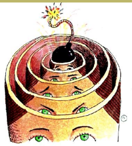
Recap
Grammars
BNF Grammars for . . .

Module Home Page
Title Page
Page 1 of 12
Back
Full Screen
Close
Quit

Lecture 8:
Programming Languages: Syntax

Aims:

- To look at how to define the syntax of programming languages using grammars in Backus-Naur Form.



Recap
Grammars
BNF Grammars for . . .

Module Home Page
Title Page
Page 2 of 12
Back
Full Screen
Close
Quit

8.1. Recap

- A language is a set of strings. There are many ways of defining a language.
  - Since it's just a set, we could give an extensional definition, e.g.:

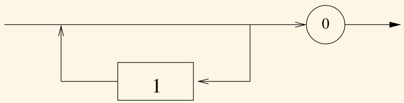
    $$L_1 =_{\text{def}} \{0, 10, 110, 1110\}$$

    . . . but only if the language is finite and, preferably, small.
  - Equally, we can give an intensional definition, e.g.:

    $$L_2 =_{\text{def}} \{w \in \{0,1\}^* \mid w = 1^*0\}$$

  - Then again, we might use a recursive definition, e.g.:

    Base case: 0 is in L_3.
    Recursive case: If w is in L_3, then $1w$ is in L_3.
    Closure: Nothing else is in L_3
  - And the last approach we saw was the use of syntax diagrams, e.g.:

    These are often used in textbooks and manuals, since they are easily understood by humans. However, they are not very compact, and they're not easy to enter into a computer.
  - In *this* lecture, we see another way: grammars. This approach is used when describing languages to machines.

Recap
Grammars
BNF Grammars for . . .

Module Home Page
Title Page
Page 3 of 12
Back
Full Screen
Close
Quit

8.2. Grammars

8.2.1. Backus-Naur Form (BNF)

- *Backus-Naur Form* (BNF) is a way of writing a grammar to define a language.
- A BNF grammar uses some symbols, specifically $::=$, $\langle$ and $\rangle$. These are *metasymbols*. It is crucial that you realise that these are part of the metalanguage; they are not part of the object language.

  It is also crucial that you realise that $::=$ is a BNF symbol and is completely different from :=, which is the DECAFF and MOCCA symbol used in assignment commands. In this lecture, we are writing grammars (using $::=$), not algorithms/programs (using :=)!
- Here is a very simple BNF grammar:

  $\langle \mathrm{S} \rangle ::= \mathrm{a}\langle \mathrm{S} \rangle$
  $\langle \mathrm{S} \rangle ::= \epsilon$

  - Symbols inside metalanguage brackets $\langle$ and $\rangle$ are called *nonterminals*. These correspond to the names inside rectangles in syntax diagrams.
  - One of the non-terminals must be designated the *start symbol*. In this case, the start symbol is $\langle \mathrm{S} \rangle$. (It is the only non-terminal in this example!).
  - Object language symbols are called *terminals*. These correspond to the names inside circles in syntax diagrams.
  - The metalanguage symbol $::=$ stands for 'is defined as' or 'rewrites as'.
  - Each line of the grammar is called a *grammar rule*.

8.2.2. Derivations

- To determine whether a particular string of terminals is a member of the language defined by a grammar, we try to find a sequence of rewrites that leads from the start symbol to the string in question.
- In the lecture we will show that *aaaa* is a member of the language defined by the grammar from above.

- In some cases, one string may have more than one derivation.
- E.g. consider this grammar with start symbol $\langle \text{S} \rangle$:

  $$\begin{aligned} \langle \text{S} \rangle &::= \langle \text{X} \rangle \langle \text{Y} \rangle \\ \langle \text{X} \rangle &::= \text{a} \\ \langle \text{Y} \rangle &::= \text{b} \end{aligned}$$

- There are two ways to derive the string ab.

8.2.3. The language defined by a grammar

- The language defined by a grammar is the set of all strings of terminals that can be derived from the start symbol.
- The language defined by this grammar:

  $$\begin{aligned} \langle \text{S} \rangle &::= \text{a}\langle \text{S} \rangle \\ \langle \text{S} \rangle &::= \epsilon \end{aligned}$$

  is $\{\epsilon, a, aa, aaa, aaaa, aaaaa, \ldots\}$, i.e. a^*.

- The language defined by this grammar:

$$\begin{aligned} \langle S \rangle &::= \langle X \rangle \langle Y \rangle \\ \langle X \rangle &::= \text{a} \\ \langle Y \rangle &::= \text{b} \end{aligned}$$

  is just $\{ab\}$.

Class Exercise

- Here is a grammar, whose start symbol is $\langle S \rangle$:

$$\begin{aligned} \langle S \rangle &::= \langle X \rangle \text{aa} \langle X \rangle \\ \langle X \rangle &::= \text{a} \langle X \rangle \\ \langle X \rangle &::= \text{b} \langle X \rangle \\ \langle X \rangle &::= \epsilon \end{aligned}$$

  1. Is *bab* a member of the language defined by this grammar?
  2. What about *baab*?
  3. *baaa*?
  4. Describe in words the language defined by this grammar.

8.2.4. Parse Trees

- *Parse trees* are a graphical representation of the grammar rules used to derive a string. Parse trees have the advantage that they make explicit the hierarchical structure of the strings.
- To draw a parse tree,
  - put the start symbol of the grammar at the root of the tree;

Recap
Grammars
BNF Grammars for . . .

- each time you use a rule ⟨A⟩ ::= α to replace nonterminal ⟨A⟩ by a sequence of terminals and/or nonterminals α, then install the members of α as children of ⟨A⟩.

- E.g. consider this grammar with start symbol ⟨S⟩:

  ⟨S⟩ ::= a⟨S⟩
  ⟨S⟩ ::= ϵ

- *aaa* is a member of the language defined by this grammar, and in the lecture we will draw the parse tree.

Class Exercise

- The following grammar has start symbol ⟨S⟩:

  ⟨S⟩ ::= ⟨X⟩aa⟨X⟩
  ⟨X⟩ ::= a⟨X⟩
  ⟨X⟩ ::= b⟨X⟩
  ⟨X⟩ ::= ϵ

- Draw a parse tree for string *baab*.

8.2.5. Ambiguity

- A grammar is *ambiguous* if the language it defines contains at least one string that has *two or more possible derivations which correspond to different parse trees.*

- We'll first revisit an example where there isn't ambiguity!

Recap
Grammars
BNF Grammars for . . .

Module Home Page
Title Page
◀◀ ▶▶
◀ ▶

Back
Full Screen
Close
Quit

- We saw earlier that we can derive string *ab* from the following grammar in two ways.

  $\langle \mathrm{S} \rangle ::= \langle \mathrm{X} \rangle\langle \mathrm{Y} \rangle$
  $\langle \mathrm{X} \rangle ::= \mathrm{a}$
  $\langle \mathrm{Y} \rangle ::= \mathrm{b}$

- However, both derivations give us the same parse tree. Hence, the grammar is unambiguous.

- But now consider this grammar (start symbol $\langle \mathrm{S} \rangle$):

  $\langle \mathrm{S} \rangle ::= \mathrm{a}\langle \mathrm{S} \rangle$
  $\langle \mathrm{S} \rangle ::= \langle \mathrm{S} \rangle\mathrm{a}$
  $\langle \mathrm{S} \rangle ::= \mathrm{a}$

- There are four derivations of *aaa* and each one gives a different parse tree.

- This grammar is *ambiguous*.

Recap

Grammars

BNF Grammars for . . .

Module Home Page

Title Page

Page 8 of 12

Back

Full Screen

Close

Quit

8.3. BNF Grammars for Programming Languages

- We can define the syntax of a programming language use a BNF grammar.
- Here is a BNF grammar for MOCCA corresponding to the syntax diagrams we saw in the previous lecture.
- The start symbol is ⟨program⟩.

$$
\begin{aligned}
\langle\text{program}\rangle &::= \langle\text{block}\rangle \\
\langle\text{block}\rangle &::= \{\ \langle\text{command-list}\rangle\ \} \\
\langle\text{command-list}\rangle &::= \epsilon \\
\langle\text{command-list}\rangle &::= \langle\text{command}\rangle\ \langle\text{command-list}\rangle \\
\langle\text{command}\rangle &::= \langle\text{block}\rangle \\
\langle\text{command}\rangle &::= \langle\text{assignment}\rangle \\
\langle\text{command}\rangle &::= \langle\text{one-armed-conditional}\rangle \\
\langle\text{command}\rangle &::= \langle\text{two-armed-conditional}\rangle \\
\langle\text{command}\rangle &::= \langle\text{while-loop}\rangle \\
\langle\text{assignment}\rangle &::= \langle\text{var}\rangle := \langle\text{expr}\rangle \\
\langle\text{one-armed-conditional}\rangle &::= \textbf{if}\ \ \langle\text{expr}\rangle\ \langle\text{command}\rangle \\
\langle\text{two-armed-conditional}\rangle &::= \textbf{if}\ \ \langle\text{expr}\rangle\ \langle\text{command}\rangle\ \textbf{else}\ \ \langle\text{command}\rangle \\
\langle\text{while-loop}\rangle &::= \textbf{while}\ \ \langle\text{expr}\rangle\ \langle\text{command}\rangle \\
\text{etc.} &
\end{aligned}
$$

Class Exercise

- Syntax diagrams and BNF grammars have equivalent power: whatever languages you can describe with one, you can describe with the other.
- But my syntax diagrams and BNF grammar for MOCCA are not equivalent. The BNF grammar allows something that the syntax diagrams do not.

– What is it?

– How would you make them equivalent?

Parse Trees and Ambiguity

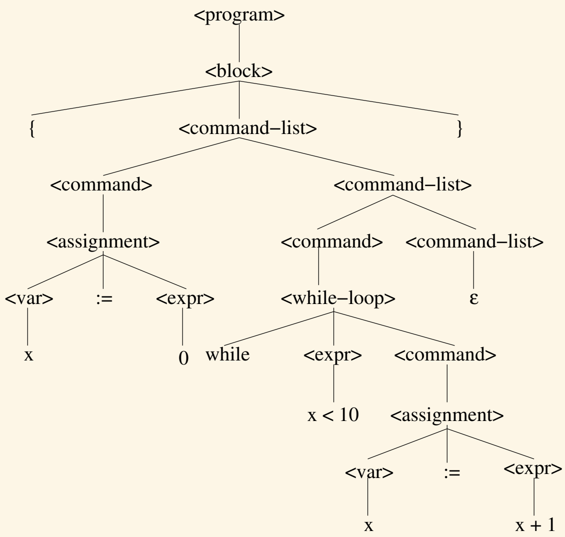
- For the purposes of illustration, here is a MOCCA program:

$$
\begin{array}{l}
\{\quad x := 0 \\
\qquad \textbf{while}\ x < 10 \\
\qquad\quad x := x + 1 \\
\}
\end{array}
$$

The BNF grammar tells us that this is a syntactically well-formed program.

- The following parse tree confirms that the program above is syntactically well-formed. It also shows the rules used to derive the program and the program's hierarchical structure.

Recap
Grammars
BNF Grammars for . . .
Module Home Page
Title Page
Page 10 of 12
Back
Full Screen
Close
Quit



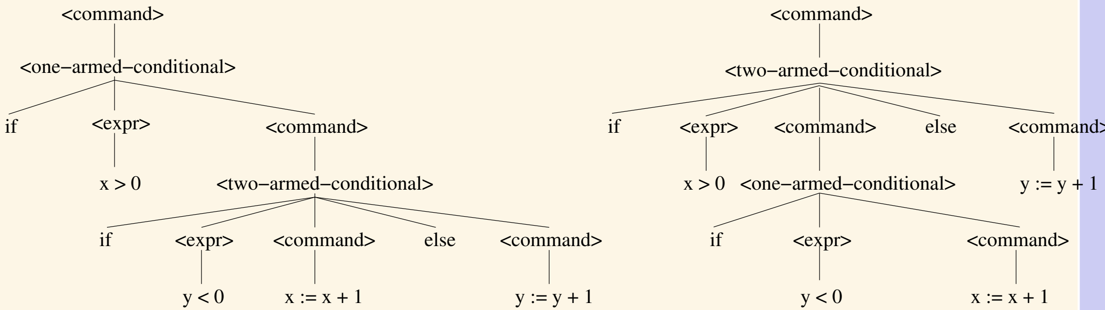
- Here's another fragment of a MOCCA program:

if $x > 0$
  if $y < 0$
    $x := x + 1$
  else
    $y := y + 1$

This MOCCA program is well-formed according to our grammar.

Recap
Grammars
BNF Grammars for . . .
Module Home Page
Title Page
Page 11 of 12
Back
Full Screen
Close
Quit

- But it has two parse trees:



- You should recall that this is an example of a dangling-**else** .
- For programming languages, ambiguity is generally undesirable. What should we do?
  - Either: abandon this grammar and come up with an unambiguous grammar.
  - Or: stick with this grammar but devise some disambiguation conventions that tell us which parse trees to discard.

Acknowledgements

Some of the grammars come from [Coh91].

Clip Art (of head with bomb) licensed from the Clip Art Gallery on DiscoverySchool.com.

References

[Coh91] D. I. A. Cohen. *Introduction to Computer Theory*. John Wiley, 2nd. edition, 1991.