Aalto University
School of Science

CS-C2160 Theory of Computation

Lecture 6: The Parsing Problem, Parse Trees and Recursive-Descent Parsing

Pekka Orponen
Aalto University
Department of Computer Science

Spring 2021

Topics:

- The parsing problem and parse trees
- Recursive-descent parsing
- LL(1) grammars
- * Excursion: Attribute grammars
- * Excursion: Parsing tools in the Scala language
- * Supplement: General definition of LL(1) grammars

Recap: Context-free grammars

Example:

A (simplified) grammar for arithmetic expressions in a C-like programming language:

$$
\begin{aligned}
E &\to T \mid E+T \\
T &\to F \mid T*F \\
F &\to a \mid (E) \mid f(L) \\
L &\to \varepsilon \mid L' \\
L' &\to E \mid E, L'
\end{aligned}
$$

Deriving the string $f(a+a)*a$ in the grammar:

$$
\begin{aligned}
\underline{E} &\Rightarrow \underline{T} &&\Rightarrow \underline{T}*F &&\Rightarrow \underline{F}*F \\
&\Rightarrow f(\underline{L})*F &&\Rightarrow f(\underline{L'})*F &&\Rightarrow f(\underline{E})*F \\
&\Rightarrow f(\underline{E}+T)*F &&\Rightarrow f(\underline{T}+T)*F &&\Rightarrow f(\underline{F}+T)*F \\
&\Rightarrow f(a+\underline{T})*F &&\Rightarrow f(a+\underline{F})*F &&\Rightarrow f(a+a)*\underline{F} \\
&\Rightarrow f(a+a)*a.
\end{aligned}
$$

The Parsing Problem and Parse Trees

6.1 The parsing problem and parse trees

We want to solve the following problem:

Given a context-free grammar G and a string x. Is $x \in \mathcal{L}(G)$?

An algorithm (program) that solves the problem is called a *parser*.

There are many alternative solution techniques, especially when the grammar G is of some (practically relevant) special form.

Derivations and parse trees

Example:

Recall the grammar G_{expr}:

$$\begin{aligned} E &\to T \mid E+T \\ T &\to F \mid T*F \\ F &\to a \mid (E) \end{aligned}$$

Some derivations of string $a+a$ in the grammar are:

$$\begin{array}{llllllll} \text{(i)} & \underline{E} & \Rightarrow & \underline{E}+T & \Rightarrow & \underline{T}+T & \Rightarrow & \underline{F}+T \\ & & \Rightarrow & a+\underline{T} & \Rightarrow & a+\underline{F} & \Rightarrow & a+a \\ \text{(ii)} & \underline{E} & \Rightarrow & E+\underline{T} & \Rightarrow & \underline{E}+F & \Rightarrow & \underline{T}+F \\ & & \Rightarrow & F+\underline{F} & \Rightarrow & \underline{F}+a & \Rightarrow & a+a \\ \text{(iii)} & \underline{E} & \Rightarrow & E+\underline{T} & \Rightarrow & E+\underline{F} & \Rightarrow & \underline{E}+a \\ & & \Rightarrow & \underline{T}+a & \Rightarrow & \underline{F}+a & \Rightarrow & a+a \end{array}$$

The underlines denote which non-terminal variable is substituted in which step.

- A derivation is a *leftmost* derivation if in each step the substitution is applied to the leftmost available non-terminal variable. (To emphasise this, we may use the symbol $\underset{\text{lm}}{\Rightarrow}$ instead of $\Rightarrow$.) Derivation (i) on the previous slide is a leftmost derivation.
- *Rightmost derivations* (symbol $\underset{\text{rm}}{\Rightarrow}$) are defined similarly. Derivation (iii) on the previous slide is a rightmost derivation.

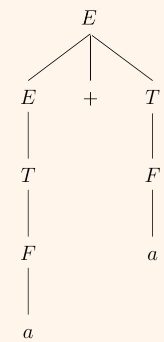
- Let $G = (V, \Sigma, R, S)$ be a context-free grammar.
- A *parse tree* for G is an ordered tree τ where:
  - The nodes in τ are labeled with elements from $V \cup \Sigma \cup \{\varepsilon\}$ so that (i) non-leaf nodes are labeled with elements in V and (ii) the root is labeled with the start variable S.
  - If A is the label of a non-leaf node and $X_1, \ldots, X_k$ are the labels of its (ordered) children, then $A \to X_1 \ldots X_k$ is a production in R.
- The string ("sentential form") *represented* by a parse tree is obtained by listing the labels of its leaf nodes in preorder ("from left to right").

- Let $G = (V, \Sigma, R, S)$ be a context-free grammar.
- A *parse tree* for G is an ordered tree [1] τ where:
  - The nodes in τ are labeled with elements from $V \cup \Sigma \cup \{\varepsilon\}$ so that (i) non-leaf nodes are labeled with elements in V and (ii) the root is labeled with the start variable S.
  - If A is the label of a non-leaf node and $X_1, \ldots, X_k$ are the labels of its (ordered) children, then $A \to X_1 \ldots X_k$ is a production in R.
- The string ("sentential form") *represented* by a parse tree is obtained by listing the labels of its leaf nodes in preorder ("from left to right").

[1] In an ordered tree the children of each node have a fixed left-to-right ordering.

Example:

A parse tree for string $a + a$ in grammar G_{expr}:



A derivation for the string:

$$\begin{aligned} E &\Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \\ &\Rightarrow a+T \Rightarrow a+F \Rightarrow a+a \end{aligned}$$

- A parse tree can be constructed from a derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_n = \gamma$$

as follows:

1. The root of the tree is labeled with S. If $n = 0$, the tree has no other nodes; otherwise
2. if the first step in the derivation applies rule $S \to X_1 X_2 \ldots X_k$, the root has k child nodes whose labels from left to right are $X_1, X_2, \ldots, X_k$;
3. if the next step applies rule $X_i \to Y_1 Y_2 \ldots Y_l$, then the ith child node of the root has l children, whose labels from left to right are $Y_1, Y_2, \ldots, Y_l$; and so on.

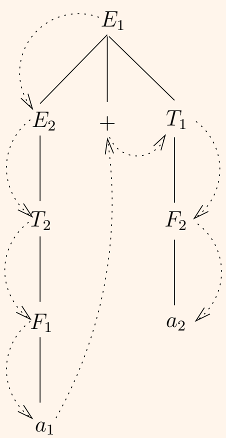
- We observe that if τ is the parse tree constructed from derivation $S \Rightarrow^* \gamma$, then the string represented by τ is γ.

- Let τ be a parse tree representing a string γ.
- We get a leftmost derivation for γ by traversing the nodes of τ in preorder ("from root to leaves, from left to right") and expanding the non-terminal variables encountered as indicated in the tree.
- A rightmost derivation can be obtained similarly by traversing τ in postorder ("from root to leaves, from right to left").
- By constructing a parse tree from a leftmost derivation and then retrieving the leftmost derivation from the tree, one obtains the original leftmost derivation. The same holds for rightmost derivations.

Example:

Retrieving a leftmost derivation for string $a+a$ from a parse tree.

Parse tree:



Nodes in preorder:

$E_1 E_2 T_2 F_1 a_1 + T_1 F_2 a_2$

Leftmost derivation:

$$E \underset{\text{lm}}{\Rightarrow} E+T \underset{\text{lm}}{\Rightarrow} T+T \underset{\text{lm}}{\Rightarrow} F+T$$
$$\underset{\text{lm}}{\Rightarrow} a+T \underset{\text{lm}}{\Rightarrow} a+F \underset{\text{lm}}{\Rightarrow} a+a$$

Lemma 6.1

Let $G = (V, \Sigma, P, S)$ be a context-free grammar.

- Each string γ that can be derived in G has a parse tree that represents γ.
- For each parse tree τ that represents a string $x \in L(G)$ there is a unique leftmost derivation $S \underset{\text{lm}}{\Rightarrow}^* x$ and a unique rightmost derivation $S \underset{\text{rm}}{\Rightarrow}^* x$.

Corollary 6.2

Each string $x \in L(G)$ has a leftmost and a rightmost derivation.

That is: parse trees, leftmost derivations and rightmost derivations are in one-to-one correspondence.
When solving the parsing problem "Is $x \in L(G)$?", one usually also produces a parse tree (or equivalently a leftmost/rightmost derivation) for x if the answer is "yes".

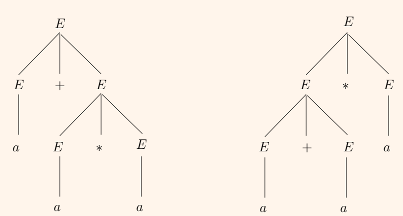
Ambiguity

Example

Let us consider the following grammar for simple arithmetic expressions:

$$G'_{\text{expr}} = \{E \to E+E,\ E \to E*E,\ E \to a,\ E \to (E)\}.$$

In this grammar, e.g. string $a+a*a$ has *two* parse trees:



- A context-free grammar G is *ambiguous* if some string $x \in L(G)$ has two different parse trees.
- Otherwise the grammar is *unambiguous*.

- Ambiguity is usually an unwanted property in computer science because it means that a string has two alternative "interpretations".
- A context-free language for which all the grammars are ambiguous is called an *inherently ambiguous language*.
- As an example, the grammar G'_{expr} is ambiguous while G_{expr} is unambiguous. The language $L_{\text{expr}} = L(G'_{\text{expr}})$ is not inherently ambiguous because it also has an unambiguous grammar G_{expr} generating it.
- On the other hand, e.g. the language

$$\{a^i b^j c^k \mid i = j \text{ or } j = k\}$$

is inherently ambiguous. (The proof of this result is rather complicated and hence omitted here.)

Recursive-Descent Parsing

6.2 Recursive-descent parsing

- One method to search for a leftmost derivation (or parse tree) for a string x in a grammar G is to (i) start from the start variable of G and then (ii) generate systematically and recursively all the possible leftmost derivations (parse trees), (iii) comparing as one proceeds the derived terminal symbols to the ones in the target string x.
- If a conflict (= non-match between derived and target symbol) is found, the search backtracks its most recent production rule choice and tries the next available rule.

Example:

Let us consider the following grammar G:

$$\begin{array}{lcl} E & \rightarrow & T+E \mid T-E \mid T \\ T & \rightarrow & a \mid (E) \end{array}$$

Recursive-descent parsing for the string $a-a$:

$$\begin{array}{llllllll}
E & \Rightarrow & T+E & \Rightarrow & a+T & \text{[conflict; backtrack]} & & \\
 & & & \Rightarrow & (E)+T & \text{[conflict; backtrack]} & & \\
 & \Rightarrow & T-E & \Rightarrow & a-E & \Rightarrow \; a-T+E & \Rightarrow & a-a+E \\
 & & & & & \text{[conflict; backtrack]} & & \\
 & & & & & & \Rightarrow & a-(E)+E \\
 & & & & & \text{[conflict; backtrack]} & & \\
 & & & & & \Rightarrow \; a-T-E & \Rightarrow & a-a-E \\
 & & & & & \text{[conflict; backtrack]} & & \\
 & & & & & & \Rightarrow & a-(E)-E \\
 & & & & & \text{[conflict; backtrack]} & & \\
 & & & & & \Rightarrow \; a-T & \Rightarrow & a-a \quad \text{[OK!]}
\end{array}$$

- This parsing method can be made efficient if the grammar has the property that at each step the *next symbol* in the input string uniquely determines which rule is to be applied when expanding the leftmost non-terminal variable.
- A grammar that has this property is called an *LL(1) grammar*.
- As an example, we can "factor" the productions of the variable E in the grammar G above and get an equivalent grammar G':

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +E \mid -E \mid \varepsilon \\ T & \rightarrow & a \mid (E) \end{array}$$

- Parsing the string $a-a$ in G' (at each step, the symbol determining the next rule is marked above the "yields" symbol):
$E \underset{\text{lm}}{\Rightarrow} TE' \overset{a}{\underset{\text{lm}}{\Rightarrow}} aE' \overset{-}{\underset{\text{lm}}{\Rightarrow}} a-E \underset{\text{lm}}{\Rightarrow} a-TE' \overset{a}{\underset{\text{lm}}{\Rightarrow}} a-aE' \overset{\varepsilon}{\underset{\text{lm}}{\Rightarrow}} a-a.$

For an LL(1) grammar, it is easy to write a parser program as a set of recursive procedures. As an example, here is a Python implementation of a parser for the grammar G':

```python
from sys import exit, stdin
def error(s): print(s); exit(1)
def e():
    print("E -> TE'")
    t(); eprime()
def eprime():
    global next
    if next=="+":
        print("E' -> +E")
        next=stdin.read(1)
        e()
    elif next=="-":
        print("E' -> -E")
        next=stdin.read(1)
        e()
    else: print("E ->")
```

Continues on the next slide...


```python
def t():
    global next
    if next=="a":
        print("T -> a")
        next=stdin.read(1)
    elif next=="(":
        print("T -> (E)")
        next=stdin.read(1)
        e()
        if next!=")": error(") expected.")
        next=stdin.read(1)
    else: error("T cannot start with %s"%(next))

next=stdin.read(1)
e()
```


When processing string `a-(a+a)`, the program outputs the following lines:

```
E → TE′
T → a
E′ → -E
E → TE′
T → (E)
E → TE′
T → a
E′ → +E
E → TE′
T → a
E′ →
E′ →
```

The output corresponds to the leftmost derivation

$$\begin{aligned} E &\Rightarrow TE' \Rightarrow aE' \Rightarrow a-E \Rightarrow a-TE' \\ &\Rightarrow a-(E)E' \Rightarrow a-(TE')E' \\ &\Rightarrow a-(aE')E' \Rightarrow a-(a+E)E' \\ &\Rightarrow a-(a+TE')E' \Rightarrow a-(a+aE')E' \\ &\Rightarrow a-(a+a)E' \Rightarrow a-(a+a). \end{aligned}$$

LL(1) Grammars

6.3 LL(1) grammars

- Let us next consider the general form of LL(1) grammars.
- LL(1) $\approx$ “parse input from **L**eft to right and produce a **L**eftmost derivation, using **1** token lookahead”.
  Here “1 token lookahead” means that one only considers the next symbol in the target string at a time.
- For instance, the grammar

$$\begin{array}{rcl} S & \to & A\,b \mid C\,d \\ A & \to & a\,A \mid \varepsilon \\ C & \to & c\,C \mid \varepsilon \end{array}$$

  is an LL(1) grammar, even though the right-hand sides of the productions don’t always start with a terminal symbol.
- The precise definition of LL(1) grammars is discussed on the supplementary slides at the end of this lecture.

6.3 LL(1) grammars

- Let us next consider the general form of LL(1) grammars.
- LL(1) $\approx$ “parse input from **L**eft to right and produce a **L**eftmost derivation, using **1** token lookahead”.
  Here “1 token lookahead” means that one only considers the next symbol in the target string at a time.
- For instance, the grammar

$$\begin{array}{rcl} S & \to & A\,b \mid C\,d \\ A & \to & a\,A \mid \varepsilon \\ C & \to & c\,C \mid \varepsilon \end{array}$$

  is an LL(1) grammar, even though the right-hand sides of the productions don’t always start with a terminal symbol.
- The precise definition of LL(1) grammars is discussed on the supplementary slides at the end of this lecture.

6.3 LL(1) grammars

- Let us next consider the general form of LL(1) grammars.
- LL(1) $\approx$ “parse input from **L**eft to right and produce a **L**eftmost derivation, using **1** token lookahead”.
  Here “1 token lookahead” means that one only considers the next symbol in the target string at a time. [2]
- For instance, the grammar

$$\begin{array}{rcl} S & \to & A\,b \mid C\,d \\ A & \to & a\,A \mid \varepsilon \\ C & \to & c\,C \mid \varepsilon \end{array}$$

  is an LL(1) grammar, even though the right-hand sides of the productions don’t always start with a terminal symbol.
- The precise definition of LL(1) grammars is discussed on the supplementary slides at the end of this lecture.

[2] There are also more general notions of ”LL(k)” and “LR(k)” grammars.

Left recursion

- Left recursion is a problem for recursive-descent parsing.

Definition 6.1

A grammar $G = (V, \Sigma, P, S)$ is *left recursive* if one can derive from some variable A with one or more steps the string $A\alpha$, where $\alpha \in (V \cup \Sigma)^*$.

Example:

The grammar G_{expr}

$$\begin{array}{rcl} E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & a \mid (E) \end{array}$$

is left recursive because $E \Rightarrow E + T$ and $T \Rightarrow T * F$.
This kind of left recursion that occurs in a single step is called *immediate left recursion*.

- Left recursion may result in infinite, non-terminating recursion in the parsing process.

Example:

In the grammar G_{expr}

$$\begin{aligned} E &\to E+T \mid T \\ T &\to T*F \mid F \\ F &\to a \mid (E) \end{aligned}$$

recursive-descent parsing may start producing the non-terminating derivation

$$\underline{E} \underset{\text{lm}}{\Rightarrow} \underline{E}+T \underset{\text{lm}}{\Rightarrow} \underline{E}+E+T \underset{\text{lm}}{\Rightarrow} \ldots$$

without ever producing a terminal symbol in the beginning of the derived string.

Example:

Also the grammar

$$\begin{aligned} S &\to ASa \mid b \\ A &\to BB \mid dA \\ B &\to b \mid \varepsilon \end{aligned}$$

is left recursive because e.g. $S \Rightarrow ASa \Rightarrow BBSa \Rightarrow BSa \Rightarrow Sa$.

Eliminating immediate left recursion

- Immediate left recursion of form

$$A \to A\beta_1 \mid \ldots \mid A\beta_n \mid \alpha_1 \mid \ldots \mid \alpha_m$$

can be eliminated by translating it into right recursion

$$\begin{aligned} A &\to \alpha_1 A' \mid \ldots \mid \alpha_m A' \\ A' &\to \beta_1 A' \mid \ldots \mid \beta_n A' \mid \varepsilon \end{aligned}$$

- Now a derivation of form

$$A \Rightarrow A\beta_1 \Rightarrow A\beta_2\beta_1 \Rightarrow \alpha_1\beta_2\beta_1$$

can be "simulated" with the derivation

$$A \Rightarrow \alpha_1 A' \Rightarrow \alpha\beta_2 A' \Rightarrow \alpha_1\beta_2\beta_1 A' \Rightarrow \alpha_1\beta_2\beta_1$$

(Also non-immediate, generic left recursion can be eliminated, see e.g. section 4.3 in the book Aho, Sethi, Ullman: "Compilers — Principles, Techniques, and Tools".)

Example:

Eliminating immediate left recursion in the grammar G_{expr}

$$\begin{aligned} E &\to E+T \mid E-T \mid T \\ T &\to T*F \mid F \\ F &\to a \mid (E) \end{aligned}$$

results in the grammar

$$\begin{aligned} E &\to TE' \\ E' &\to +TE' \mid -TE' \mid \varepsilon \\ T &\to FT' \\ T' &\to *FT' \mid \varepsilon \\ F &\to a \mid (E) \end{aligned}$$

Left factoring

- Another problematic grammar feature for recursive-descent parsing are productions that start with the same symbol.
- As an example, consider statements in the C++ language:

$$\begin{array}{rcl} \textit{stmt} & \rightarrow & \textit{selection-stmt} \mid \textit{iteration-stmt} \mid \ldots \\ \textit{selection-stmt} & \rightarrow & \textbf{if}\ (\ \textit{expr}\ )\ \textbf{then}\ \textit{stmt} \mid \\ & & \textbf{if}\ (\ \textit{expr}\ )\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \mid \\ & & \textbf{switch}\ (\ \textit{expr}\ )\ \textit{stmt} \end{array}$$

where *iteration-stmt* and others don't start with the **if** symbol.

☞ Based only on the current **if** symbol in the input string, one cannot decide whether the first or the second production for the variable *selection-stmt* should be applied.

- Common prefixes of form

$$A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma$$

can be "left factored" as follows:

$$\begin{array}{rcl} A & \rightarrow & \alpha A' \mid \gamma \\ A' & \rightarrow & \beta_1 \mid \ldots \mid \beta_n \end{array}$$

Example:

Left factoring the C++ if-then-else structure

$$\begin{array}{rcl} \textit{selection-stmt} & \rightarrow & \textbf{if}\ (\ \textit{expr}\ )\ \textbf{then}\ \textit{stmt} \mid \\ & & \textbf{if}\ (\ \textit{expr}\ )\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \mid \\ & & \textbf{switch}\ (\ \textit{expr}\ )\ \textit{stmt} \end{array}$$

results in

$$\begin{array}{rcl} \textit{selection-stmt} & \rightarrow & \textbf{if}\ (\ \textit{expr}\ )\ \textbf{then}\ \textit{stmt}\ \textit{selection-stmt}' \mid \\ & & \textbf{switch}\ (\ \textit{expr}\ )\ \textit{stmt} \\ \textit{selection-stmt}' & \rightarrow & \textbf{else}\ \textit{stmt} \mid \varepsilon \end{array}$$

* Excursion: Attribute Grammars

- Attribute grammars are a technique for associating simple semantic rules to context-free grammars.
- Each node in a parse tree, labelled with grammar symbol X, is considered an object "of type X". The fields in an object of type X are called *attributes* of X and denoted as $X.s$, $X.t$ etc. Each node "object" has its own "instances" of the attribute.
- The productions $A \rightarrow X_1 \ldots X_k$ of the grammar are associated with *evaluation rules* that describe how the values of the respective attribute instances are computed from those in the parent and child nodes.
- The evaluation rules can in principle be arbitrary functions, as long as their parameters only involve locally available information. More precisely, the evaluation rules associated with a production $A \rightarrow X_1 \ldots X_k$ can only mention attributes of the symbols $A, X_1, \ldots, X_k$.

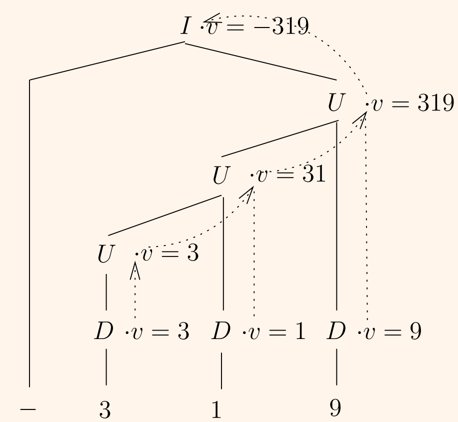
Example: Evaluating signed integers

Each node of type X in the parse tree is associated with an attribute instance $X.v$, whose value will be the numeric value of the string derived from X. In particular, the value of the instance v in the root node will be the numeric value of the whole string represented by the tree.

Productions:			*Evaluation rules:*		
I	$\to$	$+U$	$I.v$	$:=$	$U.v$
I	$\to$	$-U$	$I.v$	$:=$	$-U.v$
I	$\to$	U	$I.v$	$:=$	$U.v$
U	$\to$	D	$U.v$	$:=$	$D.v$
U	$\to$	UD	$U_1.v$	$:=$	$10 * U_2.v + D.v$
D	$\to$	0	$D.v$	$:=$	0
$\ldots$					
D	$\to$	9	$D.v$	$:=$	9

In the evaluation rule associated with production $U \to UD$, the different instances of variable symbol U are distinguished by the use of indices.

The "attributed parse tree" for string "-319":



- An attribute t is *synthetic* if the evaluation rule in each production $A \to X_1 \ldots X_k$ mentioning t is of form

$$A.t := f(A, X_1, \ldots, X_k).$$

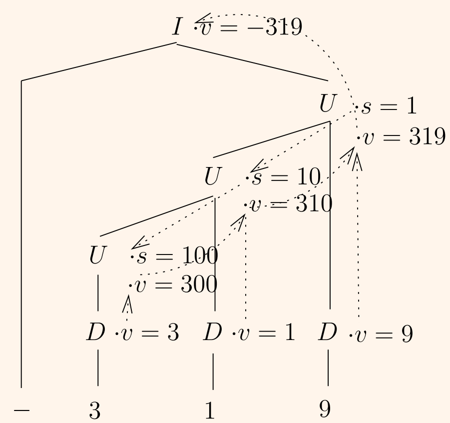
- In this case, the value of a t attribute instance depends only on the values of the attribute instances in the node itself and in its child nodes.
- Other forms of attributes are called *inherited*.
- Synthetic attributes are preferable, because their values can be evaluated in a single bottom-up traversal of the parse tree.
- Of course, one can also use inherited attributes, as long as one ensures that there are no dependency cycles in their evaluation rules.

Example

Evaluating signed integers by using an inherited "position multiplier" attribute and a synthetic "value" attribute:

Productions:			*Evaluation rules:*					
I	$\to$	$+U$	$U.s$	$:=$	$1,$	$I.v$	$:=$	$U.v$
I	$\to$	$-U$	$U.s$	$:=$	$1,$	$I.v$	$:=$	$-U.v$
I	$\to$	U	$U.s$	$:=$	$1,$	$I.v$	$:=$	$U.v$
U	$\to$	D				$U.v$	$:=$	$(D.v) * (U.s)$
U	$\to$	UD	$U_2.s$	$:=$	$10 * (U_1.s),$			
			$U_1.v$	$:=$	$U_2.v + (D.v) * (U_1.s)$			
D	$\to$	0				$D.v$	$:=$	0
$\vdots$								
D	$\to$	9				$D.v$	$:=$	9

For the string “-319” we get the following attributed parse tree:



- The values of attribute instances can often be computed “on-the-fly” without explicitly constructing the parse tree.

Example

Example. A program that transforms arithmetic expressions from infix notation to postfix notation.

We associate to our grammar G_{expr} one synthetic, string-valued attribute *pf*. The value of attribute instance $X.pf$ in each parse tree node of type X will be the postfix version of the infix-notation string derived from the node.

Productions:			*Evaluation rules:*		
E	$\to$	$T+E$	$E_1.pf$	$:=$	$(T.pf)^\frown(E_2.pf)^\frown('+')$
E	$\to$	T	$E.pf$	$:=$	$T.pf$
T	$\to$	$F*T$	$T_1.pf$	$:=$	$(F.pf)^\frown(T_2.pf)^\frown('*')$
T	$\to$	F	$T.pf$	$:=$	$F.pf$
F	$\to$	a	$F.pf$	$:=$	$'a'$
F	$\to$	(E)	$F.pf$	$:=$	$E.pf$

A recursive-descent parser for G_{expr} that also evaluates the values of attribute instances on-the-fly during parsing:

```
from sys import stdin
def error(s): print(s); exit(1)
def e():          # E -> T + E | T
    global next
    pf1=t()
    if next=="+":
        next=stdin.read(1)
        return pf1+e()+"+" # E1.pf := T.pf E2.pf +
    else: return pf1       # E.pf  := T.pf
def t():          # T -> F * T | F
    global next
    pf1=f()
    if next=="*":
        next=stdin.read(1)
        return pf1+t()+"*" # T1.pf := F.pf T2.pf *
    else: return pf1       # T.pf  := F.pf
```

Continues on the next slide...


```
def f():          # F -> a | (E)
    global next
    if next=="a":
        next=stdin.read(1)
        return "a"                # F.pf := a
    elif next=="(":
        next=stdin.read(1)
        pf=e()
        if next!=")": error(") expected.")
        next=stdin.read(1)
        return pf                 # F.pf := E.pf
    else: error("F cannot start with this.")

next=stdin.read(1)
print(e())
```

* Excursion: Parsing Tools in the Scala Language

- A library suitable for (restricted) recursive-descent parsing is included in the standard Scala language distribution
- Regular expressions can also be included in the grammar rules.

References:

- Chapter 31 in book "Programming in Scala, First Edition"
- Parsers trait
- The abstract Parser class
- The RegexParsers trait

A parser and evaluator for simple arithmetic expressions:

```scala
import util.parsing.combinator._

object parser extends RegexParsers {
  val integer = "[0-9]+".r  // Regular expression

  def expr: Parser[BigInt] = (  // E -> T + E | T
      term~"+"~expr ^^  {case t~"+"~e => t + e}
    | term ^^ {case t => t} )
  def term: Parser[BigInt] =  // T -> F * T | T
    rep1sep(factor, "*") ^^  { factors => factors.product }
  def factor: Parser[BigInt] = ( // F -> integer | ( E )
      integer ^^  { case intString => BigInt(intString) }
    | "(" ~> expr <~ ")" ^^  { case e => e }  )

  def parse(input: String): (Option[BigInt], String) = {
    parseAll(expr, input) match {
      case Success(value, _) => (Option(value), "success")
      case f => (None, f.toString)
    }
  }
}
```


- With well-formed input we get the expected result:

```
scala> parser.parse("123+4*5")
res0: (Option[BigInt], String) = (Some(143),success)
```

- while an erroneous input gives an error message:

```
scala> parser.parse("123++4*5")
res1: (Option[BigInt], String) =
(None,[1.5] failure: `(' expected but `+' found

123++4*5
    ^)
```

- A classic textbook on these topics:
  - Aho, Sethi, Ullman: Compilers — Principles, Techniques, and Tools

* Supplement: General Definition of LL(1) Grammars

* Supplement: General definition of LL(1) grammars

- In the following we will formally define LL(1) grammars
- To do this, we need two auxiliary sets
  - FIRST describes which terminal symbols can appear as the first symbols in the strings derivable from a non-terminal variable
  - FOLLOW describes which terminal symbols can follow a non-terminal variable in any of the derivations

- We also need the auxiliary concept of a nullable non-terminal variables

Definition 6.2

A non-terminal variable A is *nullable* if $A \Rightarrow^* \varepsilon$.

Example:

In the grammar

$$\begin{array}{rcl} S & \rightarrow & ASa \mid b \\ A & \rightarrow & BB \mid dA \\ B & \rightarrow & b \mid \varepsilon \end{array}$$

the variables A and B are nullable because

- $\underline{A} \Rightarrow \underline{B}B \Rightarrow \underline{B} \Rightarrow \varepsilon$
- $\underline{B} \Rightarrow \varepsilon$

FIRST-sets

- For each non-terminal variable A we define the set $\mathrm{FIRST}(A)$ of terminal symbols (incl. ε is A is nullable) that can be the first symbols in strings derivable from A:

  $\mathrm{FIRST}(A) = \{a \in \Sigma \mid A \Rightarrow^* a\gamma$ for some $\gamma \in (V \cup \Sigma)^*\} \cup \{\varepsilon \mid A \Rightarrow^* \varepsilon\}$

Example:

In the grammar

$$\begin{array}{lcl} S & \to & A\,b \mid C\,d \\ A & \to & a\,A \mid \varepsilon \\ C & \to & c\,C \mid \varepsilon \end{array}$$

- $\mathrm{FIRST}(C) = \{c, \varepsilon\}$ as $C \Rightarrow cC$ ja $C \Rightarrow \varepsilon$
- $\mathrm{FIRST}(A) = \{a, \varepsilon\}$ as $A \Rightarrow aA$ ja $A \Rightarrow \varepsilon$
- $\mathrm{FIRST}(S) = \{a, b, c, d\}$ as
  - $S \Rightarrow Ab \Rightarrow aAb$ and $S \Rightarrow Ab \Rightarrow b$
  - $S \Rightarrow Cd \Rightarrow cCd$ and $S \Rightarrow Cd \Rightarrow d$

Example:

In the grammar

$$\begin{array}{lcl} S & \to & A\,S\,a \mid b \\ A & \to & B\,B \mid d\,A \\ B & \to & b \mid \varepsilon \end{array}$$

we have

- $\mathrm{FIRST}(S) = \{b, d\}$ as $S \Rightarrow b$ and $S \Rightarrow ASa \Rightarrow dASa$
- $\mathrm{FIRST}(A) = \{b, d, \varepsilon\}$ koska $A \Rightarrow BB \Rightarrow bB$, $A \Rightarrow dA$ and $A \Rightarrow BB \Rightarrow B \Rightarrow \varepsilon$
- $\mathrm{FIRST}(B) = \{b, \varepsilon\}$ as $B \Rightarrow b$ and $B \Rightarrow \varepsilon$.

FIRST-sets (for both terminal symbols and non-terminal variables) can be computed inductively:

- If a is a terminal symbol (i.e. $a \in \Sigma$), then $\mathrm{FIRST}(a) = \{a\}$
- If $X \to \varepsilon$ is a production, then $\varepsilon \in \mathrm{FIRST}(X)$
- If $X \to X_1X_2\ldots X_k$ is a production, a terminal symbol $a \in \mathrm{FIRST}(X_i)$ for some $1 \leq i \leq k$ and $\varepsilon \in \mathrm{FIRST}(X_j)$ for all $1 \leq j < i$, then $a \in \mathrm{FIRST}(X)$
- If $X \to X_1X_2\ldots X_k$ is a production and $\varepsilon \in \mathrm{FIRST}(X_j)$ for all $1 \leq j \leq k$, then $\varepsilon \in \mathrm{FIRST}(X)$

It holds that $\varepsilon \in \mathrm{FIRST}(A)$ if and only if A is nullable.

Example:

For the grammar

$$\begin{array}{lcl} S & \to & A\,b \mid C\,d \\ A & \to & a\,A \mid \varepsilon \\ C & \to & c\,C \mid \varepsilon \end{array}$$

- $\mathrm{FIRST}(a) = \{a\}$, $\mathrm{FIRST}(b) = \{b\}$, $\mathrm{FIRST}(c) = \{c\}$, $\mathrm{FIRST}(d) = \{d\}$
- $\varepsilon \in \mathrm{FIRST}(C)$ as $C \to \varepsilon$ is a production
- $c \in \mathrm{FIRST}(C)$ as $C \to cC$ is a production
- $\varepsilon \in \mathrm{FIRST}(A)$ as $A \to \varepsilon$ is a production
- $a \in \mathrm{FIRST}(A)$ as $A \to aA$ is a production
- $a \in \mathrm{FIRST}(S)$ as $S \to Ab$ is a production and $a \in \mathrm{FIRST}(A)$
- $b \in \mathrm{FIRST}(S)$ as $S \to Ab$ is a production, $\varepsilon \in \mathrm{FIRST}(A)$ and $b \in \mathrm{FIRST}(b)$
- $c, d \in \mathrm{FIRST}(S)$ with similar argumentation

- We expand FIRST to strings over $V \cup \Sigma$ so that we can study which symbols can occur as first ones when deriving strings from the right hand sides of productions
- Let us define this expansion inductively: $\mathrm{FIRST}(X_1 \ldots X_k)$ is the smallest subset of $\Sigma \cup \{\varepsilon\}$ for which the following conditions hold:
  - $\varepsilon \in \mathrm{FIRST}(\varepsilon)$
  - $a \in \mathrm{FIRST}(a)$ for each $a \in \Sigma$
  - If $x \in \Sigma$, $x \in \mathrm{FIRST}(X_i)$ for some $1 \leq i \leq k$ and $\varepsilon \in \mathrm{FIRST}(X_j)$ for all $1 \leq j < i$, then $x \in \mathrm{FIRST}(X_1 \ldots X_k)$
  - If $\varepsilon \in \mathrm{FIRST}(X_j)$ for all $1 \leq j \leq k$, then $\varepsilon \in \mathrm{FIRST}(X_1 \ldots X_k)$

Example:

Consider again the grammar

$$
\begin{array}{rcl}
S & \to & A\,b \mid C\,d \\
A & \to & a\,A \mid \varepsilon \\
C & \to & c\,C \mid \varepsilon
\end{array}
$$

Now

- $\mathrm{FIRST}(A) = \{a, \varepsilon\}$ and $\mathrm{FIRST}(b) = \{b\}$
- $\mathrm{FIRST}(C) = \{c, \varepsilon\}$ and $\mathrm{FIRST}(d) = \{d\}$
- $\mathrm{FIRST}(S) = \{a, b, c, d\}$
- $\mathrm{FIRST}(Ab) = \{a, b\}$
- $\mathrm{FIRST}(Cd) = \{c, d\}$

☞ in the beginning of the parsing, based only on the first symbol in the string, we can decide whether the production $S \to Ab$ or $S \to Cd$ should be applied

Example:

For the grammar

$$
\begin{array}{rcl}
E & \to & E + T \mid T \\
T & \to & T * F \mid F \\
F & \to & a \mid (E)
\end{array}
$$

we have

- $\mathrm{FIRST}(F) = \{a, (\}$
- $\mathrm{FIRST}(T) = \{a, (\}$
- $\mathrm{FIRST}(E) = \{a, (\}$
- $\mathrm{FIRST}(E + T) = \{a, (\}$

☞ by applying either $E \Rightarrow E + T$ or $E \Rightarrow T$ we can get a to be the first terminal symbol in the derived string

☞ based on the first symbol in the string only, the parser cannot decide which production should be used

Example:

Grammar:

$$
\begin{array}{rcl}
E & \to & T\,E' \\
E' & \to & +\,T\,E' \mid \varepsilon \\
T & \to & F\,T' \\
T' & \to & *\,F\,T' \mid \varepsilon \\
F & \to & a \mid (E)
\end{array}
$$

FIRST-sets:

- $\mathrm{FIRST}(E) = \{a, (\}$
- $\mathrm{FIRST}(E') = \{+, \varepsilon\}$
- $\mathrm{FIRST}(T) = \{a, (\}$
- $\mathrm{FIRST}(T') = \{*, \varepsilon\}$
- $\mathrm{FIRST}(F) = \{a, (\}$

Consider parsing the string $a * a + a$

$$\underline{E} \Rightarrow \underline{T}E' \Rightarrow \underline{F}T'E' \Rightarrow a\underline{T'}E'$$

Should we now use the production $T' \Rightarrow *\,F\,T'$ or $T' \Rightarrow \varepsilon$? Why?

Example:

A small "if-then-else" grammar after left factoring:

$$S \rightarrow s \mid \textbf{if}\, C\, \textbf{then}\, S\, S'$$
$$S' \rightarrow \textbf{else}\, S \mid \varepsilon$$
$$C \rightarrow c$$

FIRST-sets:

- $\mathrm{FIRST}(S) = \{s, \textbf{if}\}$
- $\mathrm{FIRST}(S') = \{\textbf{else}, \varepsilon\}$
- $\mathrm{FIRST}(C) = \{c\}$
- $\mathrm{FIRST}(\textbf{else}\, S) = \{\textbf{else}\}$
- $\mathrm{FIRST}(\varepsilon) = \{\varepsilon\}$

Making leftmost derivation for the string **if** c **then if** c **then** s **else** s:

$$\begin{aligned}\underline{S} &\Rightarrow \textbf{if}\, \underline{C}\, \textbf{then}\, S\, S' \\ &\Rightarrow \textbf{if}\, c\, \textbf{then}\, \underline{S}\, S' \\ &\Rightarrow \textbf{if}\, c\, \textbf{then}\, \textbf{if}\, \underline{C}\, \textbf{then}\, S\, S'\, S' \\ &\Rightarrow \textbf{if}\, c\, \textbf{then}\, \textbf{if}\, c\, \textbf{then}\, \underline{S}\, S'\, S' \\ &\Rightarrow \textbf{if}\, c\, \textbf{then}\, \textbf{if}\, c\, \textbf{then}\, s\, \underline{S'}\, S'\end{aligned}$$

Should we now use the production $S' \rightarrow \textbf{else}\, S$ or $S' \rightarrow \varepsilon$?

FOLLOW-sets

- For nullable productions the FIRST-set includes the symbol ε
- How should we interpret this when deciding which production to take next?
- Let us define for each non-terminal variable A the set $\mathrm{FOLLOW}(A)$ of terminal symbols (incl. a special symbol \$ describing the end of the string) that *may follow* A in some derivation:
  - $c \in \mathrm{FOLLOW}(A)$ if $c \in \Sigma$ and $S \Rightarrow^* \alpha A c \beta$ for some $\alpha, \beta \in (V \cup \Sigma)^*$
  - $\$ \in \mathrm{FOLLOW}(A)$ if $S \Rightarrow^* \alpha A$ for some $\alpha \in (V \cup \Sigma)^*$

Example:

Grammar:

$$E \rightarrow T\,E'$$
$$E' \rightarrow +T\,E' \mid \varepsilon$$
$$T \rightarrow F\,T'$$
$$T' \rightarrow *F\,T' \mid \varepsilon$$
$$F \rightarrow a \mid (E)$$

FIRST-sets:

- $\mathrm{FIRST}(F) = \{a, (\}$
- $\mathrm{FIRST}(T') = \{*, \varepsilon\}$
- $\mathrm{FIRST}(T) = \{a, (\}$
- $\mathrm{FIRST}(E') = \{+, \varepsilon\}$
- $\mathrm{FIRST}(E) = \{a, (\}$

Now

- $\mathrm{FOLLOW}(E) = \mathrm{FOLLOW}(E') = \{\$,)\}$ as
  - E is the start variable, $E \Rightarrow^* (E)T'E'$ and $E \rightarrow TE'$
- $\mathrm{FOLLOW}(T) = \mathrm{FOLLOW}(T') = \{+, \$,)\}$ as
  - $E \Rightarrow TE' \Rightarrow T + TE'$, $E \Rightarrow TE' \Rightarrow T$ and $T \Rightarrow FT'$
- $\mathrm{FOLLOW}(F) = \{+, *, \$,)\}$ as
  - $E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow F * FT'E'$
  - $T \Rightarrow FT' \Rightarrow F$ ("what follows F, also follows T")

Example:

Grammar:

$$E \rightarrow T\,E'$$
$$E' \rightarrow +T\,E' \mid \varepsilon$$
$$T \rightarrow F\,T'$$
$$T' \rightarrow *F\,T' \mid \varepsilon$$
$$F \rightarrow a \mid (E)$$

FIRST-sets:

- $\mathrm{FIRST}(F) = \{a, (\}$
- $\mathrm{FIRST}(T') = \{*, \varepsilon\}$
- $\mathrm{FIRST}(T) = \{a, (\}$
- $\mathrm{FIRST}(E') = \{+, \varepsilon\}$
- $\mathrm{FIRST}(E) = \{a, (\}$

As

- $\mathrm{FOLLOW}(E) = \mathrm{FOLLOW}(E') = \{\$,)\}$
- $\mathrm{FOLLOW}(T) = \mathrm{FOLLOW}(T') = \{+, \$,)\}$
- $\mathrm{FOLLOW}(F) = \{+, *, \$,)\}$

we know that when making the leftmost derivation for $a * a + a$

$$\underline{E} \Rightarrow \underline{T}E' \Rightarrow \underline{F}T'E' \Rightarrow a\underline{T'}E'$$

we should apply the production $T' \Rightarrow *F\,T'$ instead of $T' \Rightarrow \varepsilon$ because the non-terminal variable T' cannot be followed by the symbol $*$ in any derivation.

Example:

A simple "if-then-else" grammar after left factoring:

$$
\begin{aligned}
S &\rightarrow s \mid \textbf{if}\, C\, \textbf{then}\, S\, S' \\
S' &\rightarrow \textbf{else}\, S \mid \varepsilon \\
C &\rightarrow c
\end{aligned}
$$

FOLLOW-sets:

- $\mathrm{FOLLOW}(S) = \{\$, \textbf{else}\}$
- $\mathrm{FOLLOW}(S') = \{\$, \textbf{else}\}$
- $\mathrm{FOLLOW}(C) = \{\textbf{then}\}$

Building a leftmost derivation for **if** c **then** **if** c **then** s **else** s:

$$
\begin{aligned}
\underline{S} &\Rightarrow \textbf{if}\, \underline{C}\, \textbf{then}\, S\, S' \\
&\Rightarrow \ldots \\
&\Rightarrow \textbf{if}\, c\, \textbf{then}\, \textbf{if}\, c\, \textbf{then}\, s\, \underline{S'}\, S'
\end{aligned}
$$

Now

- **else** $\in \mathrm{FIRST}(\textbf{else}\, S)$
- as well as $S' \rightarrow \varepsilon$ and **else** $\in \mathrm{FOLLOW}(S')$

☞ based only on the first symbol **else**, one cannot decide whether $S' \rightarrow \textbf{else}\, S$ or $S' \rightarrow \varepsilon$ should be applied

Computing FOLLOW-sets inductively

FOLLOW-sets are the smallest sets that fulfill the following conditions:

- If S is the start variable, then $\$ \in \mathrm{FOLLOW}(S)$
- If $A \rightarrow \alpha B \beta$ is a production and a terminal symbol $a \in \mathrm{FIRST}(\beta)$, then $a \in \mathrm{FOLLOW}(B)$
- If $A \rightarrow \alpha B$ is a production and $a \in \mathrm{FOLLOW}(A)$, then $a \in \mathrm{FOLLOW}(B)$
- If $A \rightarrow \alpha B \beta$ is a production, $\varepsilon \in \mathrm{FIRST}(\beta)$ and $a \in \mathrm{FOLLOW}(A)$, then $a \in \mathrm{FOLLOW}(B)$

Example:

Grammar:

$$
\begin{aligned}
E &\rightarrow T\, E' \\
E' &\rightarrow +\, T\, E' \mid \varepsilon \\
T &\rightarrow F\, T' \\
T' &\rightarrow *\, F\, T' \mid \varepsilon \\
F &\rightarrow a \mid (\, E\,)
\end{aligned}
$$

FIRST-sets:

- $\mathrm{FIRST}(F) = \{a, (\}$
- $\mathrm{FIRST}(T') = \{*, \varepsilon\}$
- $\mathrm{FIRST}(T) = \{a, (\}$
- $\mathrm{FIRST}(E') = \{+, \varepsilon\}$
- $\mathrm{FIRST}(E) = \{a, (\}$

Now

- $\$ \in \mathrm{FOLLOW}(E)$ as E is the start symbol
- $) \in \mathrm{FOLLOW}(E)$ as $F \rightarrow (\, E\,)$ is a production
- $\$,) \in \mathrm{FOLLOW}(E')$ as $E \rightarrow TE'$ is a production
- $+ \in \mathrm{FOLLOW}(T)$ as $E' \rightarrow +TE'$ and $+ \in \mathrm{FIRST}(E')$
- and so on...

Finally, we build a two-dimensional *parsing table* M, where each set-valued cell $M(A,a)$ includes all those productions that can be applied when the current non-terminal variable is A and the next input string symbol is a:

- If $A \rightarrow \alpha$ is a production and the terminal symbol $a \in \mathrm{FIRST}(\alpha)$, then $A \rightarrow \alpha \in M(A,a)$
- If $A \rightarrow \alpha$ is a production, $\varepsilon \in \mathrm{FIRST}(\alpha)$ and $b \in \mathrm{FOLLOW}(A)$, then $A \rightarrow \alpha \in M(A,b)$

Definition 6.3

A grammar is an LL(1) grammar if its parsing table contains at most production in each cell.

Example:

Let us consider again the grammar

$$
\begin{array}{rcl}
E & \to & T\,E' \\
E' & \to & +T\,E' \mid \varepsilon \\
T & \to & F\,T' \\
T' & \to & *F\,T' \mid \varepsilon \\
F & \to & a \mid (E)
\end{array}
$$

The parsing table is

	a	$+$	$*$	$($	$)$	$\$$
E	$E \to T\,E'$			$E \to T\,E'$		
E'		$E' \to +T\,E'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to F\,T'$			$T \to F\,T'$		
T'		$T' \to \varepsilon$	$T' \to *F\,T'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to a$			$F \to (E)$		