

Syntax Analysis

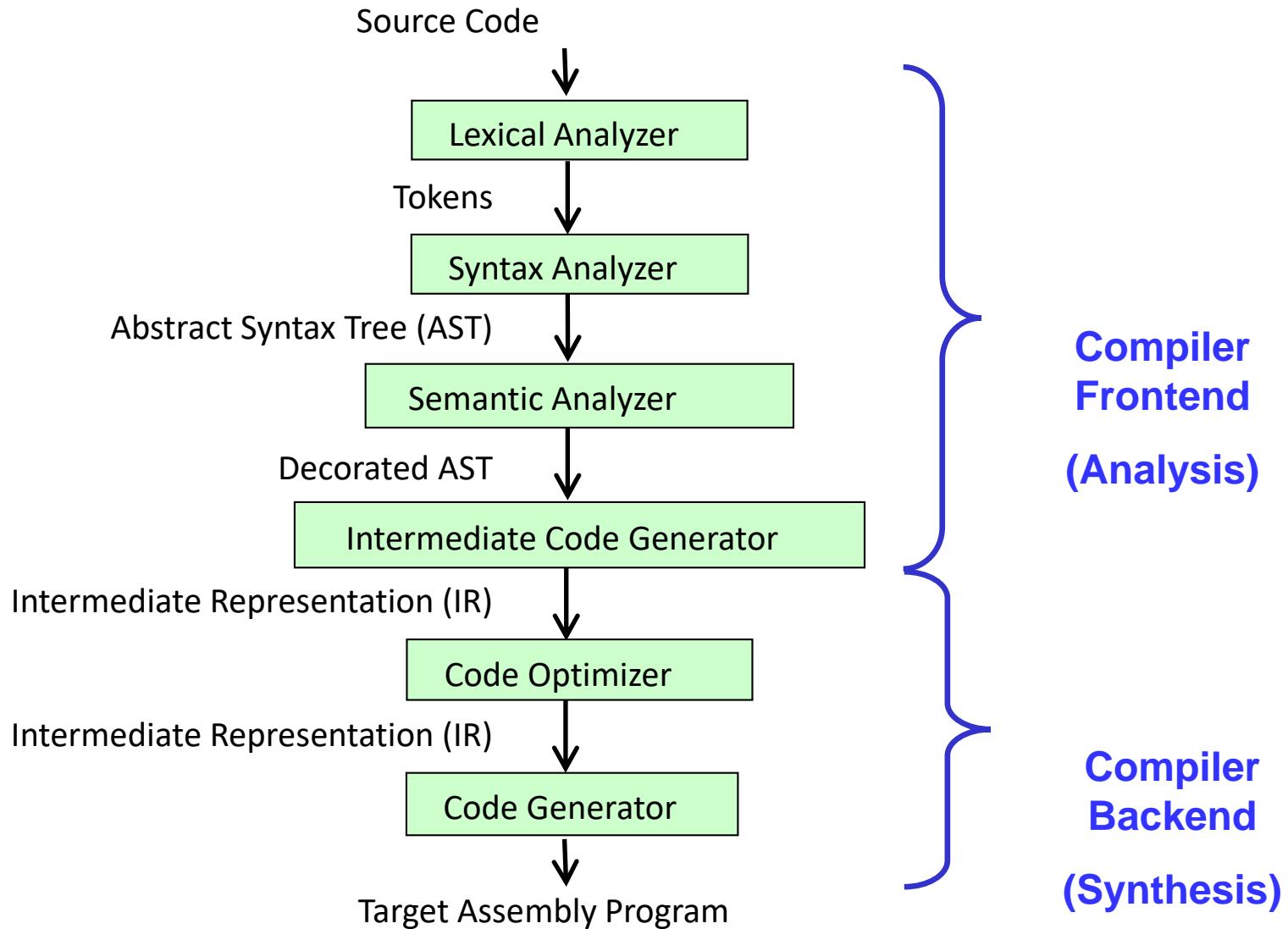
Jeonggeun Kim
Kyungpook National University



Outlook

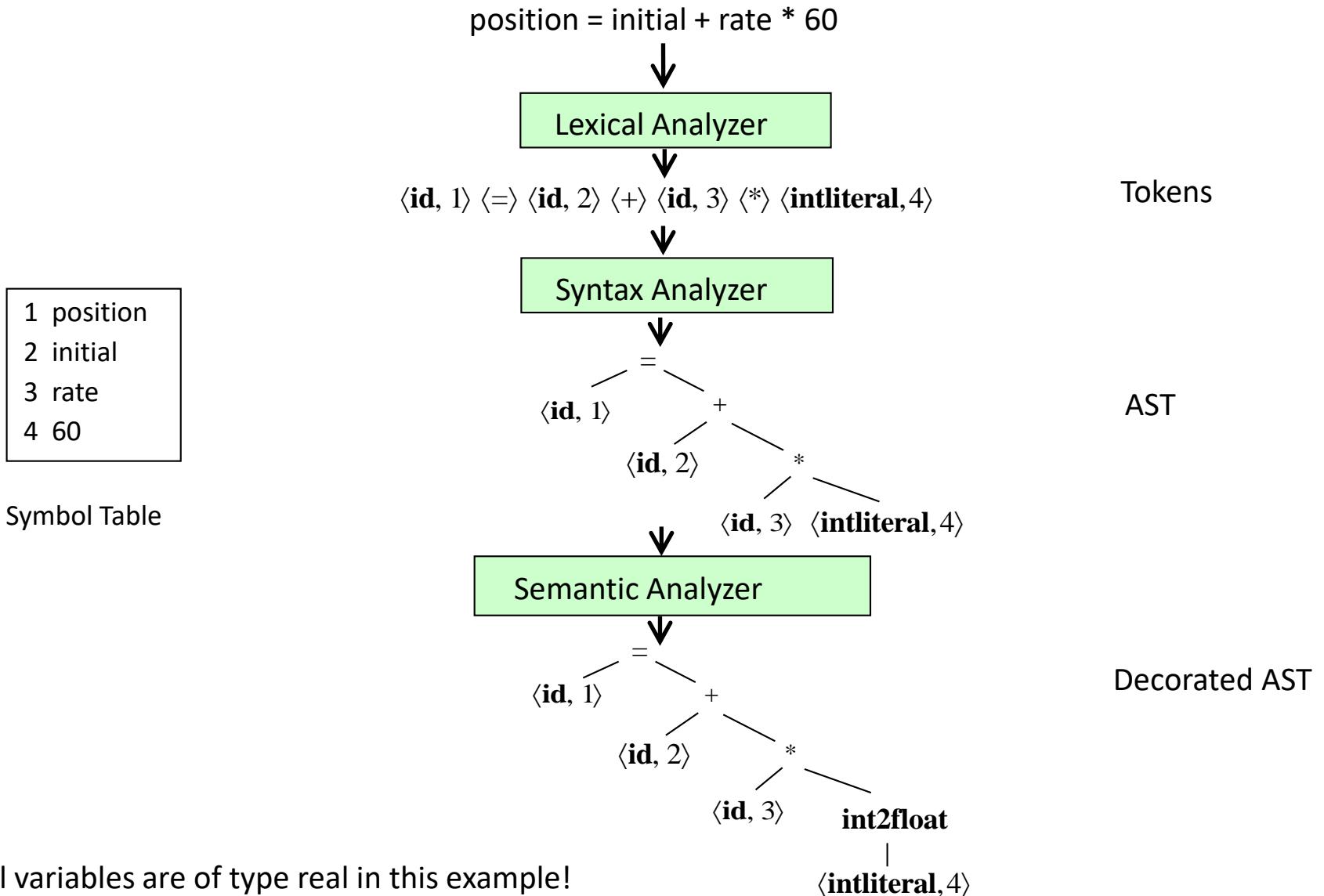
- Syntax and Semantics of Programming Languages
- Specifying the Syntax of a programming language:
 - CFGs, BNF and EBNF
 - Grammar transformations
- Parsing
 - Top-down parsing (LL)
 - Recursive descent (LL) parser construction
 - LL Grammars
- AST Construction
 - Parse trees vs. ASTs
- Chomsky's Hierarchy

Recapitulate: the structure of a compiler...



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

Recapitulate: the structure of a compiler...



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:

I like the black cat. // micro-English does not contain adjectives.

Derivations

- From a grammar we can **derive** strings (a.k.a. **sentences**) by generating sequences of tokens.
- 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' , 'greater than' , '==' , ID , INTLITERAL
 - Nonterminal Symbols (Non-terminals)
 - 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
 - Productions
 - Example: sentence ::= noun verb

Context-free Grammars

CFGs can be expressed in BNF (Backus-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
           | ...
```

Context-free Grammars

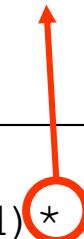
For our **convenience**, we will use EBNF or ``Extended BNF'' rather than simple BNF:

EBNF = BNF + regular expressions

* means 0 or more occurrences of ...

Example in EBNF:

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



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)
```

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** nonterminal.

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;
```

Grammar & Program, side-by-side

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

```
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 terminal s 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" | ... | "9"  
Operator ::= "+" | "-" | "*" | "/"
```

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''

Grammar Transformations (1)

Left factorization:

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

Example:

single-Command

$::= V\text{-name} ::= \text{Expression}$

$| \quad \text{if Expression then single-Command}$

$| \quad \text{if Expression then single-Command}$

$\quad \quad \quad \text{else single-Command}$

$Y = \epsilon$

X

Z

Y

single-Command

$::= V\text{-name} ::= \text{Expression}$

$| \quad \text{if Expression then single-Command}$

$(\epsilon \mid \text{else single-Command})$



Grammar Transformations (2)

Elimination of Left Recursion:

$$N ::= X \mid N Y \quad \xrightarrow{\hspace{1cm}} \quad N ::= X Y^*$$

Example:

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

left factorization



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

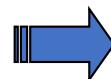
eliminate left recursion



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

Grammar Transformations (3)

Substitution of non-terminal symbols

$$\begin{array}{l} N ::= X \\ M ::= \alpha \ N \beta \end{array}$$

$$\begin{array}{l} N ::= X \\ M ::= \alpha \ X \ \beta \end{array}$$

Example:

single-Command

$::= \text{for } \text{contrVar} \ := \ \text{Expression}$

$\text{to-or-dt} \ \text{Expression} \ \text{do} \ \text{single-Command}$

$\text{to-or-dt} \ ::= \ \text{to} \ | \ \text{downto}$

single-Command $::=$

$\text{for } \text{contrVar} \ := \ \text{Expression}$

$(\text{to} \mid \text{downto}) \ \text{Expression} \ \text{do} \ \text{single-Command}$

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)
 - Recursive descent (LL) parser construction
 - LL Grammars
- AST Construction
 - Parse trees vs. ASTs
- Chomsky's Hierarchy

Parsing

We will now look at parsing.

Topics:

- Derivations and parse trees
- Ambiguous grammars
 - Operator precedence
 - Operator associativity
- Recursive descent parsing
 - What it is.
 - How to implement it given an EBNF specification.

Parse Trees

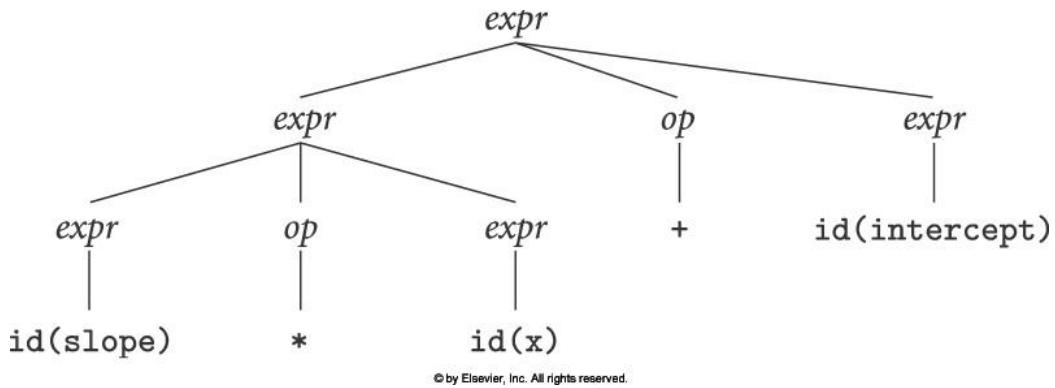
Correspondence between a **derivation** and a **parse tree**:

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

generate string “slope * x + intercept”:

expr ::= expr op expr

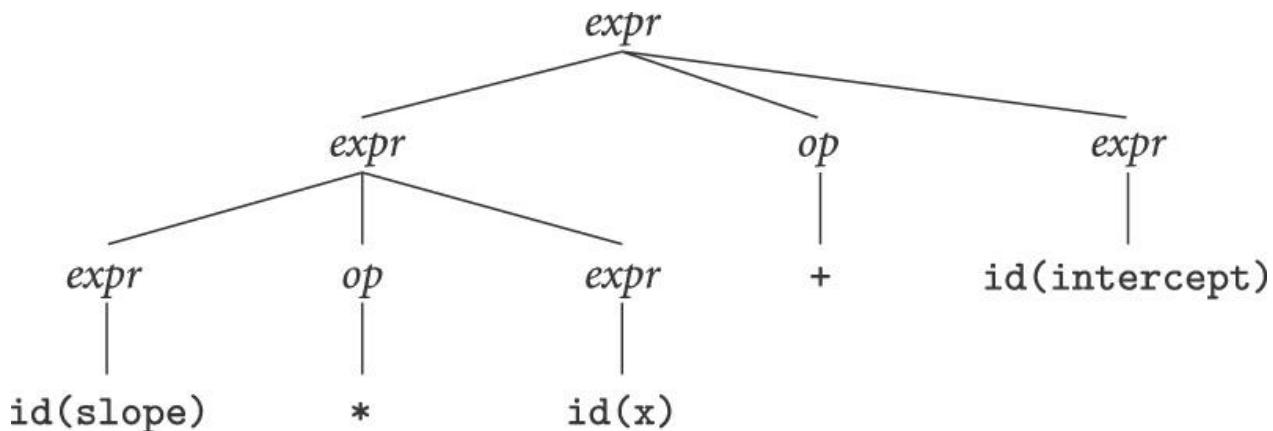
- expr op id
 - expr + id
 - expr op expr + id
 - expr op id + id
 - expr * id + id
 - id * id + id
- (slope) (x) (intercept)



Parse Trees (cont.).

Correspondence between a **derivation** and a **parse tree**:

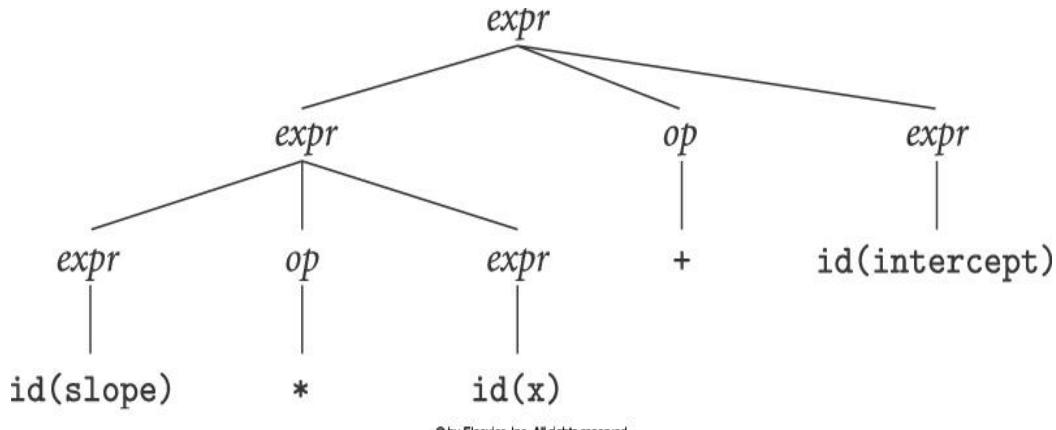
- The parse tree's internal nodes are nonterminals in the productions.
- The children of a node are the terminals and nonterminals on the right-hand side of a production.
- The leaves of the parse tree are the terminals (tokens).
- When read from left to right, the leaves make up the sentence of the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



Ambiguous Grammars

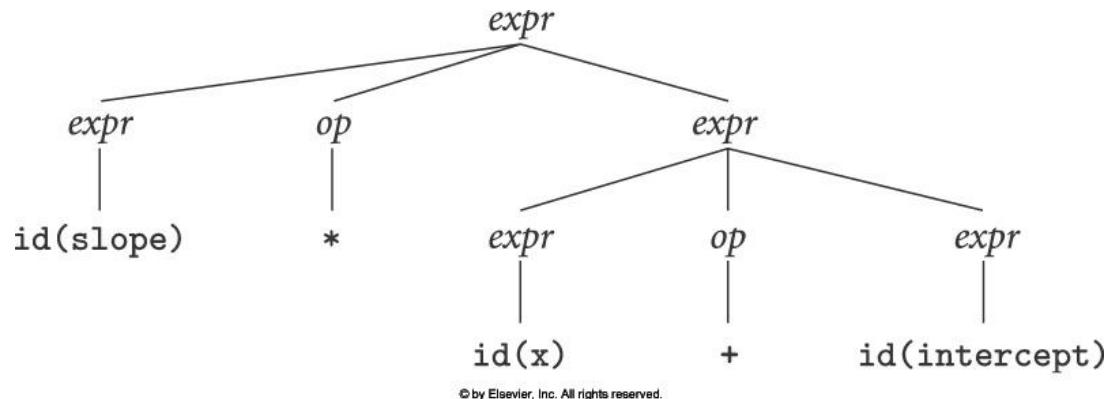
Ambiguity: one sentential form has **several** distinct parse trees.

Example: $\text{slope} * \text{x} + \text{intercept}$



Meaning implied by parse tree:

$(\text{slope} * \text{x}) + \text{intercept}$



$\text{slope} * (\text{x} + \text{intercept})$

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, **precedence** and **associativity** of operators are used to disambiguate the productions.
 - We rewrite the grammar to make it un-ambiguous:

expr and term
trees can only
grow to the left
(left-to-right
associativity)

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

Force + to be higher
than * in the parse tree
(precedence)

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 ?”.

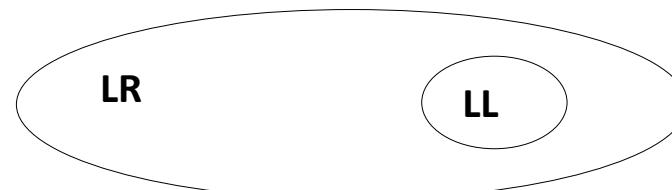
- A **recognizer** uses a CFG to check the syntax of the program.
It answers ``**Yes**'' if the program corresponds to the CFG, and
``**No**'' otherwise (**Assignment 1**).

- **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, and
 - builds the AST of the program (**Assignment 2**).

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:

$X ::= X Y \dots$

and

$X ::= Y Z \dots$

$Y ::= X \dots$

- LL(k) where k is lookahead depth: if $k=1$, alternative **productions** with **common prefixes** of length >1 **cannot** be handled:

$X ::= a b \dots | a c \dots$

- A top-down parser constructs a parse tree from the root down.
- Not too difficult to implement a predictive parser for an LL(1) grammar in BNF by hand using *recursive descent*.

Top-down Parsing Example: Micro English

- Example parsing of a sentence in ``Micro English''

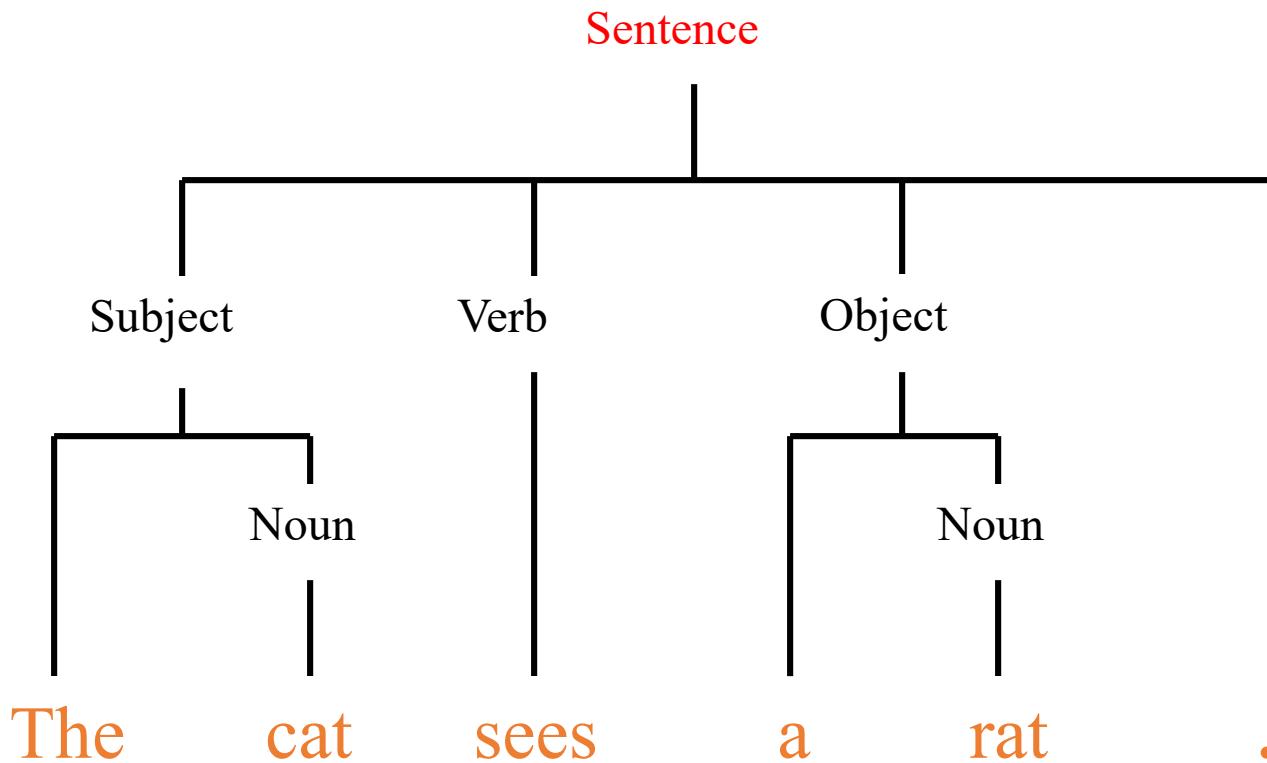
```
Sentence   ::= Subject Verb Object .
Subject    ::= I | a Noun | the Noun
Object     ::= me | a Noun | the Noun
Noun       ::= cat | mat | rat
Verb       ::= like | is | see | sees
```

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

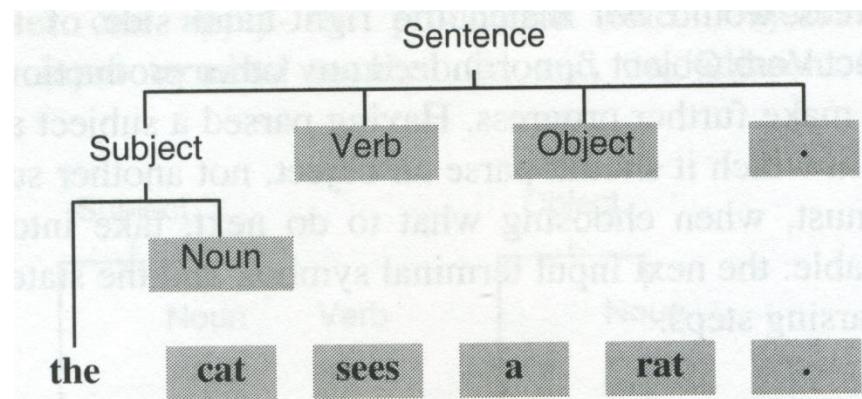
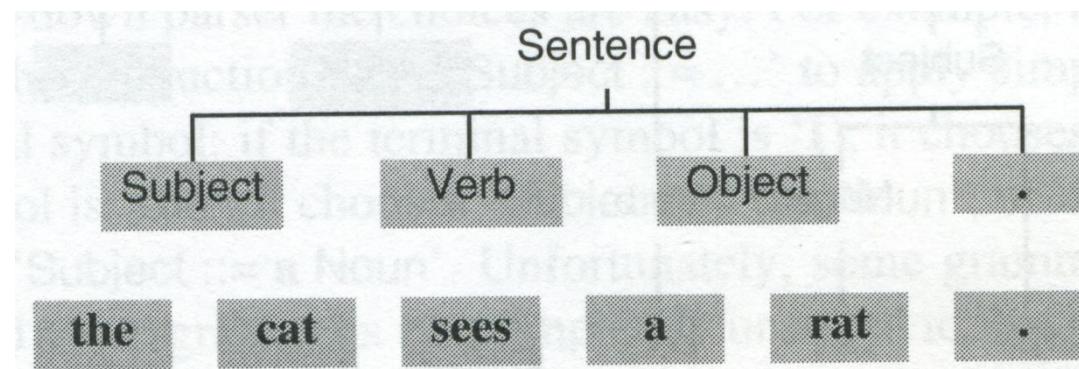
The rat like me.
I see the rat.
I sees a rat.

Top-down LL Parsing

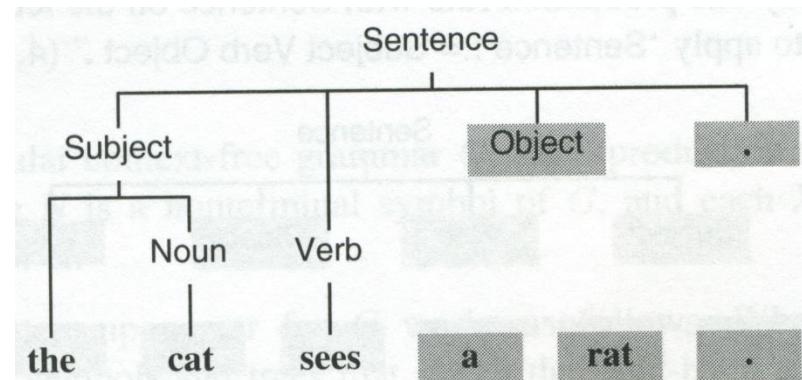
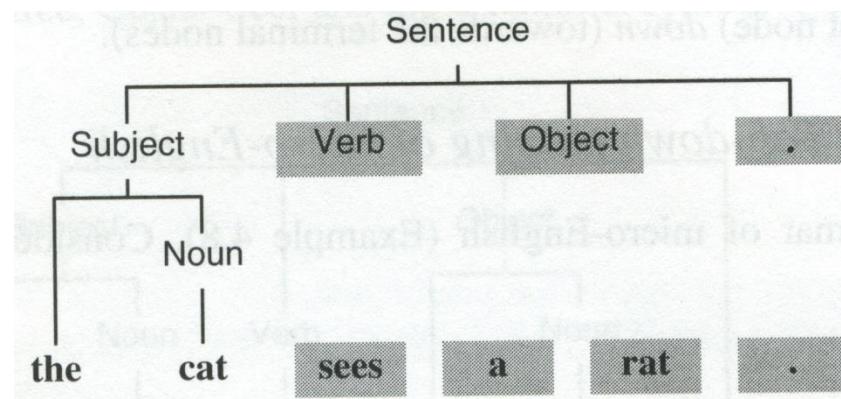
The parse tree is constructed starting at the top (**root**).



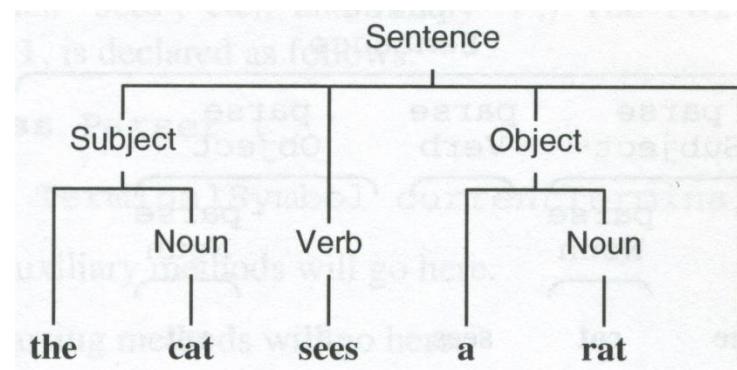
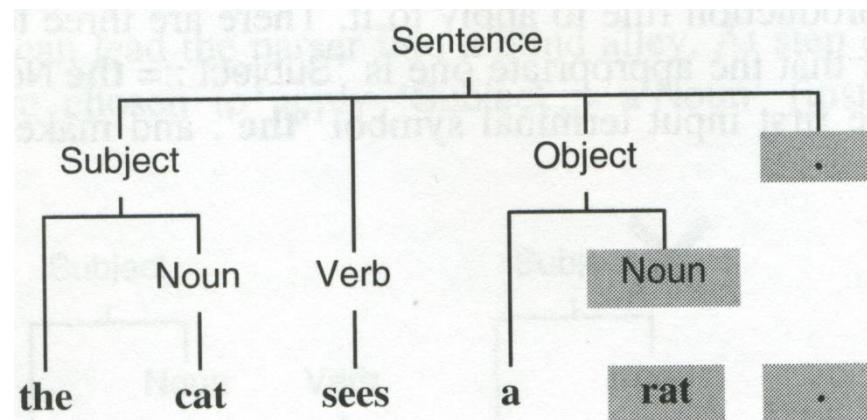
```
Sentence ::= Subject Verb Object .
Subject ::= I | a Noun | the Noun
Object ::= me | a Noun | the Noun
Noun ::= cat | mat | rat
Verb ::= like | is | see | sees
```



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



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) ✓
 - Recursive descent (LL) parser construction
 - LL Grammars
- AST Construction
 - Parse trees vs. ASTs
- Chomsky's Hierarchy

Recursive Descent Parsing

- **Recursive descent parsing** is a straightforward LL (top-down) parsing method.
- We will now look at how to develop a recursive descent parser from an EBNF specification.
- **Key idea:** the parse tree structure corresponds to the caller–callee relationship of parsing procedures that call each other at runtime.

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  
}
```

Recursive Descent Parser: Parsing Methods

```
Object ::= me | a Noun | the Noun
```

```
Verb ::= like | is | see | sees
```

```
private void parseObject() {
```

```
    ?
```

```
}
```

```
private void parseVerb() {
```

```
    ?
```

```
}
```

Systematic Development of a RD Parser

- (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** `parseN` method for each nonterminal **N**
- **public** `parse` method that
 - gets the first token from the scanner
 - calls `parseS` (**S** is the start symbol of the grammar)
 - ensure that all tokens are consumed upon return from `parseS()`

MiniC: Systematic Development of an RD Parser (1 & 2)

Common prefixes require left-factorization:

```
program ::= ( function-def | variable-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 left-factorization we get:

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

MiniC: Systematic Development of an 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.");  
    }  
}
```

The Error Reporter:

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

```
// Example error report:  
reportError("Type specifier instead of % expected",  
           Token.spell(currentToken.kind), srcpos);
```

Output: "ERROR: Type specifier instead of % expected 9..10, line 22."

MiniC Parser: 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() { // constructor
```

```
    super();
```

```
}
```

```
}
```

```
// Example syntax error generation:
```

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

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

Output: "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 2...  
    }  
    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!
- 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 XY



parse $X()$;
parse $Y()$;

Algorithm to convert EBNF into a RD parser

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.

*parsing X**



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

For an example, see next slide.

Example: parseExpr

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



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

Example: parseProgram

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



```
private 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) ✓
 - Recursive descent (LL) parser construction ✓
 - LL Grammars
- AST Construction
 - Parse trees vs. ASTs
- 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
 - Left-to-right parsing of the input stream
 - Leftmost derivation
 - 1 token look-ahead in the input stream
- What grammars are LL(1)?
 - A grammar containing left-recursion is not LL(1)
 - A grammar containing common prefixes is not LL(1)
 - An ambiguous grammar is 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

$$a, b, c, \dots \in V_t$$

$$\alpha, \beta, \gamma, \dots \in V^*$$

$$A, B, C, \dots \in V_n$$

$$u, v, w, \dots \in V_t^*$$

- 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 $\text{FIRST}(\alpha)$ 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
$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$$
- 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**?

```
private 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:

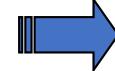
$\text{FIRST}(\text{ Expr } "+" \text{ INT }) = \{\text{INT}\}$

$\text{FIRST}(\text{ INT }) = \{\text{INT}\}$

*Problem2: infinite recursion via
parseExpr().*

Left-Recursive Grammars are not LL(1)

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

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



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

```
private 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**?

```
private 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  
    }  
}
```

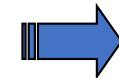
Note: this parseExpr() invocation does not lead to infinite recursion, because we have consumed input in between (e.g., Token.PLUS).

Problem of overlapping cases in FIRST (TERM) !

Grammars with Common Prefixes are not LL(1)

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

perform left-factorization:

 $X Y \mid X Z$  $X(Y \mid Z)$ 

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

```
private 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.

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

For many context-free grammars, such a grammar does not exist:

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, $\text{FIRST}(\alpha)$ is

- the set of terminal symbols that **start** sentences derived from α
 $\{c \in V_t \mid \alpha \Rightarrow^* c\beta\}$
- If $\alpha \Rightarrow^* \epsilon$ then $\epsilon \in \text{FIRST}(\alpha)$

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

1. If $X \in V_t$ then $\text{FIRST}(X) = \{X\}$

The algorithm to compute $\text{FIRST}(X)$ for all grammar symbols X .

2. If $X ::= \epsilon$ then add ϵ to $\text{FIRST}(X)$

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

3. If $X ::= Y_1 Y_2 \dots Y_k$:

Recursive invocation!

(a) Put $\text{FIRST}(Y_1) - \{\epsilon\}$ in $\text{FIRST}(X)$

(b) $\forall i : 1 < i \leq k$, if $\epsilon \in \text{FIRST}(Y_1) \cap \dots \cap \text{FIRST}(Y_{i-1})$

(i.e., $Y_1 \dots Y_{i-1} \rightarrow^* \epsilon$)

then put $\text{FIRST}(Y_i) - \{\epsilon\}$ in $\text{FIRST}(X)$

(c) if $\epsilon \in \text{FIRST}(Y_1) \cap \dots \cap \text{FIRST}(Y_k)$ then put ϵ in $\text{FIRST}(X)$

Repeat Step 3 until no more additions can be made.

FIRST-Set Example 1

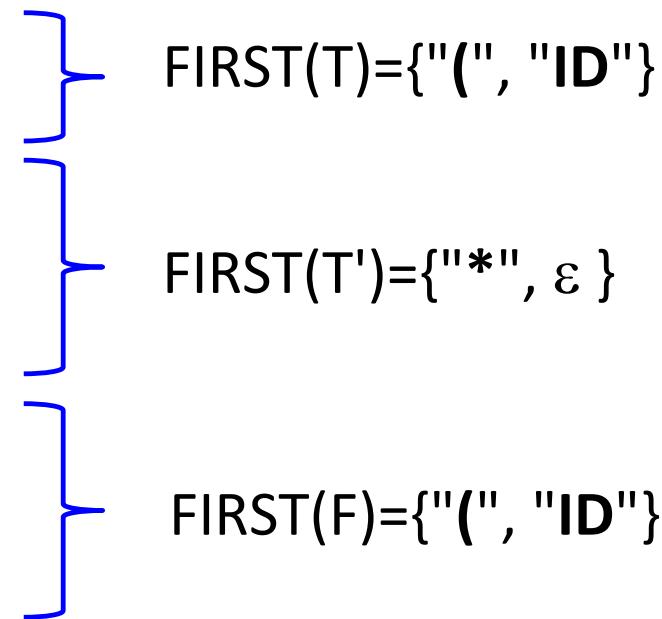
$T ::= F T'$

$T' ::= "*" F T'$

$T' ::= \epsilon$

$F ::= "(" T ")"$

$F ::= "ID"$



FIRST-Set Example 2

$T ::= S F T'$	{	$\text{FIRST}(T)=\{"-", "\text{ID}", "("\}$
$T' ::= "*" S F T'$		$\text{FIRST}(T')=\{"*\", \epsilon\}$
$T' ::= \epsilon$		
$F ::= "\text{ID}"$	{	$\text{FIRST}(F)=\{"\text{ID}\", "("\}$
$F ::= "(" E ")"$		
$S ::= "-"$		
$S ::= \epsilon$	{	$\text{FIRST}(S)=\{\epsilon, "-"\}$

- Because $S \rightarrow \epsilon$, $\text{FIRST}(T)$ contains $\text{FIRST}(F)$
 - See Rule 3(b) of the FIRST-set algorithm

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 right after it.

Note: a terminal symbol has no FOLLOW set.

1. Put $\$$ in FOLLOW(S)

' S ' is the CFG's start symbol, and ' $\$$ ' is the input end marker (EOF).

Note: initialize $\text{Follow}(X) = \{\}$ for all non-terminals.

2. If $A ::= \alpha B \beta$:

(a) Put $\text{FIRST}(\beta) - \{\epsilon\}$ in FOLLOW(B)

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

(b) if $\beta = \epsilon$ (i.e., $A ::= \alpha B$) or $\epsilon \in \text{FIRST}(\beta)$ (i.e., $\beta \rightarrow^* \epsilon$) then put FOLLOW(A) in FOLLOW(B)

Repeat Step 2 until no more additions can be made.

FOLLOW Set Example 1

s1. Put \$ in FOLLOW(S)

2. If $A ::= \alpha B \beta$:

(a) Put FIRST(β) - { ϵ } in FOLLOW(B)

(b) if $\beta = \epsilon$ (i.e., $A ::= \alpha B$) or $\epsilon \in \text{FIRST}(\beta)$ (i.e., $\beta \rightarrow^* \epsilon$) then put FOLLOW(A) in FOLLOW(B)

Repeat Step 2 until no more additions can be made.

Sentence ::= Subject Verb "."

Subject ::= "the" X Y α B β

X ::= "dog"

Y ::= "owner" | ϵ

Verb ::= "eats"

- According to Rule 2(a), FIRST(Y) - ϵ must be added to FOLLOW(X).
- Because $Y \rightarrow \epsilon$, FOLLOW(Subject) must be added to FOLLOW(X)
 - Rule 2(b)
 - $\text{FOLLOW}(X) = \{"\text{owner"} , \text{"eats"}\}$

FOLLOW-Sets: Processing Production $A ::= \alpha B \beta$

For each production $A ::= \alpha B \beta$, Rule (2) on the previous slide must be applied to add elements to $\text{FOLLOW}(B)$.

- α is a string of terminal and nonterminal symbols, including ϵ
- B is a single non-terminal
- β is a string of terminal and nonterminal symbols, including ϵ

Key-Point:

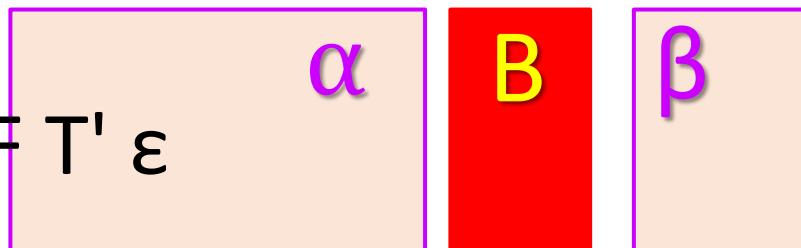
- For a given production p , the **non-terminal B** in string $\alpha B \beta$ may match in **zero, one or more than one positions** of the right-hand side of p .
 - For every non-terminal in the RHS of p , you have one matching position.
- Therefore, Rule (2) will have to be applied **zero, one, or more than one times** for production p .
- Example: see following slide.

Anchoring the $\alpha B \beta$ Pattern

Assume production $T' ::= "*" F T'$

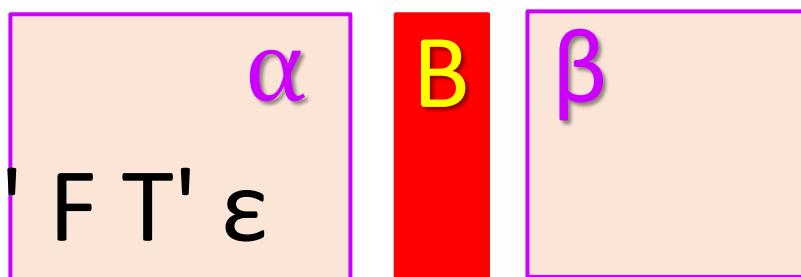
- Pattern $\alpha B \beta$ can be anchored in 3 positions on the RHS of this production:

$T' ::= \epsilon "*" F T' \epsilon$



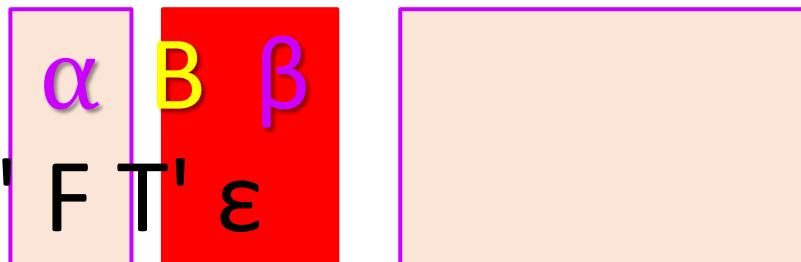
Position 1:
FOLLOW(T')

$T' ::= \epsilon "*" F T' \epsilon$



Position 2:
FOLLOW(F)

$T' ::= \epsilon "*" F T' \epsilon$



Position 3:
FOLLOW("*)
(undefined!)

FOLLOW-Set Example 2:

$T ::= F T'$

$T' ::= "*" F T'$

$T' ::= \epsilon$

$F ::= (" T ")$

$F ::= "ID"$

Symbol	FIRST
T	$\{(" ", "ID")\}$
T'	$\{"*", \epsilon\}$
F	$\{(" ", "ID")\}$

Symbol	FOLLOW
T	$\{\$\}$
T'	$\{\}$
F	$\{\}$

Iteration 1

Symbol	FOLLOW
T	$\{\$, ")"\}$
T'	$\{\$\}$
F	$\{"*", \$\}$

Iteration 2

Symbol	FOLLOW
T	$\{\$, ")"\}$
T'	$\{\$, ")"\}$
F	$\{"*", \$, ")"\}$

Iteration 3

- We processed the productions top to bottom.
 - The order is arbitrary.
 - A different order may require a different number of iterations to derive the final result.
- Processing terminates when there is no more change to a FOLLOW set.

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:

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

An Example from our MiniC CFG: for-loop

for_stmt ::= "for" "(" a_expr ";" ... ")" stmt

a_expr ::= "ID" "=" $\underbrace{\text{expr} \mid \varepsilon}_{\alpha}$ β

- If $\beta \Rightarrow^* \varepsilon$ then $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \{ \}$
 - $\text{FOLLOW}(a_expr) = \{";"\}$
 - $\text{FIRST}(\alpha) = \{"ID"\}$
 - In conclusion, the a_expr production does not prevent the above CFG from being LL(1).
- } disjoint!

Non-LL(1) Grammar Examples

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

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

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

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) overview ✓
 - Recursive descent (LL) parser construction ✓
 - LL Grammars ✓
- AST Construction
 - Parse trees vs. ASTs
- 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) EB NF 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 1 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.)

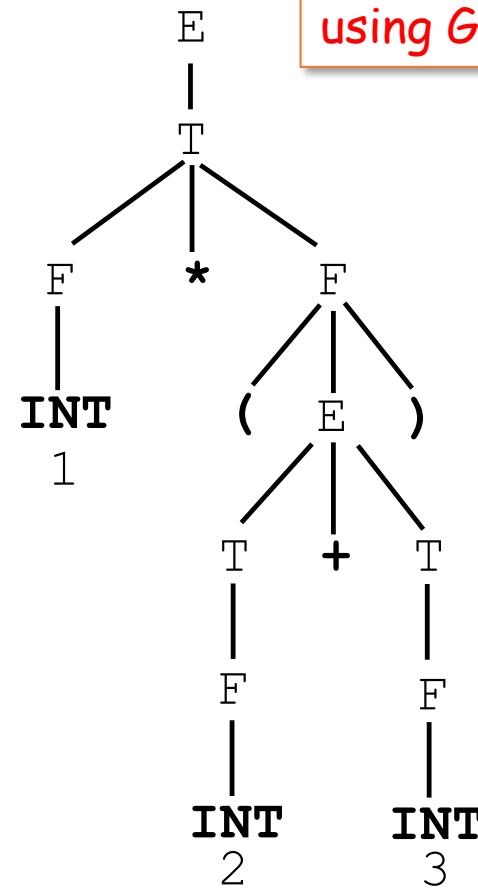
```

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.)

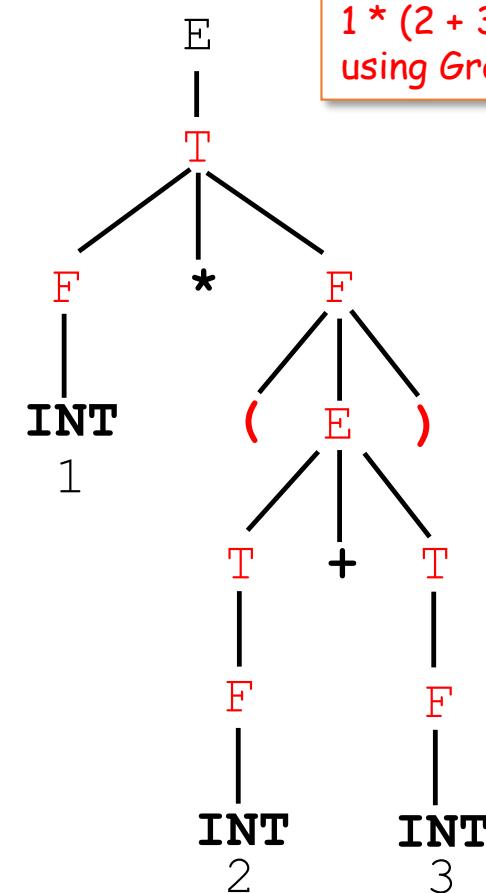
```
E ::= E+E | E-E | E*E | E/E | (E) | INT | (E)
```

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

Grammar 2

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:

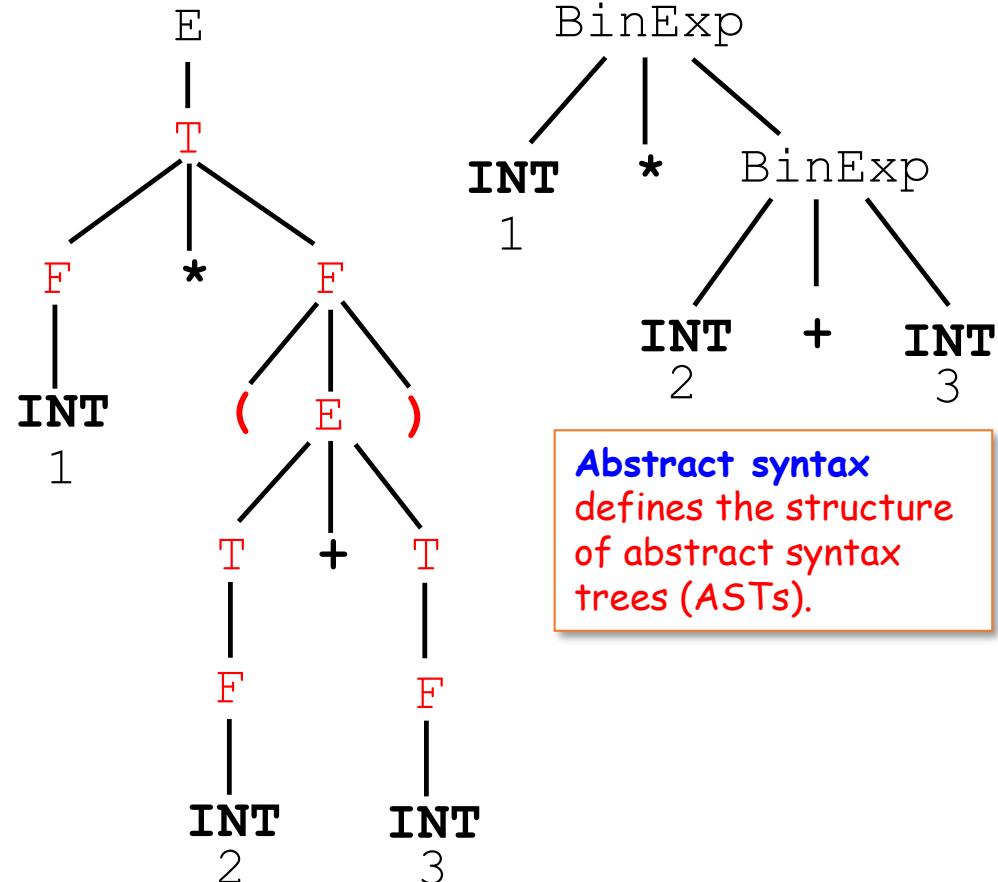
- Nonterminals that define operator precedence and associativity (this information is **implicit** in trees).
- Punctuation tokens (parentheses, semi-colons, ...)
- Keywords

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).

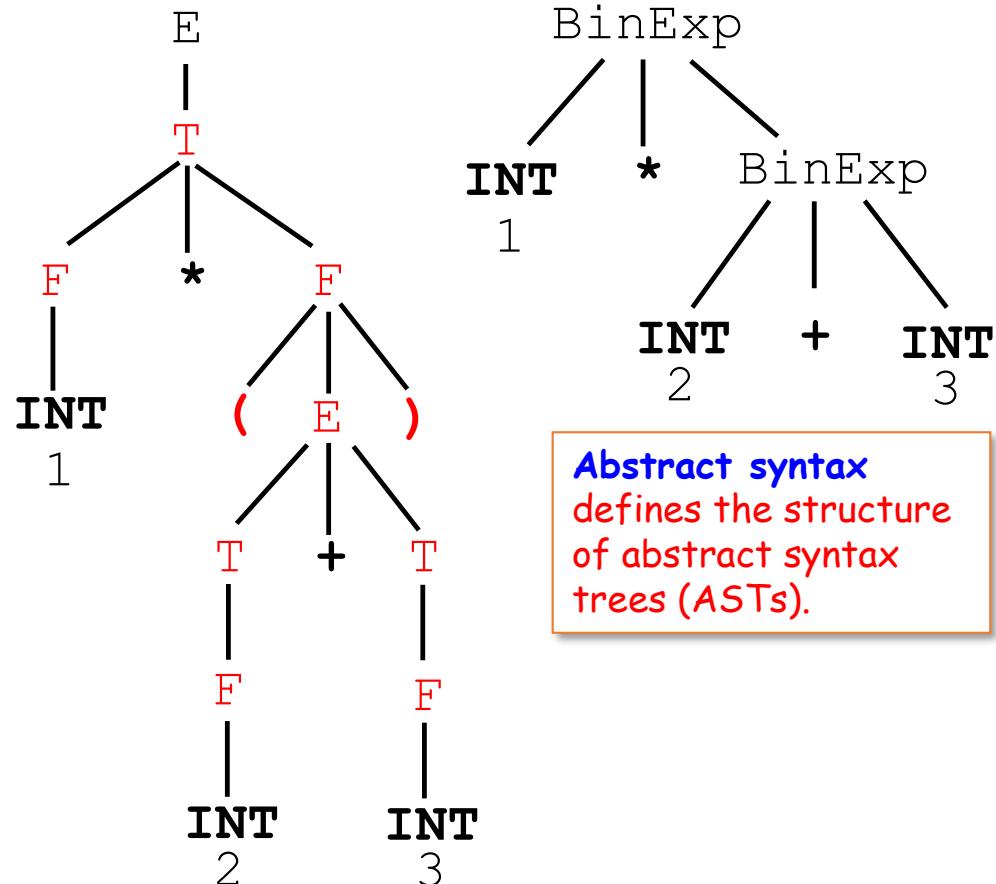
For Assignment 2, you'll get

- 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.

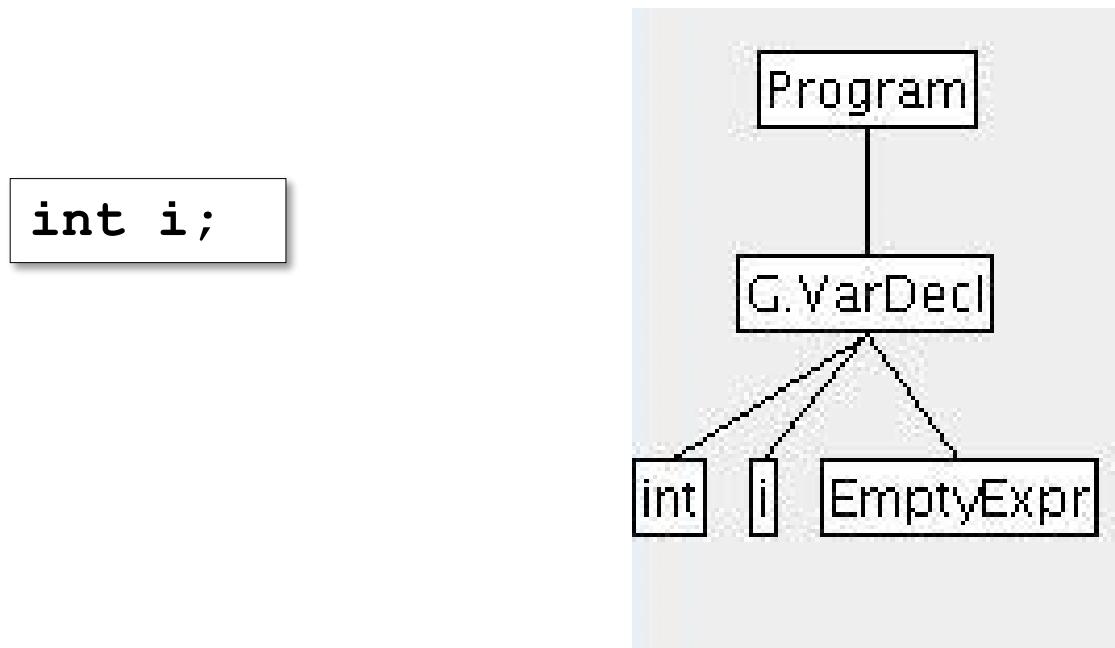
Overview of the MiniC AST Classes

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

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);
```



Assignment 2

- 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 2:

AST Classes

Parser (your Assignment 1)

Add calls to constructors of AST classes

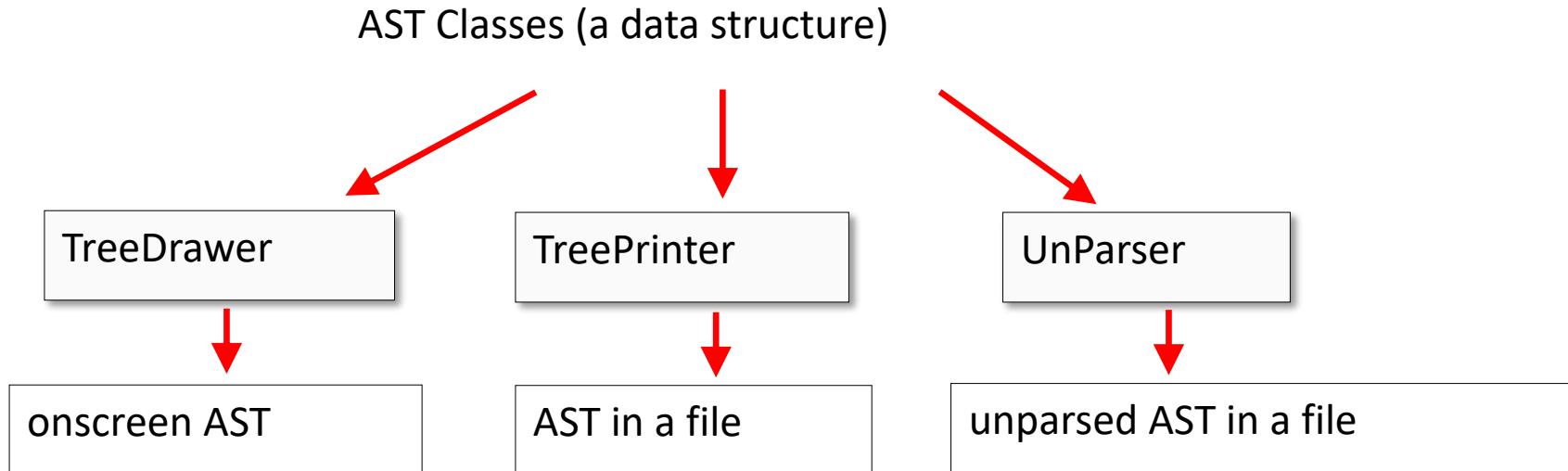
MiniC
Program

Parser that builds
an AST

AST

- The larger part of Assignment 2 consists of inserting the constructors into your parser.

Programming Environment for Assignment 2



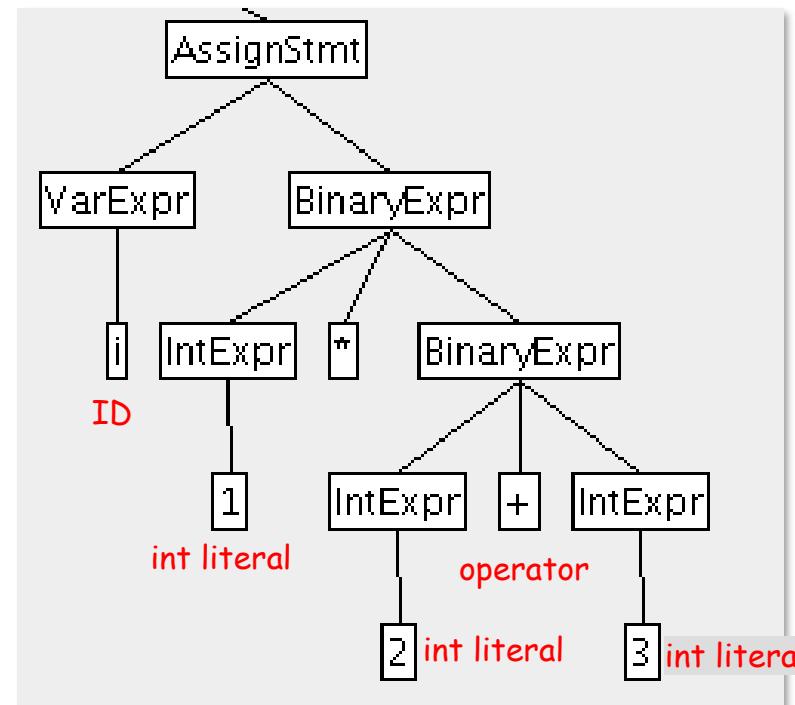
- All three packages implemented using the Visitor Design Pattern.
 - Details on this design pattern will be covered in a subsequent lecture.
- Use the design pattern to traverse the AST (= “tree-walk”)
- During traversal, generate the desired output
- We will use this pattern in Assignment 2.
- **Tree-walkers** can be generated automatically from **attribute grammars**.

Example:

- MiniC code snippet:

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

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



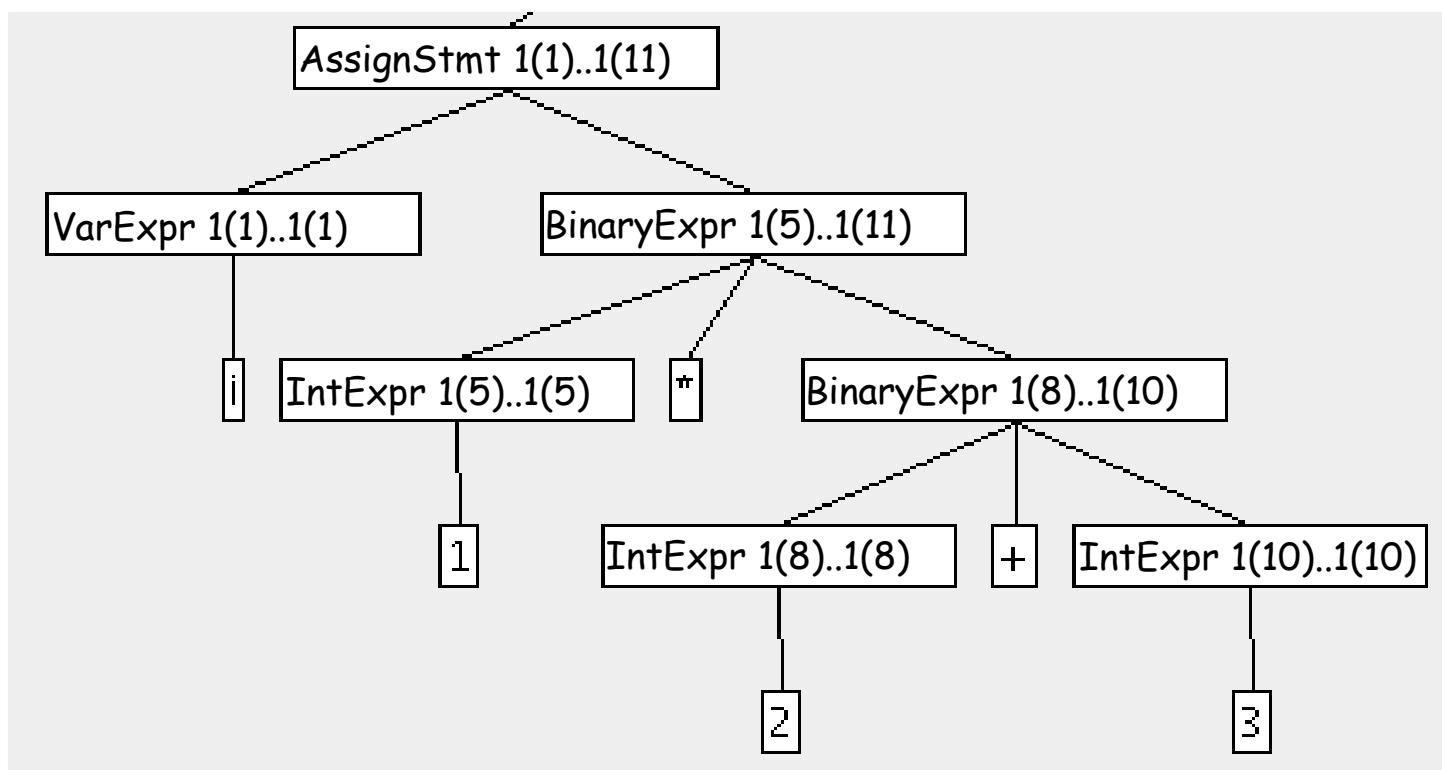
Example:

- 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**:



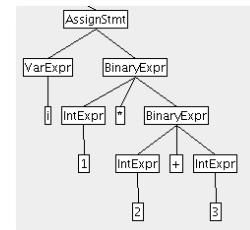
Example:

- 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 operator.

Example: MiniC Stmt ASTs

Stmt

```
::= compound-stmt  
| ID "=" Expr ";"  
| while "(" Expr ")" stmt
```

CompoundStmt {...}
AssignStmt
WhileStmt

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

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

...

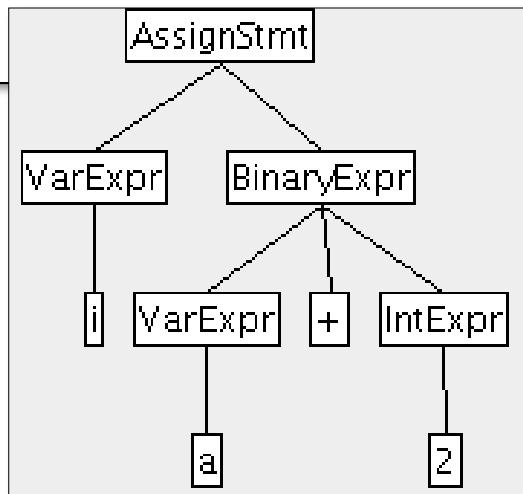
Example: MiniC Stmt ASTs

Stmt

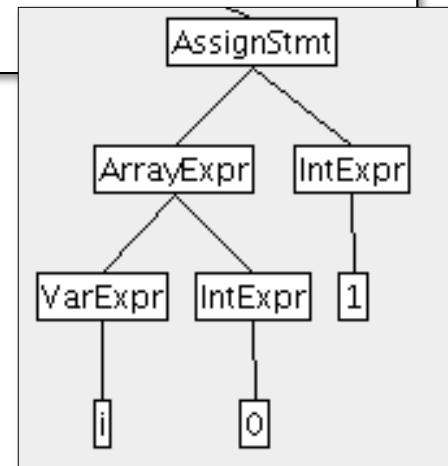
```
::= compound-stmt           CompoundStmt
  | ID "=" Expr ";"          AssignStmt
  | ID "[" Expr "]" "=" Expr ";" AssignStmt
```

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

i = a + 2;



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

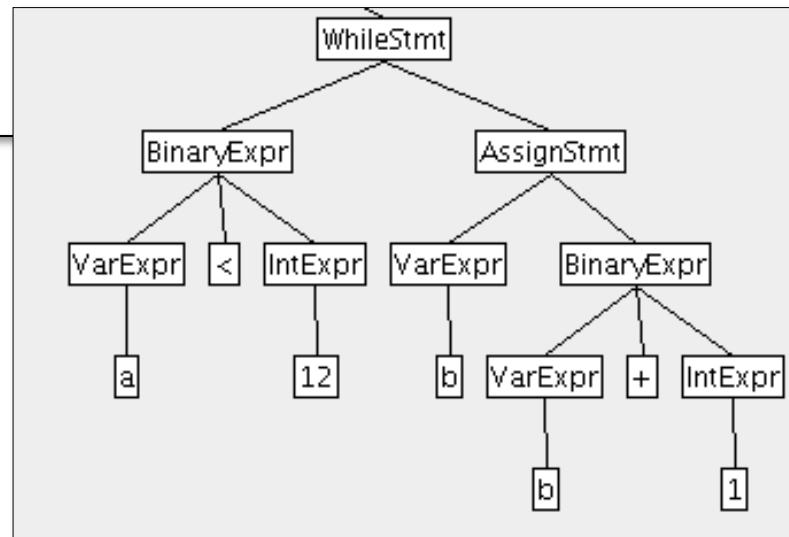


Example: MiniC Stmt ASTs

```
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; // common to all terminals  
    ...  
}  
  
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; // loop condition expression  
    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**.

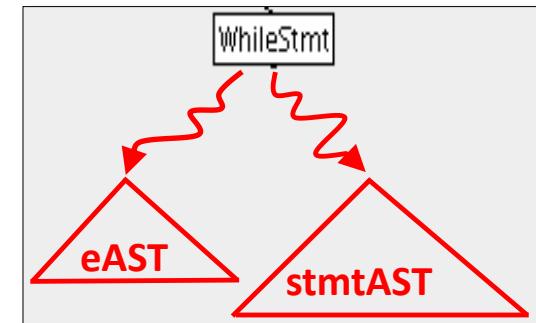
Parsing of a MiniC *while* statement

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



```
private 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 1. 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 ) ?
```



```
private 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* "}"
```



```
private 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))
    {
        // Case1:
        return new EmptyCompoundStmt (previousTokenPosition);
    } else {
        // Case 2:
        return new CompoundStmt (D, S,
                               previousTokenPosition);
    }
}
```

{ }

EmptyCompoundStmt

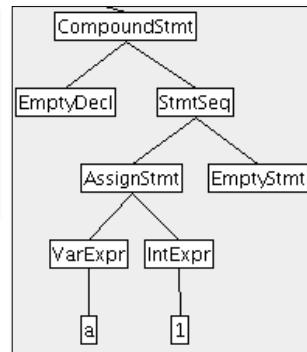
MiniC statement sequence ASTs grow to the right...

```
private 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);  
}
```

{ }

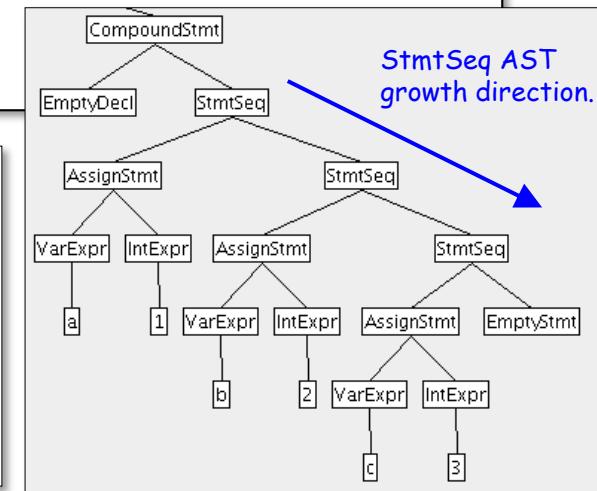
EmptyCompoundStmt

{
 a=1;
}



See Case 1 on
previous page.

{
 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
```



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

The Parsing Method for Stmt

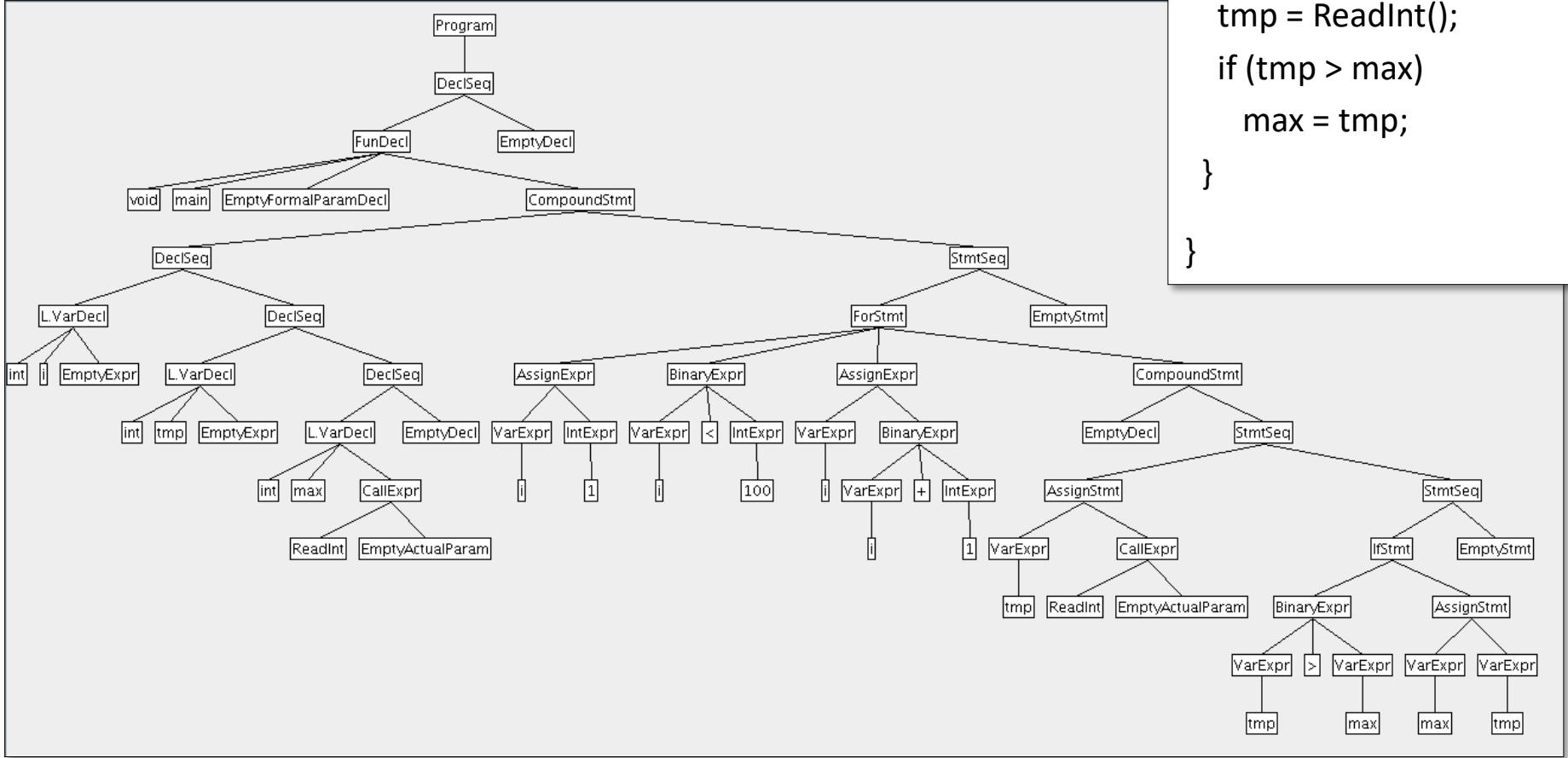
```
private Stmt parseStmt() throws SyntaxError {  
    Stmt stmtAST = null;  
    switch (currentToken.kind) {  
        case Token.LEFTBRACE:  
            stmtAST = parseCompoundStmt();  
            break;  
        case Token.IF:  
            stmtAST = parseIfStmt();  
            break;  
        ...  
        default:  
            syntaxError("'" + currentToken.lexeme + "' not a statement", Token.spell(currentToken.kind));  
    }  
    return stmtAST;  
}
```

- `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**. (See class hierarchy on Slide 95.)
- 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 **IfStmt** is a **Stmt**, but not every **Stmt** is an **IfStmt**.

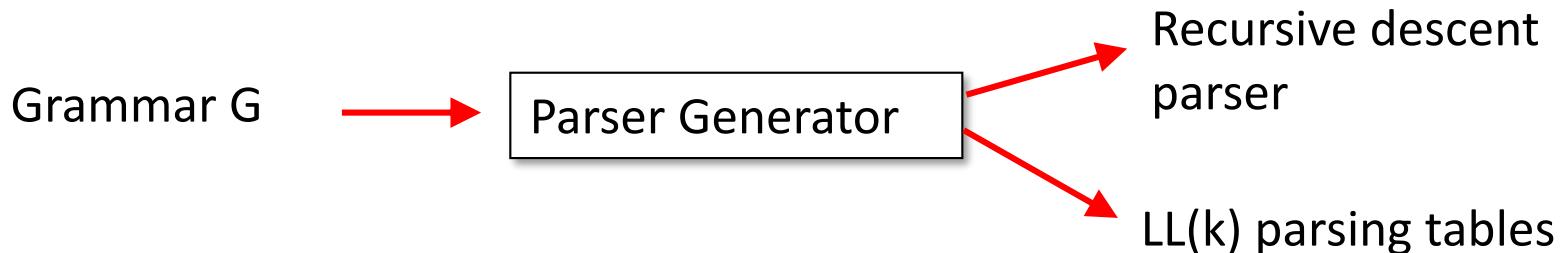
MiniC AST Example

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

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



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++, Python, Go, Java RD LL(k)

We will study the JavaCC parser generator towards the end of the course, if time permits.

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) overview ✓
 - Recursive descent (LL) parser construction ✓
 - LL Grammars ✓
- AST Construction ✓
 - Parse trees vs. ASTs ✓
- Chomsky's Hierarchy

Chomsky's Hierarchy

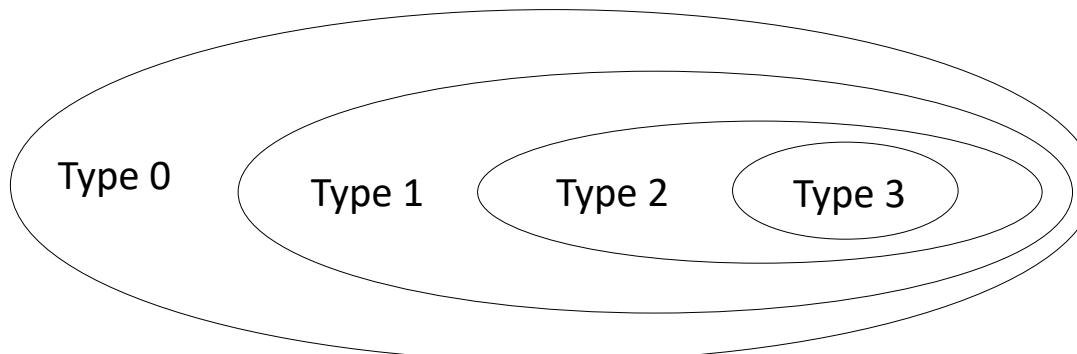
Depending on the productions $\alpha \rightarrow \beta$, four types of grammars can be distinguished:

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

Note:

- “a” is a terminal symbol
- A and B are single non-terminals
- α, β, γ are strings of terminals and/or non-terminals
- α, β : may be empty γ : never empty

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:

$$S \rightarrow \varepsilon \mid aSb$$

- 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:

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

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) overview ✓
 - Recursive descent (LL) parser construction ✓
 - LL Grammars ✓
- AST Construction ✓
 - Parse trees vs. ASTs ✓
- Chomsky's Hierarchy ✓