Language Description

Don't tell me of a man's being able to talk sense; everyone can talk sense. Can he talk nonsense?
-- William Pitt

- Syntax 3.2, 3.3
  - formal grammar
  - context-free grammars, BNF
  - derivation, parsing
  - extended BNF (EBNF)
  - ambiguous grammars
- Semantics 3.5, 3.6
  - static semantics
  - dynamic semantics
    - operational semantics
    - axiomatic semantics
    - denotational semantics

Motivation

Programming languages must be very precise. In order to use a language, a programmer must know:

- what are the *legal constructs* of the language
  - data (built-in types, complex structures, etc.)
  - control (loops, subprograms, etc.)
- what *keywords* are used to represent them
- how can constructs be *combined* to form legal *programs*
- what is the *meaning* of programs
  - further constraints on combinations of constructs
  - the results of execution

Definitions

- A *language* is a set of *sentences* built of *words* from a dictionary combined according to a set of *rules*.
- The set of rules for how words combine to form legal sentences is called the *syntax* of the language.
- The *meanings* of words and combinations of words make up the *semantics* of a language.
- Rules can be specified using various *formalisms*; one such formalism is called a *grammar*.

What are the words *in a programming language?*

What are the sentences *in a programming language?*

Premature Exemplification

We wish to define the language of *smileys*: `:-)`, `:-(`, `8-D`, `:^7`, ...

- Our dictionary consists of the following words
  - D = { `:   ;   X   8   |   -   ^   ’   )   (   D   7   b   >   <` }
- The syntax is specified with the following grammar rules
  - *eyes* can be any of { `:   ;   X   8   |` }
  - *nose* can be any of { `-   ^   ’` }
  - *mouth* can be any of { `)   (   D   7   b   >   <` }
  - *bigsmiley* can be *eyes* followed by *nose* followed by *mouth*
  - *littlesmiley* can be *eyes* followed by *mouth*
  - *smiley* can be *bigsmiley*
  - *smiley* can be *littlesmiley*

Which of the following are legal sentences in our language?

`;->   :’(   X-b   (-:   8-O   |<   *<8-{)}}`

Lexical Analysis

The syntax of a language specifies how words can be combined to form sentences. *Lexical analysis* takes a program file (sequence of characters) and extracts the words.

- Words in a programming language are called *tokens*.
  - identifiers
    - variable names, function names, labels, etc.
    - `my_counter`, `do_this_thing`, `OUTER_LOOP`, etc.
  - keywords (a.k.a. reserved words)
    - control words, type names, built-in operators, etc.
    - `while`, `char`, `mod`, `%`, etc.
  - literals (a.k.a. constants)
    - `42`, `4.2E+01`, "`throwdown at the hoedown`", etc.
  - punctuation
    - `;`, `(`, `)`, `[`, `]`, `,`, `"`, `'`, etc.

Syntactic Analysis

Recall *syntax*: rules for combining words into legal sentences.

Recall *grammar*: a formalism for defining the rules.

How can you specify the legal sentences of a language?

- build a *recognizer*
  - "accepts" sequences of words that satisfy rules in the grammar
  - "rejects" sequences of words that don't satisfy the grammar rules
- build a *generator*
  - starting with the "most general" rule, apply rules until a sequence of words is generated
  - all sequences generated in this way are legal sentences

A *parser* is a recognizer that keeps a record of which rules were used in the process of accepting or rejecting a sentence.

Formal Grammars

A *formal grammar* is a language for describing the syntax of another language. It consists of four components:

- *terminal symbols*
  - individual language elements
  - *tokens* in programming languages
  - *words* in natural languages
- *nonterminal symbols*
  - symbols in the grammar that correspond to combinations of one or more terminals and nonterminals
- a *goal* symbol (a.k.a. the *start* symbol)
  - the top-level symbol representing *sentences* in the language
- *production rules* (a.k.a. *rewrite* rules)
  - rules for combining terminals (and nonterminals) to form more general structures

Formal Grammars (cont.)

In our language of *smileys*:

- *What are the* terminal symbols*?*
- *What are the* nonterminal symbols*?*
- *What is the* goal symbol*?*
- *What are the* production rules*?*

Backus-Naur Form (BNF)

BNF is a handy notation for writing grammars. Here's a grammar for our *smiley* language written in BNF:

```
<smiley>       ::=  <bigsmiley>
<smiley>       ::=  <littlesmiley>
<bigsmiley>    ::=  <eyes> <nose> <mouth>
<littlesmiley> ::=  <eyes> <mouth>
<eyes>         ::=  : | ; | X | 8 | |
<nose>         ::=  - | ^ | '
<mouth>        ::=  ) | ( | D | 7 | b | > | <
```

'::=' means "is composed of"

'|' means "or"

Derivations

- How do we know what the legal sentences in a language are?
- Given a sequence of words, how can we tell it is a legal sentence in a language?

By using the grammar as a recognizer *or* generator!

Derivations (cont.)

- from the *start* symbol, *produce* more and more specific sequences by replacing *nonterminals* (LHS) with their *definitions* (RHS)
  - any sequence of all terminals produced (generated) in this way will be a legal sentence in the language
  - a *top-down derivation*

- *reduce* a sequence into more and more general forms by replacing *definitions* (RHS) with their corresponding *nonterminals* (LHS)
  - if the reduction eventually leads to the *goal* symbol, the original sequence was a legal sentence in the language
  - a *bottom-up derivation*

59

Generating Smileys

Start with the *start* symbol:

<smiley> ⇒

<bigsmiley> ⇒

<eyes> <nose> <mouth> ⇒

: <nose> <mouth> ⇒

: **-** <mouth> ⇒

: **-** **)**

```
<smiley>        ::=  <bigsmiley>
<smiley>        ::=  <littlesmiley>
<bigsmiley>     ::=  <eyes>  <nose>  <mouth>
<littlesmiley>  ::=  <eyes>  <mouth>
<eyes>          ::=  : | ; | X | 8 | |
<nose>          ::=  - | ^ | '
<mouth>         ::=  ) | ( | D | 7 | b | > | <
```

60

Generating Smileys

Start with the *start* symbol:

<smiley> ⇒

<littlesmiley> ⇒

<eyes> <mouth> ⇒

; <mouth> ⇒

; >

<smiley> ::= <bigsmiley>
<smiley> ::= <littlesmiley>
<bigsmiley> ::= <eyes> <nose> <mouth>
<littlesmiley> ::= <eyes> <mouth>
<eyes> ::= : | ; | X | 8 | |
<nose> ::= - | ^ | ’
<mouth> ::=) | (| D | 7 | b | > | <

Recognizing Smileys

Start with the sequence of terminal symbols:

:’(⇒

<eyes> **’(** ⇒

<eyes> <nose> **(** ⇒

<eyes> <nose> <mouth> ⇒

<bigsmiley> ⇒

<smiley>

<smiley> ::= <bigsmiley>
<smiley> ::= <littlesmiley>
<bigsmiley> ::= <eyes> <nose> <mouth>
<littlesmiley> ::= <eyes> <mouth>
<eyes> ::= : | ; | X | 8 | |
<nose> ::= - | ^ | ’
<mouth> ::=) | (| D | 7 | b | > | <

Derivation Trees

For both top-down and bottom-up derivations, we can keep a record of the production rules that are applied during the derivation. Often, we use a tree as a record of applied rules (called a *derivation tree* or an *abstract syntax tree* or a *parse tree*).

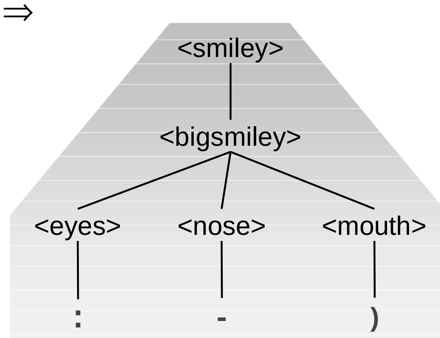
- for top-down derivations, the start symbol is the *root* of the tree; every time a LHS is rewritten as the corresponding RHS of a rule, the elements of the RHS become children of the LHS symbol in the tree
- for bottom-up derivations, the terminals in the sentence are the *leaves* of the tree; every time a group of symbols from the RHS of a rule is replaced with the LHS, the LHS symbol becomes the parent of those symbols

63

A Top-Down Derivation Tree

Remember our smiley derivation:

<smiley> ⇒
<bigsmiley> ⇒
<eyes> <nose> <mouth> ⇒
: <nose> <mouth> ⇒
: **-** <mouth> ⇒
: **-** **)**



64

A Bottom-Up Derivation Tree

Start with the sequence of terminal symbols:

: ’ (⇒

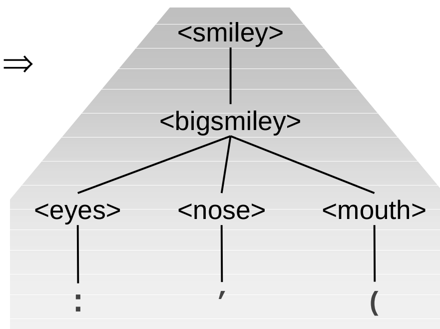
<eyes> **’ (** ⇒

<eyes> <nose> **(** ⇒

<eyes> <nose> <mouth> ⇒

<bigsmiley> ⇒

<smiley>



Note that there is no record of the order *the rules were applied!*

65

Extending the Metalanguage

A *metalanguage* is a language for describing other languages. BNF is a metalanguage for programming languages.

- *S* ::= *A* — *definition* (standard BNF)
  - S is defined as A
  - in a *production*, S can be rewritten as A
  - in a *reduction*, A can be rewritten as S
- *S* ::= *A* **|** *B* — *disjunction* (EBNF)
  - S is defined as A *or* B
  - equivalent to:
    S ::= A
    S ::= B

66

Extending the Metalanguage (cont.)

- *S ::= A [B]* — *optionality*
  - S is defined as A *optionally followed by* B — (EBNF)
  - equivalent to:

    S ::= A
    S ::= A B
- *S ::= A { B }* — *repetition*
  - S is A *followed by zero or more occurrences of* B — (EBNF)
  - equivalent to:

    S ::= A
    S ::= A B
    S ::= A B B
    S ::= A B B B
    …

Infinity

Part of the power of a grammar is that it is a *finite* description of an *infinite language* (a language with an infinite number of legal sentences).

- for example, the number of possible Pascal programs is infinite, but the grammar of Pascal is quite small (and finite!)

How could EBNF be used to describe an infinite language?

Infinity Continued

- the *repetition* notation of EBNF can be used to describe infinite languages.
  - <number> ::= <digit> { <digit> }
    <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- an even more powerful expression of infinite productions in a grammar is *recursion*:
  - <number> ::= <digit>
    <number> ::= <digit> <number>
    <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

69

A Simple Infinite Language

Here's an example of using a simple EBNF grammar to describe an infinite language of mathematical expressions:

<expr> ::= <expr> + <expr> |
<expr> × <expr> |
<number>
<number> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

(Normally, a grammar of expressions would account for numbers with more than one digit:

<number> ::= <digit> [<number>]
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

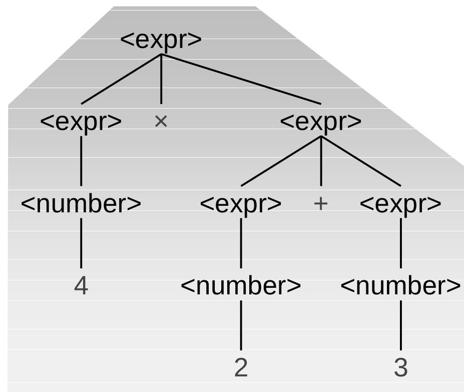
)

70

An Example

Let’s do a top-down derivation of the expression 4 × 2 + 3:

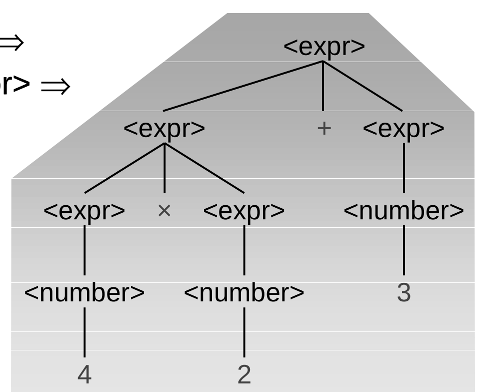
<expr> ⇒
<expr> × <expr> ⇒
<number> × <expr> ⇒
4 × <expr> ⇒
4 × <expr> + <expr> ⇒
4 × <number> + <expr> ⇒
4 × 2 + <expr> ⇒
4 × 2 + <number> ⇒
4 × 2 + 3



An Example Twisted

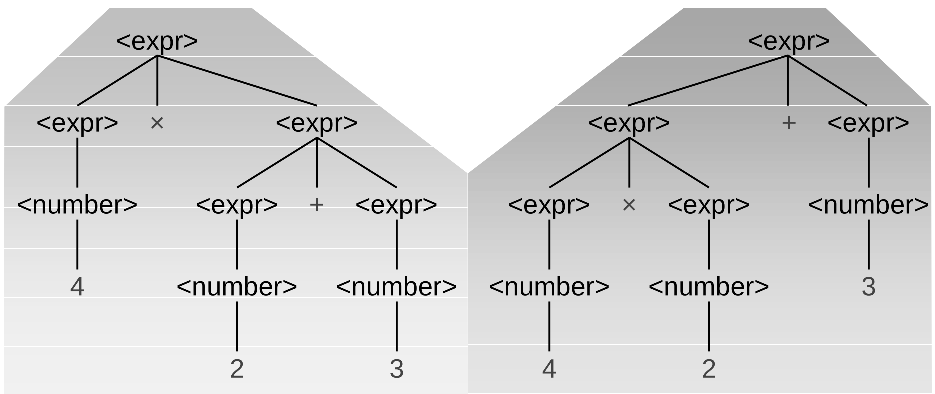
But there are no rules governing the order to apply the rules:

<expr> ⇒
<expr> + <expr> ⇒
<expr> × <expr> + <expr> ⇒
<number> × <expr> + <expr> ⇒
4 × <expr> + <expr> ⇒
4 × <number> + <expr> ⇒
4 × 2 + <expr> ⇒
4 × 2 + <number> ⇒
4 × 2 + 3



Ambiguous Grammars

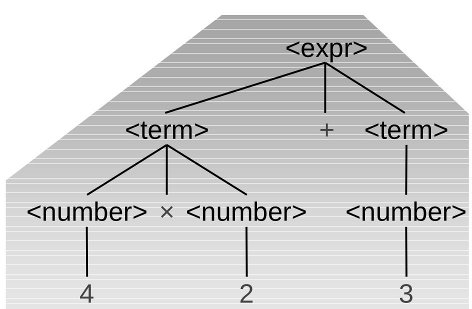
A grammar is *ambiguous* when there exists a sentence in the language that has more than one derivation tree.



An Unambiguous Grammar for Expressions

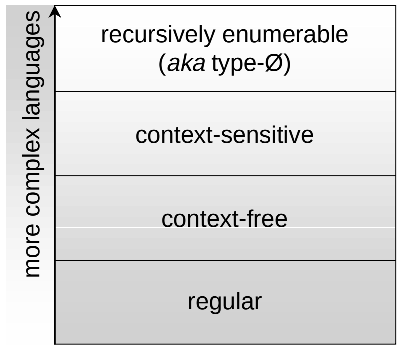
Here’s a slightly modified grammar for the infinite language of mathematical expressions, adjusted to be unambiguous:

<expr> ::= <term> { + <term> }
<term> ::= <number> { × <number> }
<number> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



Classes of Languages

Depending on the kinds of rules used to generate sentences, a language can be very simple or very complex.



Semantics

- The *syntax* of a programming language defines
  - the *structure* of combinations of basic elements
- The *semantics* of a programming language defines
  - the *meaning* of basic elements and their combinations

Unfortunately, there's no room in this course for much semantics, but check out this baby:

> **CSI 4125. Theory of Programming Languages (3,0,0) 3 cr.**
> The concept of formal semantics. Attribute grammars. Denotational semantics. Operational semantics. Axiomatic semantics. Lambda-calculus for programming language description. Resolution and the semantics of logic programming. Theory of abstract data types. Concurrent programming, process algebras, CCS, CSP.
> *Prerequisites:* CSI3104, CSI3125, CSI3310

Syntax vs. Semantics: An Example

Consider the Pascal statement:

```
myvar := (i + 3) * 2;
```

- The syntax of Pascal says that
  - the tokens **(**, **i**, **+**, **3**, **)**, ***** and **2** combine to make a valid expression
  - the statement is in the form of a legal assignment statement
- The semantics of Pascal tell us that
  - the variable named `myvar` must be a numeric type
  - the variable named `i` must be a numeric type
  - the value of `(i + 3)` is three greater than the current value of `i`
  - the value of `(i + 3) * 2` is double the value of `(i + 3)`
  - upon execution of the statement, the memory location referred to by myvar will contain the value of the expression `(i + 3) * 2`

Static Semantics

The study of programming language semantics often distinguishes two kinds of semantics:

- *static semantics*
  - those parts of the meaning of program elements that can be determined without executing the program (from the written program alone)
    - type checking
    - resolving ambiguous variable names
    - etc.
- *dynamic semantics*
  - those parts of the meaning of a program that depend upon its execution
    - evaluating expressions
    - determining loop or program termination
    - etc.

Attribute Grammars

An *attribute grammar* associates some semantic information with every symbol in a grammar. The semantic information is carried in *attributes* and combined according to *semantic rules*.

<expr$_{etype}$> ::= <term$_{ttype}$> [+ <expr$_{etype2}$>]
if(ttype = *int* and etype2 = *int*) then etype = *int* else etype = *real*

<term$_{ttype}$> ::= <number$_{ntype}$> [× <term$_{ttype2}$>]
if(ntype = *int* and ttype2 = *int*) then ttype = *int* else ttype = *real*

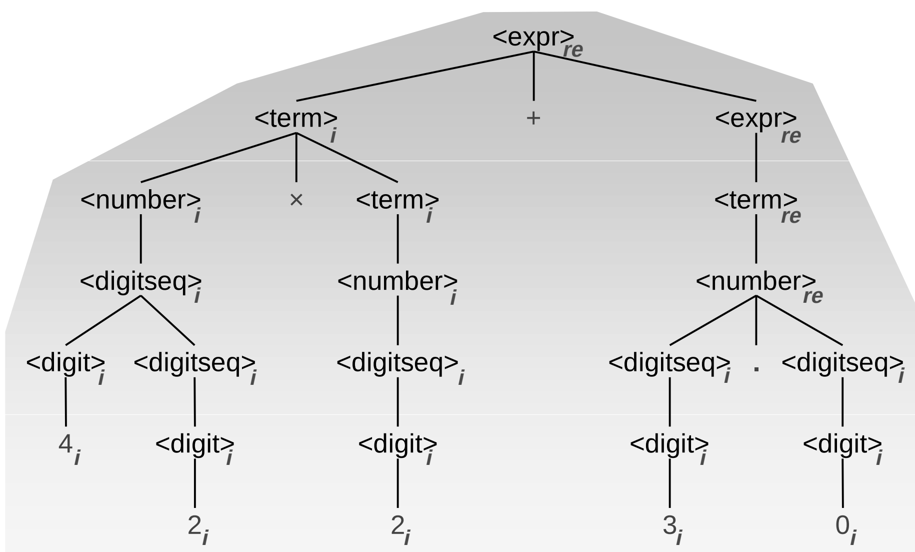
<number$_{ntype}$> ::= <digitseq$_{dtype}$>
ntype = *int*

<number$_{ntype}$> ::= <digitseq$_{dtype1}$> **.** <digitseq$_{dtype2}$>
ntype = *real*

<digitseq$_{dstype}$> ::= <digit$_{dtype1}$> [<digitseq$_{dtype2}$>]
dtype = *int*

<digit$_{dtype}$> ::= 0$_{int}$ | 1$_{int}$ | 2$_{int}$ | ...
dtype = *int*

Derivations with Attribute Grammars



Dynamic Semantics

Recall that *dynamic semantics* is concerned with those parts of the meaning of a program that depend upon its execution.

- evaluating expressions
- determining loop or program termination
- determining control flow (which statement comes next)
- resolving (some) references (pointers, subprogram parameters, etc.)
- etc.

Why do these elements of meaning depend on the execution of a program?

81

Operational Semantics

The meaning of some construct in a program is described in terms of its implementation (or the result of executing its implementation).

- usually the operational semantic description of a program element is expressed as the translation of that element into a low-level language (one with obvious semantics)
- for example:

C statement	*Operational semantics*
`for(expr1; expr2; expr3)`	`expr1;`
`{`	`L1: if expr2 = 0 goto L2`
` ...`	`...`
`}`	`expr3;`
`...`	`goto L1`
	`L2: ...`

3.6.1

82

Axiomatic Semantics

The meaning of a statement in a program is defined indirectly as the effect of its execution on the program's variables

- the effect of a statement on a program's variables is shown through assertions about those variables *before* and *after* statement execution (*preconditions* and *postconditions*)
- for example:

{ }
```
unsigned i;
```
{ i ≥ 0 }
```
i = i + x;
```
{ i ≥ x }
```
while(i > x)
   i = i / 2;
```
{ i ≤ x }

3.6.2

Denotational Semantics

The meaning of a language element is described by assigning a mathematical object to the element and defining functions to determine the object's value.

- for example, consider a grammar for integers:

  <num> ::= [<num>] <digit>

  <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

  the denotational semantic representation would be:

  *sem(*0*) = 0 sem(*1*) = 1 sem(*2*) = 2 sem(*3*) = 3...*

  sem(<digit>*) = sem(*0*) or sem(*1*) or sem(*2*)...*

  sem(<num> <digit>*) = 10×sem(*<num>*) + sem(*<digit>*)*

3.6.3

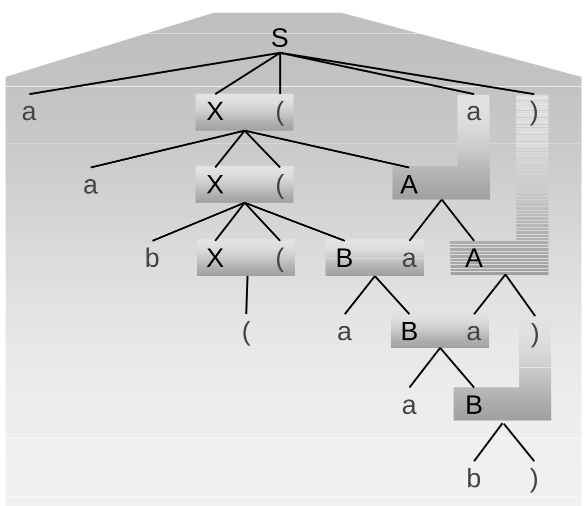
Context-Sensitve Grammars

The grammars seen so far have all denoted context-free syntax: the choice of production rule in a derivation is independent of context in which the symbols in the rule appear. The following grammar is context-sensitive:

```
S    ::=  aX(a)  |  bX(b)
X(   ::=  aX(A  |  bX(B  |  (
Aa   ::=  aA
Ab   ::=  bA
Ba   ::=  aB
Bb   ::=  bB
A)   ::=  a)
B)   ::-  b)
```

> *for convenience,*
> *nonterminals: { S, X, A, B }*
> *terminals: { a, b, (,) }*

A Derivation Using a Context-Sensitive Grammar



aab(aab)

```
S    ::=  aX(a)  |  bX(b)
X(   ::=  aX(A  |  bX(B  |  (
Aa   ::=  aA
Ab   ::=  bA
Ba   ::=  aB
Bb   ::=  bB
A)   ::=  a)
B)   ::-  b)
```