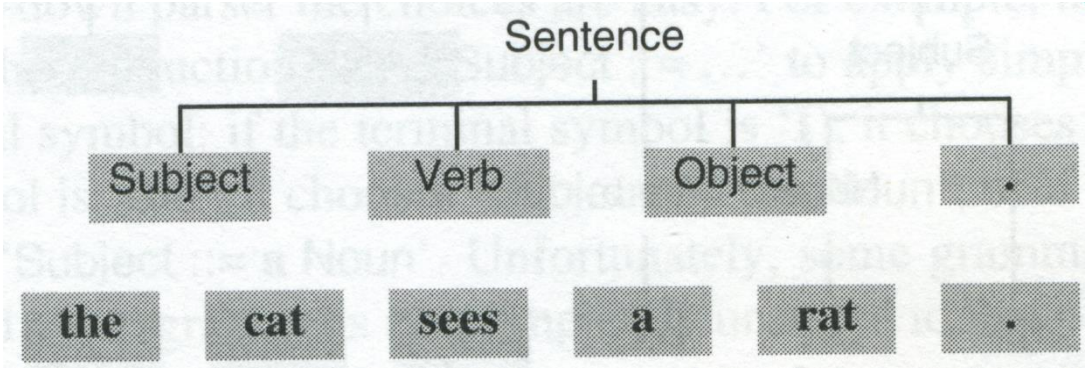
The rat like me.
I see the rat.
I sees a rat.

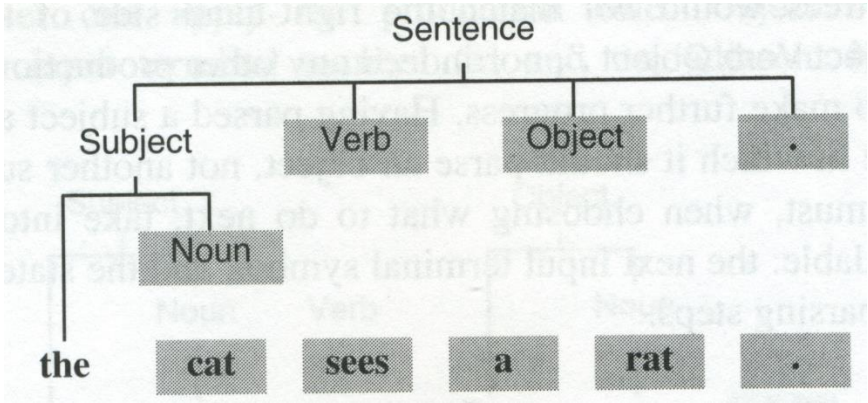
Top-down LL Parsing

The parse tree is constructed starting at the top (root).

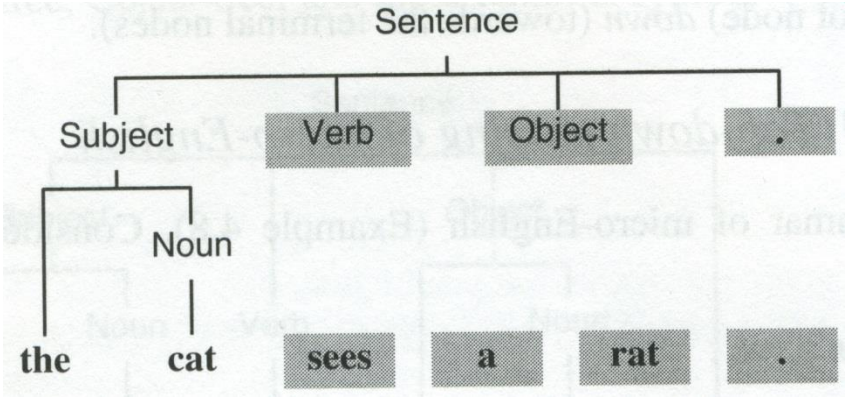


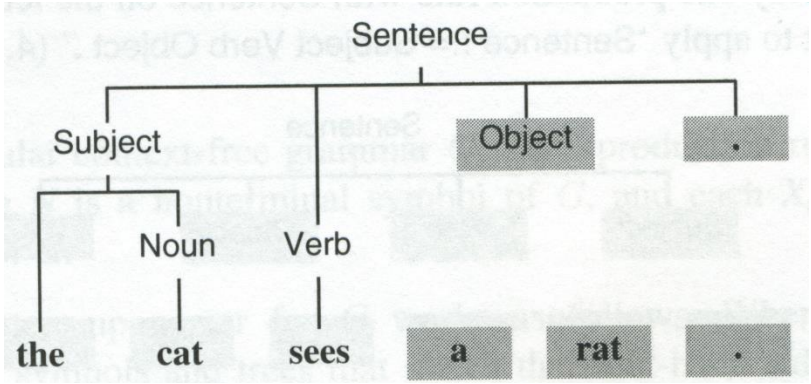
```
Sentence  ::= Subject Verb Object .
Subject   ::= I | a Noun | the Noun
Object    ::= me | a Noun | the Noun
Noun      ::= cat | mat | rat
Verb      ::= like | is | see | sees
```



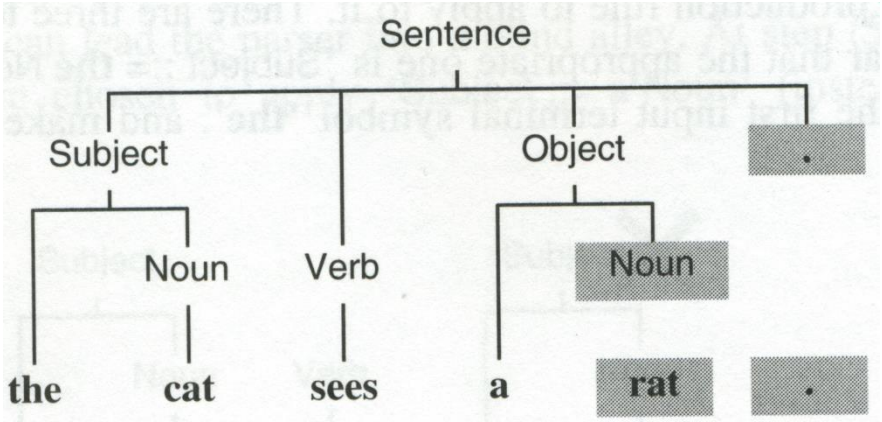


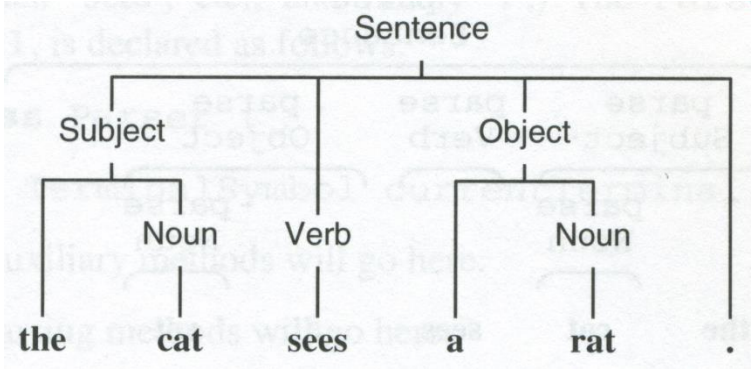
```
Sentence  ::= Subject Verb Object .
Subject   ::= I | a Noun | the Noun
Object    ::= me | a Noun | the Noun
Noun      ::= cat | mat | rat
Verb      ::= like | is | see | sees
```





```
Sentence  ::= Subject Verb Object .
Subject   ::= I | a Noun | the Noun
Object    ::= me | a Noun | the Noun
Noun      ::= cat | mat | rat
Verb      ::= like | is | see | sees
```



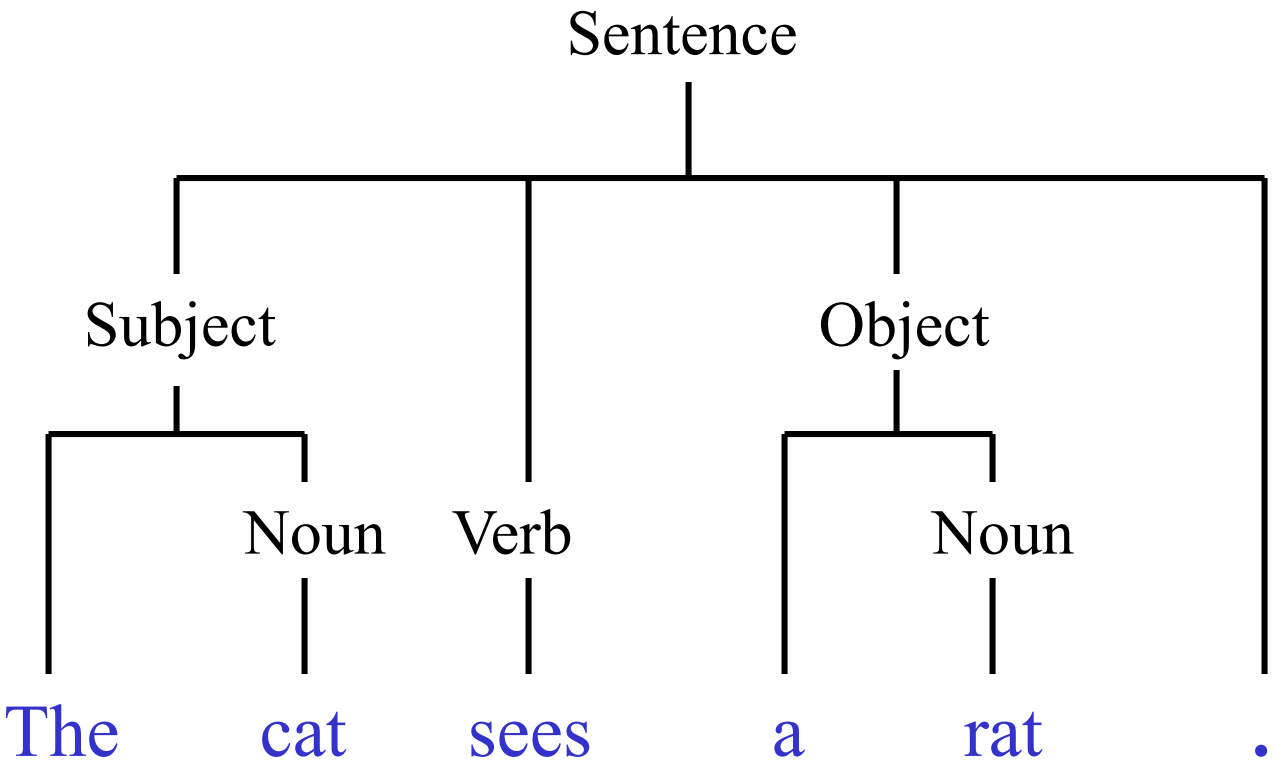


Bottom-up LR Parsing

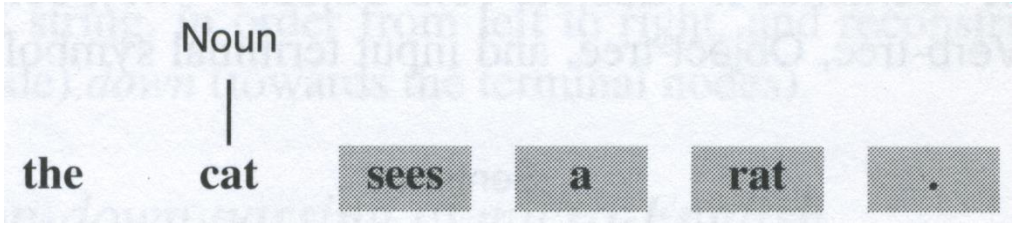
- Bottom-up parser is a parser for LR class of grammars.
- Difficult to implement by hand
- Tools (e.g. Yacc/Bison) exist that generate bottom-up parsers for LALR grammars automatically
- LR parsing is based on shifting tokens on a stack until the parser recognizes a right-hand side of a production which it then reduces to a left-hand side (nonterminal) to form a partial parse tree.

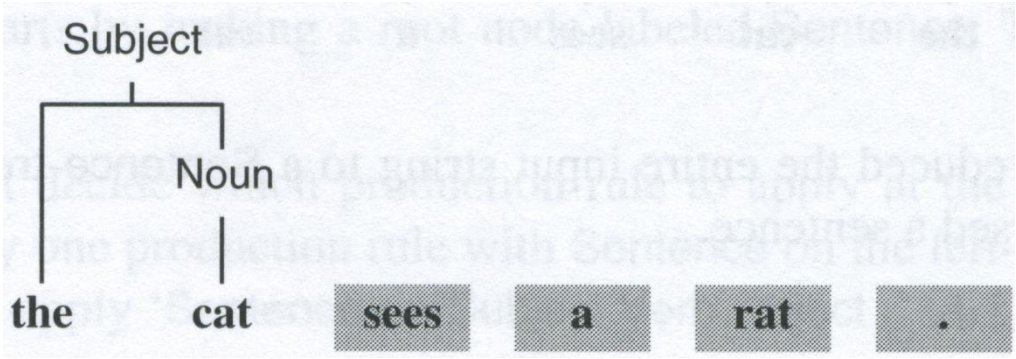
Bottom up LR parsing

The parse tree "grows" from the bottom (leafs) up to the top (root).

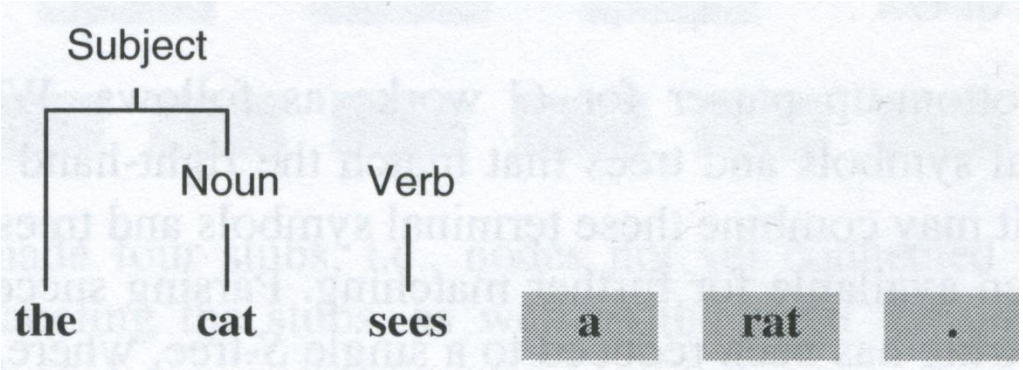


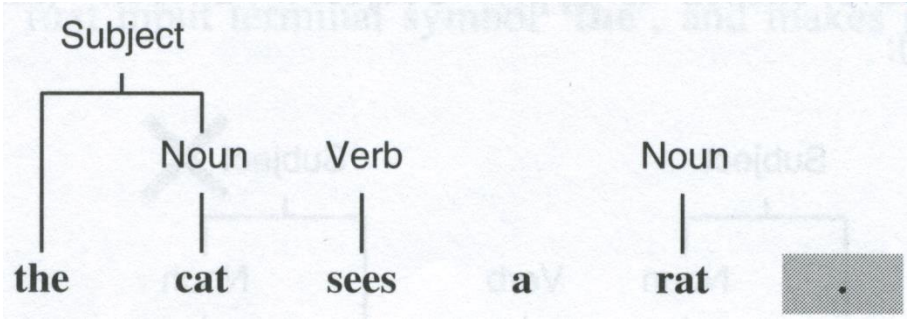
```
Sentence  ::= Subject Verb Object .
Subject   ::= I | a Noun | the Noun
Object    ::= me | a Noun | the Noun
Noun      ::= cat | mat | rat
Verb      ::= like | is | see | sees
```



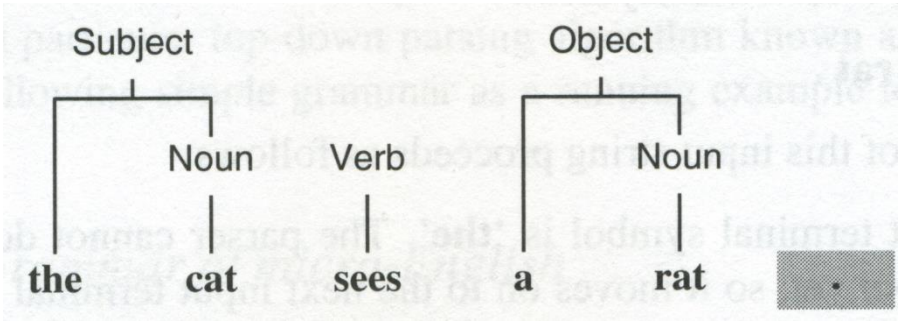


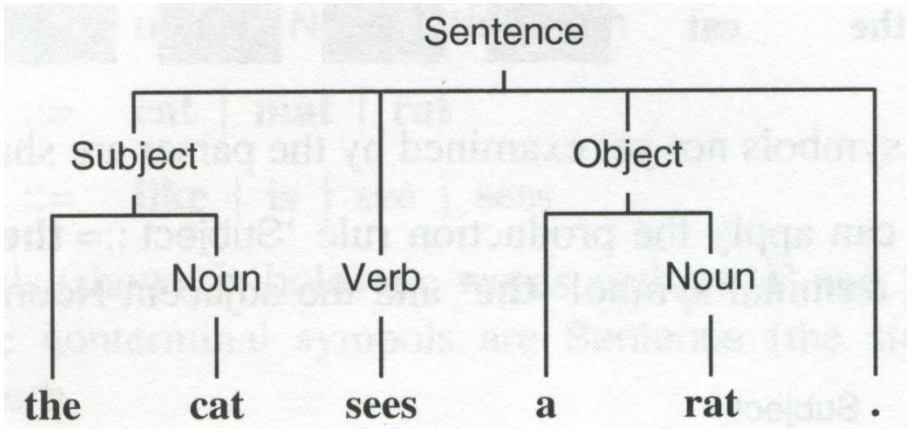
```
Sentence  ::= Subject Verb Object .
Subject   ::= I | a Noun | the Noun
Object    ::= me | a Noun | the Noun
Noun      ::= cat | mat | rat
Verb      ::= like | is | see | sees
```





```
Sentence  ::= Subject Verb Object .
Subject   ::= I | a Noun | the Noun
Object    ::= me | a Noun | the Noun
Noun      ::= cat | mat | rat
Verb      ::= like | is | see | sees
```





Bottup-Up Parsing & Rightmost derivations

The cat sees a rat .
The Noun sees a rat .
Subject sees a rat .
Subject Verb a rat .
Subject Verb a Noun .
Subject Verb Object .
Sentence

Underlined symbols are righthand sides of productions that got replaced by their lefthand side during LR parsing.

The process of bottom-up parsing produces the reverse of a rightmost derivation. A bottom-up parser starts with the input sentence and produces a sequence of sentential forms until it reaches the start symbol.

Sentence → Subject Verb Object .
→ Subject Verb a Noun .
→ Subject Verb a rat .
→ Subject sees a rat .
→ The Noun sees a rat .
→ The cat sees a rat .

This is a rightmost derivation.

Underlined symbols are lefthand sides (non-terminals) of productions that got replaced by their righthand side.

Outlook

- Syntax and Semantics of Programming Languages ✓
- Specifying the Syntax of a programming language: ✓
  - CFGs, BNF and EBNF ✓
  - Grammar transformations ✓
- Parsing
  - Top-down parsing (LL) vs. bottom-up parsing (LR) ✓
  - Recursive descent (LL) parser construction
  - LL Grammars
- Parser Generators
- AST Construction
  - Parse trees vs. ASTs
- Chomsky's Hierarchy

Recursive Descent Parsing

- Recursive descent parsing is a straightforward top-down parsing algorithm.
- We will now look at how to develop a recursive descent parser from an EBNF specification.
- Idea: the parse tree structure corresponds to the “call graph” structure of parsing procedures that call each other.

Recursive Descent Parser Construction Example

```
Sentence ::= Subject Verb Object .
Subject  ::= I | a Noun | the Noun
Object   ::= me | a Noun | the Noun
Noun     ::= cat | mat | rat
Verb     ::= like | is | see | sees
```

Define a procedure parseN for each non-terminal N

```
private void parseSentence() ;
private void parseSubject();
private void parseObject();
private void parseNoun();
private void parseVerb();
```

Recursive Descent Parser Construction Example

```
public class MicroEnglishParser {

   private Token currentToken;

   //Auxiliary methods will go here
   ...

   //Parsing methods will go here
   ...
}
```

Recursive Descent Parser Construction: Auxiliary Methods

```
public class MicroEnglishParser {

  private Token currentToken

  private void accept(Token expectedToken) {
    if (currentToken == expectedToken)
      currentToken = scanner.scan();  // get next token
                                      // from scanner
    else
      report a syntax error
   }

   ...
}
```

Recursive Descent Parser Construction: Parsing Methods

```
Sentence  ::= Subject Verb Object .
```

```
private void parseSentence( ) {
   parseSubject( );
   parseVerb( );
   parseObject( );
   accept('.');
}
```

Recursive Descent Parser: Parsing Methods

```
Subject   ::= I | a Noun | the Noun
```

```
private void parseSubject() {
   if (currentToken == 'I')
      accept('I');
   else if (currentToken == 'a') {
      accept('a');
      parseNoun();
   }
   else if (currentToken == 'the') {
      accept('the');
      parseNoun();
   }
   else
     report a syntax error
}
```

Recursive Descent Parser: Parsing Methods

```
Noun    ::= cat | mat | rat
```

```
private void parseNoun( ) {
   if (currentToken == 'cat')
      accept('cat');
   else if (currentToken == 'mat')
      accept('mat');
   else if (currentToken == 'rat')
      accept('rat');
   else
     report a syntax error
}
```

Systematic Development of a RD Parser

recursive descent

(1) Express grammar in EBNF

(2) Grammar Transformations:

Left factorization and left recursion elimination

(3) Create a parser class with

- private variable `currentToken`
- methods to call the `scanner: accept` and `acceptIt`

(4) Convert EBNF into a RD parser:

Implement private parsing methods:

- add **private** `parse`***N*** method for each nonterminal ***N***
- **public** `parse` method that
  - gets the first token form the scanner
  - calls `parse`***S*** (***S*** is the start symbol of the grammar)

Systematic Development of a RD Parser (1 & 2)

common prefixes require left - factorization.

Left factorization:

```
program      ::= ( variable-def  |  function-def )*
function-def ::= typespec ID params compoundstmt
variable-def ::= typespec init-decl ("," init-decl)* ";"
init-decl    ::= declarator (“=” initializer) ?
declarator   ::= ID | ID "[" INTLITERAL "]"
typespec     ::= void | int | bool | float
params       ::= "(" params-list?  ")"
```

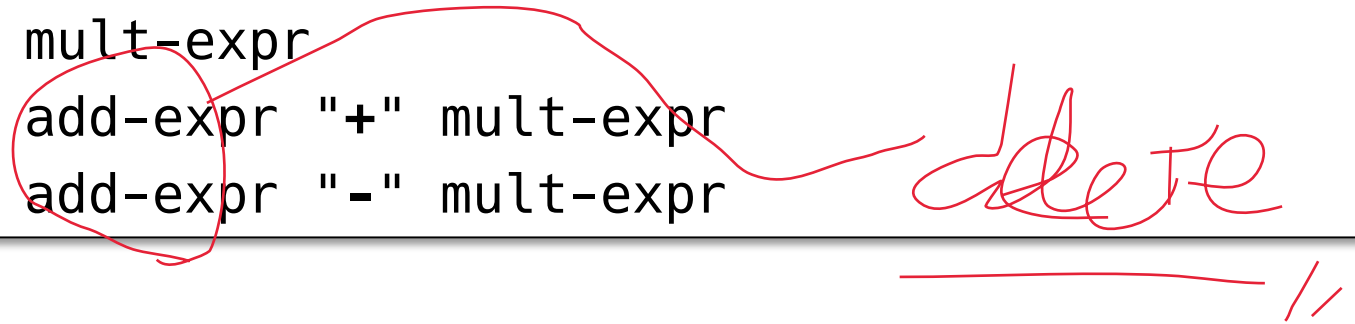
After factorization we get:

```
program  ::= ( typespec ID (VarPart | FuncPart ))*
FuncPart ::= ( "(" ParamsList? ")" compoundstmt )
VarPart  ::= ( "[" INTLITERAL "]" )? ( "=" initializer ) ?
                                     ( "," init_decl)* ";"
```

Systematic Development of a RD Parser (1 & 2)

Left recursion elimination:

```
add-expr ::= mult-expr
           |  add-expr "+" mult-expr
           |  add-expr "-" mult-expr
```



After elimination of the left-recursion we get:

```
add-expr ::= mult-expr ( ("+"|"-") mult-expr)*
```

You will have to transform a few other MiniC productions (easy!).

Developing a RD Parser for MiniC

The compiler driver in MiniC.java:

```
static void compileProgram (String sourceName) {
    ...
    scanner  = new Scanner(source);
    reporter = new ErrorReporter();
    parser   = new Parser(scanner, reporter);
    parser.parse();

    boolean successful = (reporter.numErrors == 0);
    if (successful) {
       System.out.println("Compilation was successful.");
    } else {
       System.out.println("Compilation was unsuccessful.");
    }
}
```

exception error handling

using error exception nicely hand error - !!

error check

The Error Reporter:

```
public class ErrorReporter {
  int numErrors;
  ErrorReporter( ) { numErrors = 0; }

                                                  token
  public void reportError (String m, String tok, SourcePos pos) {
    System.out.print ("ERROR: ");
    for (int c = 0; c < m.length( ); c++) {
      if (m.charAt(c) == '%')
       System.out.print(tok);
      else
       System.out.print(m.charAt(c));
    }
   System.out.println(" " + pos.StartCol + ".." + pos.EndCol + ", line +
                       pos.StartLine + ".");
    numErrors++;
  }
}
```

reportError("Type specifier instead of **%** expected", Token.spell(currentToken.kind));

→ “ERROR: Type specifier instead of IF expected 9..10, line 22.”

Reporting errors and throwing exceptions:

```
void syntaxError(String Template, String tok) throws SyntaxError {

   SourcePos pos = currentToken.GetSourcePos( );
   errorReporter.reportError(Template, tok, pos);
   throw(new SyntaxError( ));
}

class SyntaxError extends Exception {

  SyntaxError( ) {
    super( );
  };
}
```

See slides on "Java Exceptions" for an explanation of the Java exception mechanism.

syntaxError("Type specifier instead of % expected", currentToken.GetLexeme());

→ "ERROR: Type specifier instead of IF expected 9..10, line 22."

Exception-Handling in the MiniC Parser

```
////////////////////////////////////////////////////////////////////////
// toplevel MiniC parse() routine:
////////////////////////////////////////////////////////////////////////
public void parse() {

    currentToken = scanner.scan(); // get first token from scanner...

    try {
       parseProgram();
       if (currentToken.kind != Token.EOF) {
            syntaxError("\"%\" not expected after end of program",
                        currentToken.GetLexeme());
       }
    }
    catch (SyntaxError s) {
       // Here we catch all SyntaxError exceptions that did not get
       // handled in parseProgram(), plus the one thrown in
       // syntaxError() above.
       return; // To be refined in Assignment 3...
    }
    return;
}
```

Algorithm to convert EBNF into a RD parser

- The conversion of an EBNF specification into a Java implementation for a recursive descent parser is so “mechanical” that it can easily be automated!

=> JavaCC “Java Compiler Compiler”

- We can describe the algorithm by a set of mechanical rewrite rules.

N ::= *X*

```
private void parseN() {
   parse X();
}
```

Algorithm to convert EBNF into a RD parser

parsing t where *t* is a terminal

accept(*t*);

parsing N where *N* is a non-terminal

parse*N*();

parsing ε

// do nothing, a dummy statement

parsing X Y

parse X();
parse Y();

Algorithm to convert EBNF into a RD parser

*parsing X**

```
while (currentToken.kind is in  FIRST [X]) {
   parse X();
}
```

For an example, see slide 70.

parsing X | Y

```
switch (currentToken.kind) {
   cases in  FIRST [X]:
     parse X
     break;
   cases in  FIRST [Y]:
     parse Y
     break;
   default: report syntax error
}
```

FIRST [X] denotes the set of terminal symbols that start sentences derived from X.

Example: parseExpr

```
Expr  ::=  AndExpr ( "||" AndExpr)*
```

```
public void parseExpr() throws SyntaxError {
      parseAndExpr();
      while (currentToken.kind == Token.OR) {
          acceptIt();
          parseAndExpr();
      }
}
```

Example: parseProgram

program ::= ((**VOID** | **INT** | **BOOL** | **FLOAT**) **ID** (FunPart | VarPart))*

```
public void parseProgram() throws SyntaxError {
    while (isTypeSpecifier (currentToken.kind)) {
        acceptIt();
        accept(Token.ID);
        if(currentToken.kind == Token.LEFTPAREN) {
            parseFunPart();
        } else {
            parseVarPart();
        }
    }
}
```

isTypeSpecifier is true iff the current Token is either Token.VOID, Token.INT, Token.BOOL or Token.FLOAT.

Outlook

- Syntax and Semantics of Programming Languages ✓
- Specifying the Syntax of a programming language: ✓
  - CFGs, BNF and EBNF ✓
  - Grammar transformations ✓
- Parsing
  - Top-down parsing (LL) vs. bottom-up parsing (LR) ✓
  - Recursive descent (LL) parser construction ✓
  - LL Grammars
- AST Construction
  - Parse trees vs. ASTs
- Parser Generators
- Chomsky's Hierarchy

LL(1) Grammars

- The presented algorithm to convert EBNF into a parser does not work for all possible grammars.
  - It only works for so called “LL(1)” grammars
- **LL(1)** is an acronym for
  - **L**eft-to-right parsing of the input stream
  - **L**eftmost derivation
  - **1** token look-ahead in the input stream
- What grammars are LL(1)?
  - Left-recursion means a grammar is not LL(1)
  - Common prefixes mean a grammar is not LL(1)
  - Ambiguous grammars are not LL(1)

  → See examples on next slides...

  → See LL(1) definition further down the road...

Notation and Terminology

Given a context-free grammar G:

- V_t is the set of terminal symbols in the grammar
- V_n is the set of nonterminals.
- P is a finite set of productions
- $V = V_t \cup V_n$ is the vocabulary of G

$a, b, c, \ldots \in V_t$ $\quad\quad$ $\alpha, \beta, \gamma, \ldots \in V^*$

$A, B, C, \ldots \in V_n$ $\quad\quad$ $u, v, w, \ldots \in V_t^*$

- If $A ::= \gamma$ then $\alpha A \beta \rightarrow \alpha \gamma \beta$ is a single-step derivation using $A \rightarrow \gamma$

- Similarly, $\rightarrow^*$ and $\rightarrow^+$ denote derivations of >= 0 and >= 1 steps.

Predictive Parsing (aka Recursive Descent Parsing)

Basic idea:

- For any two productions $A \rightarrow \alpha \mid \beta$ we would like a distinct way of choosing the correct production to expand.
- For some right-hand-side $\alpha \in G$, define FIRST(α) as the set of tokens that appear first in some string derived from α
- Key property:
  Whenever two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

$$\mathrm{FIRST}(\alpha) \cap \mathrm{FIRST}(\beta) = \emptyset$$

- This would allow the parser to predict the correct production with a lookahead of only one symbol!

Left-Recursive Grammars are not LL(1)

```
Expr ::= Expr "+" INT
       |  INT
```

What happens if we don't perform **left-recursion elimination**?

```
public void parseExpr() {
 switch (currentToken.kind) {
   case Token.INT:
        parseExpr();
        accept(Token.PLUS);
        accept(Token.INT);
        break;
   case Token.INT:
        accept(Token.INT);
        break;
     default: report syntax error
  }
}
```

*Problem1: **overlapping cases**:*

FIRST (Expr "+" INT) = {**INT**}

FIRST (**INT**) = {**INT**}

*Problem2: **infinite recursion** via* parseExpr().

Left-Recursive Grammars are not LL(1)

```
Expr ::= INT
       |  Expr "+" INT
```

eliminate left-recursion: $N ::= X \mid N\ Y$ ⇒ $N ::= X\ Y^*$

```
Expr ::= INT ( "+" INT)*
```

```
public void parseExpr() {
    accept(Token.INT);
    while (currentToken.kind == Token.PLUS) {
        accept(Token.PLUS);
        accept(Token.INT);
    }
}
```

Grammars with Common Prefixes are not LL(1)

```
Expr ::= Term "+" Expr
       |  Term
```

What happens if we don't perform **left-factorization**?

```
public void parseExpr() {
  switch (currentToken.kind) {
    case currentToken.kind ∈ FIRST (Term):
      parseTerm();
      accept(Token.PLUS);
      parseExpr();
      break;
    case currentToken.kind ∈ FIRST (Term):
      parseTerm();
      break;
    default: report syntax error
  }
}
```

Note: this parseExpr() invocation does not lead to infinite recursion, because we have consumed input in between (e.g., Token.PLUS).

Problem of overlapping cases in FIRST (TERM) *!*

Grammars with Common Prefixes are not LL(1)

```
Expr ::= Term "+" Expr
       |  Term
```

perform left-factorization: $X\ Y\ |\ X\ Z \Rightarrow X(\ Y\,|\,Z\ )$

```
Expr ::= Term ( "+" Expr | ε )
```

```
public void parseExpr() {
    parseTerm();
    if (currentToken.kind == Token.PLUS) {
        accept(Token.PLUS);
        parseExpr();
    }
}
```

Intuition behind LL(1) Grammars:

parse X | Y

```
switch (currentToken.kind) {
   case currentToken.kind ∈ FIRST (X):
     parse X
     break;
   case currentToken.kind ∈ FIRST (Y):
     parse Y
     break;
   default: report syntax error
}
```

Condition 1: FIRST (X) and FIRST (Y) must be disjoint sets.

*parse X**

```
while (currentToken.kind is in FIRST (X)) {
   parse X
}
```

Condition 2: FIRST (X) must be disjoint from the set of tokens that can immediately follow *X* *

Generality

Question:

By left-factoring and elimination of left-recursion, can we transform any grammar such that it can be parsed with a single token lookahead?

Answer:

Given a context-free grammar that does not meet our conditions, it is undecidable whether an equivalent grammar exists that meets our conditions.

Many context-free grammars do not have such a grammar:

$$\{ a^n 0 b^n \mid n \geq 1 \} \bigcup \{ a^n 1 b^{2n} \mid n \geq 1 \}$$

Must look past an arbitrary number of a's to discover the 0 or 1 and so determine the derivation.
However: most programming languages can be expressed with an LL(1) grammar!

FIRST

For a string α of grammar symbols, FIRST(α) is

- the set of terminal symbols that **start** sentences derived from α :

  $\{c \in V_t \mid \alpha \Rightarrow^* c\beta\}$

- If $\alpha \Rightarrow^* \varepsilon$ then $\varepsilon \in \mathrm{FIRST}(\alpha)$

$\mathrm{FIRST}(\alpha)$ contains the set of tokens valid in the initial position in α

The algorithm to compute FIRST(X) for all grammar symbols X.

Note: initialize FIRST(X)={} for all non-terminals and terminals X. The algorithm terminates when the sets FIRST(X) stabilize (no further addition possible).

1. If $X \in V_t$ then $\mathrm{FIRST}(X) = \{X\}$
2. If $X ::= \epsilon$ then add ϵ to $\mathrm{FIRST}(X)$
3. If $X ::= Y_1 Y_2 \ldots Y_k$:
   (a) Put $\mathrm{FIRST}(Y_1) - \{\epsilon\}$ in $\mathrm{FIRST}(X)$
   (b) $\forall i : 1 < i \leq k$, if $\epsilon \in \mathrm{FIRST}(Y_1) \cap \cdots \cap \mathrm{FIRST}(Y_{i-1})$
   (i.e., $Y_1 \ldots Y_{i-1} \rightarrow^* \epsilon$)
   then put $\mathrm{FIRST}(Y_i) - \{\epsilon\}$ in $\mathrm{FIRST}(X)$
   (c) if $\epsilon \in \mathrm{FIRST}(Y_1) \cap \cdots \cap \mathrm{FIRST}(Y_{i-1})$ then put ϵ in $\mathrm{FIRST}(X)$

   Repeat Step 3 until no more additions can be made.

Recursive invocation !

FOLLOW

For a non-terminal *A*, we define FOLLOW(*A*) as

the set of terminals that can appear **immediately to the right** of *A* in some sentential form.

A non-terminal's FOLLOW-set specifies the tokens that can legally appear after it.

Note: a terminal symbol has no FOLLOW set.

'S' is the CFG's start symbol, and '$' is the input end marker (EOF).
Note: initialize Follow(X)={} for all non-terminals.

The algorithm iterates until the FOLLOW sets for all non-terminals have been computed.

1. Put \$ in FOLLOW(S).
2. If $A ::= \alpha B \beta$:
   (a) Put FIRST(β) $- \{\epsilon\}$ in FOLLOW(B)
   (b) if $\beta = \epsilon$ (i.e., $A ::= \alpha B$) or $\epsilon \in$ FIRST(β) (i.e., $\beta \rightarrow^* \epsilon$) then put FOLLOW($A$) in FOLLOW($B$)

   Repeat Step 2 until no more additions can be made.

LL(1) Grammars

Given FIRST and FOLLOW sets, we have now everything in place to define LL(1) grammars.

A grammar *G* is LL(1) iff it is not left-recursive and for each pair of productions $A \rightarrow \alpha \mid \beta$ the following conditions hold:

- $\mathrm{FIRST}(\alpha) \cap \mathrm{FIRST}(\beta) = \{\}$
- If $\alpha \Rightarrow^* \varepsilon$ then $\mathrm{FIRST}(\beta) \cap \mathrm{FOLLOW}(A) = \{\}$
- If $\beta \Rightarrow^* \varepsilon$ then $\mathrm{FIRST}(\alpha) \cap \mathrm{FOLLOW}(A) = \{\}$

Non-LL(1) Grammar Examples

A grammar *G* is LL(1) iff it is not left-recursive and for each pair of productions $A \rightarrow \alpha \mid \beta$ the following conditions hold:

1) $\mathrm{FIRST}(\alpha) \cap \mathrm{FIRST}(\beta) = \{\}$
2) If $\alpha \Rightarrow^{*} \varepsilon$ then $\mathrm{FIRST}(\beta) \cap \mathrm{FOLLOW}(A) = \{\}$
3) If $\beta \Rightarrow^{*} \varepsilon$ then $\mathrm{FIRST}(\alpha) \cap \mathrm{FOLLOW}(A) = \{\}$

Grammar	Not LL(1) because:
$S \rightarrow S$ **a** | **a**	Left recursive
$S \rightarrow$ **a** S | **a**	FIRST(**a** S) $\cap$ FIRST(**a**) $\neq$ {}
$S \rightarrow$ **a** $R \mid \varepsilon$ $R \rightarrow S \mid \varepsilon$	For R: $S \Rightarrow^{*} \varepsilon$ and $\varepsilon \Rightarrow^{*} \varepsilon$
$S \rightarrow$ **a** R **a** $R \rightarrow S \mid \varepsilon$	For R: FIRST(S) $\cap$ FOLLOW(R) $\neq \varnothing$

Outlook

- Syntax and Semantics of Programming Languages ✓
- Specifying the Syntax of a programming language: ✓
  - CFGs, BNF and EBNF ✓
  - Grammar transformations ✓
- Parsing
  - Top-down parsing (LL) vs. bottom-up parsing (LR) ✓
  - Recursive descent (LL) parser construction ✓
  - LL Grammars ✓
- AST Construction
  - Parse trees vs. ASTs
- Parser Generators
- Chomsky's Hierarchy

Abstract Syntax Trees (ASTs)

- So far we have discussed how to build a recursive descent parser which recognizes a given language described by an LL(1) EBNF grammar.

- However, at the end of the syntactic analysis phase, we need an AST.

- Now we will look at
  - how parse trees differ from ASTs,
  - how to represent ASTs as data structures,
  - how to refine our parser from Assignment 2 to construct AST data structures.

Parse Trees vs. ASTs

Several expression grammars:

A grammar with left-recursion:

```
E ::= E + T | E - T | T
T ::= T * F | T / F | F
F ::= INT | (E)
```

Grammar 1

A grammar without left recursion:

```
E ::= T ( (+|-) T)*
T ::= F ( (*|/) F)*
F ::= INT | (E)
```

Grammar 2

The ambiguous grammar that we used in the beginning:

```
E ::= E+E | E-E | E*E | E/E |(E)| INT | (E)
```

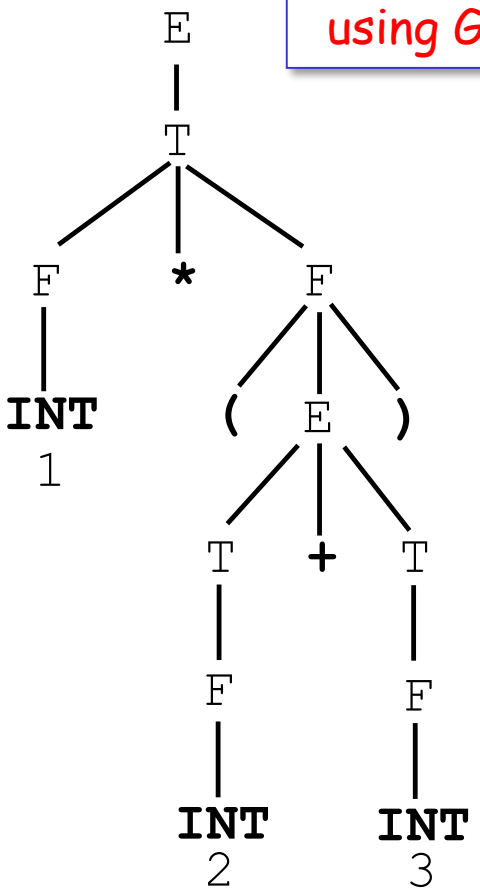
Grammar 3

Parse Trees vs. ASTs (cont.)

```
E ::= T ( (+|-) T)*
T ::= F ( (*|/) F)*
F ::= INT | (E)
```

Grammar 2

The parse tree for 1 * (2 + 3) using Grammar 2.



- Parse tree specifies the syntactic structure of the input
  - (1-1 correspondence with derivations)
  - leaves of tree correspond to tokens
- one internal node for each production used during parsing
- We can use Grammar 2 for LL(1) parsing of expressions
  - (Grammar 2 is unambiguous)
- However: the resulting parse tree is cluttered with information relevant only during parsing.
  - See next slide...

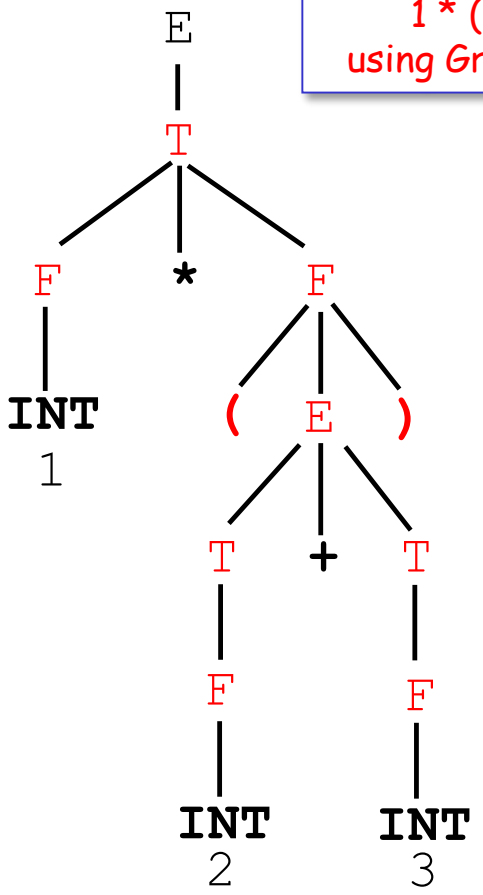
Parse Trees vs. ASTs (cont.)

```
E ::= E+E | E-E | E*E | E/E |(E)| INT | (E)
```

```
E ::=  T ( (+|-) T)*
T ::=  F ( (*|/) F)*
F ::=  INT | (E)
```

Grammar 2

The parse tree for 1 * (2 + 3) using Grammar 2.



However: the resulting parse tree is cluttered with information relevant only during parsing:

- T and F have only been introduced to enforce operator precedence (* before +).
- Parentheses () are needed only in source code to group subexpressions. The tree structure encodes this information using the level of the operator in the tree (operators further down are evaluated first).
- Once the parse tree is constructed, grouping and operator precedence follow from the structure of the tree (depth-first traversal).
- There is no need to store the information shown in red in the tree...
- A parse-tree mirrors the **concrete syntax**.

Parse Trees vs. ASTs (cont.)

We can leave out all information which is not relevant for later phases of the compiler:

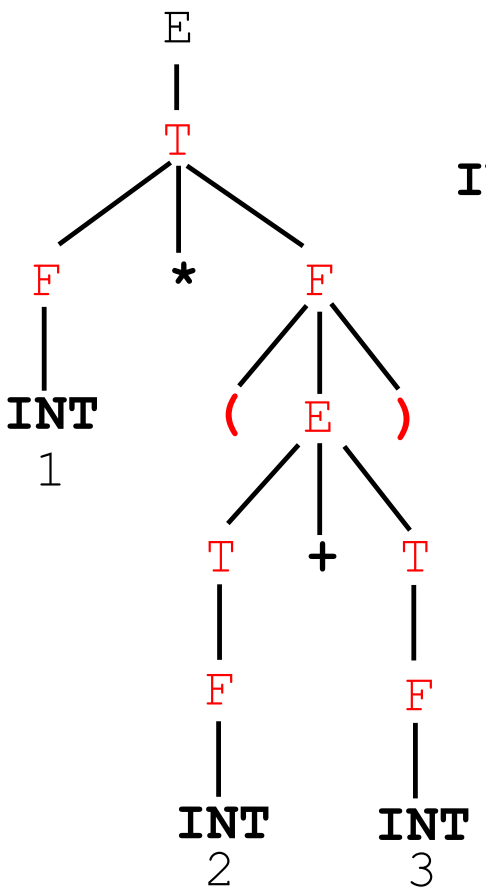
- Nonterminals that define operator precedence and associativity (this information is implicit in trees).
- Punctuation tokens (parentheses, semi-colons, ...)
- Keywords

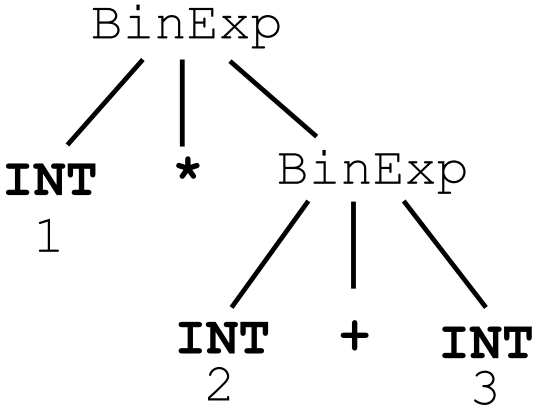
Yields more compact representation than the parse tree.

- Called **abstract syntax** (it abstracts away unnecessary details of the programming language).

Parsing is done using the concrete syntax, e.g., Grammar 2.

- During parsing, we generate an AST, not a parse tree.
- For subsequent phases of the compiler, the AST is sufficient.





Abstract syntax defines the structure of abstract syntax trees (ASTs).

Concrete syntax defines the structure of parse trees (Grammar 2).

Parse Trees vs. ASTs (cont.)

Abstract syntax (and hence ASTs) can also be specified via context-free grammars.

- Would be very similar to our simple initial grammar (Grammar 3).

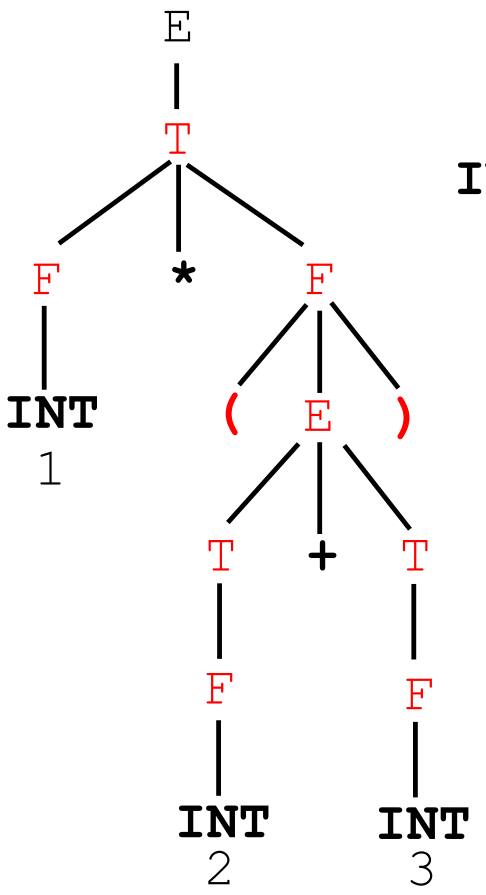
For Assignment 3, you'll get

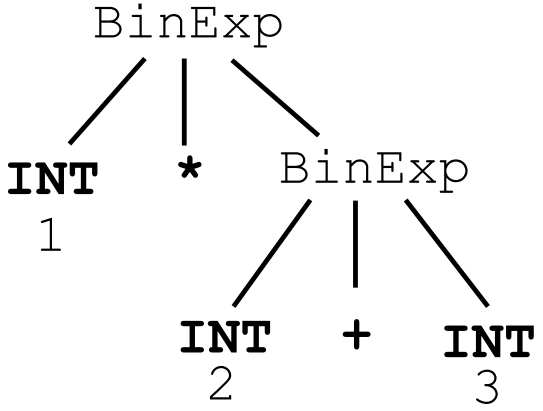
1) simple source code examples
2) the corresponding ASTs

Tells you how MiniC ASTs should look like.

ASTs are a compact intermediate representation (IR) of the input program, used for

- type-checking
- code optimizations
- code generation





Abstract syntax defines the structure of abstract syntax trees (ASTs).

Concrete syntax defines the structure of parse trees (Grammar 2).

Design of AST Classes

We want Java classes for all parts of an AST.

- During parsing, we can use those classes to generate the AST of the input program.

ASTs can be specified formally using a context-free grammar

- AST classes could then be generated automatically.

Overall structure of AST-classes in our MiniC compiler:

- AST.java is the top-level abstract class
- One concrete class for each symbol from the abstract grammar

With MiniC

- EmptyXYZ AST classes are used to avoid NULL pointers.

Overview of the MiniC AST Classes

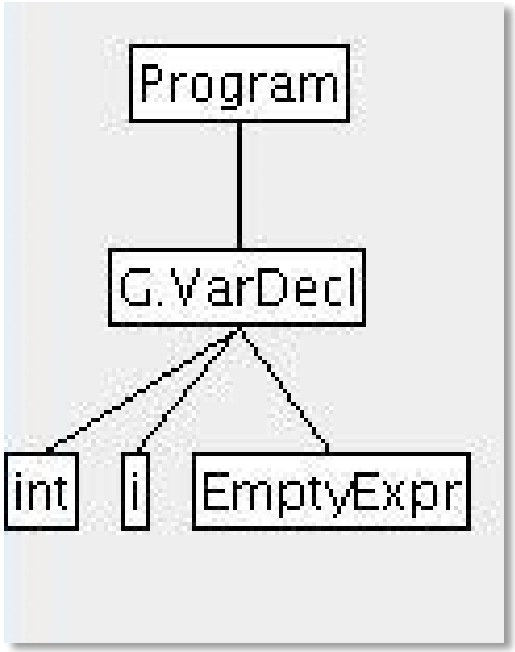
+ AST
  =Program(Decl D)
  + Decl
    = FunDecl(Type tAST, ID idAST, Decl paramsAST, Stmt stmtAST)
    = VarDecl(Type tAST, ID idAST, Expr eAST)
    = FormalParamDecl(Type astType, ID astIdent)
  + Stmt
    = CompoundStmt(Decl astDecl, Stmt astStmt)
    = IfStmt(Expr eAST, Stmt s1AST (, Stmt s2AST)?)
    = WhileStmt(Expr eAST, Stmt stmtAST)
    = ForStmt (Expr e1AST, Expr e2AST, Expr e3AST, Stmt stmtAST)
    = ReturnStmt(Expr eAST)
    = AssignStmt(Expr lAST, Expr rAST)
  + Expr
    =BinaryExpr(Expr lAST, Operator oAST, Expr rAST)
    =UnaryExpr(Operator oAST, Expr eAST)
    ....
  + Terminal
    =ID(String Lexeme)
    =Operator(String Lexeme)
    =IntLiteral(String Lexeme)
    =FloatLiteral(String Lexeme)
    ...

Classes marked with "+" are abstract classes, "=" denotes concrete classes. The arguments for the constructors are given in parentheses (), with source position arguments omitted for claritly.

Use of AST Classes

```
SourcePosition pos = new SourcePos();

Type T = new IntType (pos);
ID Ident = new ID ("i", pos);
Expr E = new EmptyExpr(pos);

VarDecl VD = new VarDecl (T, Ident, E, pos);
Program P = new Program (VD, pos);
```

```
int i;
```



Assignment 3

- Involved Java packages:

Package	Functionality
MiniC.AstGen	AST classes to create tree nodes
MiniC.Parser	Parser (where we will generate the AST)
MiniC.TreeDrawer	Draws your AST on the screen
MiniC.TreePrinter	Print an AST in ASCII representation
MiniC.UnParser	Walk the AST to print a MiniC program

- Additional MiniC Compiler options:

-ast display the AST of the input program

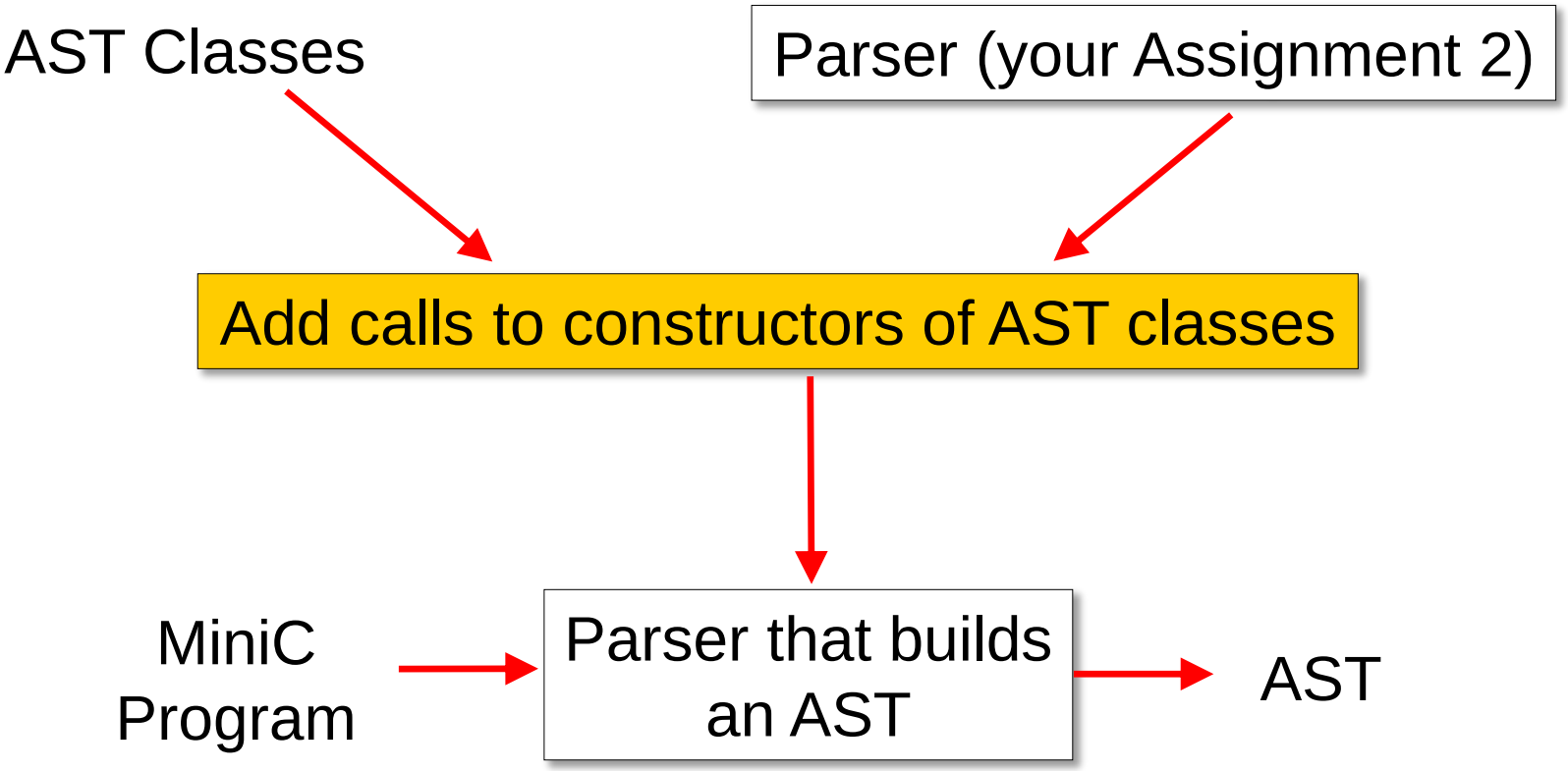
-astp display the AST plus source positions of the input program

-t <file> write AST to <file>

-u <file> unparse the AST into <file>

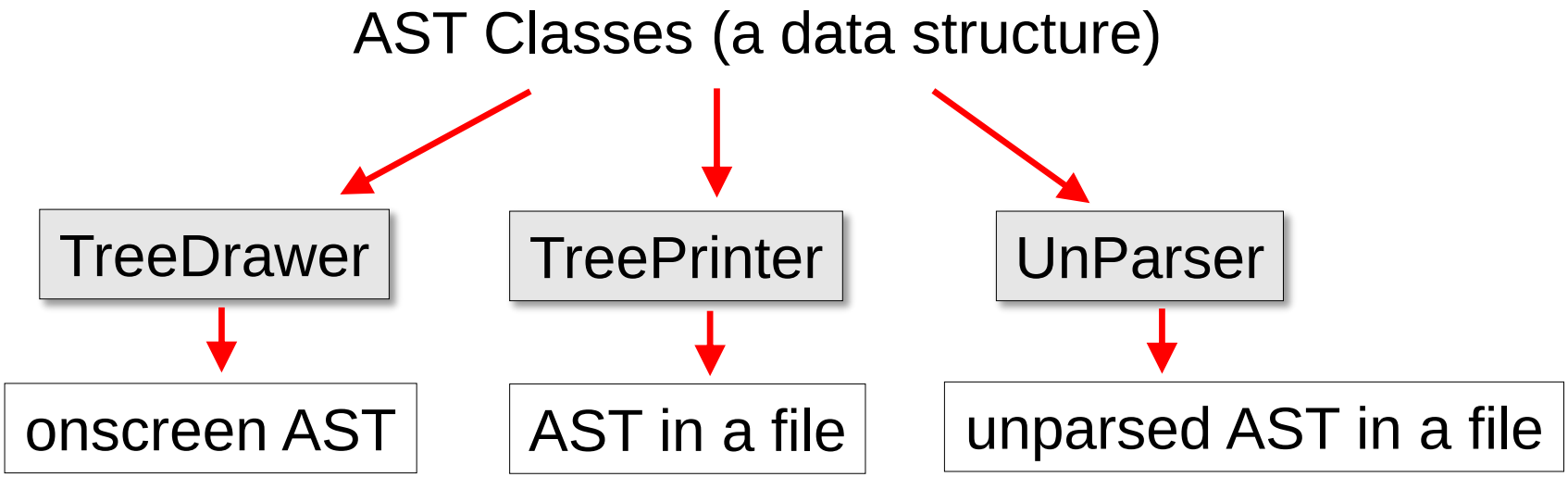
See examples on following slides...

Constructing ASTs in Assignment 3:



- The larger part of Assignment 3 consists of inserting the constructors into your parser.

Programming Environment for Assignment 3



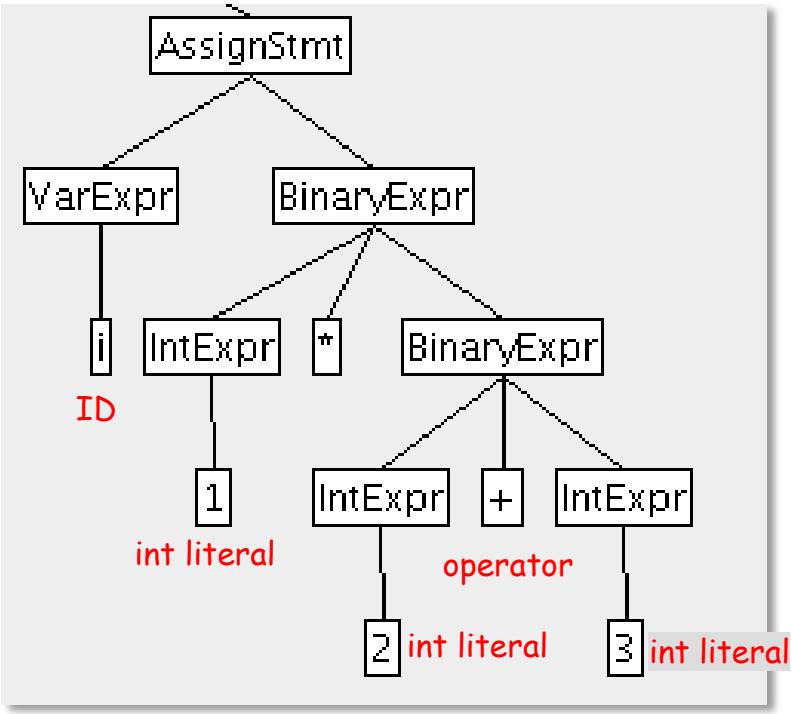
- All three packages implemented using the Visitor Design Pattern. Book available at http://www.patterndepot.com/put/8/JavaPatterns.htm
- Use the design pattern to traverse the AST (= “tree-walk”)
- during traversal, generate the desired output
- We will use this pattern in Assignment 4 and 5.
- Tree-walkers can be generated automatically from attribute grammars.

Example:

- MiniC code snippet:

```
i = 1*(2+3);
```

- The corresponding AST, using the MiniC compiler option **–ast**:



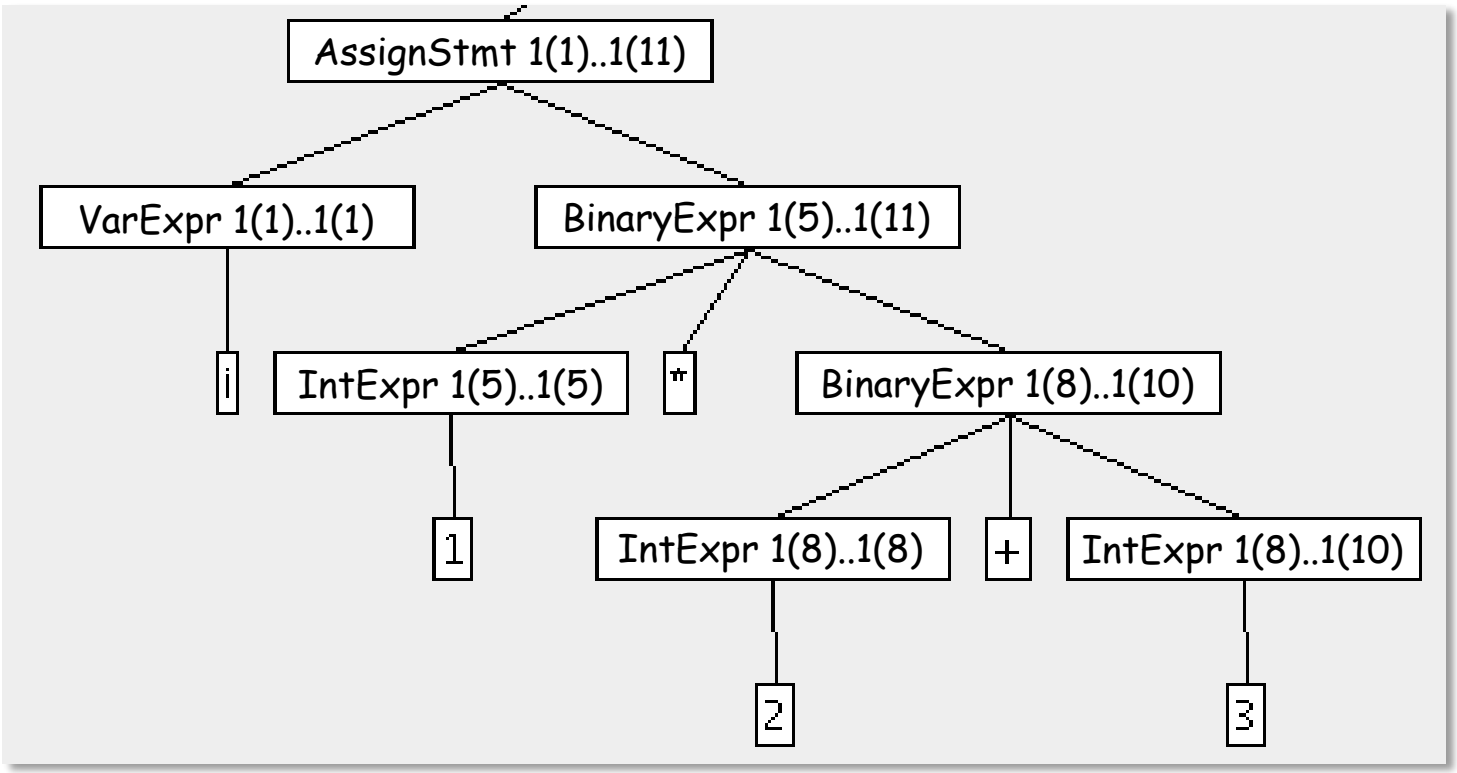
Example:

- MiniC code snippet:

```
i = 1*(2+3);
```

Option -astp includes the start and end positions of the phrase in the source code.

- The corresponding AST, using the MiniC compiler option **–astp**:



Example:

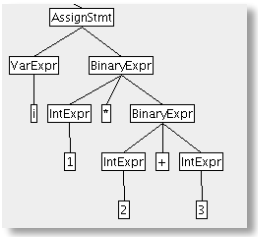
- MiniC code snippet:

```
i = 1*(2+3);
```

- The corresponding AST, using the MiniC compiler option **–t**:

```
AssignStmt
    VarExpr
        ID: i
    BinaryExpr
        IntExpr
            IntLiteral: 1
        Operator: *
        BinaryExpr
            IntExpr
                IntLiteral: 2
            Operator: +
            IntExpr
                IntLiteral: 3
```

Indentation of an AST node in the tree dump corresponds to the node's level in the AST. Children of a node follow below the node, at the next higher indentation level.




```
AssignStmt
    VarExpr
        ID: i
    BinaryExpr
        IntExpr
            IntLiteral: 1
        Operator: *
        BinaryExpr
            IntExpr
                IntLiteral: 2
            Operator: +
            IntExpr
                IntLiteral: 3
```

AST & indentation levels: 0 1 2 3 4

Example:

- MiniC code snippet:

```
i = 1*(2+3);
```

- The corresponding code snippet, using the MiniC compiler option **–u**:

```
i = (1 * (2 + 3));
```

- The ‘unparsed’ output uses full parenthesization!
  - One pair of parentheses for each and every operator.

Example: MiniC Stmt ASTs

```
Stmt
  ::= compound-stmt                    CompoundStmt {...}
    | ID "=" Expr ";"                  AssignStmt
    | while "(" Expr ")" stmt          WhileStmt
```

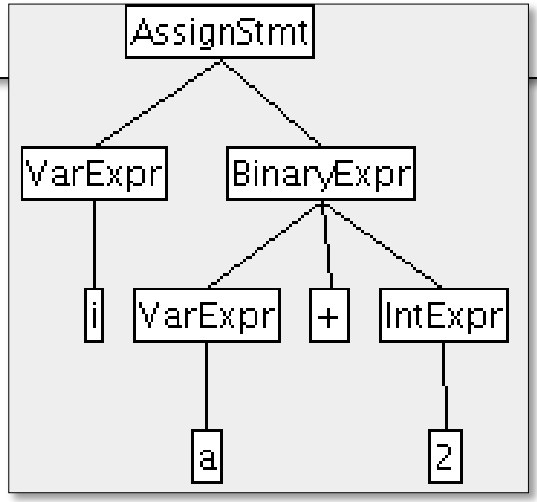
```
public abstract class Stmt extends AST { ... }

public class AssignStmt extends Stmt { ... }
public class CompoundStmt extends Stmt { ... }
public class WhileStmt extends Stmt { ... }
...
```

Example: MiniC Stmt ASTs

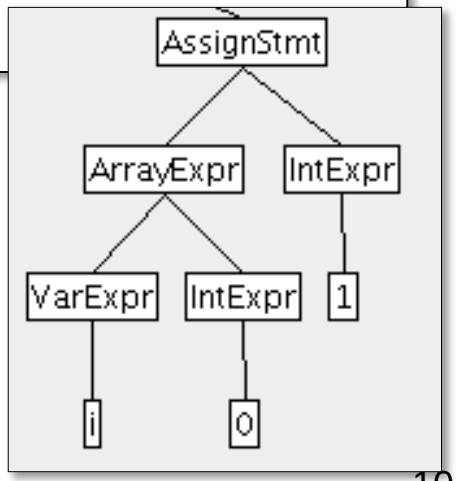
```
Stmt
  ::= compound-stmt                          CompoundStmt
    | ID "=" Expr ";"                        AssignStmt
    | ID "[" Expr "]" "=" Expr ";"           AssignStmt
```

```
public class AssignStmt extends Stmt {
   public Expr lAST;   // L-value of the assignment.
   public Expr rAST;   // R-value (what to assign to the L-value).
   ...
}
```

i = a + 2;



// int i[10];
i[0] = 1;

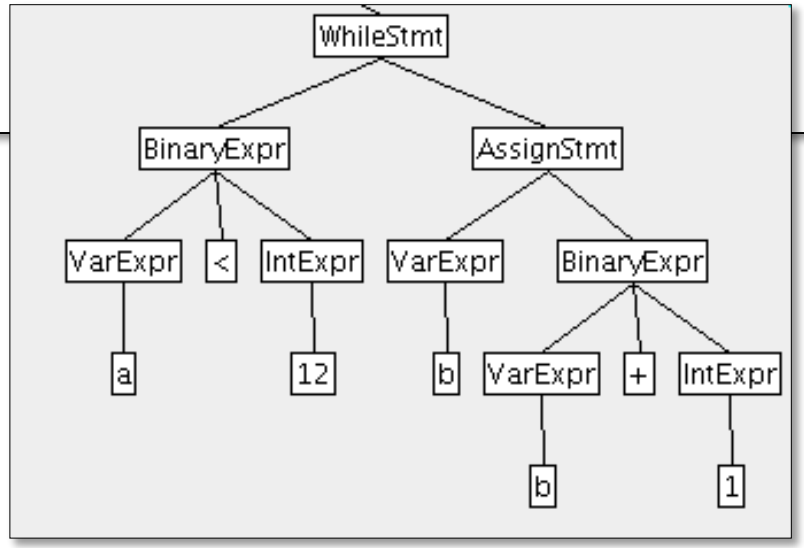


Example: MiniC Stmt ASTs

```
Stmt
  ::= compound-stmt                          CompoundStmt
    | ...
    | while "(" Expr ")" stmt                WhileStmt
```

```
public class WhileStmt extends Stmt {
   public Expr eAST ;       // loop condition
   public Stmt stmtAST;   // loop body
   ...
}
```

```
while (a < 12) {
    b = b + 1;
}
```



AST Terminal Nodes

```
public abstract class Terminal extends AST {
   public String Lexeme;
   ...
}

public class ID extends Terminal { ... }

public class IntLiteral extends Terminal { ... }

public class BoolLiteral extends Terminal { ... }

public class FloatLiteral extends Terminal { ... }

public class StringLiteral extends Terminal { ... }

public class Operator extends Terminal { ... }
```

Note:
Due to the fact that we're using an AST and not a parse tree as our program representation (IR), it is not necessary to have Terminal subclasses for:

- punctuation characters: (){}[];, etc.
- type specifiers: void, int, float, bool
- keywords: for, while, if, etc.

AST Construction

First, every concrete AST class needs a **constructor.**

```
public class WhileStmt extends Stmt {
   public Expr eAST;       // expression of the loop condition.
   public Stmt stmtAST;  //  loop body

   public WhileStmt(Expr eAST, Stmt stmtAST) {
      this.eAST = eAST;
      this.stmtAST = stmtAST;
   }
}
```

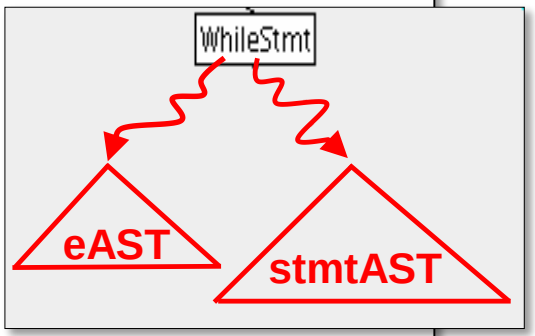
The constructor takes the AST subtrees of a WhileStmt as arguments and assigns them to the member variables **eAST** and **stmtAST**.

Parsing of a MiniC *while* statement

```
WhileStmt ::= while "(" Expr ")" Stmt
```

```
public Stmt parseWhileStmt() throws SyntaxError {
    Expr eAST;
    Stmt stmtAST;
    accept(Token.WHILE);
    accept(Token.LEFTPAREN);
    eAST = parseExpr();
    accept(Token.RIGHTPAREN);
    stmtAST = parseStmt();
    return new WhileStmt (eAST, stmtAST);
  }
```

The parts in red need to be added to your parser from Assignment 2. They construct an AST from the subtrees returned by the parse* functions as shown below..



Parsing of a MiniC *if* statement

```
IfStmt ::= if "(" Expr ")" Stmt ( "else" Stmt ) ?
```

```
public Stmt parseIfStmt() throws SyntaxError {
      Expr eAST;
      Stmt thenAST, elseAST;
      accept(Token.IF);
      accept(Token.LEFTPAREN);
      eAST = parseExpr();
      accept(Token.RIGHTPAREN);
      thenAST = parseStmt();
      if (currentToken.kind == Token.ELSE) {
          acceptIt();
          elseAST = parseStmt();
          return new IfStmt (eAST, thenAST, elseAST);
      } else {
          return new IfStmt(eAST, thenAST);
      }
}
```

Parsing of a MiniC *Compound* statement

```
CompoundStmt ::= "{" VariableDef* Stmt* "}"
```

```
public CompoundStmt parseCompoundStmt() throws SyntaxError {
    accept(Token.LEFTBRACE);
    Decl D = parseCompoundDecls();
    Stmt S = parseCompoundStmts();
    accept(Token.RIGHTBRACE);
    if ( (D.getClass() == EmptyDecl.class) &&
         (S.getClass() == EmptyStmt.class))
    {
        return new EmptyCompoundStmt (previousTokenPosition); // Case 1
    } else {
        return new CompoundStmt (D, S, previousTokenPosition);  // Case 2
    }
}
```

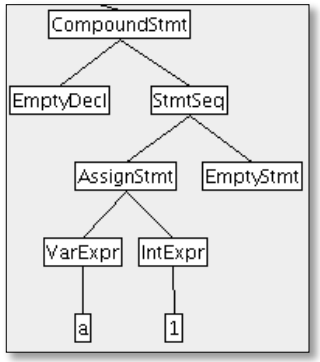
MiniC statement sequence ASTs grow to the right...

```
public Stmt parseCompoundStmts () throws SyntaxError {
    if ( ! (currentToken.kind == Token.LEFTBRACE ||
            currentToken.kind == Token.IF ||
            currentToken.kind == Token.WHILE ||
            currentToken.kind == Token.FOR ||
            currentToken.kind == Token.RETURN ||
            currentToken.kind == Token.ID)   )
    {
        return new EmptyStmt();  // terminate recursion
    }
    Stmt S = parseStmt();
    Stmt Remainder = parseCompoundStmts();
    return new StmtSequence (S, Remainder);
}
```

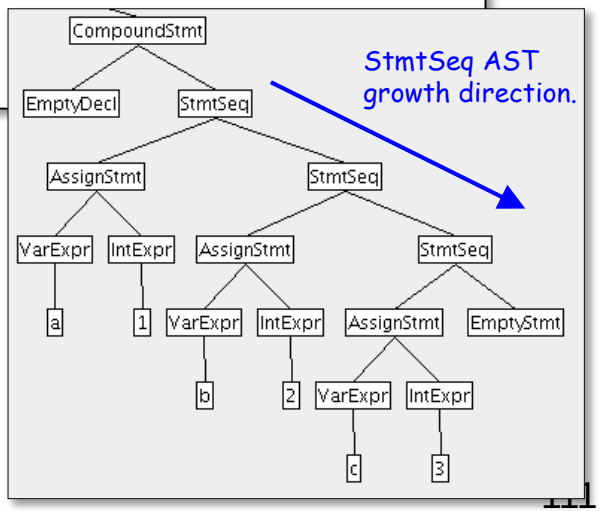
```
{ }
```

See Case 1 on previous page.

```
{
  a=1;
}
```




```
{
   a=1;
   b=2;
   c=3;
}
```



AST Construction (in general)

We will now show how to refine our recursive descent parser to actually construct an AST.

```
N ::= X
```

```
public N parseN() {
  N theAST;
  parse X at the same time constructing theAST
  return theAST;
}
```

The Parsing Method for Stmt

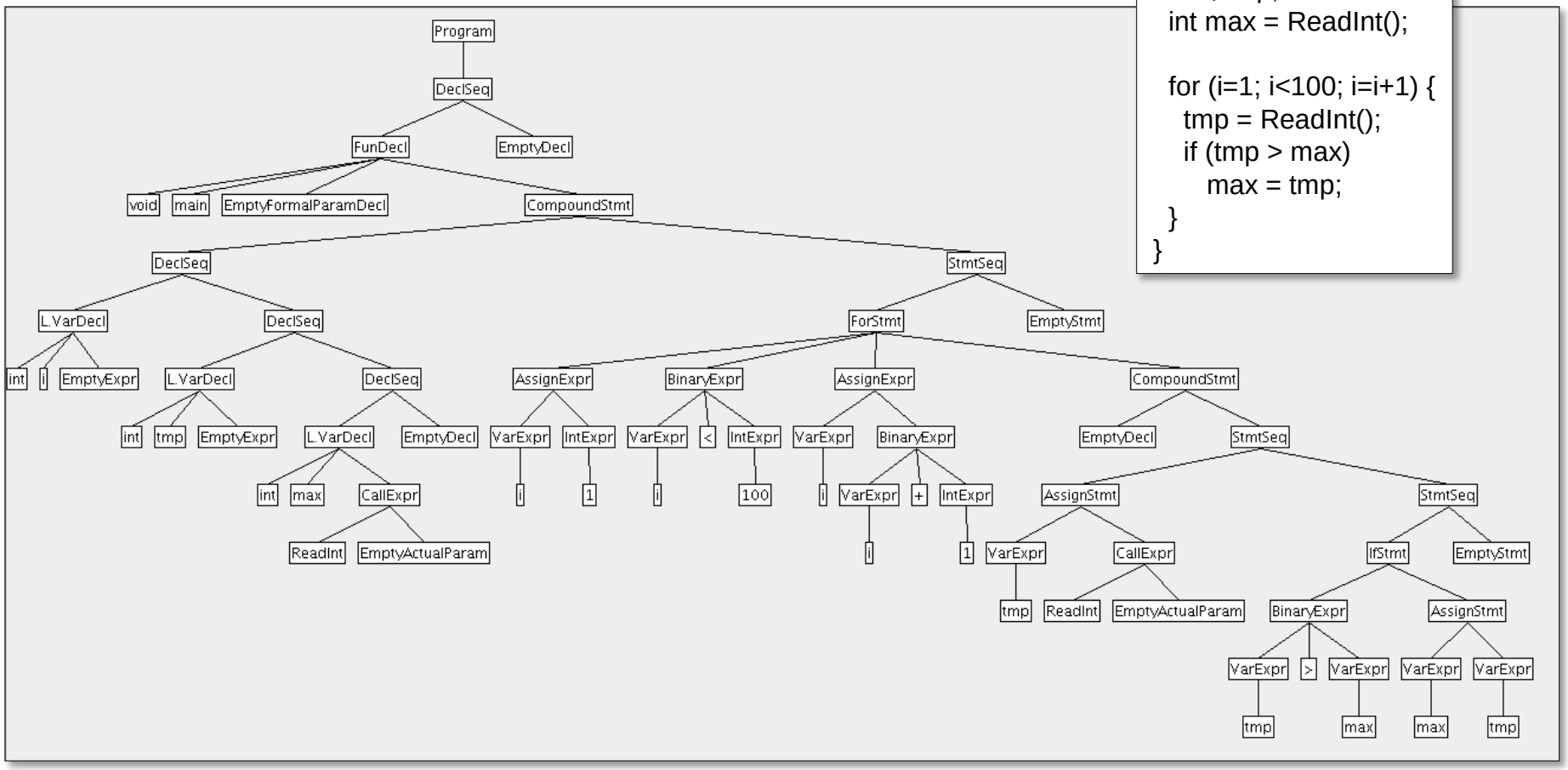
```
public Stmt parseStmt() throws SyntaxError {
      Stmt retStmt = null;
       switch (currentToken.kind) {
       case Token.LEFTBRACE:
          retStmt = parseCompoundStmt();
          break;
       case Token.IF:
          retStmt = parseIfStmt();
          break;
      ...
       default:
          syntaxError("\"%\" not a statement", Token.spell(currentToken.kind));
       }
       return retStmt;
}
```

- parseCompoundStmt(), parseIfStmt(), ... return concrete objects from the MiniC class-hierarchy.
- the return-type of parseStmt() is abstract. 'Stmt' is the abstract base class for all MiniC statements.
- This works, because with OO, every derived class "is-a" base-class.
  - called **subtype-polymorphism**: a sub-class can always be substituted for the base-class.
- The opposite does not hold.

  Example: every if-stmt is a stmt, but not every stmt is an if-stmt.

MiniC AST Example

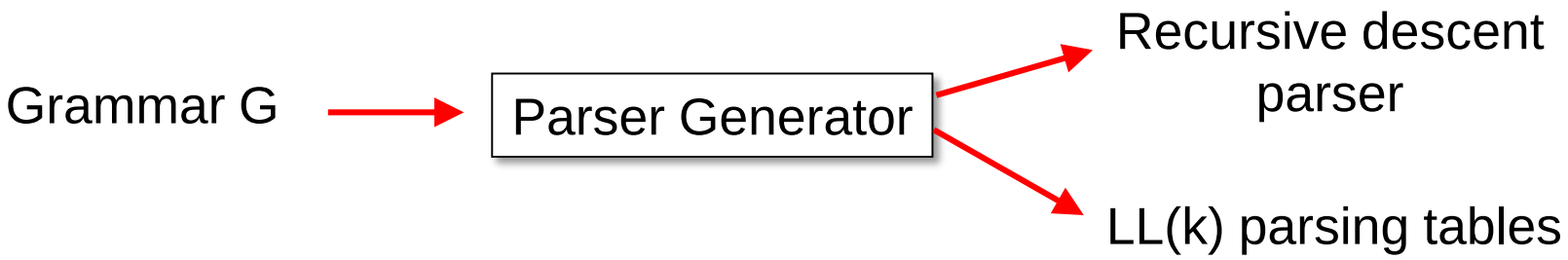
```
void main()
{
  int i, tmp;
  int max = ReadInt();

  for (i=1; i<100; i=i+1) {
    tmp = ReadInt();
    if (tmp > max)
       max = tmp;
  }
}
```



Outlook

- Syntax and Semantics of Programming Languages ✓
- Specifying the Syntax of a programming language: ✓
  - CFGs, BNF and EBNF ✓
  - Grammar transformations ✓
- Parsing
  - Top-down parsing (LL) vs. bottom-up parsing (LR) ✓
  - Recursive descent (LL) parser construction ✓
  - LL Grammars ✓
- AST Construction ✓
  - Parse trees vs. ASTs ✓
- Parser Generators
- Chomsky's Hierarchy

Parser Generators generating Top-Down Parsers



Tool	accepted grammar	generated parsers and their implementation languages
JavaCC	LL(k) EBNF	Java RD LL(1) with some LL(k) portions
COCO/R	LL(k) EBNF	Pascal, C,C++,Java etc RD LL(1)
ANTLR	Predicated LL(k)	C, C++, Java RD LL(k)

Predicate: a conditional evaluated by the parser at run-time to determine which of the two conflicting productions to use

Q ::= [if (lookahead is “else”)] “**else”** S | ε

where the condition inside [...] resolves the dangling-else problem.

Parser Generators generating Bottom-Up Parsers

Tool	accepted grammar	generated parsers and their implementation languages
Yacc	BNF-like	LR(1) table-driven, C
JavaCUP	BNF-like	LR(1) table-driven, Java
Jacc	BNF-like	LR(1) table-driven, Java

We might deal with LR parsing at the end of the semester...

The JavaCC Compiler Compiler

- Kind of a “Lex and Yacc” for Java
- Based on LL(k) grammars
- Grammars specified in EBNF
- JavaCC transforms EBNF into an LL(k) parser
- The JavaCC EBNF grammar can have embedded action code written in Java.
- The lookahead can be changed by writing LOOKAHEAD(...).
- The whole input is specified in just one file (not two).
  - Advantage over using Lex (scanner generator) in conjunction with Yacc (parser generator), which would require 2 input files (for scanner & parser).

The JavaCC Input Format

One file:

- header
- token specifications for lexical analysis
- EBNF grammar

Example 1: JavaCC Spec for matching braces

CFG

```
Input ::= MatchedBraces ("\n"|"\r")*
MatchedBraces ::= "{"  MatchedBraces ? "}"
```

```
PARSER_BEGIN(Simple1)

public class Simple1 {

  public static void main(String args[]) throws ParseException {
    Simple1 parser = new Simple1(System.in);
    parser.Input();
  }

}

PARSER_END(Simple1)

void Input() :                                  // LHS of production 1
{}                                              // definitions for production 1
{
  MatchedBraces() ("\n"|"\r")* <EOF>            // RHS for production 1
}

void MatchedBraces() :                          // LHS of production 2
{}
{                                               // RHS for production 2
  "{" ( MatchedBraces() )? "}"
}
```

JavaCC Spec for above CFG

Generating a parser with JavaCC

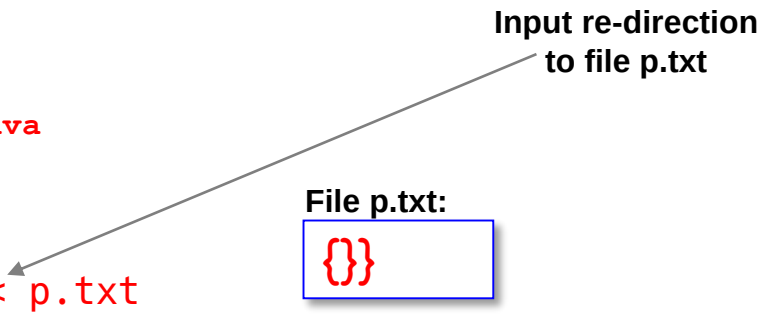
javacc Simple1.jj // generates a parser with a specified name

javac Simple1.java // Simple1.java contains a call to the parser

java Simple1 < p.txt // parses the program in file p.txt

Javacc is installed on the elc1 server.
The example .jj files are provided on yscec.

```
[bburg@elc1 src]$ javacc Simple1.jj
Java Compiler Compiler Version 4.0 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file Simple1.jj . . .
Parser generated successfully.
[bburg@elc1 src]$ javac Simple1.java
[bburg@elc1 src]$ java Simple1
{{}} <CTRL-D for EOF>
[bburg@elc1 src]$ java Simple1 < p.txt
Exception in thread "main" ParseException: Encountered "}" at line 2, column 1.
```



Augmenting a JavaCC Specification with Semantic Actions

- The previous example was a recognizer only.
- We can add declarations and Java code to a JavaCC specification
  - E.g., to build an AST, similar to Assignment 2
  - Code is added to productions of the grammar
  - Code executes when parser reduces a production.
    - Such code is called a **semantic action**.

- On the next slide, additional code has been added to the ``matching braces'' parser spec to count the number of braces.
  - Added code is shown in red.
- Further information on JavaCC can be found at

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

Augmenting a JavaCC Specification with Semantic Actions

```
PARSER_BEGIN(Simple1)

public class Simple1 {
  public static void main(String args[]) throws
    Simple1 parser = new Simple1(System.in);
    int val = parser.Input();
    System.out.println(val);
  }
}

PARSER_END(Simple1)

int Input() :
{ int NrBraces=0; }
{
  NrBraces=MatchedBraces()
  { System.out.println("Finished braces...\n"); }
  ("\n"|"\r")* <EOF>
  { return NrBraces; }
}

int MatchedBraces() :
{ int NrBraces=0; }
{
  "{" ( NrBraces=MatchedBraces() )? "}"
  { return NrBraces + 2; }
}
```

```
void Input() :
{}
{
  MatchedBraces() ("\n"|"\r")* <EOF>
}

void MatchedBraces() :
{}
{
  "{" ( MatchedBraces() )? "}"
}
```

JavaCC spec without actions

Augmenting a JavaCC Specification with Semantic Actions

```
PARSER_BEGIN(Simple1)

public class Simple1 {
  public static void main(String args[]) throws ParseException {
    Simple1 parser = new Simple1(System.in);
    int val = parser.Input();
    System.out.println(val);
  }
}

PARSER_END(Simple1)

int Input() :
{ int NrBraces=0; }
{
  NrBraces=MatchedBraces()
  { System.out.println("Finished braces...\n"); }
  ("\n"|"\r")* <EOF>
  { return NrBraces; }
}

int MatchedBraces() :
{ int NrBraces=0; }
{
  "{" ( NrBraces=MatchedBraces() )? "}"
  { return NrBraces+2; }
}
```

- Input() and MatchedBraces() now return int value
  - the number of matched braces
- Added declaration for NrBraces in definitions part of productions.
- Added Java code in RHSs to compute the number of matched braces in NrBraces variable.
- Can add **arbitrary** Java code anywhere within RHS of production, if between **{...}**
  - called a **semantic action**
  - semantic actions can be used to construct an AST

Example 2: JavaCC Spec for simple language

```
PARSER_BEGIN(SimpleLanguage)
...
PARSER_END(SimpleLanguage)

SKIP : { " " | "\t" | "\n" }

TOKEN :
{ < WHILE: "while" >
| < BEGIN: "begin" >
| < END: "end" >
| < DO:  "do" > | < IF:  "if" > | < THEN: "then" > | < ELSE: "else" >
| < ID: ["a"-"z"](["a"-"z"]|["0"-"9"])* >
}

void Prog() :
{} { StmtList() <EOF> }

void StmtList() :
{} { Stmt() StmtListPrime() }

void StmtListPrime() :
{} { ( ";" Stmt() StmtListPrime() )? }

void Stmt() :
{} { <ID> "=" <ID>
| <WHILE> <ID> "do" Stmt()
| <BEGIN> StmtList() <END>
| LOOKAHEAD( <IF> <ID> <THEN> Stmt() <ELSE> )
  <IF> <ID> <THEN> Stmt() <ELSE> Stmt() | <IF> <ID> <THEN> Stmt()
}
```

- **Syntactic lookahead** to resolve tangling else problem.
  - Syntactic lookahead specifies a varying number of lookahead tokens!
  - If the specified clause matches, the following production is chosen.
  - More information on javacc lookahead:
    - https://javacc.java.net/doc/lookahead.html

Outlook

- Syntax and Semantics of Programming Languages ✓
- Specifying the Syntax of a programming language: ✓
  - CFGs, BNF and EBNF ✓
  - Grammar transformations ✓
- Parsing
  - Top-down parsing (LL) vs. bottom-up parsing (LR) ✓
  - Recursive descent (LL) parser construction ✓
  - LL Grammars ✓
- AST Construction ✓
  - Parse trees vs. ASTs ✓
- Parser Generators ✓
- Chomsky's Hierarchy

Chomsky's Hierarchy

Depending on the productions $\alpha \rightarrow \beta$, four types of grammars can be distinguished:

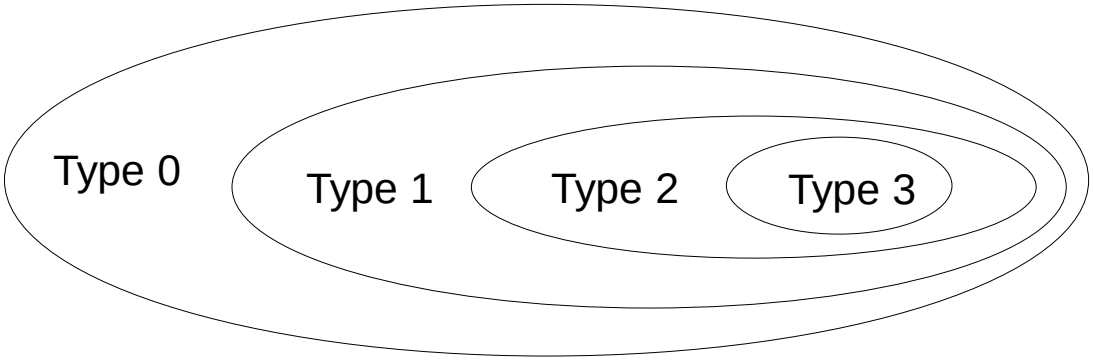
Grammar	Name	Definition
Type 0	unrestricted grammars	$\alpha \rightarrow \beta$ no restrictions
Type 1	context-sensitive grammars	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type 2	context-free grammars	$A \rightarrow \alpha$
Type 3	regular grammars	$A \rightarrow a \mid aB$

Note:

- "a" is a terminal symbol
- A and B are single non-terminals

α, β, γ are strings of terminals and nonterminals

Relationships between the 4 types of languages



- A Type *k* language is a proper subset of a Type *k-1* language.
- The existence of a Type 0 language is proven Hopcroft, Ullman, “Introduction to Automata Theory, Languages, and Computation”
- But any programming language one can think of turns out to be context-sensitive.

Limitations of Regular Grammars

- Cannot generate nested constructs
- The following language is not regular

$$L = \{a^n b^n \mid n \geq 0\}$$

- But L is context/free:

$$S \rightarrow \varepsilon \mid a\,S\,b$$

- Regular grammars (expressions) powerful enough to specify tokens which are not nested.
- Regular grammars (finite automata) cannot count.

Limitations of Context-Free Grammars

Context-free languages include only a subset of all languages.

Examples of non-context-free constructs:

- An abstraction of a variable-declaration-before-use:

$$L = \{ w c w \mid w \text{ is in } (a \mid b)^* \}$$

  where the first w represents a declaration and the second represents its use.

- A method called with the right number of arguments:

$$L = \{ a^n b^m c^n d^m \mid n \geq 1, m \geq 1 \}$$

  where a^n and b^m represent formal parameter lists in two methods with n and m arguments, and c^n and d^m represent actual parameter lists in two calls to the two methods.

All language features that cannot be captured by CFGs need to be checked during semantic analysis.

Outlook

- Syntax and Semantics of Programming Languages ✓
- Specifying the Syntax of a programming language: ✓
  - CFGs, BNF and EBNF ✓
  - Grammar transformations ✓
- Parsing
  - Top-down parsing (LL) vs. bottom-up parsing (LR) ✓
  - Recursive descent (LL) parser construction ✓
  - LL Grammars ✓
- Parser Generators ✓
- AST Construction ✓
  - Parse trees vs. ASTs ✓
- Chomsky's Hierarchy ✓