Principles of programming languages (2007) Lecture 2

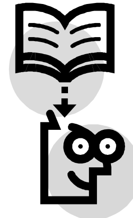
http://few.vu.nl/~nsilvis/PPL/2007

Natalia Silvis-Cividjian
e-mail: nsilvis@few.vu.nl

vrije Universiteit *amsterdam*

Imagine: You are a **novice.** You start to learn a new (programming) language.

What do you want to know about this language ?



Language definition

§Programming language **syntax** = how programs look, their form and structure
Language syntax is defined by a **grammar**

§Programming language **semantics** = what programs do, their behavior and meaning
Language semantics is more difficult to define but grammars can help.

Grammars

- Part I. Grammars and syntax
- Part II. Grammars and semantics

Part I

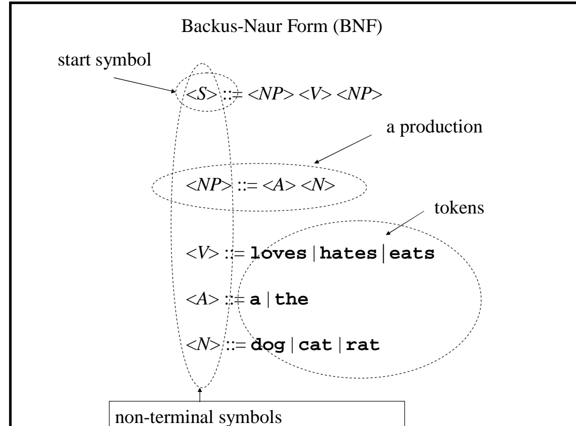
Grammars and syntax

WHAT is a **GRAMMAR**?

A set of rules to define a language

Example 1: A grammar for a subset of English language

A sentence is a noun phrase, a verb, and a noun phrase.	<*S*> ::= <*NP*> <*V*> <*NP*>
A noun phrase is an article and a noun.	<*NP*> ::= <*A*> <*N*>
A verb can be the word `loves` or `hates` or `eats`	<*V*> ::= **`loves`** | **`hates`** | **`eats`**
An article is…	<*A*> ::= **`a`** | **`the`**
A noun is...	<*N*> ::= **`dog`** | **`cat`** | **`rat`**

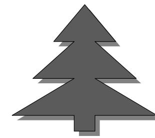


Backus-Naur Form

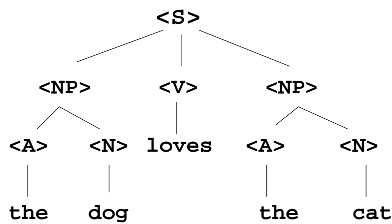
- First used for the syntax of ALGOL60
- BNF is a metalangauge = language used to define another language
- BNF is context free if the left part of a production is always a single nonterminal.

HOW can a **GRAMMAR** generate a **LANGUAGE**?

Using parse trees

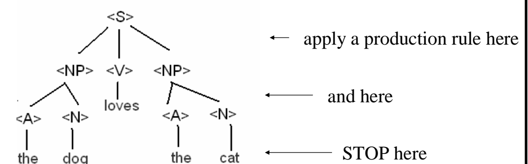


A possible parse tree



`the dog loves the cat`

Algorithm to build a parse tree

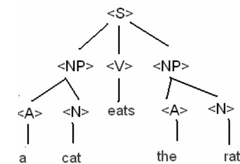


Generated string: **`the dog loves the cat`**

How to build a parse tree

- START: Put <*S*> at the root of the tree
- The grammar's rules say how children can be added at any point in the tree
- For instance, the rule
  <*S*> ::= <*NP*> <*V*> <*NP*>
  says you can add nodes <*NP*>, <*V*>, and <*NP*>, in that order, as children of <*S*>
- Apply a production rule to build another level in the tree, until you get to tokens. Then STOP.

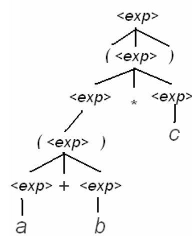
Another parse tree



Generated string: a cat eats the rat

Example 2: A grammar for a Java subset

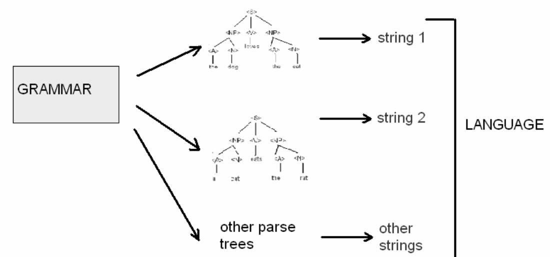
```
<exp> ::= <exp> + <exp> | <exp> * <exp> |
          ( <exp> ) | a | b | c
```



A possible parse tree

Generated string:

`((a+b)*c)`



A grammar generates a language

Who needs the grammars?

- **Novices** want to find out what legal programs look like
- **Experts**—advanced users and language system implementers (compiler builders) want an exact, detailed definition
- **Tools**—parser and scanner generators, want an exact, detailed definition in a particular, machine-readable form

How does the compiler check if a program is valid?

Syntax analysis

- **Syntax analysis** = check if the program is written according to the language grammar
- **Parsing** = build a parse tree for each string following the grammar rules
- If success = string is in the language
- If failed = string is rejected = syntax error

Tokens

- Tokens are the smallest pieces of a program text (atoms)
- Identifiers (**`count`**), keywords (**`if`**), operators (**`==`**), constants (**`123.4`**), etc.
- Programs stored in files are just sequences of characters
- How is such a file divided into a sequence of tokens?

Lexical and Phrase Structure

- Grammars so far have defined **phrase structure**: how a program is built from a sequence of tokens
- We also need to define **lexical structure**: how a text file is divided into tokens

One Grammar For Both?

- You could do it all with one grammar by using characters as the only tokens
- Not done in practice: things like white space and comments would make the grammar too messy to be readable

<if-stmt> ::= **`if`** *<white-space> <expr> <white-space>*
`then` *<white-space>*
<stmt> <white-space> <else-part>
<else-part> ::= **`else`** *<white-space> <stmt>* | *<empty>*

Separate Grammars

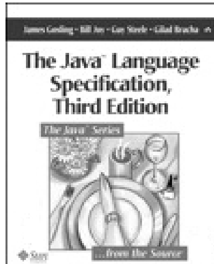
- Usually there are two separate grammars
  - One says how to construct a sequence of tokens from a file of characters = **lexical grammar**
  - One says how to construct a parse tree from a sequence of tokens = **syntactic grammar**

Lexical grammar

<program-file> ::= *<end-of-file>* | *<element> <program-file>*
<element> ::= *<token>* | *<one-white-space>* | *<comment>*
<one-white-space> ::= *<space>* | *<tab>* | *<end-of-line>*
<token> ::= *<identifier>* | *<operator>* | *<constant>* | …

Example

http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html



25

Example

WhileStatement:
while (*Expression*) *Statement*

DoStatement:
do *Statement* while (*Expression*) ;

ForStatement:
for ($ForInit_{opt}$; $Expression_{opt}$; $ForUpdate_{opt}$) *Statement*

[from *The Java™ Language Specification*, James Gosling et. al. (link on the website)]

26

Separate Compiler Passes

- The ***scanner*** reads the input file and divides it into tokens according to the first grammar, discarding white space and comments = lexical analysis
- The ***parser*** constructs a parse tree (or at least goes through the motions—more about this later) from the token stream according to the second grammar = syntactic analysis

27

Historical Note #1

- Early languages sometimes did not separate lexical structure from phrase structure
  - Early Fortran and Algol dialects allowed spaces anywhere, even in the middle of a keyword
  - Other languages like PL/I allow keywords to be used as identifiers
- This makes them harder to scan and parse and it also reduces readability

28

Historical Note #2

- Some languages have a *fixed-format* lexical structure—column positions are significant
  - One statement per line (i.e. per card)
  - First few columns for statement label
  - Etc.
- Early dialects of Fortran, Cobol, and Basic
- Almost all modern languages are *free-format*: column positions are ignored

29

Other Grammar Forms

- BNF variations
- EBNF variations
- Syntax diagrams

Principles of programming languages - 2007 - Lecture 2

30

BNF Variations

- Some use → or = instead of ::=
- Some leave out the angle brackets and use a distinct typeface for tokens
- Some allow single quotes around tokens, for example to distinguish '|' as a token from | as a meta-symbol

E(xtended)BNF

- Additional syntax to simplify some grammar chores:
  - {x} to mean zero or more repetitions of x
  - [x] to mean x is optional (i.e. x | <*empty*>)
  - () for grouping
  - | anywhere to mean a choice among alternatives
  - Quotes around tokens, if necessary, to distinguish from all these meta-symbols

EBNF Examples

<*if-stmt*> ::= **if** <*expr*> **then** <*stmt*> [**else** <*stmt*>]

<*stmt-list*> ::= {<*stmt*> **;**}

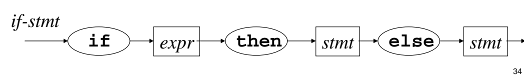
<*thing-list*> ::= { (<*stmt*> | <*declaration*>) **;**}

- Anything that extends BNF this way is called an **Extended BNF: EBNF**
- There are many variations

Syntax Diagrams

- Syntax diagrams ("railroad diagrams")
- Start with an EBNF grammar
- A simple production is just a chain of boxes (for nonterminals) and ovals (for terminals):

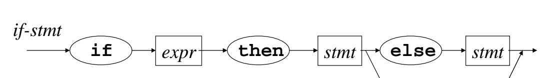
<*if-stmt*> ::= **if** <*expr*> **then** <*stmt*> **else** <*stmt*>



Bypasses

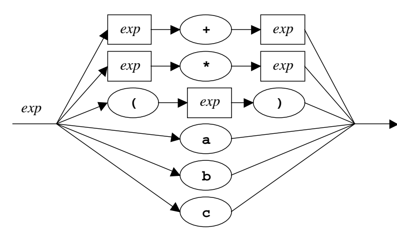
- Square-bracket pieces from the EBNF get paths that bypass them

<*if-stmt*> ::= **if** <*expr*> **then** <*stmt*> [**else** <*stmt*>]



Branching

<*exp*> ::= <*exp*> **+** <*exp*> | <*exp*> ***** <*exp*> | **(** <*exp*> **)**
| **a** | **b** | **c**



Syntax Diagrams, Pro and Con

- Easier for people to read, used in tutorials
- Harder to read precisely: what will the parse tree look like?
- Harder to make machine readable (for automatic parser-generators)

Grammar influence on design

- A simple, readable and short grammar makes the language easy to learn and to use.

Summary

- Syntax describes how a correct program looks like, its form
- Syntax is defined by a grammar
- A grammar generates a language
- Novices use grammars to learn how to write a correct program
- Compilers use grammars to check if a program is valid or not = syntax analysis

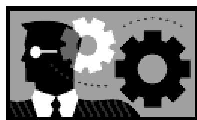
Part II

Grammars and semantics

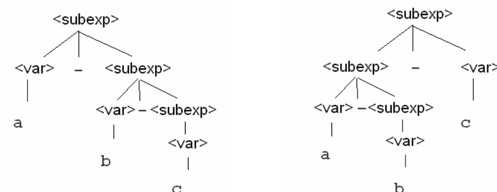
Imagine :

You are a language **designer.**

How to write a grammar to specify your language?



What **means** a-b-c ?



Semantics

- The 2 grammars generate the same language (fringes of the parse trees are the same)
- BUT…The internal structure of the 2 parse trees is different
- Semantics depends on the parse tree structure

Formal semantics definition

- Algol 60 used BNF, but now ISO and ANSI use English.
- Other formal methods :
  - denotational semantics
  - axiomatic semantics

Working Grammar

Take this grammar :

```
G1: <exp> ::= <exp> + <exp>
            | <exp> * <exp>
            | (<exp>)
            | a | b | c
```

And this string :

a + b * c

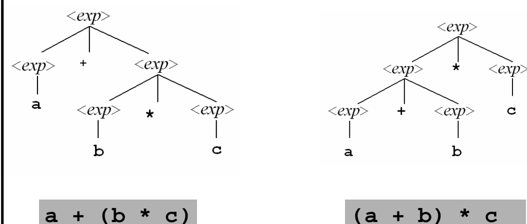
What means a + b * c ?

Ambiguity

- **G1 is an ambiguous grammar** = a BNF grammar that allows different interpretations of the same sentence (different parse trees for the same string)
- Grammars have to generate unique parse trees, so grammars have to try to be unambiguous.

Ambiguity case #1: precedence

The grammar can generate 2 different parse trees:



`a + (b * c)`   `(a + b) * c`

Operator Precedence

- Applies when the order of evaluation is not completely decided by parentheses
- Each operator has a *precedence level,* and those with higher precedence are performed before those with lower precedence, as if parenthesized
- Most languages put `*` at a higher precedence level than `+`, so that
  `a+b*c = a+(b*c)`

Precedence Examples

- C (15 levels of precedence—too many?)

```
a = b < c ? * p + b * c : 1 << d ()
```

- Pascal (5 levels—not enough?)

```
a <= 0 or 100 <= a
```

Error!

- Smalltalk (1 level for all binary operators)

```
a + b * c
```

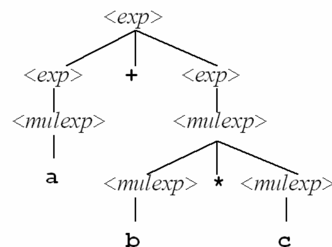

Fix the problem in the grammar

```
G1:   <exp> ::= <exp> + <exp>
              | <exp> * <exp>
              | (<exp>)
              | a | b | c
```

To fix the precedence problem, we modify the grammar so that it is forced to put `*` below `+` in the parse tree.

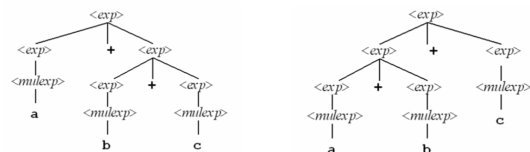
```
G2:   <exp> ::= <exp> + <exp> | <mulexp>
      <mulexp> ::= <mulexp> * <mulexp>
              | (<exp>)
              | a | b | c
```


Correct Precedence



What means `a+b+c` for G2?

Ambiguity case #2: Associativity



Grammar G2 generates both these trees for **a+b+c**. The first one is not the usual convention for operator *associativity*.

Operator Associativity

- Applies when the order of evaluation is not decided by parentheses or by precedence
- *Left-associative* operators group left to right: `a+b+c+d = ((a+b)+c)+d`
- *Right-associative* operators group right to left: `a+b+c+d = a+(b+(c+d))`
- Most operators in most languages are left-associative, but there are exceptions

Associativity Examples

- C
  - `a<<b<<c` — most operators are left-associative
  - `a=b=0` — right-associative (assignment)
- ML
  - `3-2-1` — most operators are left-associative
  - `1::2::nil` — right-associative (list builder)
- Fortran
  - `a/b*c` — most operators are left-associative
  - `a**b**c` — right-associative (exponentiation)

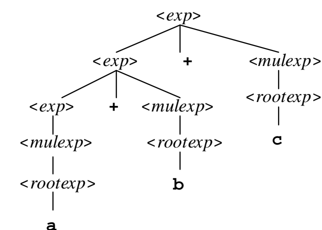
Fix the problem in the Grammar

```
G2:  <exp> ::= <exp> + <exp> | <mulexp>
     <mulexp> ::= <mulexp> * <mulexp>
                | (<exp>)
                | a | b | c
```

To fix the associativity problem, we modify the grammar to make trees of **+**s grow down to the left (and likewise for *****s)

```
G3:  <exp> ::= <exp> + <mulexp> | <mulexp>
     <mulexp> ::= <mulexp> * <rootexp> | <rootexp>
     <rootexp> ::= (<exp>)
                 | a | b | c
```


Correct Associativity



Ambiguity case #3: Dangling Else

- A chain of **if-then-else** constructs can be very hard for people to read
- Especially true if some but not all of the else parts are present

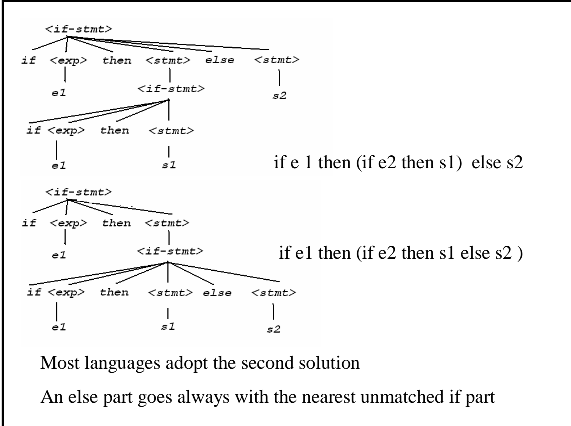
The dangling else

Take this grammar:

```
<stmt> ::= <if-stmt> | s1 | s2
<if-stmt> ::= if <expr> then <stmt> else <stmt>
            | if <expr> then >stmt>
<expr> ::= e1 | e2
```

What means

```
if e1 then if e2 then s1 else s2
```



if e 1 then (if e2 then s1) else s2

if e1 then (if e2 then s1 else s2)

Most languages adopt the second solution

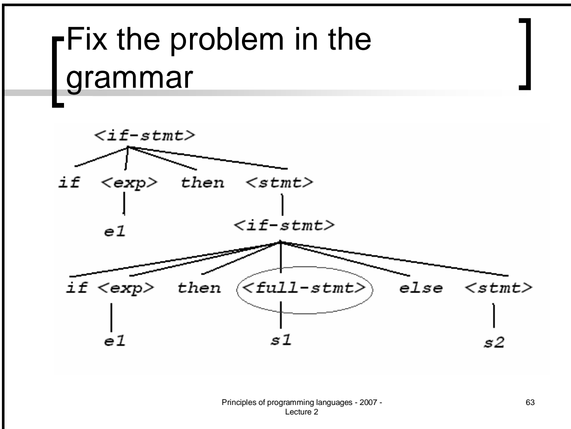
An else part goes always with the nearest unmatched if part

Fix the dangling else problem

G4:

```
<stmt> ::= <if-stmt> | s1 | s2
<if-stmt> ::= if <expr> then <full-stmt> else
<stmt> | if <expr> then >stmt>
<expr> ::= e1 | e2
<full-stmt> ::=<full-if> | s1 | s2
<full-if> ::= if <expr> then <full-stmt> else
<full-stmt>
```


Fix the problem in the grammar



Languages That Don’t Dangle

- Some languages define if-then-else in a way that forces the programmer to be more clear
- Algol does not allow the **`then`** part to be another **`if`** statement – though it can be a block containing an **`if`** statement
- Ada requires each **`if`** statement to be terminated with an **`end if`**

`What happens when we modify the grammar to eliminate ambiguity?`

Clutter

- The new if-then-else grammar is harder for people to read than the old one
- It has a lot of **clutter**: more productions and more non-terminals
- Same with G2, G3 and G4: we eliminated the ambiguity but made the grammar harder for people to read
- This is not always the right trade-off

Design trade-off: ambiguity vs. clutter

- Rewrite grammar to eliminate ambiguity (tools are happy, people not)

  OR
- Leave ambiguity but explain in accompanying text how things like associativity, precedence, and the dangling else should be parsed

  OR
- Do both in separate grammars

67

Conclusion

- Grammars define syntax, *and more*
- They define not just a set of legal programs, but a parse tree for each program
- The structure of a parse tree corresponds to the order in which different parts of the program are to be executed
- Thus, grammars contribute (a little) to the definition of semantics

68

Exercise (exam type)

Given this grammar:

<exp> ::= *<exp>* **+** *<exp>* | *<exp>* ***** *<exp>* | **(** *<exp>* **)**
| **a** | **b** | **c**

Show a parse tree for each of these strings:

```
a+b
a*b+c
(a+b)
(a+(b))
```

Show that the grammar is ambiguous and try to eliminate the ambiguity.