Describing Lists

• There are many situations in which a programming language allows a list of items (e.g. parameter list, argument list).

• Such a list can typically be as short as empty or consisting of one item.

• Such lists are typically not bounded.

• How is their structure described?
Describing lists

• The are described using *recursive rules*.
• Here is a pair of rules describing a list of identifiers, whose minimum length is one:

  \[
  \langle \text{ident_list} \rangle \rightarrow \text{ident} \\
  \hspace{1cm} \mid \text{ident}, \langle \text{ident_list} \rangle
  \]

• Notice that ‘,’ is part of the *object language* (the language being described by the grammar).
Derivation of sentences from a grammar

• A *derivation* is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)
Recall example 2

\[ G_2 = \langle \{a, \text{the}, \text{dog}, \text{cat}, \text{chased}\}, \]
\[ \{S, \text{NP}, \text{VP}, \text{Det}, \text{N}, \text{V}\}, \]
\[ \{S \rightarrow \text{NP VP}, \text{NP} \rightarrow \text{Det N}, \text{Det} \rightarrow a \mid \text{the}, \]
\[ \text{N} \rightarrow \text{dog} \mid \text{cat}, \text{VP} \rightarrow \text{V} \mid \text{VP NP}, \text{V} \rightarrow \text{chased}\}, \]
\[ S \rangle \]
Example: derivation from $G_2$

- Example: derivation of *the dog chased a cat*

  $S \rightarrow NP \ VP$
  $\rightarrow$ Det N VP
  $\rightarrow$ the N VP
  $\rightarrow$ the dog VP
  $\rightarrow$ the dog V NP
  $\rightarrow$ the dog chased NP
  $\rightarrow$ the dog chased Det N
  $\rightarrow$ the dog chased a N
  $\rightarrow$ the dog chased a cat
Example 3

$L_3 = \{ 0, 1, 00, 11, 000, 111, 0000, 1111, \ldots \}$

$G_3 = ( \{ 0, 1 \},$

$\{ S, \text{ZeroList, OneList} \},$

$\{ S \rightarrow \text{ZeroList | OneList},$

$\text{ZeroList} \rightarrow 0 | 0 \text{ ZeroList},$

$\text{OneList} \rightarrow 1 | 1 \text{ OneList } \},$

$S )$
Example: derivations from $G_3$

• Example: derivation of $00000$
  \[
  S \to \text{ZeroList} \\
  \to 0 \text{ZeroList} \\
  \to 00 \text{ZeroList} \\
  \to 000 \text{ZeroList} \\
  \to 0000 \\
  \]

• Example: derivation of $111$
  \[
  S \to \text{OneList} \\
  \to 1 \text{OneList} \\
  \to 11 \text{OneList} \\
  \to 111 \\
  \]
Observations about derivations

• Every string of symbols in the derivation is a sentential form.

• A sentence is a sentential form that has only terminal symbols.

• A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded.

• A derivation can be leftmost, rightmost, or neither.
An example programming language grammar fragment

\[
\begin{align*}
\langle \text{program} \rangle & \rightarrow \langle \text{stmt-list} \rangle \\
\langle \text{stmt-list} \rangle & \rightarrow \langle \text{stmt} \rangle \\
& \quad | \langle \text{stmt} \rangle ; \langle \text{stmt-list} \rangle \\
\langle \text{stmt} \rangle & \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle \\
\langle \text{var} \rangle & \rightarrow a \\
& \quad | b \\
& \quad | c \\
& \quad | d \\
\langle \text{expr} \rangle & \rightarrow \langle \text{term} \rangle + \langle \text{term} \rangle \\
& \quad | \langle \text{term} \rangle - \langle \text{term} \rangle \\
\langle \text{term} \rangle & \rightarrow \langle \text{var} \rangle \\
& \quad | \text{const}
\end{align*}
\]
A leftmost derivation of
a = b + const

<program>  =>  <stmt-list>
 =>  <stmt>
 =>  <var> = <expr>
 =>  a = <expr>
 =>  a = <term> + <term>
 =>  a = <var> + <term>
 =>  a = b + <term>
 =>  a = b + const
Parse tree

- A *parse tree* is an hierarchical representation of a derivation:

```
<program>
  |
<stmt-list>
  |
<stmt>
  
  <var> = <expr>
  
  a <term> + <term>
  
  <var> const
  
  b
```
Parse trees and compilation

• A compiler builds a parse tree for a program (or for different parts of a program).
• If the compiler cannot build a well-formed parse tree from a given input, it reports a compilation error.
• The parse tree serves as the basis for semantic interpretation/translation of the program.
Extended BNF

• Optional parts are placed in brackets []
  \[<\text{proc\_call}> \rightarrow \text{ident} \ [(\langle\text{expr\_list}\rangle)]\]

• Alternative parts of RHSs are placed inside parentheses and separated via vertical bars
  \[<\text{term}> \rightarrow <\text{term}> (+|-) \text{ const}\]

• Repetitions (0 or more) are placed inside braces
  \[\{\} \]
  \[<\text{ident}> \rightarrow \text{letter} \ \{\text{letter|digit}\}\]
Comparison of BNF and EBNF

• sample grammar fragment expressed in BNF

  \[<expr> \rightarrow <expr> + <term>\]
  \[\quad | \quad <expr> - <term>\]
  \[\quad | \quad <term>\]

  \[<term> \rightarrow <term> * <factor>\]
  \[\quad | \quad <term> / <factor>\]
  \[\quad | \quad <factor>\]

• same grammar fragment expressed in EBNF

  \[<expr> \rightarrow <term> \{ (+ | - ) <term> \}\]
  \[<term> \rightarrow <factor> \{ (* | / ) <factor> \}\]
Ambiguity in grammars

• A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees

• Operator precedence and operator associativity are two examples of ways in which a grammar can provide an unambiguous interpretation.
Operator precedence ambiguity

The following grammar is ambiguous:

\[
\text{<expr>} \rightarrow \text{<expr>} \text{<op>} \text{<expr>} \mid \text{const} \\
\text{<op>} \rightarrow / \mid -
\]

The grammar treats the '/' and '-' operators equivalently.
An ambiguous grammar for arithmetic expressions

<expr> -> <expr> <op> <expr> | const
<op> -> / | -

```
const - const / const
```

```
const - const / const
```
Disambiguating the grammar

• If we use the parse tree to indicate precedence levels of the operators, we can remove the ambiguity.
• The following rules give / a higher precedence than -

\[
\begin{align*}
\langle \text{expr} \rangle & \rightarrow \langle \text{expr} \rangle - \langle \text{term} \rangle \quad | \quad \langle \text{term} \rangle \\
\langle \text{term} \rangle & \rightarrow \langle \text{term} \rangle / \text{const} \quad | \quad \text{const}
\end{align*}
\]
Links to BNF-style grammars for actual programming languages

Below are some links to grammars for real programming languages. Look at how the grammars are expressed.

- http://www.sics.se/isl/sicstuswww/site/documentation.html

In the ones listed below, find the parts of the grammar that deal with operator precedence.

- http://www.lykkenborg.no/java/grammar/JLS3.html
- http://www.lrz-muenchen.de/~bernhard/Pascal-EBNF.html
Derivation of $2 + 5 \times 3$

using C grammar