

# Some Principles of Induction

Mooly Sagiv  
msagiv@acm.org  
Tuesday 11-13, Schriber 317  
TA: Oded Padon  
Email: odedp@mail.tau.ac.il

<http://www.cs.tau.ac.il/~msagiv/courses/pl18.html>

# Goals

- Mathematical Induction
- Inductively Defined Sets
- Derivation Trees
- Structural Induction

# Syntax vs. Semantics

- The pattern of formation of sentences or phrases in a language
- Examples
  - Regular expressions
  - Context free grammars
- The study or science of meaning in language
- Examples
  - Interpreter
  - Compiler
  - Better mechanisms will be given in the course

# Induction in Programming Languages

- The syntax of programming languages is inductively defined
  - A number is **expression**
  - If  $e_1$  and  $e_2$  are **expressions** so is  $e_1 + e_2$
- Types are inductively defined
  - int is a **type**
  - if  $t_1, t_2, \dots, t_k$  are **types** then  
struct  $\{ t_1 i_1; t_2 i_2; \dots, t_k i_k; \}$  is a **type** where  $i_1, i_2, \dots, i_k$  are identifiers
- Recursive functions
  - $\text{fac}(n) = \text{if } n = 1 \text{ then } 1 \text{ else } n * \text{fac}(n-1)$   
 $\text{fac}_1=1, \text{fac}_n=n*\text{fac}_{n-1}$
- Semantics is inductively defined

# Mathematical Induction

- $P(n)$  is a property of natural number  $n$
- To show that  $P(n)$  holds for every  $n$ , it suffices to show that:
  - $P(0)$  is true
  - If  $P(m)$  is true then  $P(m+1)$  is true for every number  $m$
- In logic
  - $(P(0) \wedge \forall m \in \mathbb{N}. P(m) \Rightarrow P(m+1)) \Rightarrow \forall n \in \mathbb{N}. P(n)$



# Simple Example

- Show that  $0 + 1 + 2 + \dots + n = n(n + 1)/2$

# Fibonacci Numbers

- A sequence
  - $\text{fib}_0=1$
  - $\text{fib}_1=1$
  - $\text{fib}_n = \text{fib}_{n-2} + \text{fib}_{n-1}$

1 1 2 3 5

# The Golden Ratio

- $\text{fib}(n) = (a^n - b^n)/(a-b)$  where  $a=(1+ \sqrt{5})/2$  and  $b=(1- \sqrt{5})/2$
- How can this be proved?





# Course of values Induction

- $P(n)$  is a property of natural number  $n$
- To show that  $P(n)$  holds for every  $n$ , it suffices to show that for all  $m$  if for all  $k < m$ ,  $P(k)$  holds then  $P(m)$
- In logic
  - $\forall m \in \mathbb{N}. (\forall k < m. P(k)) \Rightarrow P(m) \Rightarrow \forall n \in \mathbb{N}. P(n)$





# The Induction Hypothesis

- The art of inductive proofs is coming up with induction hypothesis
- Given a property  $Q$  find a stronger property  $P$  which is inductive  $P \Rightarrow Q$  or  $P \subseteq Q$

# Induction on a ball game

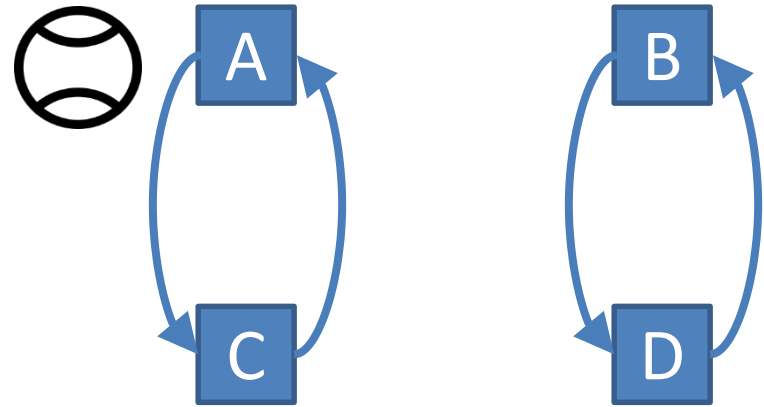
- Four players pass a ball:

- A will pass to C

- B will pas to D

- C will pass to A

- D will pass to B



- The ball starts at player A
- Can the ball get to D?

# Induction on a ball game

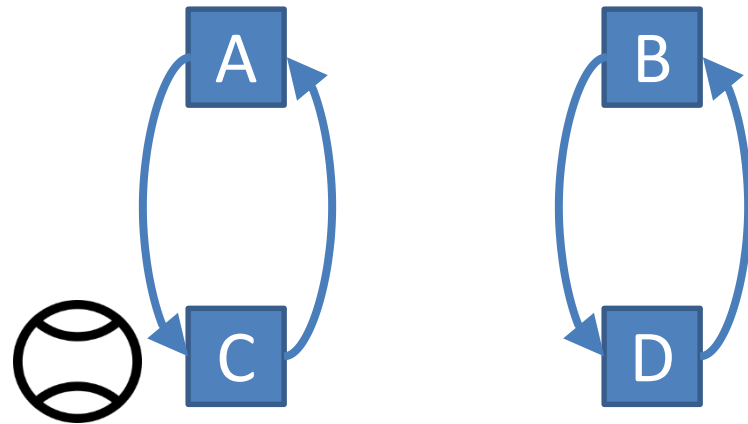
- Four players pass a ball:

- A will pass to C

- B will pas to D

- C will pass to A

- D will pass to B

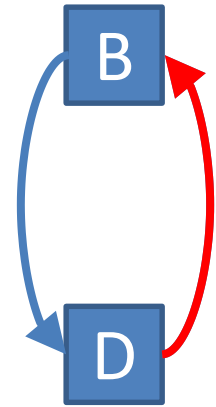


- The ball starts at player A
- Can the ball get to D?

# Formalizing with induction

- $x_0 = A$

- $x_{n+1} = \begin{cases} C & \text{if } x_n = A \\ D & \text{if } x_n = B \\ A & \text{if } x_n = C \\ B & \text{if } x_n = D \end{cases}$



- Prove by induction  $\forall n. x_n \neq D$

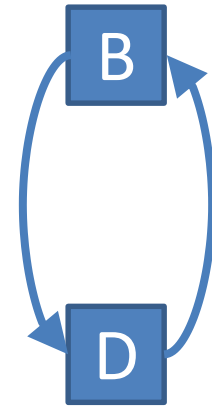
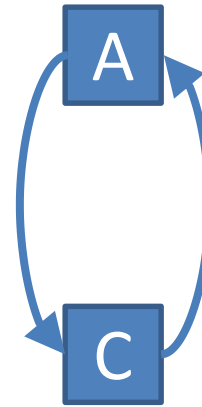
- $x_0 \neq D$  ?

- $x_m \neq D \Rightarrow x_{m+1} \neq D$  ?

# Formalizing with induction

- $x_0 = A$

- $x_{n+1} = \begin{cases} C & \text{if } x_n = A \\ D & \text{if } x_n = B \\ A & \text{if } x_n = C \\ B & \text{if } x_n = D \end{cases}$



- Prove a stronger claim by induction  $\forall n. x_n \neq B \wedge x_n \neq D$ 
  - $x_0 \neq B \wedge x_0 \neq D$
  - $x_m \neq B \wedge x_m \neq D \Rightarrow x_{m+1} \neq B \wedge x_{m+1} \neq D$

# Simple Program

- Consider the sequence
  - $x_0 = 0, y_0 = 0$
  - $x_n = x_{n-1} + y_{n-1}, y_n = y_{n-1} + 2$
- How can we show that for all  $n$ :  $x_n$  is even
- What is the inductive claim?

```
{  
x = 0, y = 0;  
while (1) {  
    assert x%2 == 0 ;  
    x = x + y;  
    y = y + 2;  
}  
}
```

# Inductively Defined Sets



# Inductively Defined Sets

- It is convenient to define sets by rules
  - Base elements
  - Inference rules for more elements
  - Define the **minimal** set that is closed under rule application

# The Natural Numbers

zero-axiom

$$0 \in \mathbb{N}$$

succ-rule

$$\begin{array}{l} m \in \mathbb{N} \\ \text{succ}(m) \in \mathbb{N} \end{array}$$

# The Even Numbers

zero-even

---

$$0 \in \text{Even}$$

even-rule

---

$$m \in \text{Even}$$
$$\text{succ}(\text{succ}(m)) \in \text{Even}$$

How can we show that  $6 \in \text{Even}$

How can we show that  $3 \notin \text{Even}$

# Driven (Proof) Tree

- A proof that element is inductive set
- The leafs are axioms
- Internal nodes are inference rules

# A Proof Tree for $6 \in \text{Even}$

zero-even  $\frac{}{0 \in \text{Even}}$

even-rule  $\frac{}{\text{succ}(\text{succ}(0)) \in \text{Even}}$

even-rule  $\frac{}{\text{succ}(\text{succ}(\text{succ}(\text{succ}(0)))) \in \text{Even}}$

even-rule  $\frac{}{\text{succ}(\text{succ}(\text{succ}(\text{succ}(\text{succ}(\text{succ}(0)))))) \in \text{Even}}$

root



Proving that  $3 \notin \text{Even}$



Why is this relevant to programming languages?

# Expression Definition

- A number is an **expression**
- If **e** is an **expression** then so is **(e)**
- If **e1** and **e2** are expressions then **e1+e2** is an expression
- This is an inductively defined set



# Rules for expressions

exp-num-axiom

---

number  $\in$  Exp

exp-paren

---

$e \in$  Exp  
 $(e) \in$  Exp

exp-plus

---

$e_1 \in$  Exp    $e_2 \in$  Exp  
 $e_1 + e_2 \in$  Exp

# Example 5 + (7+3)

exp-num-axiom  $\frac{}{7 \in \text{Exp}}$       exp-num-axiom  $\frac{}{3 \in \text{Exp}}$

exp-plus  $\frac{}{7+3 \in \text{Exp}}$

exp-num-axiom  $\frac{}{5 \in \text{Exp}}$

exp-paren  $\frac{}{(7+3) \in \text{Exp}}$

exp-plus  $\frac{}{5+(7+3) \in \text{Exp}}$

# Benefits of formal definitions

- Intellectual
- Better understanding
- Formal proofs
- Mechanical checks by computer
- Tool generation
  - Consistency
  - Entailment
  - Query evaluation

# What is a good formal definition?

- Natural
- Concise
- Easy to understand
- Permits effective mechanical reasoning

# Benefits of formal syntax for programming language

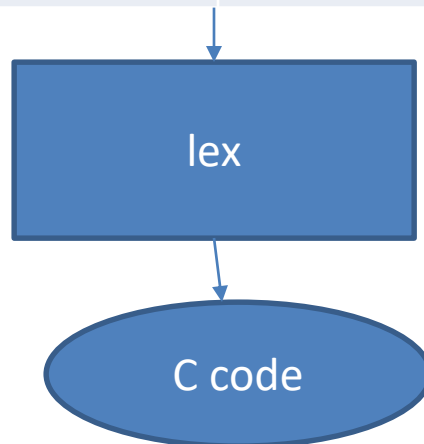
- Intellectual
- Simplicity
- Better understanding
  - Interaction between different parts
- Abstraction
  - Portability
- Tool generations
  - Parser

How can the syntax of programming language be formally defined?

# Tokens

- Basic units of the programming language
- Usually defined by regular expressions
- Good tools: lex, awk

Regexp	action
[0-9]+	printf("number");
"if"	printf("if");



# Example Tokens

- Keywords
  - “if”
  - “while”
- Identifier {letter}({letter}|{digit}|\_)\*
- Numbers [0-9]+



# Example Tokens

Type	Examples
ID	foo n_14 last
NUM	73 00 517 082
REAL	66.1 .5 10. 1e67 5.5e-10
IF	if
COMMA	,
NOTEQ	!=
LPAREN	(
RPAREN	)

# Example Non Tokens

Type	Examples
comment	<code>/* ignored */</code>
preprocessor directive	<code>#include &lt;foo.h&gt;</code>
	<code>#define NUMS 5, 6</code>
macro	<code>NUMS</code>
whitespace	<code>\t \n \b</code>

# Recursive Syntax Definitions

- The syntax of programming languages is naturally defined recursively
- Can be also defined using inductive definitions
- Valid program are represented as syntax trees

# Expression Definitions

- Every **identifier** is an **expression**
- If E1 and E2 are **expressions** and **op** is a binary operation then so is '**E<sub>1</sub> op E<sub>2</sub>**' is an **expression**

$\langle E \rangle \rightarrow id \mid \langle E \rangle \langle op \rangle \langle E \rangle$   
 $\langle op \rangle \rightarrow + \mid - \mid * \mid /$

plus-ax  $\frac{}{+ \in Op}$

minus-ax  $\frac{}{- \in Op}$

mul-ax  $\frac{}{* \in Op}$

div-ax  $\frac{}{/ \in Op}$

id-E  $\frac{id \in ID}{id \in E}$

bin-E  $\frac{e_1 \in E \ e_2 \in E \ op \in Op}{e_1 \ op \ e_2 \in E}$

# Statement Definitions

- If **id** is a **identifier** and **E** is an expression then '**id := E**' is a **statement**
- If **S<sub>1</sub>** and **S<sub>2</sub>** are statements and **E** is an expression then
  - '**S<sub>1</sub> ; S<sub>2</sub>**' is a statement
  - '**if (E) S<sub>1</sub> else S<sub>2</sub>**' is a statement

```
<S> → id := <E>  
<S> → <S> ; <S>  
<S> → if (<E>) <S> else <S>
```

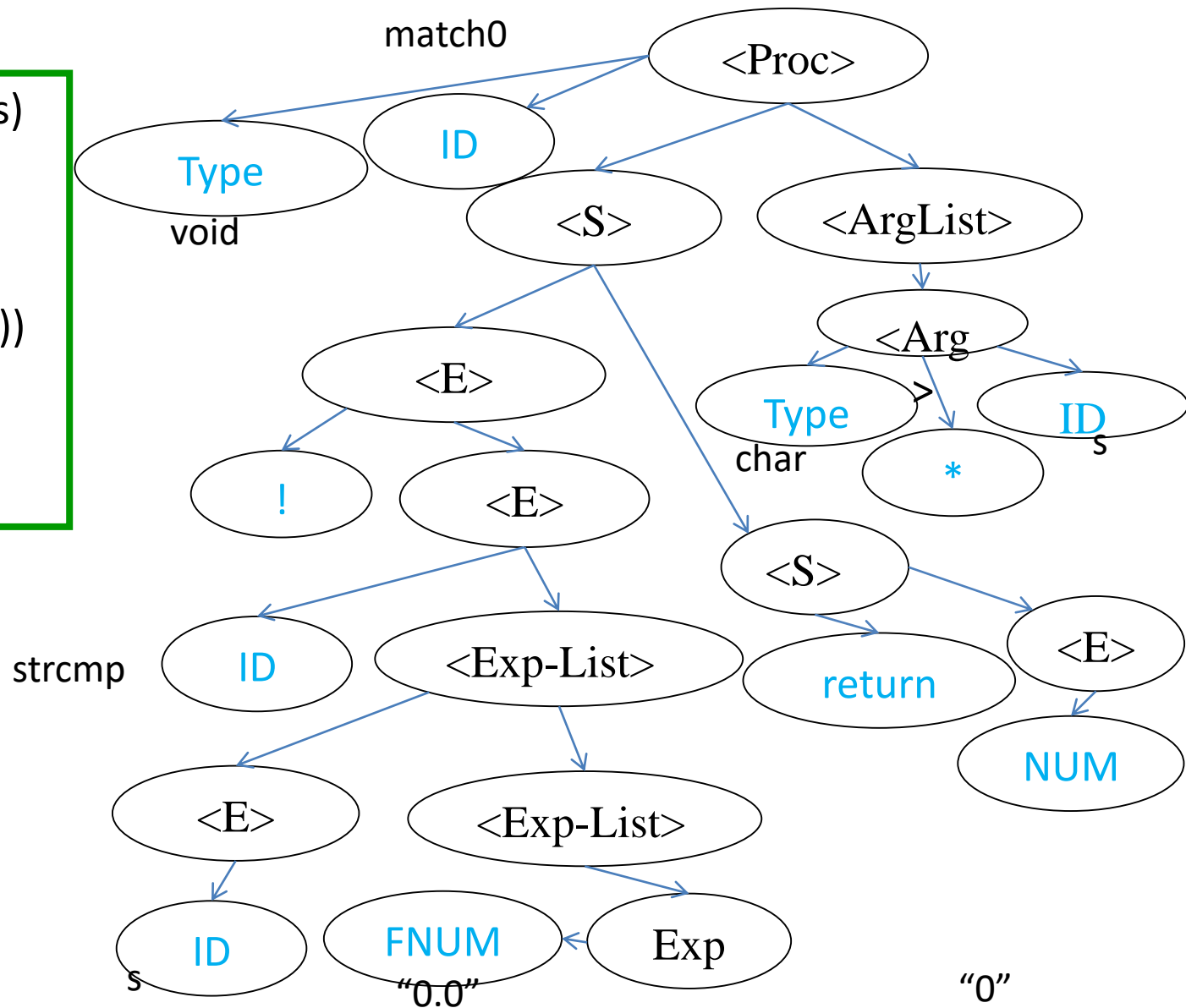
assign-rule  $\frac{id \in Id, e \in E}{id := E \in S}$

seq-s  $\frac{S_1 \in S \quad S_2 \in S}{S_1 ; S_2 \in S}$

conditional-s  $\frac{e \in E \quad S_1 \in S \quad S_2 \in S}{if (E) S_1 else S_2 \in S}$

# C Example

```
void match0(char *s)
/* find a zero */
{
  if (!strcmp(s, "0.0"))
    return 0 ;
}
```



# Context Free Grammars

- Non-terminals
  - Start non-terminal
- Terminals (tokens)
- Context Free Rules  
 $\langle \text{Non-Terminal} \rangle \rightarrow \text{Symbol Symbol} \dots \text{Symbol}$

# Example Context Free Grammar

- 1  $\langle S \rangle \rightarrow \langle S \rangle ; \langle S \rangle$
- 2  $\langle S \rangle \rightarrow \text{id} := \langle E \rangle$
- 3  $\langle S \rangle \rightarrow \text{print} (\langle L \rangle)$
- 4  $\langle E \rangle \rightarrow \text{id}$
- 5  $\langle E \rangle \rightarrow \text{num}$
- 6  $\langle E \rangle \rightarrow \langle E \rangle + \langle E \rangle$
- 7  $\langle E \rangle \rightarrow (\langle S \rangle, \langle E \rangle)$
- 8  $\langle L \rangle \rightarrow \langle E \rangle$
- 9  $\langle L \rangle \rightarrow \langle L \rangle, \langle E \rangle$



# Derivations

- Show that a sentence is in the grammar (valid program)
  - Start with the start symbol
  - Repeatedly replace one of the non-terminals by a right-hand side of a production
  - Stop when the sentence contains terminals only
- Rightmost derivation
- Leftmost derivation

# Parse Trees

- The trace of a derivation
- Every internal node is labeled by a non-terminal
- Each symbol is connected to the deriving non-terminal

# Example Parse Tree

<<S>>

<S> ; <S>

<S> ; id := E

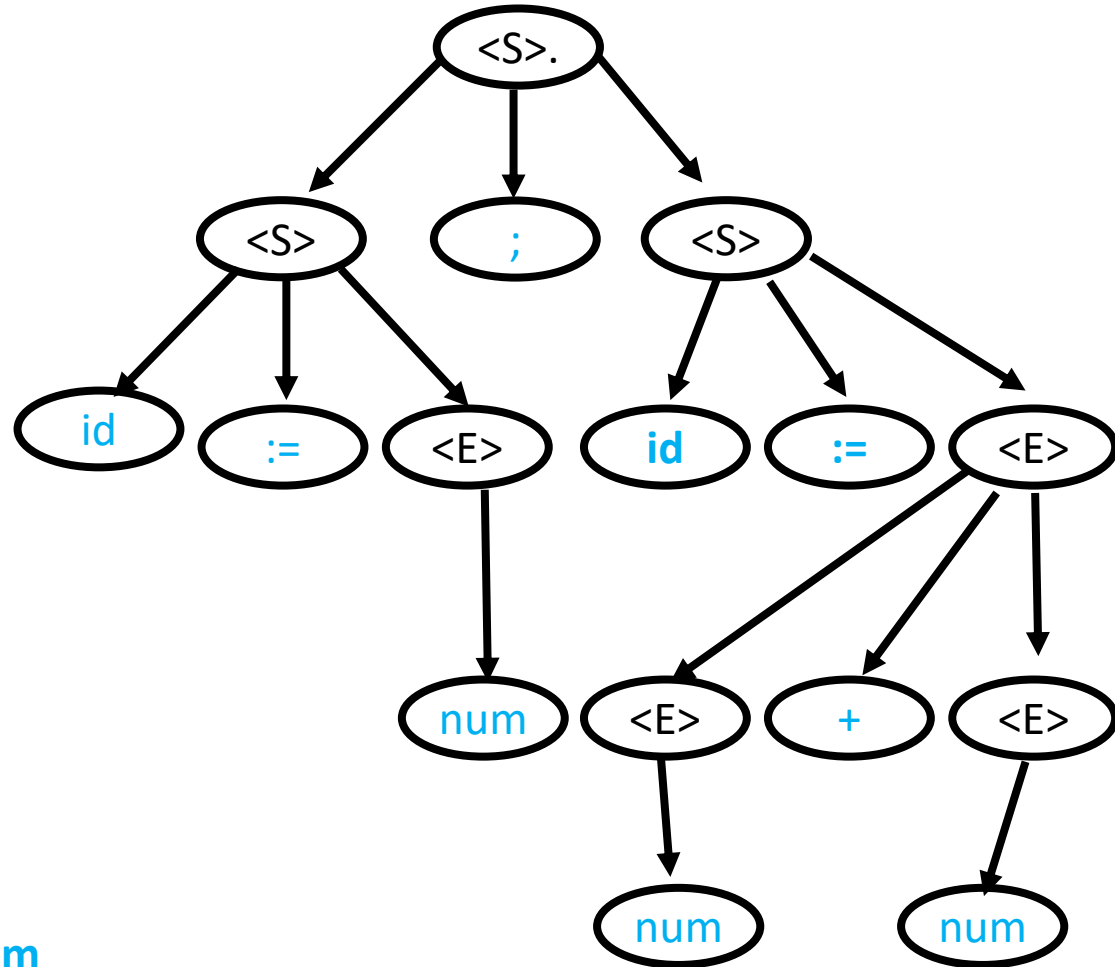
id := <E> ; id := <E>

id := num ; id := <E>

id := num ; id := <E> + <E>

id := num ; id := <E> + num

id := num ; id := num + num



# Ambiguous Grammars

- Two leftmost derivations
- Two rightmost derivations
- Two parse trees

# A Grammar for Arithmetic Expressions

1  $\langle E \rangle \rightarrow \langle E \rangle + \langle E \rangle$

2  $\langle E \rangle \rightarrow \langle E \rangle * \langle E \rangle$

3  $\langle E \rangle \rightarrow \text{id}$

4  $\langle E \rangle \rightarrow (\langle E \rangle)$

# Drawbacks of Ambiguous Grammars

- Ambiguous semantics
- Parsing complexity
- But how can we express the syntax of PL using non-ambiguous grammars?

# Non Ambiguous Grammar for Arithmetic Expressions

Ambiguous grammar

- 1  $\langle E \rangle \rightarrow \langle E \rangle + \langle E \rangle$
- 2  $\langle E \rangle \rightarrow \langle E \rangle * \langle E \rangle$
- 3  $\langle E \rangle \rightarrow \text{id}$
- 4  $\langle E \rangle \rightarrow (\langle E \rangle)$

- 1  $\langle E \rangle \rightarrow \langle E \rangle + \langle T \rangle$
- 2  $\langle E \rangle \rightarrow \langle T \rangle$
- 3  $\langle T \rangle \rightarrow \langle T \rangle * \langle F \rangle$
- 4  $\langle T \rangle \rightarrow \langle F \rangle$
- 5  $\langle F \rangle \rightarrow \text{id}$
- 6  $\langle F \rangle \rightarrow (\langle E \rangle)$

How do we show that the above grammars are the same?

# Non Ambiguous Grammars for Arithmetic Expressions

Ambiguous grammar

1  $\langle E \rangle \rightarrow \langle E \rangle + \langle E \rangle$   
2  $\langle E \rangle \rightarrow \langle E \rangle * \langle E \rangle$   
3  $\langle E \rangle \rightarrow \text{id}$   
4  $\langle E \rangle \rightarrow (\langle E \rangle)$

1  $\langle E \rangle \rightarrow \langle E \rangle + \langle T \rangle$   
2  $\langle E \rangle \rightarrow \langle T \rangle$   
3  $T \rightarrow \langle T \rangle * \langle F \rangle$   
4  $T \rightarrow \langle F \rangle$   
5  $F \rightarrow \text{id}$   
6  $F \rightarrow (\langle E \rangle)$

1  $\langle E \rangle \rightarrow \langle E \rangle * \langle T \rangle$   
2  $\langle E \rangle \rightarrow \langle T \rangle$   
3  $\langle T \rangle \rightarrow \langle F \rangle + \langle T \rangle$   
4  $\langle T \rangle \rightarrow \langle F \rangle$   
5  $\langle F \rangle \rightarrow \text{id}$   
6  $\langle F \rangle \rightarrow (\langle E \rangle)$



# Language Equivalence

- What does it mean for two grammars to be equivalent?
- How do we show that two grammars are equivalent?

# Structural Induction

- Prove that the property holds for all simple trees
- Prove that the property holds for all composite trees using the fact that it holds for subtrees
  - For each rule assume that the property holds for its premises (induction hypothesis) and prove it holds for the conclusion of the rule

# Rule Induction

- Prove that the property holds for all simple derivation trees by showing it holds for axioms
- Prove that the property holds for all composite trees:
  - For each rule assume that the property holds for its premises (induction hypothesis) and prove it holds for the conclusion of the rule

# Inductive Rules for Interpretation

Operational Semantics

# Expression Evaluation

- The rules for expression evaluation can also be defined inductively
- $Val \subseteq Exp \times Int$ 
  - Denote  $e, k \in Val$  by  $e \rightarrow k$
- Leads to simple evaluation rules
- Easy to implement

# Rules for expression evaluation

vexp-num-ax

---

number  $\rightarrow$  number

vexp-paren

---

$e \rightarrow v$   
 $(e) \rightarrow v$

vexp-plus

---

$e_1 \rightarrow v_1$                        $e_2 \rightarrow v_2$   
 $e_1 + e_2 \rightarrow v_1 + v_2$

# Example 2+3

vexp-num-ax



$2 \rightarrow 2$

vexp-num-ax



$3 \rightarrow 3$

vexp-plus



$2+3 \rightarrow 5$

# Theorem: Values are deterministic

- For every expression  $e \in \text{Exp}$  and values  $v_1, v_2 \in \text{Int}$ 
  - If  $e \rightarrow v_1$  and  $e \rightarrow v_2$  then  $v_1 = v_2$



# Well Founded Induction

- Inductive arguments work since they cannot go forever
- Eventually we reach the basis

# Well Founded Relations

- A binary relation  $\prec$  on a set  $A$  is **well founded** if there are no infinite descending chains

$$\dots \prec a_i \prec \dots \prec a_1 \prec a_0$$

- Examples

- $A = \mathbb{N}$  and  $x \prec y$  if  $y = x + 1$

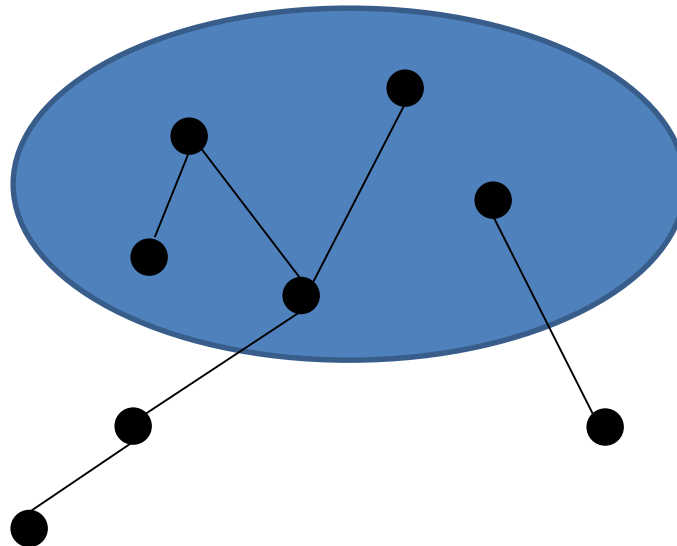
- $A = \mathbb{N}$  and  $x \prec y$  if  $x < y$

- $A$  is set of strings and  $x \prec y$  if  $x$  is a prefix of  $y$ , i.e. there exists  $w$  such that  $x.w = y$

- $A$  is set of strings and  $x \prec y$  if  $y$  is a prefix of  $x$ , i.e. there exists  $w$  such that  $y.w = x$

# Minimal Elements

- A binary relation  $<$  on a set  $A$  is well founded if and only if every non-empty subset  $Q$  of  $A$  has a minimal element  $m$  such that
  - $m \in Q \wedge \forall b. b < m \Rightarrow b \notin Q$



# The Principle of Well-Founded Induction

- Let  $\prec$  be a well-founded relation on a set  $A$  and  $P$  be a property then  
 $\forall a. P(a)$   
iff  
 $\forall c. [\forall b. (b \prec c \wedge P(b)) \Rightarrow P(c)]$
- Examples
  - For  $A = \mathbb{N}$  and  $x \prec y$  if we  $y = x + 1$  we get mathematical induction
  - For  $A = \mathbb{N}$  and  $x \prec y$  if we  $x < y$  we get Course of values Induction
  - For  $A =$  Derivation Trees and  $x \prec y$  if  $x$  is a subtree of  $y$  we get structural induction

# Further Reading

- Winskel: The formal semantics of programming languages: Chapter 3

# Summary

- Induction is powerful
  - Mathematical
  - Course of values
  - Rule induction
  - Structural
  - Well Founded
- Coming up with induction hypothesis may be hard
- Induction is a key technique in programming languages