3. Syntax

3. Syntax

What is a Language?

Concrete Syntax

Abstract Syntax

List Structures in Syntax

What is a Language?

Language: A system of communication in a structured way

Natural language

- *used for arbitrary communication*
- *complex, nuanced, and imprecise*

English, Chinese, Hindi, Spanish, …

Programming language

- *used to describe computation*
- *programs have **structure** and **meaning***

Elm, Java, C, Python, SQL, XML, …

Object vs. Metalanguage

Important to distinguish two ***kinds of languages***:

- ***Object language***: the language we're defining
- ***Metalanguage***: the language we're using to define the structure and meaning of the object language!

A single language can fill both roles at different times!
Examples: English, Elm

Syntax vs. Semantics & Metalanguages

Two main ***aspects of a language***:

- ***Syntax***: the structure of its programs
- ***Semantics***: the meaning of its programs

Scope of Metalanguages

	Syntax	*Denotational Semantics*	*Operational Semantics*	*Type Systems*
Regular Expressions	●			
Grammars	●			
Elm	●	●		●
Inference Rules	●	●	●	●

3. Syntax

Grammar Metalanguage

Grammar Concepts

- ***Grammar:*** *Set of* ***productions*** *(or* ***rules****)*
- ***Production:*** L ::= R *where*
  - *L: nonterminal symbol (in a context-free grammar)*
  - *R: sequence of terminal & nonterminal symbols*
- ***Derivation:*** *Sequence of substitutions ("L by R")*
- ***Sentence:*** *Sequence of terminals derivable from nonterminal*
- ***Language:*** *Set of derivable sentences*

Context-Free Grammars

Formal Grammar Definition

*A **grammar** is a four-tuple* (N, Σ, P, S) *where:*

- N *is a set of **nonterminal symbols***
- Σ *is a set of **terminal symbols** with* $N \cap \Sigma = \varnothing$
- $P \subseteq N \times (N \cup \Sigma)^*$ *is a set of **productions***
- $S \in N$ *is the **start symbol**.*

Grammar for Binary Numbers

$$(\{dig, bin\}, \{\mathtt{0}, \mathtt{1}\}, \{(dig, \mathtt{0}), (dig, \mathtt{1}), (bin, dig), (bin, dig\, bin)\}, bin)$$

Backus-Naur Form (BNF)

Grammar for Binary Numbers

$$(\{\mathit{dig}, \mathit{bin}\}, \{\mathtt{0}, \mathtt{1}\}, \{(\mathit{dig}, \mathtt{0}), (\mathit{dig}, \mathtt{1}), (\mathit{bin}, \mathit{dig}), (\mathit{bin}, \mathit{dig}\ \mathit{bin})\}, \mathit{bin})$$

BNF $\approx$ Only Productions

$$\begin{array}{lcl} \mathit{dig} & ::= & \mathtt{0} \\ \mathit{dig} & ::= & \mathtt{1} \\ \mathit{bin} & ::= & \mathit{dig} \\ \mathit{bin} & ::= & \mathit{dig}\ \mathit{bin} \end{array}$$

Grouping RHSs

$$\begin{array}{lcl} \mathit{dig} & ::= & \mathtt{0} \mid \mathtt{1} \\ \mathit{bin} & ::= & \mathit{dig} \mid \mathit{dig}\ \mathit{bin} \end{array}$$

Example Derivations

Binary numbers

$$
\begin{array}{llll}
\mathit{dig} & ::= & \texttt{0} & \text{(P1)} \\
\mathit{dig} & ::= & \texttt{1} & \text{(P2)} \\
\mathit{bin} & ::= & \mathit{dig} & \text{(P3)} \\
\mathit{bin} & ::= & \mathit{dig}\ \mathit{bin} & \text{(P4)}
\end{array}
$$

$$
\begin{array}{lll}
 & \mathit{bin} & \\
\Rightarrow & \mathit{dig}\ \mathit{bin} & \text{(P4)} \\
\Rightarrow & \mathit{dig}\ \mathit{dig}\ \mathit{bin} & \text{(P4)} \\
\Rightarrow & \mathit{dig}\ \mathit{dig}\ \mathit{dig} & \text{(P3)} \\
\Rightarrow & \texttt{1}\ \mathit{dig}\ \mathit{dig} & \text{(P2)} \\
\Rightarrow & \texttt{1}\ \texttt{0}\ \mathit{dig} & \text{(P1)} \\
\Rightarrow & \texttt{1}\ \texttt{0}\ \texttt{1} & \text{(P2)}
\end{array}
\qquad
\begin{array}{lll}
 & \mathit{bin} & \\
\Rightarrow & \mathit{dig}\ \mathit{bin} & \text{(P4)} \\
\Rightarrow & \mathit{dig}\ \mathit{dig}\ \mathit{bin} & \text{(P4)} \\
\Rightarrow & \mathit{dig}\ \mathit{dig}\ \mathit{dig} & \text{(P3)} \\
\Rightarrow & \mathit{dig}\ \mathit{dig}\ \texttt{1} & \text{(P2)} \\
\Rightarrow & \mathit{dig}\ \texttt{0}\ \texttt{1} & \text{(P1)} \\
\Rightarrow & \texttt{1}\ \texttt{0}\ \texttt{1} & \text{(P2)}
\end{array}
$$

Note: One sentence may have different derivations!

Derivation $\approx$ Trace
(only substitutions, no simplifications)

Grammars Define Languages

Language

Grammar $G = (N, \Sigma, P, S)$ *defines the* ***language*** $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$

Example

$L(\{dig, bin\}, \{\mathtt{0}, \mathtt{1}\}, \{(dig, \mathtt{0}), (dig, \mathtt{0}), \ldots\}, bin) = \{b^k \mid b \in \{\mathtt{0}, \mathtt{1}\} \wedge k > 0\}$

Exercise ...

Question 1

Consider the language L defined by the following grammar.

$$
\begin{aligned}
S &::= A\,B \\
A &::= \mathtt{0}A \mid \mathtt{0} \\
B &::= \mathtt{1}B \mid \mathtt{1}
\end{aligned}
$$

Which of the following statements about are true for words/sentences of L? Each sentence contains ...

(a) ... one or more `1`s

(b) ... one or more `0`s

(c) ... at least as many `1`s as `0`s

(d) ... at least as many `0`s as `1`s

(e) ... exactly as many `1`s as `0`s

(f) ... at least two digits

(g) All `0`s precede all `1`s

a, b, f, g

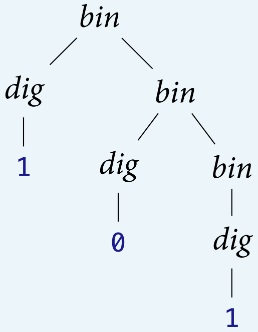
Sentence Structure = Parse Tree

Binary numbers

$$
\begin{array}{lcl}
\mathit{dig} & ::= & \texttt{0} \mid \texttt{1} \\
\mathit{bin} & ::= & \mathit{dig} \mid \mathit{dig}\ \mathit{bin}
\end{array}
$$

Rules can be interpreted as instructions to build trees:

"Add R as children to L"



Internal nodes:
Nonterminals

Leaves:
Terminals

Parse tree ignores the order of rule applications
$\Longrightarrow$ *Appropriate representation of sentence structure*

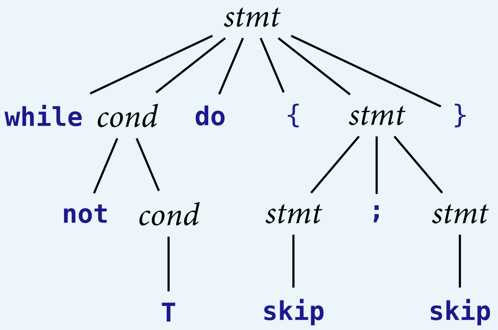
Exercise ...

Concrete Syntax

$$
\begin{array}{rcl}
\mathit{cond} & ::= & \mathtt{T} \\
 & \mid & \mathtt{not}\ \mathit{cond} \\
 & \mid & (\ \mathit{cond}\ ) \\
\mathit{stmt} & ::= & \mathtt{skip} \\
 & \mid & \mathtt{while}\ \mathit{cond}\ \mathtt{do}\ \{\ \mathit{stmt}\ \} \\
 & \mid & \mathit{stmt}\ \mathtt{;}\ \mathit{stmt}
\end{array}
$$

Question 2

Create a parse tree for the following sentence: **`while not T do {skip ; skip}`**



Ambiguity

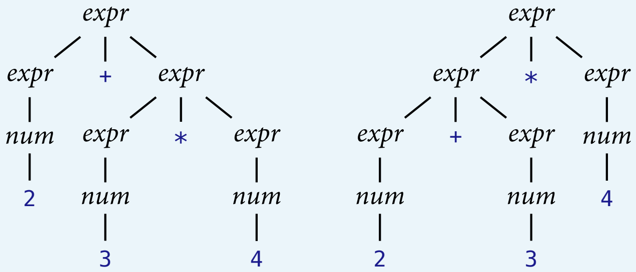
Ambiguous Grammar

Some sentences have more than one parse tree.

Ambigous Grammar

$$\begin{aligned} num &::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \\ expr &::= num \\ &\mid \quad expr + expr \\ &\mid \quad expr * expr \end{aligned}$$

Parse tree for `2 + 3 * 4`



In a Nutshell ...

What is a Grammar?

*A formalism to define linear representations for **typed** tree data structures*

Each nonterminal ($\approx$ type) denotes a set of trees with a specific structure.

Parser: *Algorithm to recover tree structures from strings*

Pretty Printer: *Algorithm to turn tree structures into strings*

Why Grammar Matters

Video Clip

WARNING: This video contains R-rated language!

3. Syntax

What is a Language?

Concrete Syntax

Abstract Syntax

List Structures in Syntax

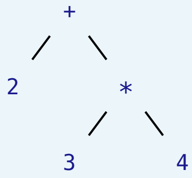
Programs are Trees!

Abstract Syntax Tree (AST)

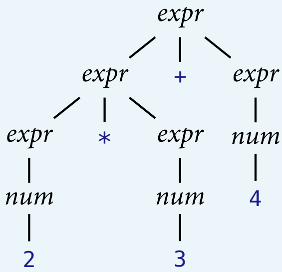
The essential structure of a program
(everything needed to determine its semantics)

Internal nodes: **Operations**

Leaves: **Values**



AST for `2 + 3 * 4`



Parse tree for `2 + 3 * 4`

More AST Examples

Abstract Syntax Tree (AST)

The essential structure of a program
(everything needed to determine its semantics)

*Internal nodes: **Operations***

*Leaves: **Values***

```
    +
   / \
  2   *
     / \
    3   4

2 + 3 * 4
```

```
        *
      /   \
    +       +
   / \     / \
  5   6   7   8

(5 + 6) * (7 + 8)
```

```
         if
      /  |   \
  true   +    5
        / \
       2   3

if true then 2+3 else 5
```

Concrete vs. Abstract Syntax

	Concrete Syntax	Abstract Syntax
Program	Sequence of words	Tree of constructors & values
Language	Set of sentences	Set of ASTs
Description	Context-free grammar	*Elm type definitions*
Structure	Parse tree	AST = *Value of an Elm type*
Ambiguity	can be a problem	ruled out by formalism
Keywords	yes	no
Parentheses	yes	no[1]
Convenience		smaller in size

[1] Parentheses of the metalanguage (i.e., Elm) may be used

Abstract Syntax via Data Types

Concrete Syntax

$$
\begin{array}{lcl}
\textit{term} & ::= & \mathbf{true} \\
 & \mid & \mathbf{false} \\
 & \mid & \mathbf{not}\ \textit{term} \\
 & \mid & \mathbf{if}\ \textit{term}\ \mathbf{then}\ \textit{term}\ \mathbf{else}\ \textit{term} \\
 & \mid & (\ \textit{term}\ )
\end{array}
$$

Abstract Syntax

```
type Term = Tru
          | Fls
          | Not Term
          | If Term Term Term
```

Translation

Nonterminal $\longrightarrow$ Type
Terminal $\longrightarrow$ Value / Constructor
Production RHS $\longrightarrow$ Constructor (with arguments)
Nonterminals on RHS $\longrightarrow$ Type arguments for constructor

Example

term $\rightarrow$ `Term`
true $\rightarrow$ `Tru`
not *term* $\rightarrow$ `Not Term`
term $\rightarrow$ `Term`

Exercise ...

Concrete Syntax

$$
\begin{array}{lcl}
\mathit{num} & ::= & \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \ldots \\
\mathit{expr} & ::= & \mathit{num} \\
 & \mid & \mathtt{-}\mathit{expr} \\
 & \mid & \mathit{expr} \mathbin{\mathtt{+}} \mathit{expr} \\
 & \mid & \mathit{expr} \mathbin{\mathtt{*}} \mathit{expr} \\
 & \mid & \mathtt{(}\ \mathit{expr}\ \mathtt{)}
\end{array}
$$

Question 3

Define a data type `Expr` *to represent the abstract syntax.*

Solution to Exercise

Concrete Syntax

$$
\begin{array}{lcl}
num & ::= & \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \ldots \\
expr & ::= & num \\
 & \mid & -expr \\
 & \mid & expr + expr \\
 & \mid & expr * expr \\
 & \mid & (\ expr\ )
\end{array}
$$

Abstract Syntax

```
type Expr = Num Int
          | Neg Expr
          | Plus  Expr Expr
          | Times Expr Expr
```

Note: *The* `Num` *constructor is needed to embed* `Int` *into* `Expr`*.*

Program = Tree = Data Type Value

Linear/textual form

```
2 + 3 * 4
```

Elm values

```
Plus (Num 2) (Times (Num 3) (Num 4))
Times (Plus (Num 2) (Num 3)) (Num 4)
```

Informal AST notation

```
      +
     / \
    2   *
       / \
      3   4
```

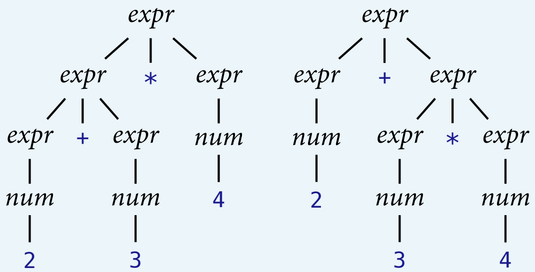
Elm ASTs

```
      Plus                       Times
      /  \                       /  \
   Num   Times               Plus    Num
    |    /  \                /  \     |
    2  Num   Num           Num   Num  4
        |     |             |     |
        3     4             2     3
```

Abstract Syntax Trees vs. Parse Trees

Ambigous Grammar

$$num ::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots$$
$$expr ::= num \mid expr + expr \mid expr * expr$$

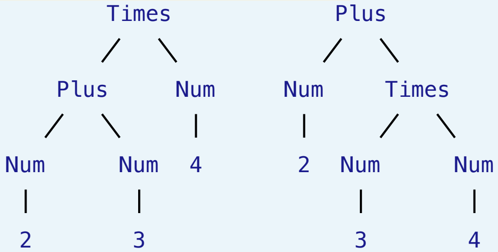


2 parse trees for

`2 + 3 * 4`

Abstract Syntax

```
type Expr = Num Int
          | Plus  Expr Expr
          | Times Expr Expr
```



Forced Unique ASTs for

`2 + 3 * 4`

Grammars into Data Types

Grammar		**_Data Type_**	
Basic nonterminal	*num*, *name*, ...	Predefined type	`Int`, `String`, ...
Non-basic nonterminal	*expr*, *term*, ...	Data type	`Expr`, `Term`, ...
Terminal for operation	**+**, **if**, ...	Constructor	`Plus`, `If`, ...
Grouping/filler terminal	**begin**, (,), **do**, ...	—	—
“Subtype“ nonterminal	*num*	Extra constructor	`Num Int`

Nonterminals $\approx$ ***Types***

Terminals $\approx$ ***Values / Operations***

Exercise ...

Concrete Syntax

$$
\begin{array}{rcl}
\mathit{cond} & ::= & \mathtt{T} \\
 & | & \mathtt{not}\ \mathit{cond} \\
 & | & (\ \mathit{cond}\ ) \\
\mathit{stmt} & ::= & \mathtt{skip} \\
 & | & \mathtt{while}\ \mathit{cond}\ \mathtt{do}\ \{\ \mathit{stmt}\ \} \\
 & | & \mathit{stmt}\ \mathtt{;}\ \mathit{stmt}
\end{array}
$$

Question 4

Define the data types `Cond` *and* `Stmt` *to represent the abstract syntax.*

Solution to Exercise

Concrete Syntax

$$
\begin{array}{rcl}
\mathit{cond} & ::= & \mathtt{T} \\
& | & \mathtt{not}\ \mathit{cond} \\
& | & (\ \mathit{cond}\ ) \\
\mathit{stmt} & ::= & \mathtt{skip} \\
& | & \mathtt{while}\ \mathit{cond}\ \mathtt{do}\ \{\ \mathit{stmt}\ \} \\
& | & \mathit{stmt}\ \mathtt{;}\ \mathit{stmt}
\end{array}
$$

Abstract Syntax

```
type Cond = T | Not Cond

type Stmt = Skip
          | While Cond Stmt
          | Seq Stmt Stmt
```

Exercise ...

Question 5

Translate the following program into abstract syntax, i.e., write it as a value of type `Stmt`*.*

```
while not(not(T)) {
  while T {
    skip; skip
  }
}
```

Abstract Syntax

```
type Cond = T | Not Cond

type Stmt = Skip
          | While Cond Stmt
          | Seq Stmt Stmt
```

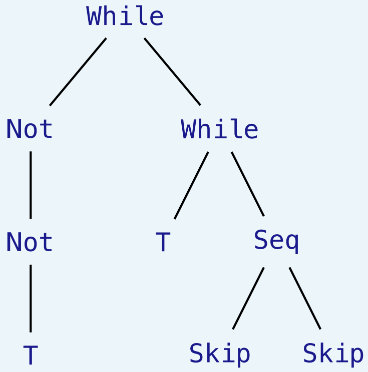
```
While (Not (Not T))
      (While T
             (Seq Skip Skip))
```

Exercise ...

Question 6

Draw the abstract syntax tree for the following program.

```
while not(not(T)) {
  while T {
    skip; skip
  }
}
```

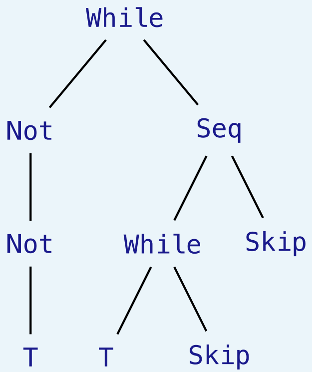


Exercise …

Question 7

Draw the abstract syntax tree for the following program.

```
while not(not(T)) {
  while T {skip}; skip
}
```



3. Syntax

What is a Language?

Concrete Syntax

Abstract Syntax

List Structures in Syntax

Syntactic Repetition via Recursion

Concrete Syntax

$$
\begin{array}{rcl}
n \in \mathit{name} & ::= & \text{(strings)} \\
\mathit{fundef} & ::= & n\ \mathit{pars} = \mathit{expr} \\
\mathit{pars} & ::= & n\ \mathit{pars} \mid \epsilon
\end{array}
$$

Abstract Syntax

```
type alias Name = String
type FunDef = Fun Name Pars Expr
type Pars = P Name Pars | Empty
```

ϵ-rule allows removal of pars nonterminal

```
f m x y = m*x + y
```

```
Fun "f" (P "m" (P "x" (P "y" Empty))) (Plus (Times ...) ...)
```

Syntactic Repetition via Lists

Concrete Syntax

$$
\begin{aligned}
n \in \mathit{name} \quad &::= \quad \text{(strings)} \\
\mathit{fundef} \quad &::= \quad n\ n^{*} = \mathit{expr}
\end{aligned}
$$

Abstract Syntax

```
type alias Name = String
type FunDef = Fun Name (List Name) Expr
```

```
f m x y = m*x + y
```

```
Fun "f" ["m","x","y"] (Plus (Times ...) ...)
```

Representing Non-Zero Repetitions (A)

Concrete Syntax

$$
\begin{aligned}
n \in \mathit{name} \quad &::= \quad \text{(strings)} \\
\mathit{fundef} \quad &::= \quad n \ \mathit{pars} = \mathit{expr} \\
\mathit{pars} \quad &::= \quad n \ \mathit{pars} \mid n
\end{aligned}
$$

Abstract Syntax (A)

```
type alias Name = String
type FunDef = Fun Name Pars Expr
type Pars = P Name Pars | N Name
```

```
f m x y = m*x + y
```

```
Fun "f" (P "m" (P "x" (N "y"))) (Plus (Times ...) ...)
```

Representing Non-Zero Repetitions (B)

Concrete Syntax

$$
\begin{array}{rcl}
n \in \mathit{name} & ::= & \text{(strings)} \\
\mathit{fundef} & ::= & n \ \mathit{pars} = \mathit{expr} \\
\mathit{pars} & ::= & n \ \mathit{pars} \mid n
\end{array}
$$

Abstract Syntax (B)

```
type alias Name = String
type alias FunDef = (Name,Pars,Expr)
type alias Pars = (Name,List Name) -- precise
type alias Pars = List Name        -- also ok
```

```
f m x y = m*x + y
```

```
("f",["m","x","y"],Plus (Times ...))
```

Discussion …

The following type definition represents the abstract syntax of a specific form of integer lists.

```
type S = A Int
       | B Int Int S
```

Which type corresponds most closely to S?

(One) `List Int`
(Two) `List (Int,Int)`
(Three) `(Int,List Int)`
(Four) `(List Int,List Int)`
(Five) `(Int,List (Int,Int))` ✓