Languages and Compilers (SProg og Oversættere)

Lecture 7
Top Down Parsing

Bent Thomsen

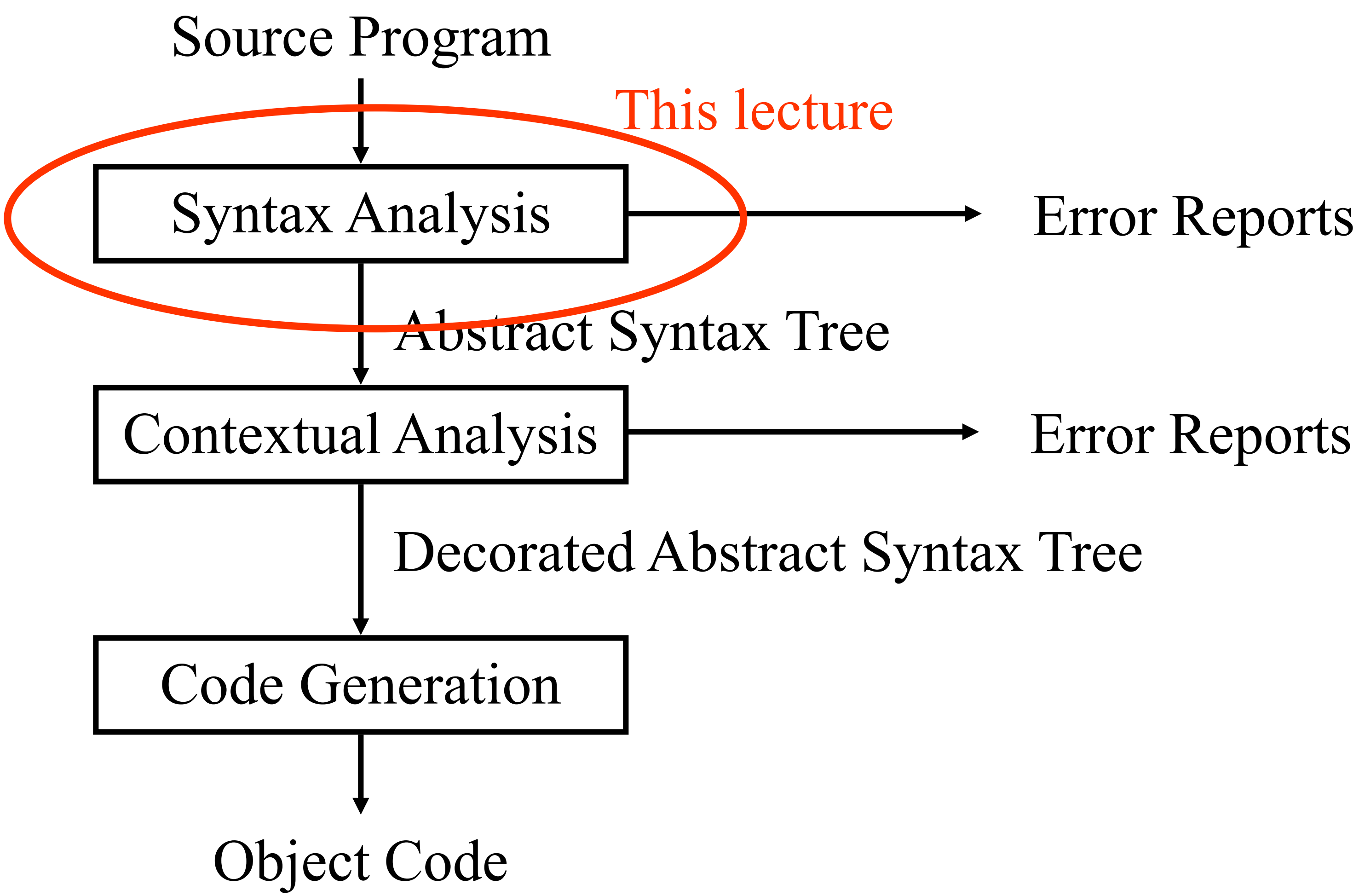
Department of Computer Science

Aalborg University

Learning goals

- To understand top down parsing
- To understand recursive decent parsers
- To understand the role of LL grammers
- To get an overview of table driven top down parsing
- To get an overview of top down parsing tools

The "Phases" of a Compiler



Syntax Analysis

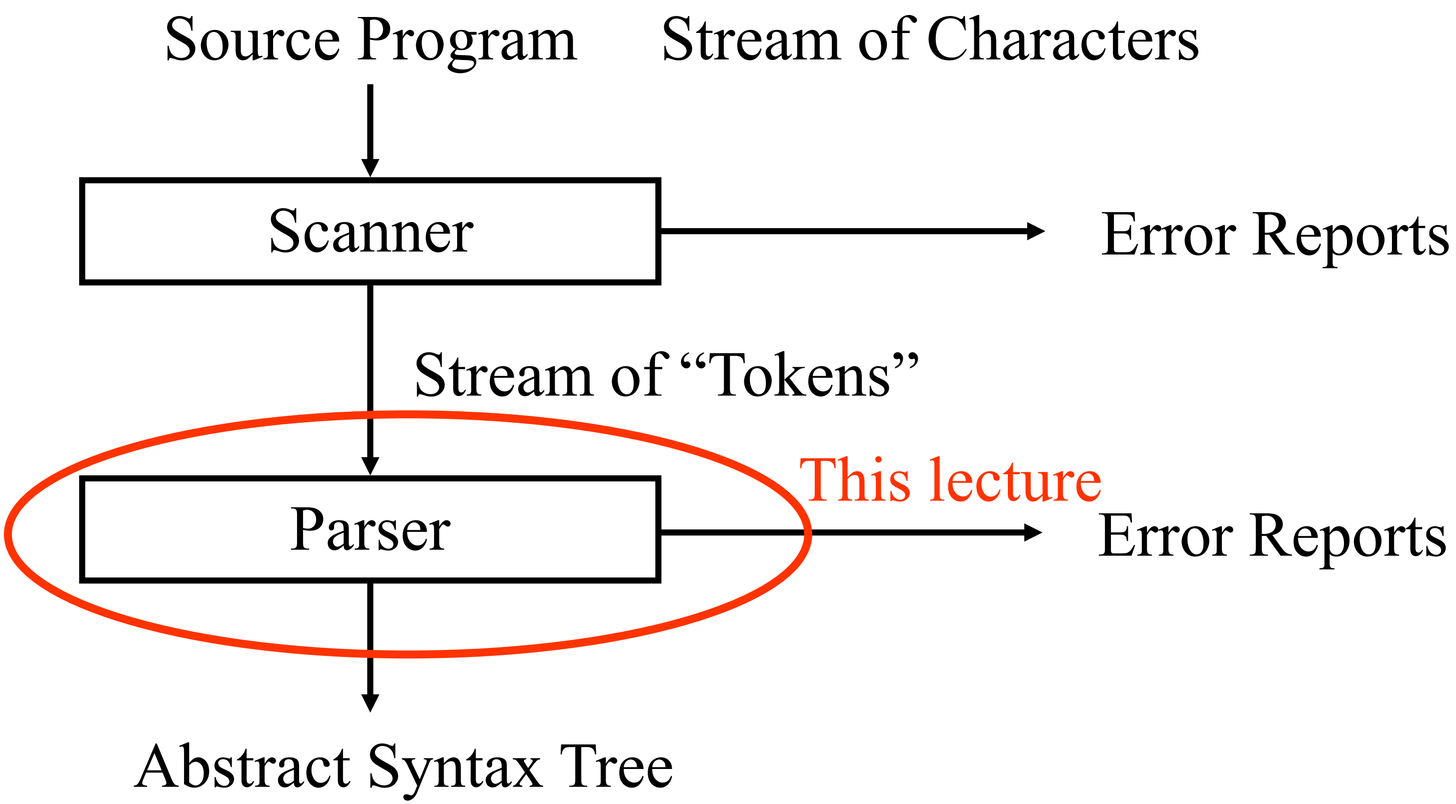
- The "job" of syntax analysis is to read the source text and determine its phrase structure.
- Subphases
  - Scanning
  - Parsing
  - Construct an internal representation of the source text that reifies the phrase structure (usually an AST)

Reify - To regard or treat (an abstraction) as if it had concrete or material existence

Note: A single-pass compiler usually does not construct an AST.

Syntax Analysis

Dataflow chart



1) Scan: Divide Input into Tokens

An example ac source program:

```
f b
i a
a = 5
b = a + 3.2
p b
```

Lexems are “words” in the input, for example keywords, operators, identifiers, literals, etc.
Tokens is a datastructure for lexems and additional information

scanner

floatdl	*id*	*intdcl*	*id*	*id*	*assign*	*inum*	…
f	b	i	a	a	=	5	

…	*assign*	*id*	*plus*	*fnum*	*print*	*id*	*eot*
	=	a	+	3.2	p	b	

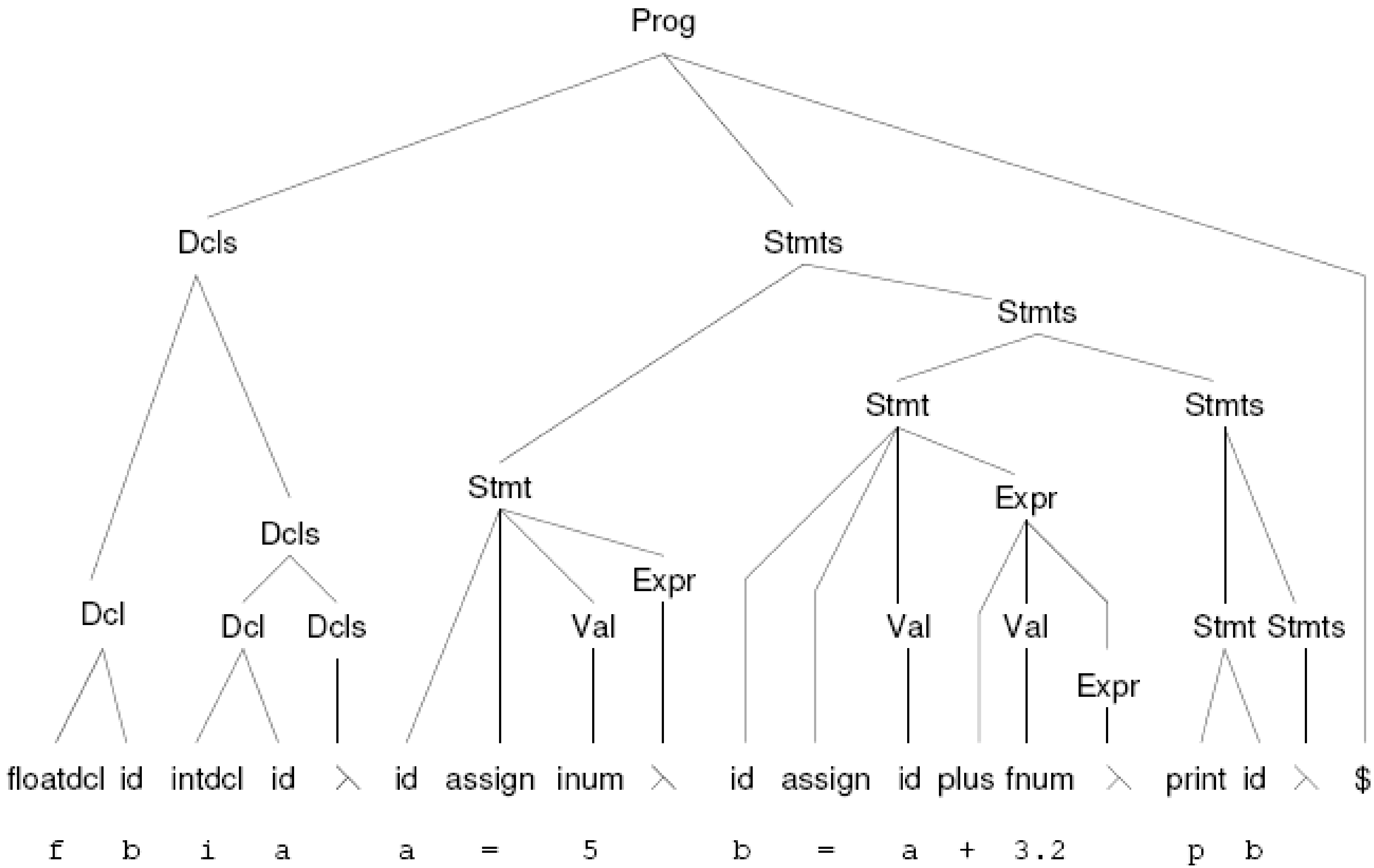


Figure 2.4: An ac program and its parse tree.

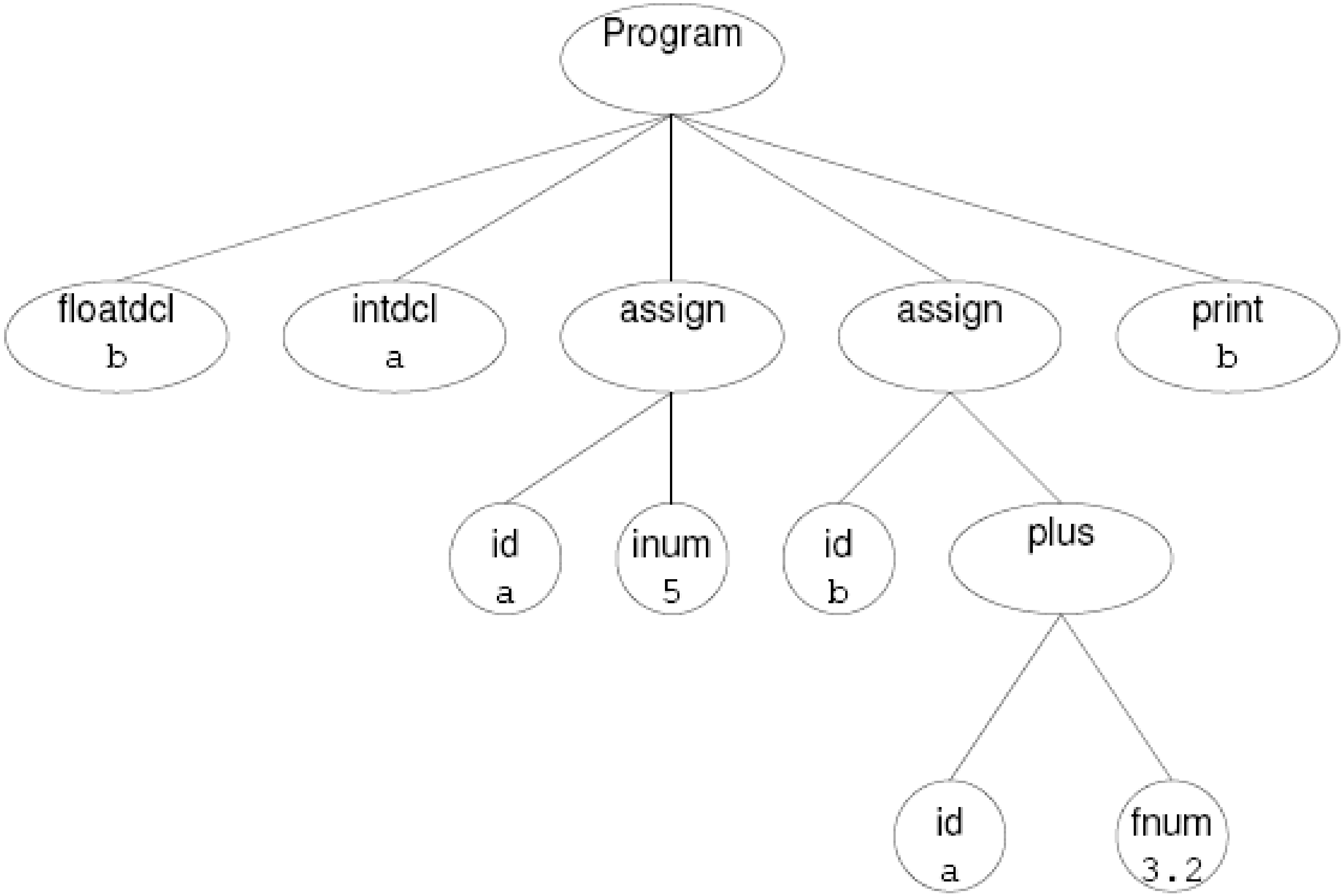
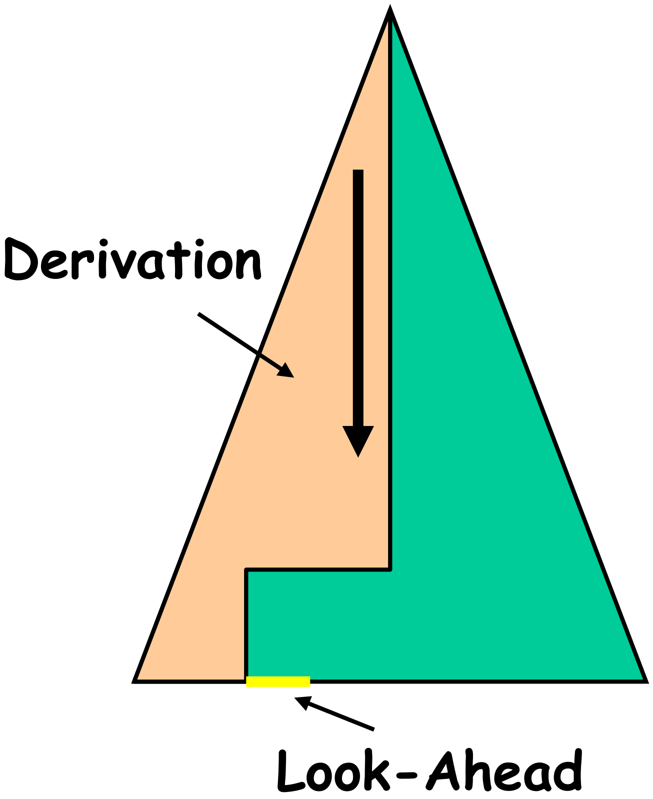


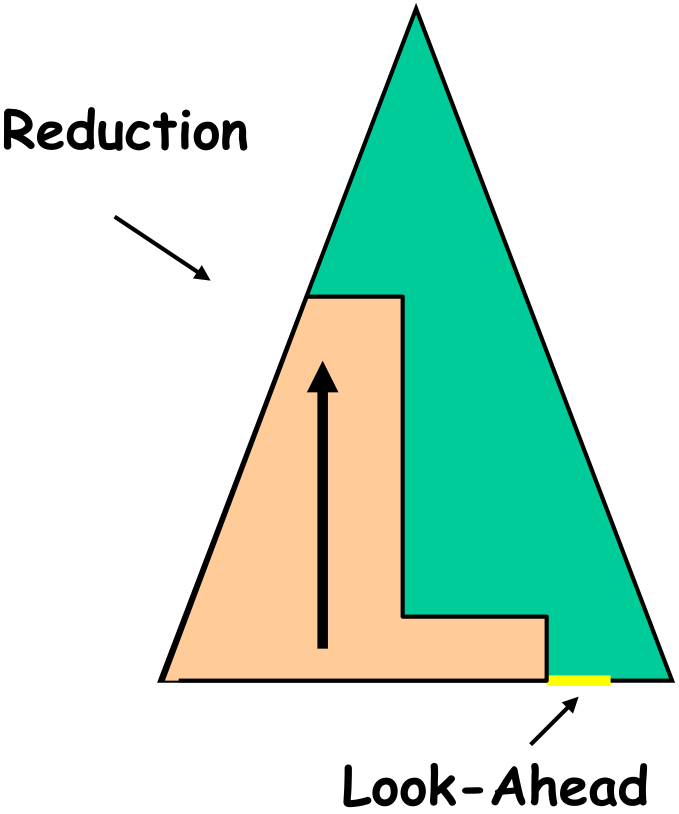
Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

Top-Down vs. Bottom-Up parsing

LL-Analyse (Top-Down)
Left-to-Right Left Derivative



LR-Analyse (Bottom-Up)
Left-to-Right Right Derivative



Top Down Parsing Algorithms

- 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.

Left derivations

```
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
```

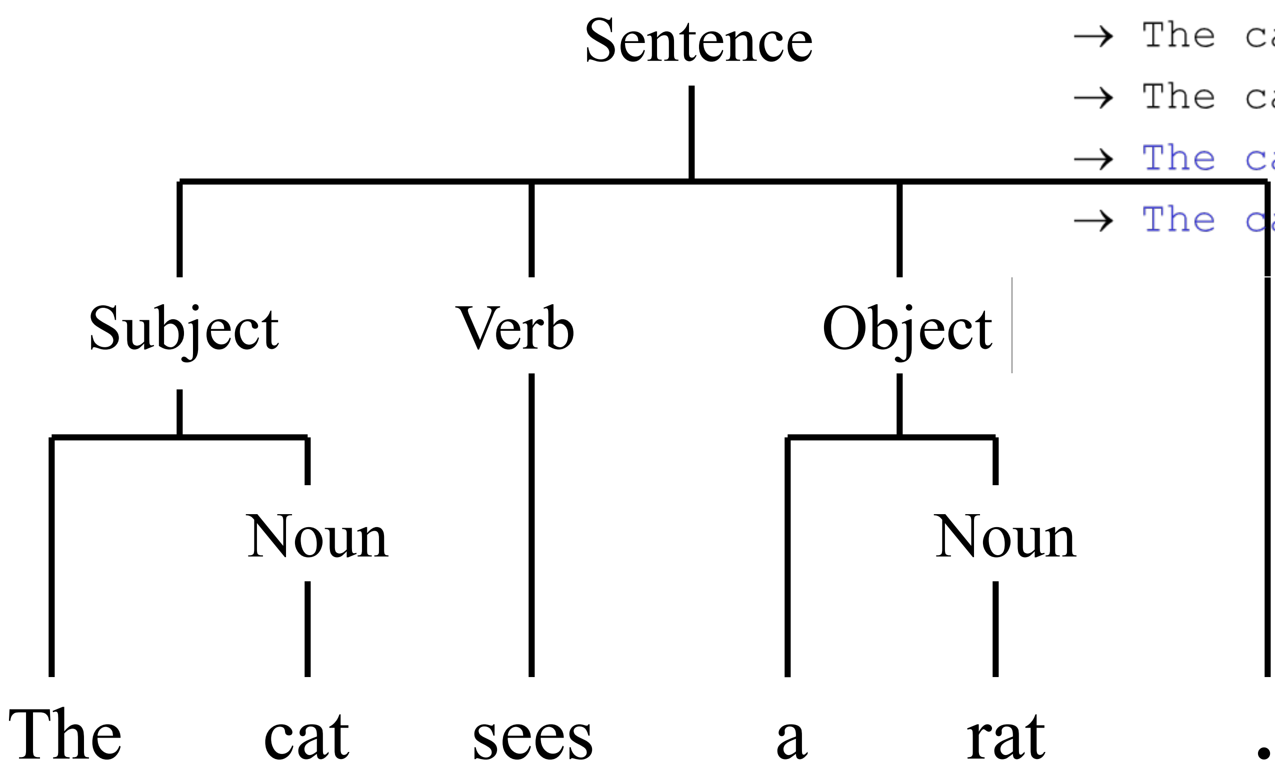
```
Sentence
→ Subject Verb Object .
→ The Noun Verb Object.
→ The cat Verb Object.
→ The cat sees Object.
→ The cat sees a Noun .
→ The cat sees a rat .
```

Top-down parsing

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

Corresponds to following left derivations

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



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 for a simple LL(1) grammar.
- Idea: the parse tree structure corresponds to the “call graph” structure of parsing procedures that call each other recursively.

Recursive Descent Parsing

```
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 Parsing: Auxiliary Methods

```
public class MicroEnglishParser {

   private TerminalSymbol currentTerminal

   private void accept(TerminalSymbol expected) {
      if (currentTerminal matches expected)
                currentTerminal = next input terminal ;
      else
                report a syntax error
   }

   ...
}
```

Recursive Descent Parsing: Parsing Methods

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

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

Recursive Descent Parsing: Parsing Methods

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

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

Recursive Descent Parsing: Parsing Methods

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

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

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 and Coco/R does that in fact

- 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

parse t where *t* is a terminal

```
accept(t);
```

parse N where *N* is a non-terminal

```
parseN();
```

parse ε

```
// a dummy statement
```

parse XY

parse X
parse Y

Algorithm to convert EBNF into a RD parser

*parse X**

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

parse X|Y

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

Note: first[X] is sometimes called starters(X)

Systematic Development of 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) Implement private parsing methods:

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

Recursive Descent Parsing with AST

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

Recursive Descent Parsing: Parsing Methods

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

```
private AST parseSentence() {
   AST theAST;
   AST subject = parseSubject();
   AST verb = parseVerb();
   AST object = parseObject();
   accept('.');
   theAST = new Sentence(subject,verb,object);
   return theAST;
}
```

Converting EBNF into RD parsers

- 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"

JavaCC

- JavaCC is a parser generator
- JavaCC can be thought of as “Lex and Yacc” for implementing parsers in Java
- JavaCC is based on LL(k) grammars
- JavaCC transforms an EBNF grammar into an LL(k) parser
- The lookahead can be change by writing LOOKAHEAD(…)
- The JavaCC can have action code written in Java embedded in the grammar
- JavaCC has a companion called JJTree which can be used to generate an abstract syntax tree

JavaCC input format

- One file with extension .jj containing
  - Header
  - Token specifications
  - Grammar
- Example:

```
TOKEN:
{
        <INTEGER_LITERAL: ([“1”-”9”]([“0”-”9”])*|”0”)>
}

void StatementListReturn() :
{}
{
        (Statement())* “return” Expression() “;”
}
```

JavaCC token specifications use regular expressions

- Characters and strings must be quoted
  - “;”, “int”, “while”
- Character lists […] is shorthand for |
  - [“a”-”z”] matches “a” | “b” | “c” | … | “z”
  - [“a”,”e”,”i”,”o”,u”] matches any vowel
  - [“a”-”z”,”A”-”Z”] matches any letter
- Repetition shorthand with * and +
  - [“a”-”z”,”A”-”Z”]* matches zero or more letters
  - [“a”-”z”,”A”-”Z”]+ matches one or more letters
- Shorthand with ? provides for optionals:
  - (“+”|”-”)?[“0”-”9”]+ matches signed and unsigned integers
- Tokens can be named
  - TOKEN : {<IDENTIFIER:<LETTER>(<LETTER>|<DIGIT>)*>}
  - TOKEN : {<LETTER: [“a”-”z”,”A”-”Z”] >|<DIGIT:[“0”-”9”]>}
  - Now <IDENTIFIER> can be used in defining syntax

ac in BNF and EBNF

```
prog - > dcls stmts
dcls -> dcl dcls | epsilon
dcl -> floatdcl id
     | intdcl id
stmts -> stmt stmts  | epsilon
stmt - > id assign val expr
       | print id
expr - > plus val expr
      | minus val expr
      | epsilon
val - > id | fnum | inum
```

```
prog - > dcl* stmt*
stmt - > id assign val expr?
         | print id
expr - > plus val expr?
         | minus val expr?
```

JavaCC Grammar for ac

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

TOKEN : /* OPERATORS */
{
  < PLUS : "+" >
| < MINUS : "-" >
| < FLOATDCL : "f" >
| < INTDCL : "i" >
| < PRINT : "p" >
| < ASSIGN : "=" >
}

TOKEN :
{
  < INUM : (< DIGIT >)+ >
| < FNUM : (< DIGIT >)+ (".") (< DIGIT >)+ >
| < #DIGIT : [ "0"-"9" ] >
| < ID : ["a"-"e"]|["g"-"h"]|["j"-"o"]|["q"-"z"] >
}
```

```
void prog() :
{}
{(dcl())+ (stmt())*
}

void dcl() :
{}
{
  < FLOATDCL > <ID > | < INTDCL > <ID >
}

void stmt() :
{}
{
  < ID ><ASSIGN > val() (expr())?
| < PRINT > <ID >
}

void val() :
{}
{
  < INUM > | < FNUM > | <  ID >
}

void expr() :
{}
{
      < PLUS > val() (expr())?
    | < MINUS > val() (expr())?
}
```

Adding AST actions for ac

```
AST prog() :
{Prog itsAST = new Prog(new ArrayList<AST >());
 AST dcl;
 AST stm;
}
{(
  dcl = dcl()
  {itsAST.prog.add(dcl);}
  )+
  (stm = stmt()
  {itsAST.prog.add(stm);}
  )*
  {return itsAST;}

}
```

```
AST dcl() :
{Token t;}
{
  (< FLOATDCL > t = <ID >)
  {return new FloatDcl(t.image);}
 | (< INTDCL > t = <ID >)
  {return new IntDcl(t.image);}
}

AST stmt() :
{Boolean b = true;
 AST v;
 Computing e = null;
 Token t;
}
{
  (t = < ID ><ASSIGN > v = val() ((e = expr()){b = false;})?)
  {if (b) return v; else { e.child1 = v; return e;}}
| (< PRINT > t = <ID >)
  {return new Printing(t.image);}
}
```

Generating a parser with JavaCC

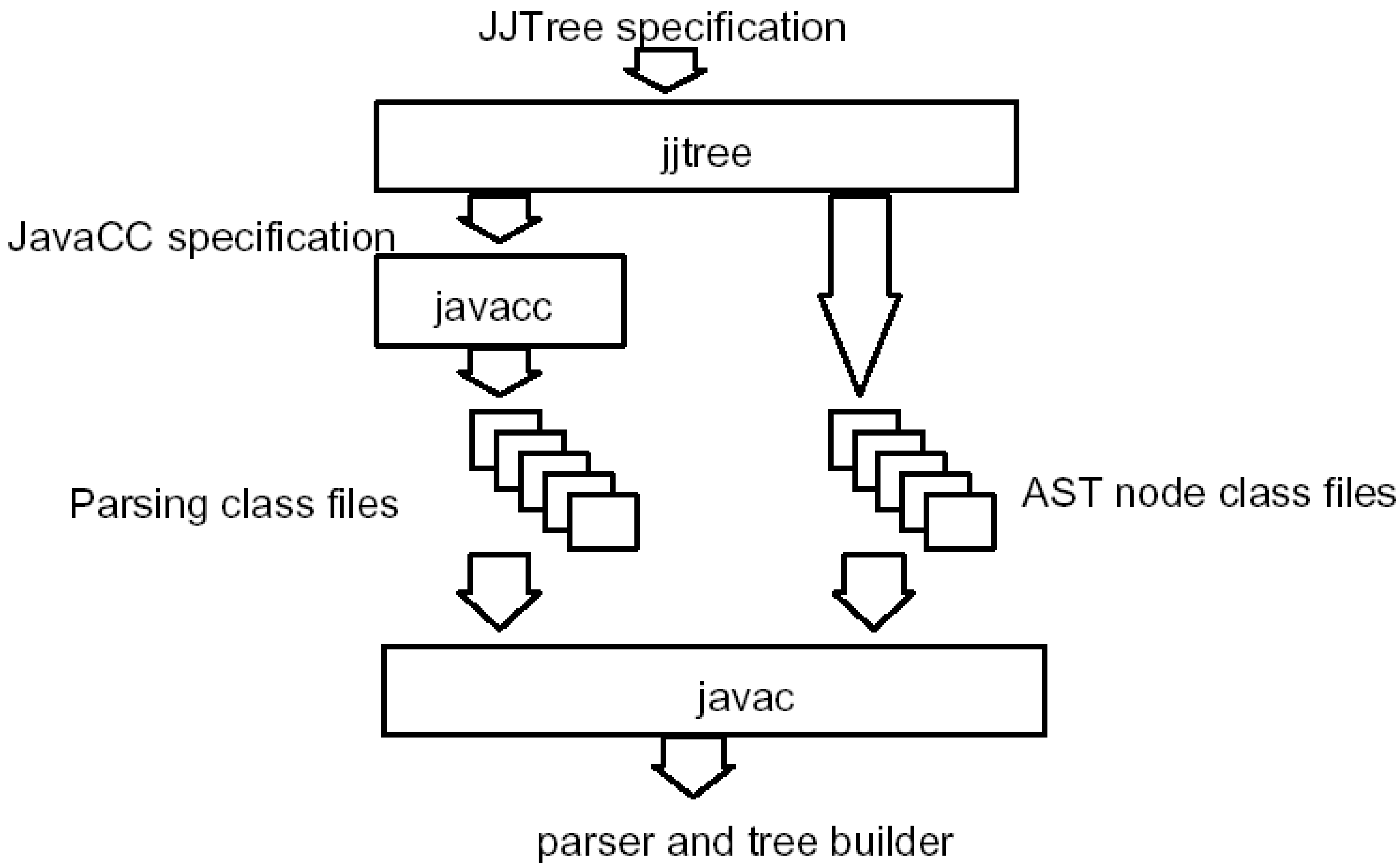
- javacc *filename.jj*
  - generates a parser with specified name
  - Lots of .java files
- javac *.java
  - Compile all the .java files
- There is a plug-in for eclipse

- Note the parser doesn't do anything on its own.
- You have to either
  - Add actions to grammar by hand
  - Use JJTree to generate actions for building AST
  - Use JBT to generate AST and visitors

JavaCC and JJTree

- JavaCC is a parser generator
  - Inputs a set of token definitions, grammar and actions
  - Outputs a Java program which performs syntatic analysis
    - Finding tokens
    - Parses the tokens according to the grammar
    - Executes actions
- JJTree is a preprocessor for JavaCC
  - Inputs a grammar file
  - Inserts tree building actions
  - Outputs JavaCC grammar file with actions
- From this you can add code to traverse the tree to do static analysis, code generation or interpretation.

JavaCC and JJTree



Using JJTree

- JJTree is a preprocessor for JavaCC
- JTree transforms a bare JavaCC grammar into a grammar with embedded Java code for building an AST
  - Classes Node and SimpleNode are generated
  - Can also generate classes for each type of node
- All AST nodes implement interface Node
  - Useful methods provided include:
    - `Public void jjtGetNumChildren()` - returns the number of children
    - `Public void jjtGetChild(int i)` - returns the i'th child
  - The "state" is in a parser field called jjtree
    - The root is at `Node rootNode()`
    - You can display the tree with
    - `((SimpleNode)parser.jjtree.rootNode()).dump(" ");`
- JJTree supports the building of abstract syntax trees which can be traversed using the visitor design pattern

JBT

- JBT – Java Tree Builder is an alternative to JJTree
- It takes a plain JavaCC grammar file as input and automatically generates the following:
  - A set of syntax tree classes based on the productions in the grammar, utilizing the Visitor design pattern.
  - Two interfaces: Visitor and ObjectVisitor. Two depth-first visitors: DepthFirstVisitor and ObjectDepthFirst, whose default methods simply visit the children of the current node.
  - A JavaCC grammar with the proper annotations to build the syntax tree during parsing.
- New visitors, which subclass DepthFirstVisitor or ObjectDepthFirst, can then override the default methods and perform various operations on and manipulate the generated syntax tree.

The Visitor Pattern

For object-oriented programming the *visitor pattern* enables the definition of a *new operator* on an *object structure* without *changing the classes* of the objects

When using visitor pattern

- The set of classes must be fixed in advance
- Each class must have an accept method
- Each accept method takes a visitor as argument
- The purpose of the accept method is to invoke the visitor which can handle the current object.
- A visitor contains a visit method for each class (overloading)
- A method for class C takes an argument of type C

- The advantage of Visitors: New methods without recompilation!

Pause

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 simple LL(1) grammars.
- What grammars are LL(1)?
- Basically, an **LL(1) grammar** is a grammar which can be parsed with a **top-down parser** with a **lookahead** (in the input stream of tokens) of **one token**.

How can we recognize that a grammar is (or is not) LL(1)?

⇒There is a formal definition

⇒We can deduce the necessary conditions from the parser generation algorithm.

Formal definition of LL(1)

A grammar G is LL(1) iff
for each set of productions $X ::= X_1 \mid X_2 \mid ... \mid X_n$:

1. *first*[X_1], *first*[X_2], …, *first*[X_n] are all pairwise disjoint
2. *If* $X_i \Rightarrow^* \varepsilon$ then *first*[X_j]$\cap$ *follow*[X]=Ø, for $1 \leq j \leq n. i \neq j$

If G is ε-free then 1 is sufficient

NOTE: first[X_1] is sometimes called *starters*[X_1]

first[X] = {t in Terminals | X =>* t β }
Follow[X] = {t in Terminals | S =>+ α X t β }

LL(1) Grammars

*parse X**

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

Condition: *first*[*X*] must be disjoint from the set of tokens that can immediately follow *X* *

parse X|*Y*

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

Condition: *first*[*X*] and *first*[*Y*] must be disjoint sets.

First Sets

Informal Definition:

The starter set of a RE *X* is the set of terminal symbols that can occur as the start of any string generated by *X*

Example :

first[`(+|-|ε)(0|1|...|9)*`] = {`+`,`-`, `0`,`1`,…,`9`}

Formal Definition:

$first[\varepsilon] = \{\}$

$first[t] = \{t\}$ (where *t* is a terminal symbol)

$first[X\ Y] = first[X] \cup first[Y]$ (if *X* generates ε)

$first[X\ Y] = first[X]$ (if not *X* generates ε)

$first[X \mid Y] = first[X] \cup first[Y]$

$first[X^*] = first[X]$

´First Sets (ctd)

Informal Definition:

The starter set of RE can be generalized to extended BNF

Formal Definition:

$first[N] = first[X]$ (for production rules N ::= X)

Example :

$$
\begin{aligned}
first[\text{Expression}] &= first[\text{PrimaryExp (Operator PrimaryExp)*}] \\
&= first[\text{PrimaryExp}] \\
&= first[\text{Identifiers}] \cup first[\textbf{(}\text{Expression}\textbf{)}] \\
&= first[\textbf{a} \mid \textbf{b} \mid \textbf{c} \mid \ldots \mid \textbf{z}] \cup \{\textbf{(}\} \\
&= \{\textbf{a}, \textbf{b}, \textbf{c}, \ldots, \textbf{z}, \textbf{(}\}
\end{aligned}
$$

```
function First(α) returns Set
    foreach A ∈ NonTerminals( ) do VisitedFirst(A) ← false      (9)
    ans ← InternalFirst(α)
    return (ans)
end
function InternalFirst(𝒳β) returns Set
    if 𝒳β = ⊥                                                  (10)
    then return (∅)
    if 𝒳 ∈ Σ                                                    (11)
    then return ({𝒳})
    /★  𝒳 is a nonterminal.                                 ★/ (12)
    ans ← ∅
    if not VisitedFirst(𝒳)
    then
        VisitedFirst(𝒳) ← true                                 (13)
        foreach rhs ∈ ProductionsFor(𝒳) do
            ans ← ans ∪ InternalFirst(rhs)                      (14)
    if SymbolDerivesEmpty(𝒳)                                    (15)
    then ans ← ans ∪ InternalFirst(β)
    return (ans)                                                (16)
end
```

Figure 4.8: Algorithm for computing First(α).

```
function Follow(A) returns Set
    foreach A ∈ NonTerminals( ) do
        VisitedFollow(A) ← false                                   (17)
    ans ← InternalFollow(A)
    return (ans)
end
function InternalFollow(A) returns Set
    ans ← ∅
    if not VisitedFolow(A)                                          (18)
    then
        VisitedFollow(A) ← true                                     (19)
        foreach a ∈ Occurrences(A) do                               (20)
            ans ← ans ∪ First(Tail(a))                              (21)
            if AllDeriveEmpty(Tail(a))                              (22)
            then
                targ ← LHS(Production(a))
                ans ← ans ∪ InternalFollow(targ)                    (23)
    return (ans)                                                    (24)
end
function AllDeriveEmpty(γ) returns Boolean
    foreach 𝒳 ∈ γ do
        if not SymbolDerivesEmpty(𝒳) or 𝒳 ∈ Σ
        then return (false)
    return (true)
end
```

Figure 4.11: Algorithm for computing Follow(A).

A variant on First and Follow sets

Rules for First Sets

1. If X is a terminal **then** First(X) is just X!
2. If there is a Production X → ε **then** add ε to first(X)
3. If there is a Production X → Y1Y2..Yk **then** add first(Y1Y2..Yk) to first(X)
4. First(Y1Y2..Yk) is **either**
    1. First(Y1) (if First(Y1) doesn't contain ε)
    2. **OR** (if First(Y1) does contain ε) then First (Y1Y2..Yk) is everything in First(Y1) <except for ε > as well as everything in First(Y2..Yk)
    3. If First(Y1) First(Y2)..First(Yk) all contain ε **then** add ε to First(Y1Y2..Yk) as well.

Rules for Follow Sets

1. First put $ (the end of input marker) in Follow(S) (S is the start symbol)
2. If there is a production A → aBb, (where a can be a whole string) **then** everything in FIRST(b) except for ε is placed in FOLLOW(B).
3. If there is a production A → aB, **then** everything in FOLLOW(A) is in FOLLOW(B)
4. If there is a production A → aBb, where FIRST(b) contains ε, **then** everything in FOLLOW(A) is in FOLLOW(B)

Source: https://www.jambe.co.nz/UNI/FirstAndFollowSets.html

First and Follow in KfG Edit

```
S -> A C
C -> c
   | EPSILON
A -> a B C d
   | B Q
B -> b B
   | EPSILON
Q -> q
   | EPSILON
```

LL(1) first condition fulfilled!

```
FIRST (S) = {a, b, EPSILON, q, c}
FOLLOW(S) = {$}
FIRST (S) ∩ FOLLOW(S) = ∅
```

```
FIRST (C) = {c, EPSILON}
FOLLOW(C) = {$, d}
FIRST (C) ∩ FOLLOW(C) = ∅
```

```
FIRST (A) = {a, b, EPSILON, q}
FOLLOW(A) = {$, c}
FIRST (A) ∩ FOLLOW(A) = ∅
```

```
FIRST (B) = {b, EPSILON}
FOLLOW(B) = {c, d, $, q}
FIRST (B) ∩ FOLLOW(B) = ∅
```

```
FIRST (Q) = {q, EPSILON}
FOLLOW(Q) = {$, c}
FIRST (Q) ∩ FOLLOW(Q) = ∅
```

LL(1) second condition fulfilled!

```
function IsLL1(G) returns Boolean
    foreach A ∈ N do
        PredictSet ← ∅
        foreach p ∈ ProductionsFor(A) do
            if Predict(p) ∩ PredictSet ≠ ∅                                4
            then return (false)
            PredictSet ← PredictSet ∪ Predict(p)
    return (true)
end
```

Figure 5.4: Algorithm to determine if a grammar G is LL(1).

```
function Predict(p : A → X_1 ... X_m) : Set
    ans ← First(X_1 ... X_m)                                            1
    if RuleDerivesEmpty(p)                                              2
    then
        ans ← ans ∪ Follow(A)                                           3
    return (ans)
end
```

Figure 5.1: Computation of Predict sets.

1	S	$\rightarrow$	A C \$
2	C	$\rightarrow$	c
3		|	λ
4	A	$\rightarrow$	a B C d
5		|	B Q
6	B	$\rightarrow$	b B
7		|	λ
8	Q	$\rightarrow$	q
9		|	λ

Figure 5.2: A CFGs.

Rule Number	A	$X_1 \dots X_m$	First($X_1 \dots X_m$)	Derives Empty?	Follow(A)	Answer
1	S	A C $	a,b,q,c,$	No		a,b,q,c,$
2	C	c	c	No		c
3		λ		Yes	d,$	d,$
4	A	a B C d	a	No		a
5		B Q	b,q	Yes	c,$	b,q,c,$
6	B	b B	b	No		b
7		λ		Yes	q,c,d,$	q,c,d,$
8	Q	q	q	No		q
9		λ		Yes	c,$	c,$

Figure 5.3: Predict calculation for the grammar of Figure 5.2.

1 S → A C $
2 C → c
3 | λ
4 A → a B C d
5 | B Q
6 B → b B
7 | λ
8 Q → q
9 | λ

```
procedure A(ts)
    switch (...)
        case ts.PEEK( ) ∈ Predict(p_1)
            /*    Code for p_1                                  */
        case ts.PEEK( ) ∈ Predict(p_i)
            /*    Code for p_2                                  */
        /*    .                                                 */
        /*    .                                                 */
        /*    .                                                 */
        case ts.PEEK( ) ∈ Predict(p_n)
            /*    Code for p_n                                  */
        case default
            /*    Syntax error                                  */
end
```

Figure 5.6: A typical recursive-descent procedure. Successful LL(1) analysis ensures that only one of the **case** predicates is **true**.

```
procedure S( )
    switch (...)
        case ts.PEEK( ) ∈ { a, b, q, c, $ }
            call A( )
            call C( )
            call MATCH($)
end
procedure C( )
    switch (...)
        case ts.PEEK( ) ∈ { c }
            call MATCH(c)
        case ts.PEEK( ) ∈ { d, $ }
            return ()
end
procedure A( )
    switch (...)
        case ts.PEEK( ) ∈ { a }
            call MATCH(a)
            call B( )
            call C( )
            call MATCH(d)
        case ts.PEEK( ) ∈ { b, q, c, $ }
            call B( )
            call Q( )
end
procedure B( )
    switch (...)
        case ts.PEEK( ) ∈ { b }
            call MATCH(b)
            call B( )
        case ts.PEEK( ) ∈ { q, c, d, $ }
            return ()
end
procedure Q( )
    switch (...)
        case ts.PEEK( ) ∈ { q }
            call MATCH(q)
        case ts.PEEK( ) ∈ { c, $ }
            return ()
end
```

Figure 5.7: Recursive-descent code for the grammar shown in Figure 5.2. The variable *ts* denotes the token stream produced by the scanner.

1	S	→	A C $
2	C	→	c
3		|	λ
4	A	→	a B C d
5		|	B Q
6	B	→	b B
7		|	λ
8	Q	→	q
9		|	λ

```
procedure MATCH(ts, token)
    if ts.PEEK( ) = token
    then  call ts.ADVANCE( )
    else  call ERROR(Expected token)
end
```

Figure 5.5: Utility for matching tokens in an input stream.

Recursive Decent Parser for ac

```java
 * Recursive-descent parser based on the grammar given
 *   in Figure 2.1
 * @author cytron
 *
 */
public class Parser {

    private TokenStream ts;

    public Parser(CharStream s) {
        ts = new TokenStream(s);
    }


    public void Prog() {
        if (ts.peek() == FLTDCL || ts.peek() == INTDCL || ts.peek() == ID || ts.peek() == PRINT || ts.peek() == EOF) {
            Dcls();
            Stmts();
            expect(EOF);
        }
        else error("expected floatdcl, intdcl, id, print, or eof");
    }

    public void Dcls() {
        if (ts.peek() == FLTDCL || ts.peek() == INTDCL) {
            Dcl();
            Dcls();
        }
        else if (ts.peek() == ID || ts.peek() == PRINT || ts.peek() == EOF) {
            // Do nothing for lambda-production
        }
        else error("expected floatdcl, intdcl, id, print, or eof");
    }

    public void Dcl() {
        if (ts.peek() == FLTDCL) {
            expect(FLTDCL);
            expect(ID);
        }
        else if (ts.peek() == INTDCL) {
            expect(INTDCL);
            expect(ID);
        }
        else error("expected float or int declaration");
    }
```

prog - > dcls stmts

dcls -> dcl dcls | epsilon

dcl -> floatdcl id

| intdcl id

stmts -> stmt stmts | epsilon

stmt - > id assign val expr

| print id

expr - > plus val expr

| minus val expr

| epsilon

val - > id | fnum | inum

Recursive Decent Parser for ac

```java
/**
 * Figure 2.7 code
 */
public void Stmts() {
    if (ts.peek() == ID || ts.peek() == PRINT) {
        Stmt();
        Stmts();
    }
    else if (ts.peek() == EOF) {
        // Do nothing for lambda-production
    }
    else error("expected id, print, or eof");

}

public void Stmt() {
    if (ts.peek() == ID) {
        expect(ID);
        expect(ASSIGN);
        Val();
        Expr();
    }
    else if (ts.peek() == PRINT) {
        expect(PRINT);
        expect(ID);
    }
    else error("expected id or print");

}

public void Expr() {
    if (ts.peek() == PLUS) {
        expect(PLUS);
        Val();
        Expr();
    }
    else if (ts.peek() == MINUS) {
        expect(MINUS);
        Val();
        Expr();

    }
    else if (ts.peek() == ID || ts.peek() == PRINT || ts.peek() == EOF) {
        // Do nothing for lambda-production
    }
    else error("expected plus, minus, id, print, or eof");

}
```


```java
public void Expr() {
    if (ts.peek() == PLUS) {
        expect(PLUS);
        Val();
        Expr();
    }
    else if (ts.peek() == MINUS) {
        expect(MINUS);
        Val();
        Expr();

    }
    else if (ts.peek() == ID || ts.peek() == PRINT || ts.peek() == EOF) {
        // Do nothing for lambda-production
    }
    else error("expected plus, minus, id, print, or eof");

}
```


```java
public void Val() {
    if (ts.peek() == ID) {
        expect(ID);
    }
    else if (ts.peek() == INUM) {
        expect(INUM);
    }
    else if (ts.peek() == FNUM) {
        expect(FNUM);
    }
    else error("expected id, inum, or fnum");

}

private void expect(int type) {
    Token t = ts.advance();
    if (t.type != type) {
        throw new Error("Expected type "
                + Token.token2str[type]
                              + " but received type "
                              + Token.token2str[t.type]);

    }
}

private void error(String message) {
    throw new Error(message);
}
}
```

stmts -> stmt stmts | epsilon

stmt - > id assign val expr

| print id

expr - > plus val expr

| minus val expr

| epsilon

val - > id | fnum | inum

Recursive Decent Parser for ac with AST

```java

public class ASTParser {
    private TokenStream ts;

    public ASTParser(CharStream s) {
        ts = new TokenStream(s);
    }


    public AST Prog() {
        Prog itsAST = new Prog(new ArrayList<AST>());
        if (ts.peek() == FLTDCL || ts.peek() == INTDCL || ts.peek() == ID || ts.peek() == PRINT || ts.peek() == EOF) {
            ArrayList<AST> dcllist = Dcls();
            ArrayList<AST> stmlist = Stmts();
            expect(EOF);
            if (dcllist != null) itsAST.prog.addAll(dcllist);
            if (stmlist != null) itsAST.prog.addAll(stmlist);
        }
        else error("expected floatdcl, intdcl, id, print, or eof");
        return itsAST;
    }

    public ArrayList<AST> Dcls() {
        ArrayList<AST> astlist = new ArrayList<AST>();
        if (ts.peek() == FLTDCL || ts.peek() == INTDCL) {
            AST dcl = Dcl();
            ArrayList<AST> dcls = Dcls();
            astlist.add(dcl);
            astlist.addAll(dcls);
        }
        else if (ts.peek() == ID || ts.peek() == PRINT || ts.peek() == EOF) {
            // Do nothing for lambda-production
        }
        else error("expected floatdcl, intdcl, id, print, or eof");
        return astlist;
    }

    public AST Dcl() {
        AST itsAst = null;
        if (ts.peek() == FLTDCL) {
            expect(FLTDCL);
            Token t = expect(ID);
            itsAst = new FloatDcl(t.val);
        }
        else if (ts.peek() == INTDCL) {
            expect(INTDCL);
            Token t = expect(ID);
            itsAst = new IntDcl(t.val);
        }
        else error("expected float or int declaration");
        return itsAst;
```

Recursive Decent Parser for ac with AST

```java

    /**
     * Figure 2.7 code
     */
    public ArrayList<AST> Stmts() {
        ArrayList<AST> astlist = new ArrayList<AST>();
        if (ts.peek() == ID || ts.peek() == PRINT) {
            AST stmt = Stmt();
            ArrayList<AST> stms = Stmts();
            astlist.add(stmt);
            astlist.addAll(stms);
        }
        else if (ts.peek() == EOF) {
            // Do nothing for lambda-production
        }
        else error("expected id, print, or eof");
        return astlist;

    }

    public AST Stmt() {
        AST itsAst = null;
        if (ts.peek() == ID) {
            Token tid = expect(ID);
            expect(ASSIGN);
            AST val = Val();
            Computing expr = Expr();
            if (expr == null) itsAst = new Assigning(tid.val,val);
            else {expr.child1 = val; itsAst = new Assigning(tid.val, expr);};
        }
        else if (ts.peek() == PRINT) {
            expect(PRINT);
            Token tid = expect(ID);
            itsAst = new Printing(tid.val);
        }
        else error("expected id or print");
        return itsAst;

    }

```

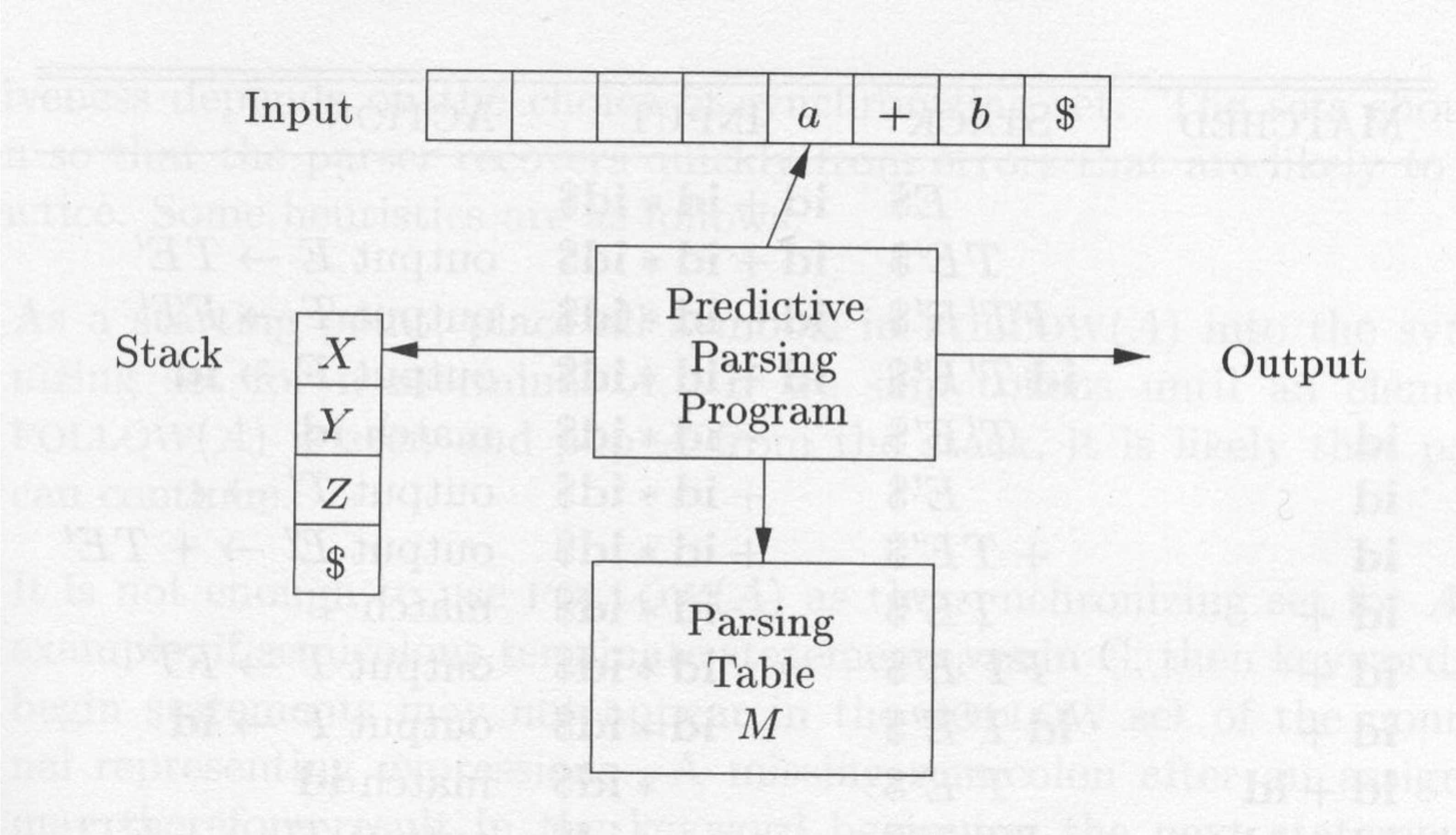
Recursive Decent Parser for ac with AST

```java
    }

    public Computing Expr() {
        Computing itsAst = null;
        if (ts.peek() == PLUS) {
            expect(PLUS);
            AST val = Val();
            Computing expr = Expr();
            //The construction of the AST is a little messy as the grammar for the ac language is Expr -> (+|-) Val Expr
            //which will be used in the Stm -> Id assign Val Expr production. However, we really want the AST
            //to have an Assigning node corresponding to Id assign Expr where Expr -> Val (+|-) Expr i.e. a Computing node
            //thus we create a Computing node in this parse method with an empty left child and
            //in the parse method for STM we adjust the AST with the correct left child
            if (expr != null) {expr.child1 = val; itsAst = new Computing("+",null, expr);}
            else itsAst = new Computing("+",null,val);
        }
        else if (ts.peek() == MINUS) {
            expect(MINUS);
            AST val = Val();
            Computing expr = Expr();
            if (expr != null) {expr.child1 = val; itsAst = new Computing("-",null, expr);}
            else itsAst = new Computing("-",null,val);

        }
        else if (ts.peek() == ID || ts.peek() == PRINT || ts.peek() == EOF) {
            // Do nothing for lambda-production
        }
        else error("expected plus, minus, id, print, or eof");
        return itsAst;

    }

    public AST Val() {
        AST itsAst = null;
        if (ts.peek() == ID) {
            Token tid = expect(ID);
            itsAst = new SymReferencing(tid.val);
        }
        else if (ts.peek() == INUM) {
            Token tid = expect(INUM);
            itsAst = new IntConsting(tid.val);
        }
        else if (ts.peek() == FNUM) {
            Token tid = expect(FNUM);
            itsAst = new FloatConsting(tid.val);
        }
        else error("expected id, inum, or fnum");
        return itsAst;

    }

```

Table-Driven LL(1) Parsers

- Creating recursive-descent parsers can be automated, but
  - Size of parser code
  - Inefficiency: overhead of method calls and returns
- To create table-driven parsers, we use stack to simulate the actions by MATCH() and calls to nonterminals' procedures
  - Terminal symbol: MATCH
  - Nonterminal symbol: table lookup
  - (Fig. 5.8)

Model of a table-driven predictive parser



```
procedure LLparser( ts )
    call push( S )
    accepted ← false
    while not accepted do                                              5
        if TOS( ) ∈ Σ                                                  6
        then
            call match( ts, TOS( ) )                                   7
            if TOS( ) = $                                              8
            then  accepted ← true
            call pop( )                                                9
        else
            p ← LLtable[TOS( ), ts.peek( )]                            10
            if p = 0
            then
                call error( Syntax error—no production applicable )
            else  call apply( p )
end
procedure apply( p : A → X_1 ... X_m )
    call pop( )                                                        11
    for i = m downto 1 do                                              12
        call push( X_i )
end
```

Figure 5.8: Generic LL(1) parser.

How to Build LL(1) Parse Table

procedure FillTable(*LLtable*)
  foreach $A \in N$ **do**
    foreach $a \in \Sigma$ **do** $LLtable[A][a] \leftarrow 0$
  foreach $A \in N$ **do**
    foreach $p \in ProductionsFor(A)$ **do**
      foreach $a \in \text{Predict}(p)$ **do** $LLtable[A][a] \leftarrow p$
end

Figure 5.9: Construction of an LL(1) parse table.

1 $S \rightarrow A\ C\ \$$
2 $C \rightarrow c$
3 $\quad |\ \lambda$
4 $A \rightarrow a\ B\ C\ d$
5 $\quad |\ B\ Q$
6 $B \rightarrow b\ B$
7 $\quad |\ \lambda$
8 $Q \rightarrow q$
9 $\quad |\ \lambda$

	Lookahead					
Nonterminal	a	b	c	d	q	$
S	1	1	1		1	1
C			2	3		3
A	4	5	5		5	5
B		6	7	7	7	7
Q			9		8	9

Figure 5.10: LL(1) table. The blank entries should trigger error actions in the parser.

ANTLR

- ANTLR is a popular lexer and parser generator in Java.
- Regexp FSM (lexer machine) for tokens
- It allows LL(*) grammars.
  - Does top-down parsing
  - Uses lookahead tokens to decide which path to take
  - Is table driven
  - Each match could
    - invoke a custom action
    - write some text via StringTemplate,
    - generate a Parse tree (or an Abstract Syntax Tree ANTLR v.3)

  - Note LL(*) means that ANTLR uses a parse algorithm that uses k lookahead (usually k=1) as often as possible, but can use regular expressions or even backtracking when making decision. Theory elaborated in 2011 PLDI paper

Java

```
grammar SimpleCalc;

tokens {
    PLUS    = '+' ;
    MINUS   = '-' ;
    MULT    = '*' ;
    DIV = '/' ;
}

@members {
    public static void main(String[] args) throws Exception {
        SimpleCalcLexer lex = new SimpleCalcLexer(new ANTLRFileStream(args[0]));
        CommonTokenStream tokens = new CommonTokenStream(lex);

        SimpleCalcParser parser = new SimpleCalcParser(tokens);

        try {
            parser.expr();
        } catch (RecognitionException e)  {
            e.printStackTrace();
        }
    }
}

/*------------------------------------------------------------------
 * PARSER RULES
 *------------------------------------------------------------------*/

expr    : term ( ( PLUS | MINUS )  term )* ;

term    : factor ( ( MULT | DIV ) factor )* ;

factor  : NUMBER ;


/*------------------------------------------------------------------
 * LEXER RULES
 *------------------------------------------------------------------*/

NUMBER  : (DIGIT)+ ;

WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+    { $channel = HIDDEN; } ;

fragment DIGIT  : '0'..'9' ;
```

What can you do in your projects now?

- You should now be able to define the lexical grammar for your langauge
- Implement the Lexer (scanner) by hand or using JLex

- Define the CFG for your language
- Check it is LL(1) or LL(n) for some n
- If it is LL(n) you should be able to implement a parser
  - Recursive decent by hand
  - Recursive decent by using a tool like JavaCC or CoCo/R
  - Table driven by using a tool like ANTLR

Remarks

- Tools
  - Many different tools
  - Downloading and installing them is part of the exercises
  - Judging if a tool is worthwhile using include judging how difficult it is to install and how difficult it is to use
  - Sometimes it is easier to do things by hand than using a tool
  - But if you haven’t tried you don’t know when

  - Try out the different tools and techniques on a small language or a subset of your own language.
  - Write down proc and cons for each.
  - Lo and behold – you have a section for your report!

Error Reporting

- A common technique is to print the offending line with a pointer to the position of the error.
- The parser might add a diagnostic message like "semicolon missing at this position" if it knows what the likely error is.
- The way the parser is written may influence error reporting is:

```
private void parseAorB () {
        switch (currentToken.kind) {
        case Token.A: {
                acceptIT();
                …
        }
        break;
        case Token.B: {
                acceptIT();
                …
        }
        break;
        default:
                report a syntax error
        }
}
```

Error Reporting

```
private void parseAorB () {
        if (currentToken.kind == Token.A) {
                acceptIT();
                …
        } else {
                acceptIT();
                …
        }
}
```

How to handle Syntax errors

- Error Recovery : The parser should try to recover from an error quickly so subsequent errors can be reported. If the parser doesn’t recover correctly it may report spurious errors.

- Possible strategies:
  - Panic-mode Recovery
  - Phase-level Recovery
  - Error Productions

Panic-mode Recovery

- Discard input tokens until a synchronizing token (like; or end) is found.
- Simple but may skip a considerable amount of input before checking for errors again.
- Will not generate an infinite loop.

Phrase-level Recovery

- Perform local corrections
- Replace the prefix of the remaining input with some string to allow the parser to continue.
  - Examples: replace a comma with a semicolon, delete an extraneous semicolon or insert a missing semicolon. Must be careful not to get into an infinite loop.

Recovery with Error Productions

- Augment the grammar with productions to handle common errors
- Example:

```
param_list
 ::= identifier_list : type
 |   param_list, identifier_list : type
 |   param_list; error identifier_list : type
    ("comma should be a semicolon")
```

```
1  S → [ E ]
2    | ( E )
3  E → a
```

```
procedure S( ts, termset )
    switch ()
        case ts.PEEK( ) ∈ { [ }
            call MATCH( [ )
            call E( ts, termset ∪ { ] } )                18
            call MATCH( ] )
        case ts.PEEK( ) ∈ { ( }
            call MATCH( ( )
            call E( ts, termset ∪ { ) } )                19
            call MATCH( ) )
end
procedure E( ts, termset )
    if ts.PEEK( ) = a
    then  call MATCH( ts, a )
    else
        call ERROR( Expected an a )
        while ts.PEEK( ) ∉ termset do  call ts.ADVANCE( )
end
```

Figure 5.26: A grammar and its Wirth-style, error-recovering parser.