

### Programming Languages

http://www.cse.iitd.ac.in/~sak/courses/pl/2021-22/index.html

### S. Arun-Kumar

Department of Computer Science and Engineering
I. I. T. Delhi, Hauz Khas, New Delhi 110 016.

December 28, 2021



PL December 28, 2021

44 | 4 | >



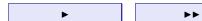
#### Contents

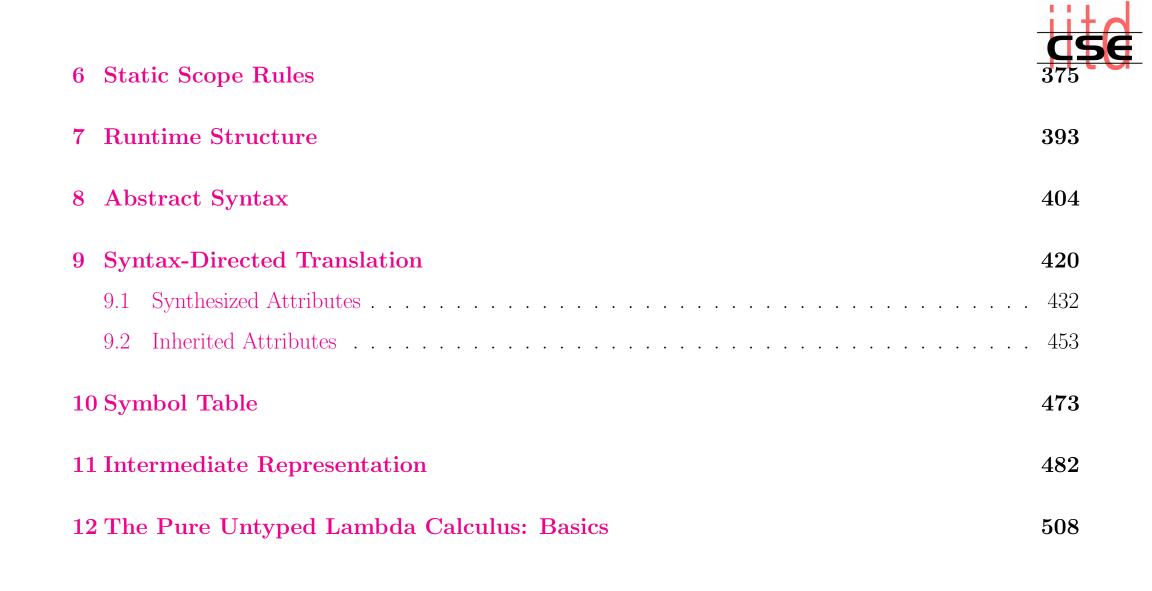
1	The	e Programming Languages Overview	3
2	Int	roduction to Compiling	25
3	Sca	nning or Lexical Analysis	45
	3.1	Regular Expressions	58
	3.2	Nondeterministic Finite Automata (NFA)	81
	3.3	Deterministic Finite Automata (DFA)	116
4	Par	rsing or Syntax Analysis	137
	4.1	Grammars	137
	4.2	Context-Free Grammars	144

PL December 28, 2021



	4.4 4.5 4.6 4.7	Ambiguity	178 189 197 215	
	4.10	Shift-Reduce Parsing	303	
5	<b>Att</b> : 5.1	ributes & Semantic Analysis  Context-sensitive analysis and Semantics	<b>363</b> 366	













12.1 Motivation for $\lambda$	<b>CS</b> 6
12.2 The $\lambda$ -notation	513
13 Notions of Reduction	545
14 Confluence Definitions	560
14.1 Why confluence?	566
14.2 Confluence: Church-Rosser	575
14.3 The Church-Rosser Property	583
15 An Applied Lambda-Calculus	<b>592</b>
15.1 FL with recursion	592
15.2 Motivation and Organization	593
15.3 Static Semantics of $FL(X)$	600

▶▶

		CS
	15.3.1 Type-checking $FL(X)$ terms	601
	15.3.2 The Typing Rules	602
15.4	Equational Reasoning in $FL(X)$	609
15.5	Type-checking FL	635
15.6	$\Lambda_{RecFL(X)}$ with type rules	635
16 An	Imperative Language	652
16 An	Imperative Language         16.0.1 l-values, r-values, aliasing and indirect addressing	
		657
16.1	16.0.1 l-values, r-values, aliasing and indirect addressing	657 664
16.1 16.2	16.0.1 l-values, r-values, aliasing and indirect addressing	657 664 678
16.1 16.2 16.3	16.0.1 l-values, r-values, aliasing and indirect addressing	657 664 678 686

PL December 28, 2021









7 of 721

### 17 Logic Programming and Prolog



1. The Programming Languages Overview



## What is a Programming Language?



- A (*linear*) notation for the *precise*, *accurate and complete* description of algorithms and data-structures *implementable* on a digital computer.
- In contrast,
  - the usual *mathematical* notation is accurate and precise enough for human beings but is not necessarily implementable on a digital computer,
  - and often the usual *mathematical* notation is not linear.
  - pseudo-code for algorithms and data-structures is too abstract<sup>a</sup> to be directly executed on a digital computer or even a virtual computer.
- A *program* is a sentence in a programming language intended to describe algorithms designed for a *universal computing machine*.
- While algorithms terminate not all programs may terminate.

<sup>&</sup>lt;sup>a</sup>Too many implementation details are either left unspecified or implicit.















Close



### The World of PLs

- There are just too many actual programming languages and more are being designed every year. Impossible to master every new PL that is released.
- Often impossible to master every feature of even the PLs that are currently in use.
- Often not necessary to master all features of a PL.

44

**▶**▶

Go Back

Full Screen

Close



## Why Study the subject of PL? - 1

To understand the various major paradigms of programming.

- The same algorithm requires different design considerations in different paradigms.
- Different data-structures as part of the language,
- Different libraries provided along with the language implementation.
- Different styles of thought involved in the implementation.

44





Go Back

FULL SCREEN

Close



## Why Study the subject of PL? - 2

To understand the major features and their implementation common to large numbers of PLs.

- The same feature may be implemented differently in different PLs.
- The same algorithm is written differently in different PLs of the same paradigm depending upon
  - the data-structures available,
  - the control structures available,
  - the libraries available.

44

**b** b

PL December 28, 2021

Go Back

Full Screen

Close



## Why Study the subject of PL? - 3

To understand the major design and architectural considerations common to most PLs.

- Whether a data- or control-structure is part of the programming language itself.
- Whether certain complex (data- and control-)structures are provided as libraries of the programming language.

44

**•** 

PL December 28, 2021

Go Back

FULL SCREEN

Close



### Architectural Considerations in PL

- Compilation vs Interpretation
- Portability considerations across hardware architecures
- Virtual machines or target architectures.
- Stack architecture vs. register architecure.
- Representation and typing.
- The set of intermediate languages/representation required for the implementation.
- Support for parallelism.

44

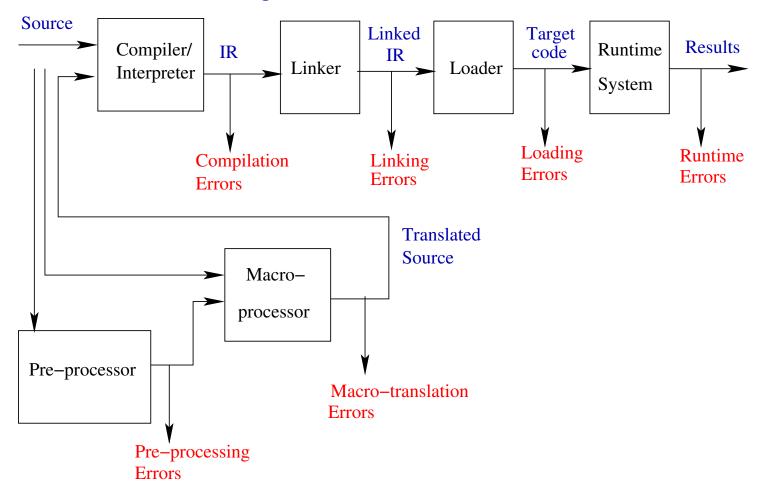








## Programs: Source to Runs

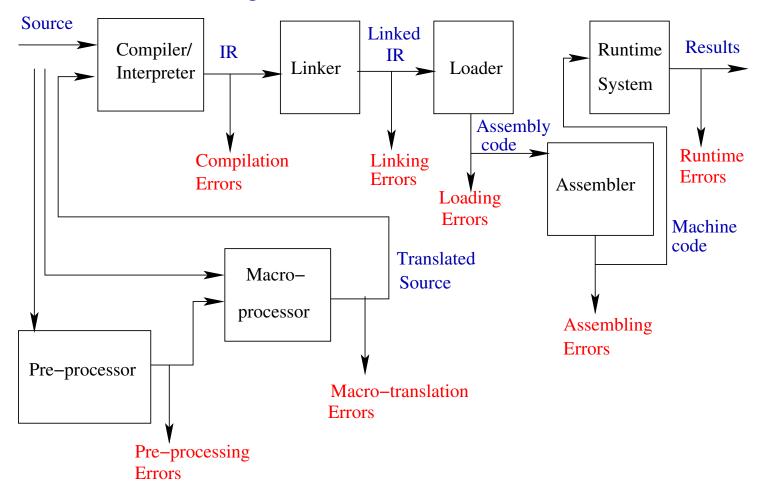


 →
 →
 Full Screen
 Close

 15 of 721



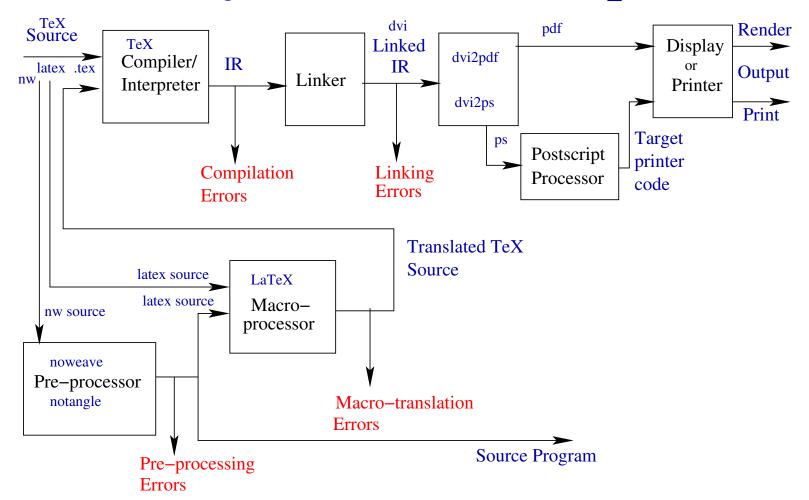
## Programs: Source to Runs-2



 ◄
 Image: Second of the propertion of the properties of the



## Programs: Source to Runs-1: LATEX

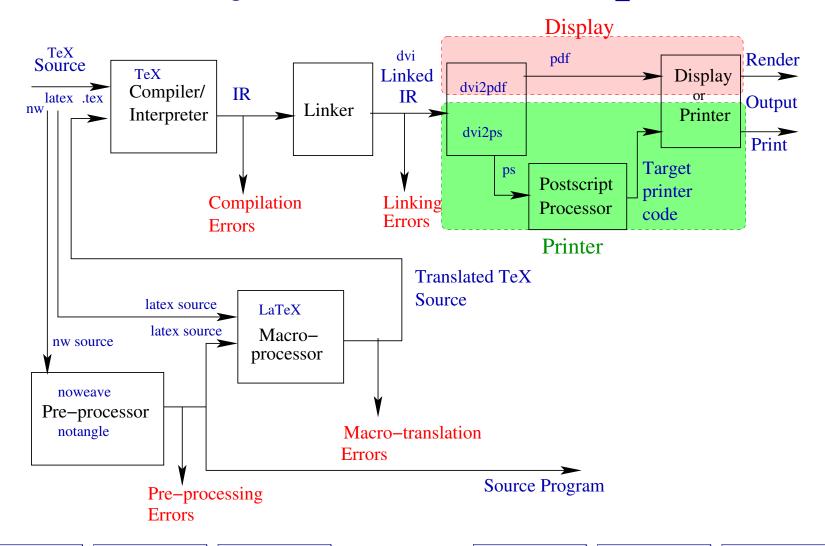


 ✓
 Full Screen

 Close
 17 of 721

## Programs: Source to Runs-2: LATEX

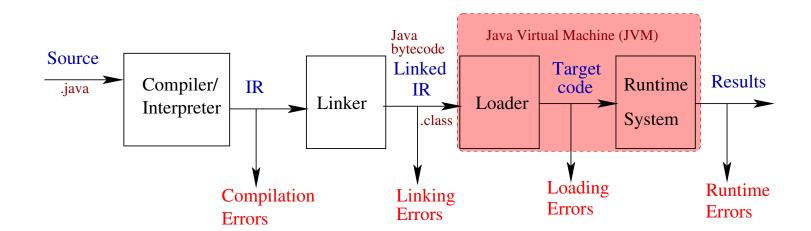




 ✓
 Image: Super the problem of the proble



# Programs: Source to Runs: Java

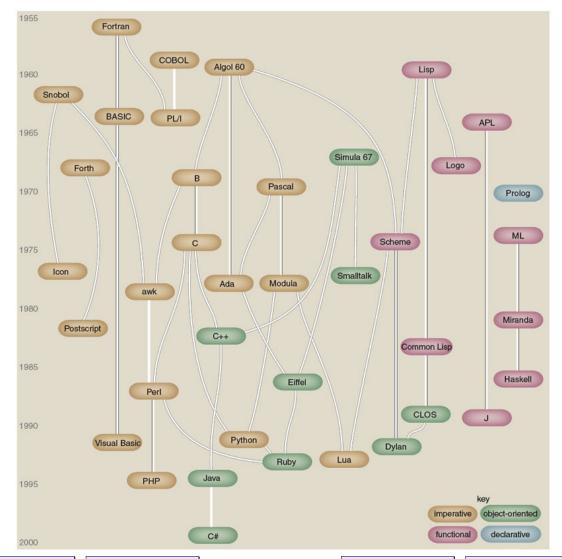


 ✓
 Full Screen

 Close
 19 of 721

# The Landscape of General PLs





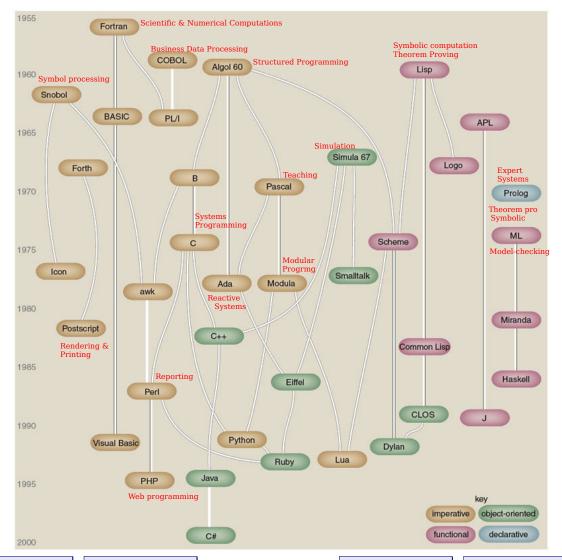






# The Usage of General PLs

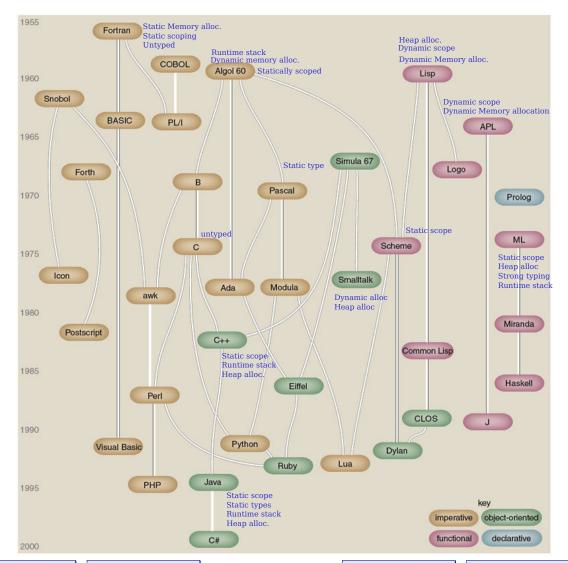




PL December 28, 2021

# The Major Features of General PLs





4

**▶** 

**>>** 

PL December 28, 2021

Go Back

Full Screen

Close



#### **FORTRAN**

- The very first high-level programming language
- Still used in scientific computation
- Static memory allocation
- Very highly compute oriented
- Runs very fast because of static memory allocation
- Parameter passing by reference

44

▶▶

PL December 28. 2021



### COBOL

- A business oriented language
- Extremely verbose
- Very highly input-oriented
- Meant to manage large amounts of data on disks and tapes and generate reports
- Not computationally friendly

44

▶▶

Go Back

Full Screen

Close



### LisP

- First functional programming language
- Introduced lists and list-operations as the only data-structure
- Introduced symbolic computation
- Much favoured for AI and NLP programming for more than 40 years
- The first programming language whose interpreter could be written in itself.

 →
 Image: A proper to the proper



### ALGOL-60

- Introduced the Backus-Naur Form (BNF) for specifying syntax of a programming language
- Formal syntax defined by BNF (an extension of context-free grammars)
- First language to implement recursion
- Introduction of block-structure and nested scoping
- Dynamic memory allocation
- Introduced the call-by-name parameter mechanism

44





PL December 28, 2021

Go Back

Full Screen

Close



### Pascal

- ALGOL-like language meant for teaching structured programming
- Introduction of new data structures records, enumerated types, sub-range types, recursive data-types
- Its simplicity led to its "dialects" being adopted for expressing algorithms in pseudo-code
- First language to be ported across a variety of hardware and OS platforms introduced the concepts of virtual machine and intermediate code (bytecode)

 ✓
 ✓

 PL December 28, 2021
 Go Back
 Full Screen
 Close
 27 of 721



#### ML

- First strongly and statically typed functional programming language
- Created the notion of an inductively defined type to construct complex types
- Powerful pattern matching facilities on complex data-types.
- Introduced type-inference, thus making declarations unnecessary except in special cases
- Its module facility is inspired by the algebraic theory of abstract data types
- The first language to introduce functorial programming between algebraic structures and modules

44



PL December 28. 2021

Go Back

FULL SCREEN

Close



## Prolog

- First Declarative programming language
- Uses the Horn clause subset of first-order logic
- Goal-oriented programming implementing a top-down methodology
- Implements backtracking as a language feature
- Powerful pattern-matching facilities like in functional programming
- Various dialects implement various other features such as constraint programming, higher-order functions etc.

PL December 28, 2021





Full Screen



2. Introduction to Compiling













Close



## Introduction to Compiling

- Translation of programming languages into executable code
- But more generally any large piece of software requires the use of compiling techniques.
- The processes and techniques of designing compilers is useful in designing most large pieces of software.
- Compiler design uses techniques from theory, data structures and algorithms.

44

▶▶

Go Back

Full Screen

Close



### Software Examples

Some examples of other software that use compiling techniques

- Almost all user-interfaces require scanners and parsers to be used.
- All XML-based software require interpretation that uses these techniques.
- All mathematical text formatting requires the use of scanning, parsing and code-generation techniques (e.g. LATEX).
- Model-checking and verification software are based on compiling techniques
- Synthesis of hardware circuits requires a description language and the final code that is generated is an implementation either at the register-transfer level or gate-level design.

PL December 28, 2021

Go Back

Full Screen



### **Books and References**

- 1. Appel A W. Modern Compiler Implementation in Java Cambridge University Press, Revised Indian Paperback edition 2001
- 2. Aho A V, Sethi R, Ullman J D. Compilers: Principles, Techniques, and Tools, Addison-Wesley 1986.
- 3. Muchnick S S. Advanced Compiler Design and Implementation, Academic Press 1997

 →
 Image: Second of the propertion of the properties of the



## A Plethora of Languages: Compiling

In general a compiler/interpreter for a a source language S written in some language C translates code written in S to a target language T.

Source  $\mathcal{S}$ 

Target  $\mathcal{T}$ 

Language of the compiler/interpreter  $\mathcal{C}$ 

Our primary concern. Compiling from a high-level source programming language to a target language using a high-level language C.

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 34 OF 721



### A Plethora of Languages: Source

The Source language S could be

- a programming language, or
- a description language (e.g. Verilog, VHDL), or
- a markup language (e.g. XML, HTML, SGML, LATEX) or
- even a "mark-down" language to simplify writing code.

44

**•** 

**▶**▶

Go Back

Full Screen

Close



## A Plethora of Languages: Target

The Target language  $\mathcal{T}$  could be

- an intermediate language (e.g. ASTs, IR, bytecode etc.)
- another programming language, assembly language or machine language, or

PL December 28, 2021

- a language for describing various objects (circuits etc.), or
- a low level language for execution, display, rendering etc. or
- even another high-level language.





Go Back

Full Screen

Close



### The Compiling Process

Besides S, C and T there could be several other intermediate languages  $\mathcal{I}_1, \mathcal{I}_2, \ldots$  (also called intermediate representations) into which the source program could be translated in the process of compiling or interpreting the source programs written in S. In modern compilers, for portability, modularity and reasons of code improvement, there is usually at least one intermediate representation.

Some of these intermediate representations could just be data-types of a modern functional or object-oriented programming language.



### Compiling as Translation

Except in the case of a source to source translation (for example, a Pascal to C translator which translates Pascal programs into C programs), we may think of the process of compiling high-level languages as one of transforming programs written in  $\mathcal S$  into programs of lower-level languages such as the intermediate representation or the target language. By a low-level language we mean that the language is in many ways closer to the architecture of the target language.

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 38 OF 721



### Phases of a Compiler

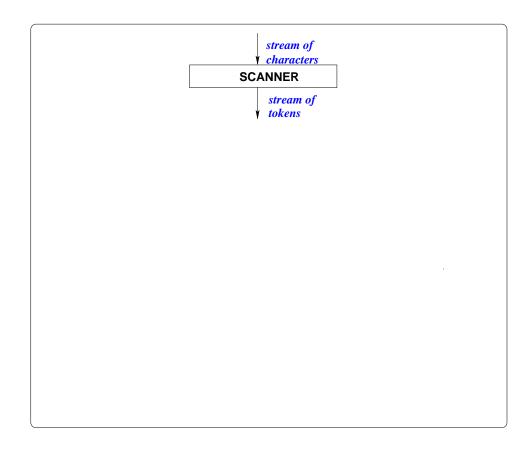
A compiler or translator is a fairly complex piece of software that needs to be developed in terms of various independent modules.

In the case of most programming languages, compilers are designed in *phases*.

The various phases may be different from the various passes in compilation













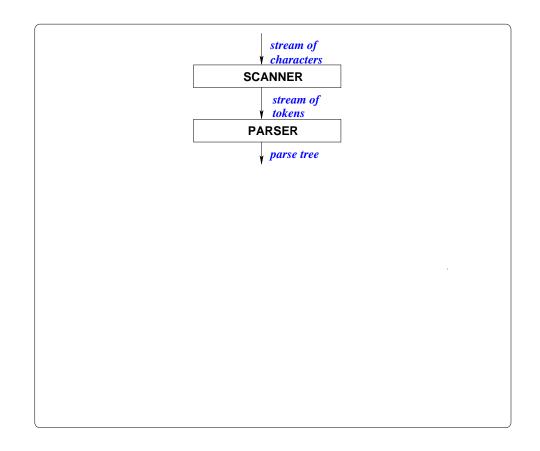




Full Screen

Close







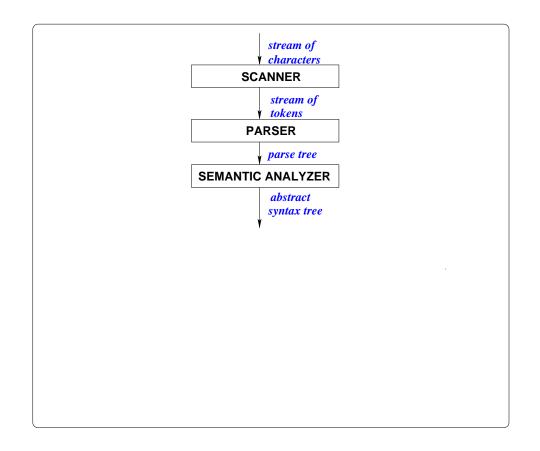














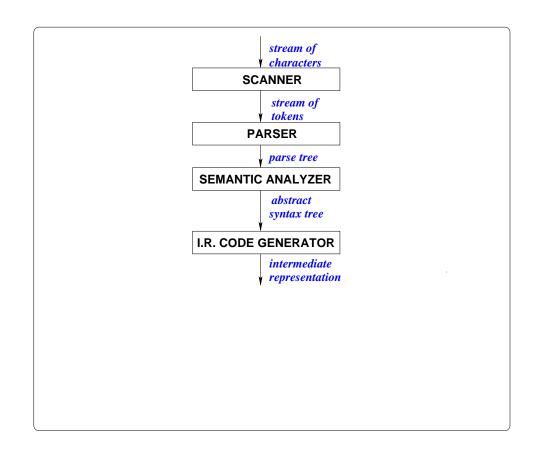
















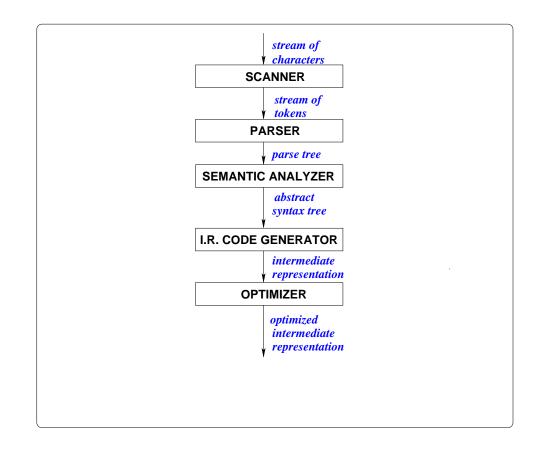














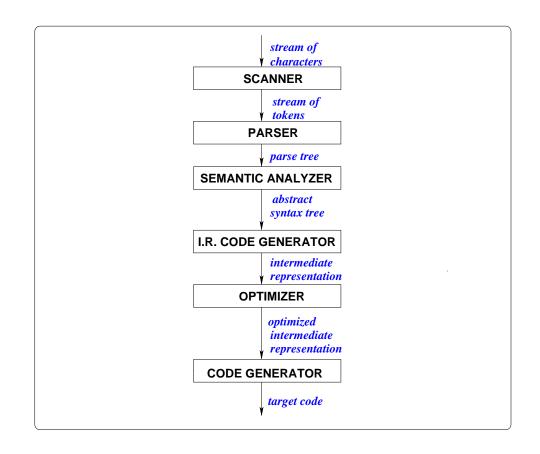












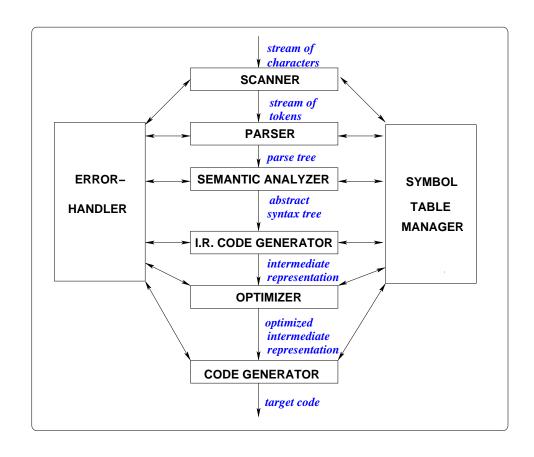










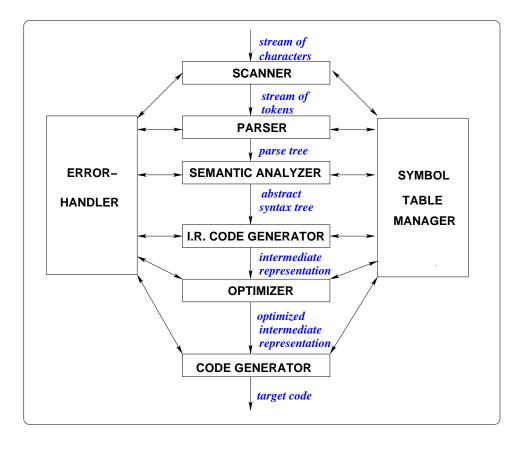


Go Back

Full Screen

Close





Scanner	Parser	Semantic Analysis	Symbol Table
IR	Run-time structure		

44

**>>** 

PL December 28, 2021

Go Back

Full Screen

Close



→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 48 OF 721



→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 49 OF 721



3. Scanning or Lexical Analysis

## Lexical Analysis











### Programming Language Elements

- Every language is built from a finite alphabet of symbols. The alphabet of a programming language consists of the symbols of the ASCII set.
- Each language has a **vocabulary** consisting of **words**. Each word is a *string of symbols* drawn from the alphabet.
- Each language has a finite set of punctuation symbols, which separate phrases, clauses and sentences.
- A programming language also has a finite set of operators.
- The phrases, clauses and sentences of a programming language are expressions, commands, functions, procedures and programs.

 ✓
 ✓

 PL December 28, 2021
 Go Back
 Full Screen
 Close
 51 of 721



### Lexical Analysis

lex-i-cal: relating to words of a language

- A source program (usually a file) consists of a stream of characters.
- Given a stream of characters that make up a *source program* the compiler must first break up this stream into a sequence of "lexemes", and other symbols.
- $\bullet$  Each such lexeme is then classified as belonging to a certain token type.
- Certain lexemes may violate the pattern rules for tokens and are considered erroneous.
- Certain sequences of characters are <u>not</u> tokens and are completely ignored (or skipped) by the compiler.

44

**>>** 

PL December 28, 2021

Go Back

FULL SCREEN

Close



#### Erroneous lexemes

Some lexemes violate all rules of tokens. Some examples common to most programming languages

- 12ab would not be an identifier or a number in most programming languages. If it were an integer in Hex code it would be written 0x12ab.
- 127.0.1 is usually not any kind of number. However 127.0.0.1 may be a valid token representing an IP address.

PL December 28, 2021

Full Screen

Close

53 OF 721



### Tokens and Non-tokens: 1

- Tokens
- Non-tokens











#### Tokens and Non-tokens: 2

### Tokens Typical tokens are

- Constants: Integer, Boolean, Real, Character and String constants.
- Identifiers: Names of variables, constants, procedures, functions etc.
- Keywords/Reserved words: void, public, main
- Operators:+, \*, /
- Punctuation: ,, :, .
- Brackets: (, ), [, ], begin, end, case, esac

#### Non-tokens

44



PL December 28, 2021

Full Screen

Close



#### Tokens and Non-tokens: 3

#### **Tokens**

Non-tokens Typical non-tokens are

- whitespace: sequences of tabs, spaces, new-line characters,
- comments: compiler ignores comments
- preprocessor directives: #include ..., #define ...
- macros in the beginning of C programs

44





Go Back

Full Screen

Close



During the scanning phase the compiler/interpreter

- takes a stream of characters and identifies tokens from the lexemes.
- Eliminates comments and redundant whitepace.
- Keeps track of line numbers and column numbers and passes them as parameters to the other phases to enable error-reporting and handling to the user.

44

▶▶

Go Back

Full Screen

Close

57 OF 721



**Definition 3.1** A lexeme is a basic lexical unit of a language consisting of one word or several words, the elements of which do not separately convey the meaning of the whole.

- Whitespace: A sequence of space, tab, newline, carriage-return, form-feed characters etc.
- Lexeme: A sequence of non-whitespace characters delimited by whitespace or special characters (e.g. operators or punctuation symbols)
- Examples of lexemes.
  - reserved words, keywords, identifiers etc.
  - Each comment is usually a single lexeme
  - preprocessor directives

44









Full Screen

Close



**Definition 3.2** A token consists of an abstract name and the attributes of a lexeme.

- Token: A sequence of characters to be treated as a single unit.
- Examples of tokens.
  - -Reserved words (e.g. begin, end, struct, if etc.)
  - Keywords (integer, true etc.)
  - -Operators (+, &&, ++ etc)
  - Identifiers (variable names, procedure names, parameter names)
  - Literal constants (numeric, string, character constants etc.)
  - Punctuation marks (:, , etc.)

44





PL December 28. 2021



Full Screen

CLOSE



- Identification of tokens is usually done by a Deterministic Finite-state automaton (DFA).
- The set of tokens of a language is represented by a large regular expression.
- This regular expression is fed to a lexical-analyser generator such as Lex, Flex or JI ex.
- A giant DFA is created by the Lexical analyser generator.

PL December 28, 2021

Go Back

Full Screen

Close



#### Lexical Rules

• Every programming language has lexical rules that define how a token is to be defined.

Example. In most programming languages identifiers satisfy the following rules.

- 1. An identifier consists of a sequence of of letters (A ... Z, a ... z), digits (0 ... 9) and the underscore  $(\_)$  character.
- 2. The first character of an identifier must be a letter.
- Any two tokens are separated by some delimiters (usually whitespace) or non-tokens in the source program.

44

1

PL December 28. 2021

Go Back

Full Screen

Close





#### 3.1. Regular Expressions

# Regular Expressions









Go Back

Full Screen

Close

Consider the string 11/12/2021? What does this string of characters represent? There are at least the following different possibilities.

- In a school mathematics text it might represent the operation of division (11/12)/2021, i.e the fraction 11/12 divided by 2021, yielding the value .00045357083951839023.
- To a student who is confused, it may also represent the operation of division 11/(12/2021) i.e. the result of dividing 11 by the fraction 12/2021 yielding the value 1852.5833333333333369846.
- In some official document from India it might represent a date (in dd/mm/yyyy format) viz. 11 December 2021<sup>1</sup>.
- In some official document from America it might represent a different date (in mm/dd/yyyy format) viz. November 12,  $2021^2$ .

The ambiguity inherent in such representations requires that (especially if the school mathematics text also uses some date format in some problems) a clearer specification of the individual elements be provided. These specifications of individual elements in programming languages are provided by *lexical rules*. These lexical rules specify "patterns" which are legal for use in the language.

PL December 28, 2021

44



<sup>&</sup>lt;sup>1</sup>Have you heard of the 26/11 attack?

<sup>&</sup>lt;sup>2</sup>Have you heard of the 9/11 attack?



## Specifying Lexical Rules

We require compact and simple ways of specifying the lexical rules of the tokens of a language. In particular,

- there are an *infinite* number of legally correct identifiers (names) in any programming language.
- we require *finite* descriptions/specifications of the lexical rules so that they can cover the infinite number of legal tokens.

One way of specifying the lexical rules of a programming language is to use regular expressions.

**◄**◄







# Regular Expressions Language



- Any set of strings built up from the symbols of A is called a language. A\* is the set of all finite strings built up from A.
- Each regular expression is a *finite* sequence of symbols made up of symbols from the alphabet and other symbols called operators.
- A regular expression may be used to describe an *infinite* collection of strings.

**Example 3.3** The regular expression used to define the set of possible identifiers as defined by the rules is

$$[A-Za-z][A-Za-z0-9_{-}]^{*}$$

Here the symbols in black are drawn from the alphabet consisting of lower-case, upper-case letters and digits. The symbols in blue are operator symbols.

44

**•** 

**>>** 

Go Back

Full Screen

Close

#### Concatenations



Consider a (finite) alphabet (of symbols) A.

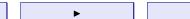
• Given any two strings x and y in a language, x.y or simply xy is the concatenation of the two strings.

**Example 3.4** Given the strings x = Mengesha and y = Mamo, x.y = MengeshaMamo and y.x = MamoMengesha.

• Given two languages X and Y, then X.Y or simply XY is the concatenation of the languages.

**Example 3.5** Let  $X = \{\text{Mengesha}, \text{Gemechis}\}$  and  $Y = \{\text{Mamo}, \text{Bekele}, \text{Selassie}\}$ . Then  $XY = \{\text{MengeshaMamo}, \text{MengeshaBekele}, \text{MengeshaSelassie}\}$  GemechisMamo, GemechisBekele, GemechisSelassie}

44





#### Note on the Concept of "language".

Unfortunately we have too many related but slightly different concepts, each of which is simply called a "language". Here is a clarification of the various concepts that we use.

- Every language has a non-empty finite set of symbols called **letters**. This non-empty finite set is called the **alphabet**.
- Each word is a finite sequence of symbols called letters.
- The words of a language usually constitute its **vocabulary**. Certain sequences of symbols may not form a word in the vocabulary. A vocabulary for a natural language is defined by a *dictionary*, whereas for a programming language it is usually defined by *formation rules*.
- A **phrase**, **clause** or **sentence** is a finite sequence of words drawn from the vocabulary.
- Every natural language or programming language is a finite or infinite set of **sentences**.
- In the case of formal languages, the formal language is the set of words that can be formed using the formation rules. The language is also said to be **generated** by the formation rules.

There are a variety of languages that we need to get familiar with.

44

**>** 

**>>** 

Go Back

Full Screen

Close

**Natural languages.** These are the usual languages such as *English*, *Hindi*, *French*, *Tamil* which we employ for daily communication and in teaching, reading and writing.

**Programming languages.** These are the languages such as *C*, *Java*, *SML*, *Perl*, *Python* etc. that are used to write computer programs in.

Formal languages. These are languages which are generated by certain formation rules.

**Meta-languages.** These are usually natural languages used to explain concepts related to programming languages or formal languages. We are using *English* as the meta-language to describe and explain concepts in programming languages and formal languages.

In addition, we do have the concept of a **dialect** of a natural language or a programming language. For example the natural languages like Hindi, English and French do have several dialects. A **dialect** (in the case of natural languages) is a particular form of a language which is peculiar to a specific region or social group. Creole (spoken in Mauritius) is a dialect of French, Similarly Brij, Awadhi are dialects of Hindi. A dialect (in the case of programming languages) is a version of the programming language. There are many dialects of C and C++. Similarly SML-NJ and poly-ML are dialects of Standard ML. The notion of a dialect does not really exist for formal languages.

44



Closer home to what we are discussing, the language of regular expressions is a *formal language* which describes the rules for forming the words of a programming language. Each regular expression represents a finite or infinite set of words in the vocabulary of a programming language. We may think of the language of regular expressions also as a *functional programming language* for describing the vocabulary of a programming language. It allows us to generate words belonging to the vocabulary of a programming language

Any formally defined language also defines an algebraic system of operators applied on a *carrier set*. Every operator in any algebraic system has a pre-defined *arity* which refers to the number of operands it requires. In the case of regular expressions, the operators are concatenation and alternation are 2-ary operators (binary operators), whereas the Kleene closure and plus closure are 1-ary operators (unary). In addition the letters of the alphabet, which are constants may be considered to be operators of arity 0.



### Simple Language of Regular Expressions

We consider a simple language of regular expressions. Assume a (finite) alphabet A of symbols. Each regular expression r denotes a set of strings  $\mathcal{L}(r)$ .  $\mathcal{L}(r)$  is also called the language specified by the regular expression r.

Symbol For each symbol a in A, the regular expression a denotes the set  $\{a\}$ .

(Con)catenation For any two regular expressions r and s, r.s or simply rs denotes the concatenation of the languages specified by r and s. That is,

PL December 28, 2021

$$\mathcal{L}(rs) = \mathcal{L}(r)\mathcal{L}(s)$$

44

**→** 

▶▶

Go Back

Full Screen

Close



### **Epsilon and Alternation**

**Epsilon**  $\epsilon$  denotes the language with a single element the **empty** string ("") i.e.

$$\mathcal{L}(\epsilon) = \{""\}$$

**Alternation** Given any two regular expressions r and s, r|s is the set union of the languages specified by the individual expressions r and s respectively.

$$\mathcal{L}(r \mid s) = \mathcal{L}(r) \cup \mathcal{L}(s)$$

Example  $\mathcal{L}(\mathbf{Menelik}|\mathbf{Selassie}|\epsilon) = \{\mathbf{Menelik}, \mathbf{Selassie}, ""\}.$ 

44

**•** 

▶▶

Go Back

Full Screen

Close



## String Repetitions

For any string x, we may use concatenation to create a string y with as many repetitions of x as we want, by defining repetitions by induction.

$$x^{0} = ""$$
 $x^{1} = x$ 
 $x^{2} = x.x$ 
 $\vdots$ 
 $x^{n+1} = x.x^{n} = x^{n}.x$ 
 $\vdots$ 

Then

$$x^* = \{x^n \mid n \ge 0\}$$

**◄**◀

PL December 28, 2021

Go Back

FULL SCREEN

Close

73 of 721



## String Repetitions Example

Example. Let x = Selassie. Then

```
x^0 = "" x^1 = \text{Selassie} x^2 = \text{SelassieSelassie} \vdots x^5 = \text{SelassieSelassieSelassieSelassieSelassie} \vdots
```

Then  $x^*$  is the language consisting of all strings that are finite repetitions of the string Selassie

PL December 28. 2021









#### Language Iteration

The \* operator can be extended to languages in the same way. For any language X, we may use concatenation to create a another language Y with as many repetitions of the strings in X as we want, by defining repetitions by induction.

$$X^{0} = "" X^{1} = X$$
 $X^{2} = X.X X^{3} = X^{2}.X$ 
 $\vdots \vdots$ 

PL December 28, 2021

In general 
$$X^{n+1}=X.X^n=X^n.X$$
 and 
$$X^*=\bigcup_{n\geq 0}X^n$$

Go Back

Full Screen

75 OF 721



## Language Iteration Example

**Example 3.6** Let  $X = \{Mengesha, Gemechis\}$ . Then

```
X^{0} = \{""\}
X^1 = \{ \texttt{Mengesha}, \texttt{Gemechis} \}
X^2 = \{ \text{MengeshaMengesha}, \text{GemechisMengesha}, \}
      MengeshaGemechis, GemechisGemechis}
X^3 = \{MengeshaMengeshaMengesha, GemechisMengeshaMengesha,
      MengeshaGemechisMengesha, GemechisGemechisMengesha,
      MengeshaMengeshaGemechis, GemechisMengeshaGemechis,
      MengeshaGemechisGemechis, GemechisGemechisGemechis}
```

44

**▶** 

PL December 28, 2021

Go Back

Full Screen

Close

76 of 721

#### Kleene Closure



Given a regular expression r,  $r^n$  specifies the n-fold iteration of the language specified by r.

Given any regular expression r, the Kleene closure of r, denoted  $r^*$  specifies the language  $(\mathcal{L}(r))^*$ .

In general

$$r^* = r^0 \mid r^1 \mid \cdots \mid r^{n+1} \mid \cdots$$

denotes an infinite union of languages.

Further it is easy to show the following identities.

$$r^* = \epsilon | r.r^*$$

$$r^* = (r^*)^*$$
(1)

44

PL December 28. 2021

Go Back

Full Screen

Close

77 of 721

# dse

#### Plus Closure

The Kleene closure allows for zero or more iterations of a language. The +-closure of a language X denoted by  $X^+$  and defined as

$$X^+ = \bigcup_{n>0} X^n$$

denotes one or more iterations of the language X.

Analogously we have that  $r^+$  specifies the language  $(\mathcal{L}(r))^+$ .

Notice that for any language X,  $X^+ = X.X^st$  and hence for any regular

expression r we have

$$r^+ = r.r^*$$

We also have the identity (1)

$$r^* = \epsilon \mid r^+$$

44

**>** 

▶▶

Go Back

FULL SCREEN

Close



## Range Specifications

We may specify ranges of various kinds as follows.

•  $[\mathbf{a} - \mathbf{c}] = \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$ . Hence the expression of Question 3 may be specified as  $[\mathbf{a} - \mathbf{c}]^*$ .

PL December 28, 2021

• Multiple ranges:  $[\mathbf{a} - \mathbf{c}\mathbf{0} - \mathbf{3}] = [\mathbf{a} - \mathbf{c}] \mid [\mathbf{0} - \mathbf{3}]$ 







#### Exercise 3.1

- 1. Try to understand what the regular expression for identifiers really specifies.
- 2. Modify the regular expression so that all identifiers start only with upper-case letters.
- 3. Give regular expressions to specify
  - real numbers in fixed decimal point notation
  - real numbers in floating point notation
  - real numbers in both fixed decimal point notation as well as floating point notation.







81 of 721

## Equivalence of Regular Expressions

**Definition 3.7** Let  $REGEXP_{A}$  denote the set of regular expressions over a a finite non-empty set of symbols A and let  $r, s \in REGEXP_{A}$ . Then

- $r \leq_{\mathbb{A}} r$  if and only if  $\mathcal{L}(r) \subseteq \mathcal{L}(s)$  and
- they are equivalent (denoted r = A s) if they specify the same language, i.e.

$$r =_{\mathbb{A}} s$$
 if and only if  $\mathcal{L}(r) = \mathcal{L}(s)$ 

We have already considered various identities (e.g. (1)) giving the equivalence between different regular expressions.

 →
 →

 Full Screen

 Close



#### Notes on bracketing and precedence of operators

In general regular expressions could be *ambiguous* (in the sense that the same expression may be interpreted to refer to different languages. This is especially so in the presence of

- multiple binary operators
- some unary operators used in prefix form while some others are used in post-fix form. The Kleene-closure and plus closure are operators in postfix form. We have not introduced any prefix unary operator in the language of regular expressions.

All expressions may be made unambiguous by specifying them in a fully parenthesised fashion. However, that leads to too many parentheses and is often hard to read. Usually rules for precedence of operators is defined and we may use the parentheses "(" and ")" to group expressions over-riding the precedence conventions of the language.

For the operators of regular expressions we will use the precedence convention that | has a lower precedence than . and that all unary operators have the highest precedence.

**Example 3.8** The language of arithmetic expressions over numbers uses the "BDMAS" convention that brackets have the highest precedence, followed by division and multiplication and the operations of addition and subtraction have the lowest precedence.

44









Full Screen

Close

82 OF 721

**Example 3.9** The regular expression r.s|t.u is ambiguous because we do not know beforehand whether it represents (r.s)|(t.u) or r.(s|t).u or even various other possibilities. By specifying that the operator | has lower precedence than . we are disambiguating the expression to mean (r.s)|(t.u).

**Example 3.10** The language of arithmetic expressions can also be extended to include the unary post-fix operation in which case an expression such as -a! becomes ambiguous. It could be interpreted to mean either (-a)! or -(a!). In the absence of a well-known convention it is best adopt parenthesisation to disambiguate the expression.

Besides the ambiguity created by multiple binary operators, there are also ambiguities created by the same operator and in deciding in what order two or more occurrences of the same operator need to be evaluated. A classic example is the case of subtraction in arithmetic expressions.

**Example 3.11** The arithmetic expression a - b - c, in the absence of any well-defined convention could be interpreted to mean either (a - b) - c or a - (b - c) and the two interpretations would yield different values in general. The problem does not exist for operators such addition and multiplication on numbers, because these operators are associative. Hence even though a + b + c may be interpreted in two different ways, both interpretations yield identical values.

**Example 3.12** Another non-associative operator in arithmetic which often leaves students confused is the exponentiation operator. Consider the arithmetic expression  $a^{b^c}$ . For a = 2, b = 3, c = 4 is this expression to be interpreted as  $a^{(b^c)}$  or as  $(a^b)^c$ ?









#### Exercise 3.2

- 1. For what regular expression r will  $r^*$  specify a finite set?
- 2. How many strings will be in the language specified by  $(\mathbf{a} \mid \mathbf{b} \mid \mathbf{c})^n$ ?
- 3. Give an informal description of the language specified by  $(\mathbf{a} \mid \mathbf{b} \mid \mathbf{c})^*$ ?
- 4. Give a regular expression which specifies the language  $\{a^k \mid k > 100\}$ .
- 5. Simplify the expression  $r^*.r^*$ , i.e. give a simpler regular expression which specifies the same language.

PL December 28, 2021

6. Simplify the expression  $r^+.r^+$ .

Go Back

Full Screen

Close





3.2. Nondeterministic Finite Automata (NFA)

## Nondeterministic Finite Automata (NFA)

PL December 28, 2021









Go Back

Full Screen

Close



#### Nondeterministic Finite Automata

A regular expression is useful in defining a *finite state automaton*. An automaton is a machine (simple program) which can be used to recognize all valid lexical tokens of a language.

A nondeterministic finite automaton (NFA) N over a finite alphabet A consists of

- a finite set Q of states,
- an initial state  $q_0 \in Q$ ,
- $\bullet$  a finite subset  $F\subseteq Q$  of states called the final states or accepting states, and
- a transition relation  $\longrightarrow \subseteq Q \times (A \cup \{\varepsilon\}) \times Q$ .

44

PL December 28. 2021

Go Back

Full Screen

Close



#### What is nondeterminstic?

• The transition relation may be equivalently represented as a function

$$\longrightarrow: Q \times (A \cup \{\varepsilon\}) \to 2^Q$$

that for each source state  $q \in Q$  and symbol  $a \in A$  associates a set of target states.

- It is non-deterministic because for a given source state and input symbol,
  - there may not be a unique target state, there may be more than one, or
  - the set of target states could be empty.
- ullet Another source of non-determinism is the empty string  $\varepsilon$ .

44

PL December 28. 2021

Go Back

Full Screen

Close

# d\$E

#### Nondeterminism and Automata

- In general the automaton *reads* the input string from left to right.
- It reads each input symbol only once and executes a transition to new state.
- ullet The arepsilon transitions represent going to a new target state *without* reading any input symbol.
- The NFA may be nondeterministic because of
  - —one or more  $\varepsilon$  transitions from the same source state *different* target states,
  - —one or more transitions on the *same input* symbol from one source state to two or more different target states,
  - choice between executing a transition on an input symbol and a transition on  $\varepsilon$  (and going to different states).

44

**•** 

**>>** 

PL December 28, 2021

Go Back

Full Screen

Closi

#### Acceptance of NFA



- For any alphabet A, A\* denotes the set of all (finite-length) strings of symbols from A.
- Given a string  $x = a_1 a_2 \dots a_n \in A^*$ , an accepting sequence is a sequence of transitions

$$q_0 \xrightarrow{\varepsilon} \cdots \xrightarrow{a_1} \xrightarrow{\varepsilon} \cdots q_1 \xrightarrow{\varepsilon} \cdots \xrightarrow{a_2} \cdots \xrightarrow{a_2} \cdots \xrightarrow{\varepsilon} \xrightarrow{a_n} \xrightarrow{\varepsilon} \cdots q_n$$

where  $q_n \in F$  is an accepting state.

• Since the automaton is nondeterministic, it is also possible that there exists another sequence of transitions

$$q_0 \xrightarrow{\varepsilon} \cdots \xrightarrow{a_1} \xrightarrow{\varepsilon} \cdots q'_1 \xrightarrow{\varepsilon} \cdots \xrightarrow{a_2} \cdots \xrightarrow{\varepsilon} \xrightarrow{a_n} \xrightarrow{\varepsilon} \cdots q'_n$$

where  $q'_n$  is not a final state.

• The automaton accepts x, if there is an accepting sequence for x.

**◄**◀

. DI

Go Back

Full Scree

Closi



#### Language of a NFA

- The language accepted or recognized by a NFA is the set of strings that can be accepted by the NFA.
- $\bullet$   $\mathcal{L}(N)$  is the language accepted by the NFA N.

44

▶

▶▶

Go Back

Full Screen

Close

#### Construction of NFAs



- We show how to construct an NFA to accept a certain language of strings from the regular expression specification of the language.
- The method of construction is by *induction on the structure* of the regular expression. That is, for each regular expression operator, we show how to construct the corresponding automaton assuming that the NFAs corresponding to individual components of expression have already been constructed.
- For any regular expression r the corresponding NFA constructed is denoted  $N_r$ . Hence for the regular expression r|s, we construct the NFA  $N_{r|s}$  using the NFAs  $N_r$  and  $N_s$  as the building blocks.
- Our method requires only one initial state and one final state for each automaton. Hence in the construction of  $N_{r|s}$  from  $N_r$  and  $N_s$ , the initial states and the final states of  $N_r$  and  $N_s$  are not initial or final unless explicitly used in that fashion.

44

**•** 

**>>** 

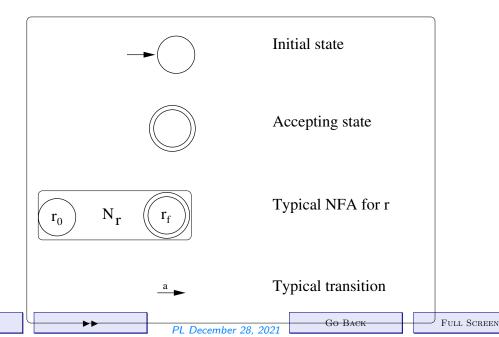
Go Back

Full Screen

Clo

## Constructing NFA

- We show the construction only for the most *basic* operators on regular expressions.
- For any regular expression r, we construct a NFA  $N_r$  whose initial state is named  $r_0$  and final state  $r_f$ .
- The following symbols show the various components used in the depiction of NFAs.



Close





We may also express the automaton in tabular form as follows:

$N_a$	Input Symbol		
State	a	• • •	$\varepsilon$
$\mathbf{a}_0$	$\{\mathbf{a}_f\}$	$\emptyset \cdots \emptyset$	$\emptyset$
$\mathbf{a}_f$	$\emptyset$	$\emptyset \cdots \emptyset$	$\emptyset$

Notice that all the cells except one have empty targets.

44





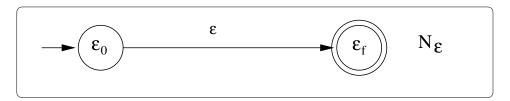
PL December 28, 2021



Full Screen

Close





$N_arepsilon$	Input Symbol		
State	a	• • •	$\varepsilon$
$\varepsilon_0$	Ø	$\emptyset \cdots \emptyset$	$\{arepsilon_f\}$
$\varepsilon_f$	$\emptyset$	$\emptyset \cdots \emptyset$	Ø

PL December 28, 2021

44



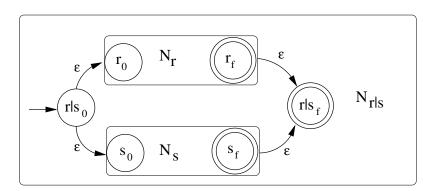
**>>** 

Go Васк

Full Screen

Close



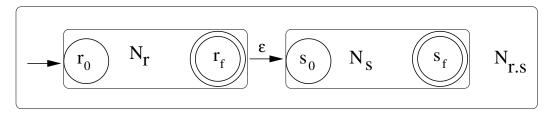


$N_{r s}$	Input Symbol		
State	a	• • •	ω
$r s_0$	Ø	• • •	$\{r_0, s_0\}$
$r_0$	• • •	• • •	• • •
:	•	•	•
$r_f$	• • •	• • •	$\{r s_f\}$
$s_0$	• • •	• • •	• • •
•	•	•	•
$s_f$	• • •	• • •	$\{r s_f\}$
$r s_f$	Ø		Ø

 ✓
 FULL SCREEN

 CLOSE
 96 OF 721





$N_{r.s}$	Input Symbol		
State	a	• • •	$\varepsilon$
$r_0$	• • •	• • •	• • •
· ·	•	•	•
$r_f$	• • •	• • •	$\{s_0\}$
$s_0$	• • •	• • •	• • •
· ·	•	•	•
Sf	• • •	• • •	• • •

Notice that the initial state of  $N_{r.s}$  is  $r_0$  and the final state is  $s_f$  in this case.

44







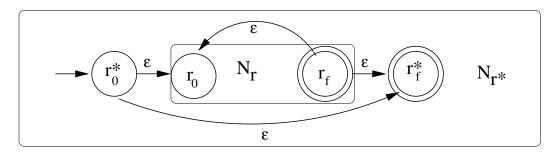




Full Screen

Close





$N_{r^*}$	Input Symbol		
State	a	• • •	$\varepsilon$
$r_0^*$	Ø	• • •	$f(r_0, r_f^*)$
$r_0$	• • •	• • •	• • •
•	•	•	•
$r_f$	• • •	• • •	$\{r_0, r_f^*\}$
$r_f^*$	Ø	Ø	Ø

PL December 28, 2021

Go Back Full Screen

Close

## Regular expressions vs. NFAs



• It is obvious that for each regular expression r, the corresponding NFA  $N_r$  is correct by construction i.e.

$$\mathcal{L}(N_r) = \mathcal{L}(r)$$

- Each regular expression operator
  - adds at most 2 new states and
  - adds at most 4 new transitions
- ullet Every state of each  $N_r$  so constructed has
  - either 1 outgoing transition on a symbol from A
  - —or at most 2 outgoing transitions on  $\varepsilon$
- Hence  $N_r$  has at most 2|r| states and 4|r| transitions.

44

PL December 28. 2021

Go Back

Full Screen

Close



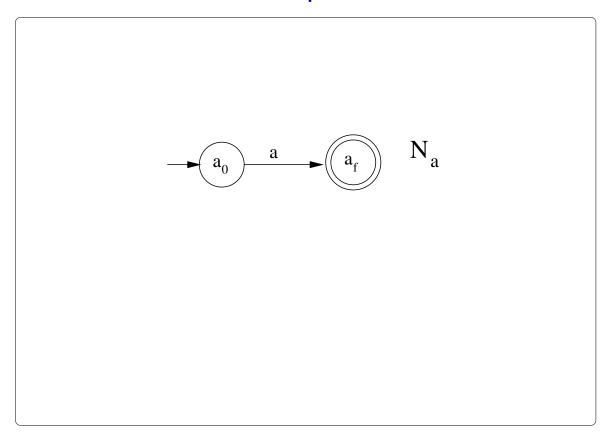
#### Example

We construct a NFA for the regular expression  $(\mathbf{a}|\mathbf{b})^*\mathbf{a}\mathbf{b}\mathbf{b}$ .

- Assume the alphabet  $A = \{a, b\}$ .
- We follow the steps of the construction as given in Constructing NFA to Regular Expressions to NFAs:5
- For ease of understanding we use the regular expression itself (subscripted by 0 and f respectively) to name the two new states created by the regular expression operator.

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 100 OF 721





Steps in NFA for  $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$ 









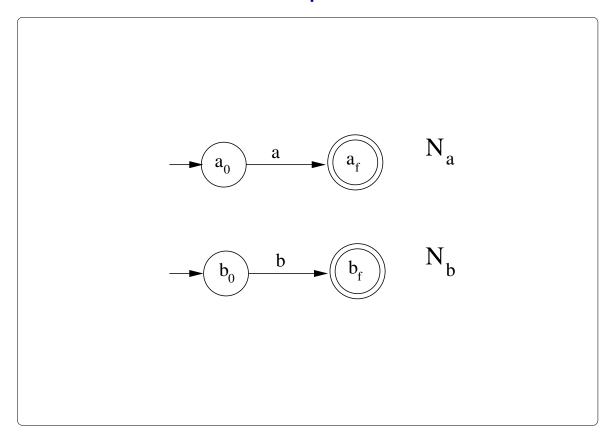






Close





Steps in NFA for  $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$ 









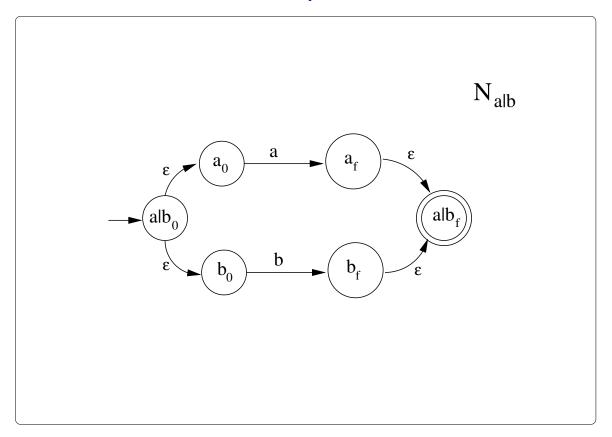






Close

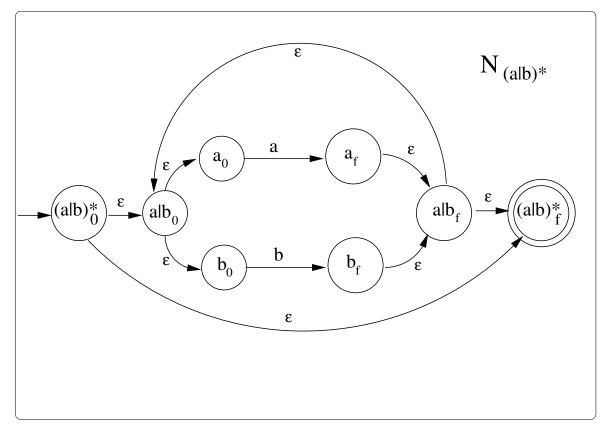




Steps in NFA for  $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$ 





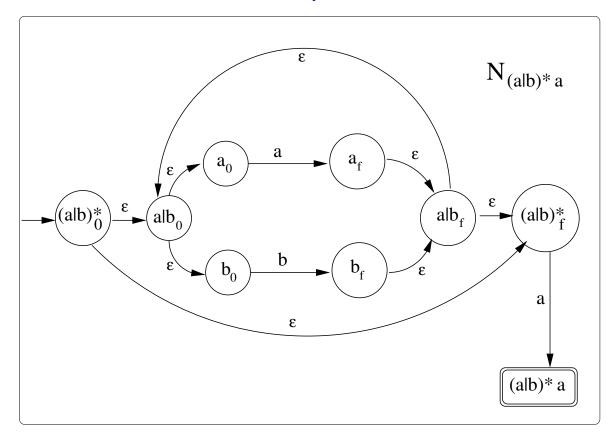


Steps in NFA for  $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$ 

 ✓
 Full Screen

 Close
 104 of 721





Steps in NFA for  $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$ 

 ■
 ■
 December 28, 2021

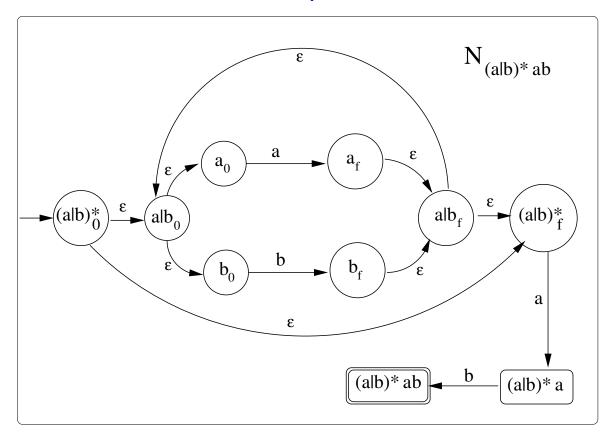
Go Back

Full Screen

Close

105 of 721





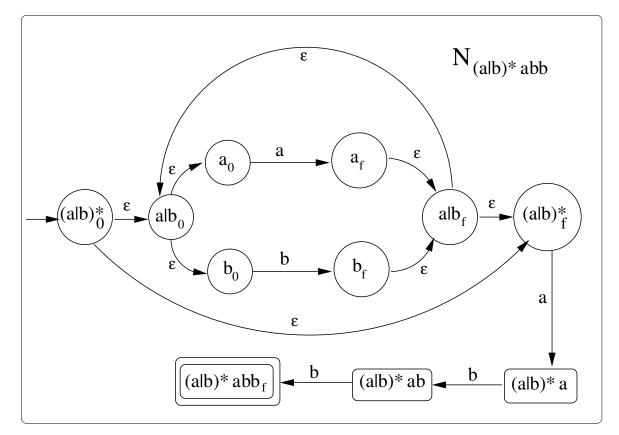
Steps in NFA for  $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$ 

 ✓
 Full Screen

 Close
 106 of 721



## Example-final



Steps in NFA for  $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$ 

 ✓
 Full Screen

 Close
 107 of 721

#### **Extensions**



We have provided constructions for only the most basic operators on regular expressions. Here are some extensions you can attempt

- 1. Show how to construct a NFA for ranges and multiple ranges of symbols
- 2. Assuming  $N_r$  is a NFA for the regular expression r, how will you construct the NFA  $N_{r^+}$ .
- 3. Certain languages like Perl allow an operator like  $r\{k,n\}$ , where

$$\mathcal{L}(r\{k,n\}) = \bigcup_{k \le m \le n} \mathcal{L}(r^m)$$

Show to construct  $N_{r\{k,n\}}$  given  $N_r$ .

4. Consider a new regular expression operator  $\hat{}$  defined by  $\mathcal{L}(\hat{}r) = A^* - \mathcal{L}(r)$  What is the automaton  $N_r$  given  $N_r$ ?

44

**>>** 

Go Back

Full Screen

Close



# Scanning Using NFAs



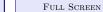








Go Back





# Scanning and Automata

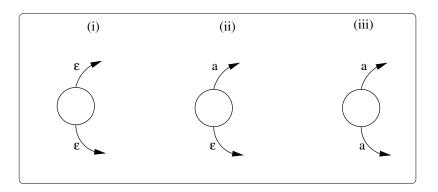
- Scanning is the only phase of the compiler in which every character of the source program is read
- The scanning phase therefore needs to be defined accurately and efficiently.
- Accuracy is achieved by regular expression specification of the tokens
- Efficiency implies that the input should not be read more than once.

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 110 OF 721

## Nondeterminism and Token Recognition



• The three kinds of nondeterminism in the NFA construction are depicted in the figure below.



- (i) It is difficult to know which  $\varepsilon$  transition to pick without reading any further input
- (ii) For two transitions on the same input symbol a it is difficult to know which of them would reach a final state on further input.
- (iii) Given an input symbol a and an  $\varepsilon$  transition on the current state it is impossible to decide which one to take without looking at further input.

**◄**◄





PL December 28, 2021

Go Back

FULL SCREEN

Close

# dse

#### Nondeterministic Features

- In general it is impossible to recognize tokens in the presence of nondeterminism without *backtracking*.
- Hence NFAs are not directly useful for scanning because of the presence of nondeterminism.
- The nondeterministic feature of the construction of  $N_r$  for any regular expression r is in the  $\varepsilon$  transitions.
- ullet The arepsilon transitions in any automaton refer to the fact that no input character is consumed in the transition.
- Backtracking usually means algorithms involving them are very complex and hence inefficient.
- To avoid backtracking, the automaton should be made deterministic

44

4

.

PL December 28, 2021

Go Back

Full Screen

Close



#### From NFA to DFA

- Since the only source of nondeterminism in our construction are the  $\varepsilon$ , we need to eliminate them without changing the language recognized by the automaton.
- Two consecutive  $\varepsilon$  transitions are the same as one. In fact any number of  $\varepsilon$  transitions are the same as one. So as a first step we compute all finite sequences of  $\varepsilon$  transitions and collapse them into a single  $\varepsilon$  transition.
- Two states q, q' are equivalent if there are only  $\varepsilon$  transitions between them. This is called the  $\varepsilon$ -closure of states.

 ✓
 ✓

 PL December 28, 2021
 Go Back
 Full Screen
 Close

 113 of 721

#### $\varepsilon$ -Closure

Given a set T of states, then  $T_{\varepsilon} = \varepsilon \text{-}closure(T)$  is the set of states which either belong to T or can be reached from states belonging to T only through a sequence of  $\varepsilon$  transitions.

# Algorithm 3.1

$$\varepsilon$$
-Closure  $(T) \stackrel{df}{=}$ 

**Require:**  $T \subseteq Q$  of NFA  $N = \langle Q, A \cup \{\varepsilon\}, q_0, F, \longrightarrow \rangle$ 

Ensures:  $T_{\varepsilon} = \varepsilon$ -Closure(T)

$$T_{\varepsilon} := T;$$

# repeat

$$\left\{ \begin{array}{l} \textbf{repeat} \\ T_{\varepsilon}' := T_{\varepsilon}; T_{\varepsilon} := T_{\varepsilon}' \cup \{q' \mid q' \not\in T_{\varepsilon}', \exists q \in T_{\varepsilon}' : q \xrightarrow{\varepsilon} q'\} \\ \textbf{until } T_{\varepsilon} = T_{\varepsilon}' \end{array} \right.$$









# Analysis of $\varepsilon$ -Closure

- If  $T = \emptyset$  then  $T_{\varepsilon} = T$  in the first iteration.
- ullet  $T_{arepsilon}$  can only grow in size through each iteration
- ullet The set  $T_{\mathcal{E}}$  cannot grow beyond the total set of states Q which is finite. Hence the algorithm always terminates for any NFA N.
- Time complexity: O(|Q|).

44

▶▶

Go Back

Full Screen

Close



# Recognition using NFA

The following algorithm may be used to recognize a string using a NFA. In the algorithm we extend our notation for targets of transitions to include sets of sources. Thus

$$S \xrightarrow{a} = \{q' \mid \exists q \in S : q \xrightarrow{a} q'\}$$

and

$$\varepsilon\text{-Closure}(S \xrightarrow{a}) = \bigcup_{\substack{q' \in S \xrightarrow{a}}} \varepsilon\text{-Closure}(q')$$

 ✓
 ✓

 FULL SCREEN

 CLOSE

 116 OF 721



# Recognition using NFA: Algorithm

# Algorithm 3.2

ACCEPT 
$$(N, x) \stackrel{df}{=}$$

**Require:** NFA  $N = \langle Q, A \cup \{\varepsilon\}, q_0, F, \longrightarrow \rangle$ , a lexeme x

**Ensures:** Boolean

$$S := \varepsilon\text{-}Closure(q_0); a := nextchar(x);$$

while  $a \neq end\_of\_string$ 

$$\mathbf{do} \ \begin{cases} S := \varepsilon\text{-}\mathbf{CLOSURE}(S \xrightarrow{a}); \\ a := \mathbf{nextchar}(x) \end{cases}$$
 return  $(S \cap F \neq \emptyset)$ 





Go Back

Full Screen

Close



# Analysis of Recognition using NFA

- Even if  $\varepsilon$ -closure is computed as a call from within the algorithm, the time taken to recognize a string is bounded by  $O(|x|.|Q_{N_r}|)$  where  $|Q_{N_r}|$  is the number of states in  $N_r$ .
- The space required for the automaton is at most O(|r|).
- Given that  $\varepsilon$ -closure of each state can be pre-computed knowing the NFA, the recognition algorithm can run in time linear in the length of the input string x i.e. O(|x|).
- Knowing that the above algorithm is deterministic once  $\varepsilon$ -closures are precomputed one may then work towards a *Deterministic* automaton to reduce the space required.

44

•

▶▶

Go Back

Full Screen

Close







#### 3.3. Deterministic Finite Automata (DFA)

## Conversion of NFAs to DFAs













Full Screen

Close



#### Deterministic Finite Automata

- A deterministic finite automaton (DFA) is a NFA in which
  - 1. there are no transitions on  $\varepsilon$  and
  - 2.  $\longrightarrow$  yields a *at most one* target state for each source state and symbol from A i.e. the transition relation is no longer a relation but a *function*<sup>a</sup>

$$\delta:Q\times \mathtt{A}\to Q$$

• Clearly if every regular expression had a DFA which accepts the same language, all backtracking could be avoided.

44

**•** 

**▶**▶

Go Back

Full Screen

Close

<sup>&</sup>lt;sup>a</sup>Also in the case of the NFA the relation  $\longrightarrow$  may not define a transition from every state on every letter

#### Transition Tables of NFAs



We may think of a finite-state automaton as being defined by a 2-dimensional table of size  $|Q| \times |A|$  in which for each state and each letter of the alphabet there is a set of possible *target* states defined. In the case of a non-deterministic automaton,

- 1. for each state there could be  $\varepsilon$  transitions to
  - (a) a set consisting of a single state or
- (b) a set consisting of more than one state.
- 2. for each state q and letter a, there could be
  - (a) an empty set of target states or
  - (b) a set of target states consisting of a single state or
  - (c) a set of target states consisting of more than one state

44

**>>** 

PL December 28. 2021

Go Back

Full Screen

Close



#### Transition Tables of DFAs

In the case of a deterministic automaton

- 1. there are no  $\varepsilon$  transitions, and
- 2. for each state q and letter a
  - (a) either there is no transition (in which case we add a new "sink" state which is a non-accepting state)
  - (b) or there is a transition to a unique state q'.

The recognition problem for the same language of strings becomes simpler and would work faster (it would have no back-tracking) if the NFA could be converted into a DFA accepting the same language.

44

1

**>>** 

PL December 28, 2021

Go Back

Full Screen

Close



#### NFA to DFA

Let  $N=\langle Q_N, {\tt A}\cup\{\varepsilon\}, s_N, F_N, \longrightarrow_N \rangle$  be a NFA . We would like to construct a DFA  $D=\langle Q_D, {\tt A}, s_D, F_D, \longrightarrow_D \rangle$  where

- $ullet Q_D$  the set of states of the DFA
- $\bullet$  A the alphabet (notice there is no  $\varepsilon$ ),
- $s_D \in Q_D$  the start state of the DFA,
- ullet  $F_D$  the final or accepting states of the DFA and
- $\bullet \ \delta_D: Q_D \times \mathbb{A} \longrightarrow Q_D$  the transition function of the DFA.

We would like  $\mathcal{L}(N) = \mathcal{L}(D)$ 

44

I

PL December 28, 2021



#### The Subset Construction

PL December 28, 2021

Non-determinism.

 $\varepsilon$ -closure

Subsets of NFA states.

Acceptance.

44





Go Back

Full

Full Screen

Close



#### The Subset Construction: Non-determinism

Non-determinism A major source of non-determinism in NFAs is the presence of  $\varepsilon$  transitions. The use of  $\varepsilon$ -CLOSURE creates a cluster of "similar" states<sup>a</sup>.

 $\varepsilon$ -closure.

Subsets of NFA states.

Acceptance.

44

**•** 

**>>** 

Go Back

FULL SCREEN

Close

127 of 721

<sup>&</sup>lt;sup>a</sup>Two states are "similar" if they are reached from the start state by the same string of symbols from the alphabet



#### The Subset Construction: $\varepsilon$ -closure

#### Non-determinism.

 $\varepsilon$ -closure. The  $\varepsilon$ -closure of each NFA state is a set of NFA states with "similar" behaviour, since they make their transitions on the same input symbols though with different numbers of  $\varepsilon$ s.

Subsets of NFA states.

Acceptance.

**⋖**◆





#### The Subset Construction: Subsets of NFA states

Non-determinism.

 $\varepsilon$ -closure.

Subsets of NFA states. Each state of the DFA refers to a subset of states of the NFA which exhibit "similar" behaviour. Similarity of behaviour refers to the fact that they accept the same input symbols. The behaviour of two different NFA states may not be "identical" because they may have different numbers of  $\varepsilon$  transitions for the same input symbol.

Acceptance

**∢**∢





## The Subset Construction: Acceptance

Non-determinism

 $\varepsilon$ -closure.

Subsets of NFA states

Acceptance. Since the notion of acceptance of a string by an automaton, implies finding an accepting sequence even though there may be other *non-accepting sequences*, the non-accepting sequences may be ignored and those non-accepting states may be clustered with the accepting states of the NFA. So two different states reachable by the same sequence of symbols may be also thought to be similar.

 ✓
 ✓

 PL December 28, 2021

 Go Back
 Full Screen

 Close
 130 of 721



```
Algorithm 3.3
 NFATODFA (N) \stackrel{df}{=}
           Requires: NFA N = \langle Q_N, A \cup \{\varepsilon\}, s_N, F_N, \longrightarrow_N \rangle
           Yields: DFA D = \langle Q_D, A, s_D, F_D, \delta_D \rangle with \mathcal{L}(N) = \mathcal{L}(D)
      s_D := \varepsilon\text{-CLOSURE}(\{s_N\}); Q_D := \{s_D\}; F_D := \emptyset; \delta_D := \emptyset;
      U := \{s_D\} Note: U is the set of unvisited states of D
      while U \neq \emptyset
                      Choose any q_D \in U; U := U - \{q_D\}; Note: q_D \subseteq Q_N
                       for each a \in A
                      \mathbf{do} \begin{cases} q'_D := \varepsilon\text{-}\mathbf{CLOSURE}(q_D \xrightarrow{a}_N); \delta_D(q_D, q) := q'_D \\ \mathbf{if} \ q'_D \cap F_N \neq \emptyset \\ \mathbf{then} \ F_D := F_D \cup \{q'_D\}; \\ \mathbf{if} \ q'_D \notin Q_D \\ \mathbf{then} \ \begin{cases} Q_D := Q_D \cup \{q'_D\}; \\ U := U \cup \{q'_D\} \end{cases} \end{cases}
```

44

PL December 28, 2021 Go BACK

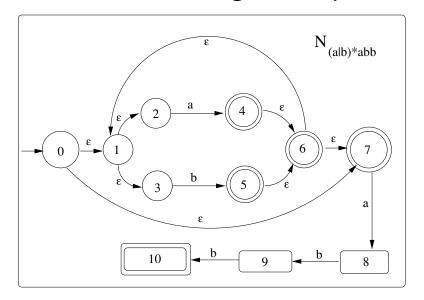
Full Screen

Close



# Example-NFA

Consider the NFA constructed for the regular expression  $(\mathbf{a}|\mathbf{b})^*\mathbf{a}\mathbf{b}\mathbf{b}$ .



and apply the NFA to DFA construction algorithm

# **Determinising**



# $N_{(\mathbf{a}|\mathbf{b})^*\mathbf{abb}}$

 $D_{(\mathbf{a}|\mathbf{b})^*\mathbf{abb}}$ 

$$EC_0 = \varepsilon$$
-Closure $(0) = \{0, 1, 2, 3, 7\}$ 

$$2 \xrightarrow{a}_N 4$$
 and  $7 \xrightarrow{a}_N 8$ . So  $EC_0 \xrightarrow{a}_D \varepsilon$ -CLOSURE $(4,8) = EC_{4,8}$ . Similarly

$$EC_0 \xrightarrow{b}_D \varepsilon$$
-Closure(5) =  $EC_5$ 

$$EC_{4.8} = \varepsilon$$
-Closure $(4, 8) = \{4, 6, 7, 1, 2, 3, 8\}$ 

$$EC_5 = \varepsilon$$
-Closure(5) =  $\{5, 6, 7, 1, 2, 3\}$ 

$$EC_5 \xrightarrow{a}_D \varepsilon$$
-CLOSURE $(4,8) = EC_{4,8}$  and  $EC_5 \xrightarrow{b}_D \varepsilon$ -CLOSURE $(5)$ 

$$EC_{4,8} \xrightarrow{a}_{D} \varepsilon\text{-CLOSURE}(4,8) = EC_{4,8} \text{ and } EC_{4,8} \xrightarrow{b}_{D} \varepsilon\text{-CLOSURE}(5,9) = EC_{5,9}$$

$$EC_{5,9} = \varepsilon$$
-Closure $(5,9) = \{5,6,7,1,2,3,9\}$ 

$$EC_{5,9} \xrightarrow{a}_{D} \varepsilon\text{-CLOSURE}(4,8) = EC_{4,8} \text{ and } EC_{5,9} \xrightarrow{b}_{D} \varepsilon\text{-CLOSURE}(5,10) = EC_{5,10}$$

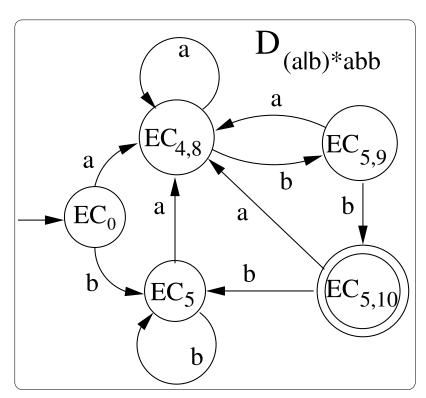
$$EC_{5,10} = \varepsilon$$
-Closure $(5, 10) = \{5, 6, 7, 1, 2, 3, 10\}$ 

$$EC_{5,10} \xrightarrow{a}_{D} \varepsilon\text{-CLOSURE}(4,8) \text{ and } EC_{5,10} \xrightarrow{b} \varepsilon\text{-CLOSURE}(5)$$

PL December 28, 2021







$D_{(\mathbf{a} \mathbf{b})^*\mathbf{abb}}$	Input Symbol	
State	a	b
$EC_0$	$EC_{4,8}$	$EC_5$
$EC_{4,8}$	$EC_{4,8}$	$EC_{5,9}$
$EC_5$	$EC_{4,8}$	$EC_5$
$EC_{5,9}$	$EC_{4,8}$	$EC_{5,10}$
$EC_{5,10}$	$EC_{4,8}$	$EC_5$

**↑** PL December 28, 2021

Go Back

Full Screen

Close

## Recognition using DFA

The following algorithm may be used to recognize a string using a DFA. Compare it with the algorithm for recognition using an NFA.

# Algorithm 3.4 ACCEPT $(D, x) \stackrel{df}{=}$ **Requires:** DFA $D = \langle Q, A, q_0, F, \delta \rangle$ , a lexeme $x \in A^*$ **Ensures:** Boolean $S := q_0; a := \mathtt{nextchar}(x);$ while $a \neq end\_of\_string$ $\mathbf{do} \ \begin{cases} \dot{S} := \delta(S, a); \\ a := \mathrm{nextchar}(x) \end{cases}$ $\mathbf{return}\ (S \in F)$

PL December 28, 2021

Go Back

Full Screen

CLOS



# Analysis of Recognition using DFA

- The running time of the algorithm is O(|x|).
- The space required for the automaton is O(|Q|.|A|).

44

•

▶▶

PL December 28, 2021

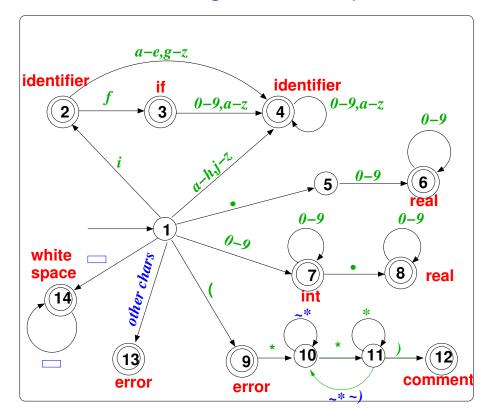
Go Back

FULL SCREEN

Close



# Scanning With output



The Big Picture













Full Screen

Close



#### DFA vis-a-vis Scanner

A scanner differs from a simple DFA in the following ways.

- A DFA may simply reach a non-accepting state in case of an unrecognizable lexeme. A scanner on the other hand needs to accept the lexeme and raise an error and proceed to the next lexeme.
- A DFA simply accepts a token or rejects a lexeme. A scanner needs to recognize and classify every token and lexeme.
- Where a token is allowed as a prefix of another (see the case of "if") scanners choose the longest lexeme that is an identifiable token.
- DFAs are often "minimised" to collapse all accepting states into one accepting state. This is not desirable in the case of scanner since tokens need to be classified separately based on the accepting states.

44

**•** 

▶▶

Go Back

Full Screen

Close



#### Exercise 3.3

- 1. Write a regular expression to specify all numbers in binary form that are multiples of of 4.
- 2. Write regular expressions to specify all numbers in binary form that are not multiples of 4.
- 3. Each comment in the C language
  - begins with the characters "//" and ends with the newline character, or
  - begins with the characters "/\*" and ends with "\*/" and may run across several lines.
  - (a) Write a regular expression to recognize comments in the C language.
  - (b) Transform the regular expression into a NFA.
  - (c) Transform the NFA into a DFA.
  - (d) Explain why most programming languages do not allow nested comments.
  - (e) modified C comments. If the character sequences "//", "/\*" and "\*/" are allowed to appear in 'quoted' form as "','/'", "',\*" and "'\*/" respectively within a C comment, then give
    - i. a modified regular expression for C comments
    - ii. a NFA for these modified C comments

44

PL December 28, 2021

Go Back

Full Screen

Close



#### iii. a corresponding DFA for modified C comments

- 4. Many systems such as Windows XP and Linux recognize commands, filenames and folder names by the their shortest unique prefix. Hence given the 3 commands chmod, chgrp and chown, their shortest unique prefixes are respectively chm, chg and cho. A user can type the shortest unique prefix of the command and the system will automatically complete it for him/her.
  - (a) Draw a DFA which recognizes all prefixes that are at least as long as the shortest unique prefix of each of the above commands.
  - (b) Suppose the set of commands also includes two more commands cmp and cmpdir, state how you will include such commands also in your DFA where one command is a prefix of another.

PL December 28, 2021

44

•



→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 141 OF 721



- 4. Parsing or Syntax Analysis
- 4.1. Grammars

Parsing Or Syntax Analysis







PL December 28, 2021

Go Васк

Full Screen

Close



143 of 721

## Generating a Language

Consider the DFA constructed earlier to accept the language defined by the regular expression  $(\mathbf{a}|\mathbf{b})^*\mathbf{a}\mathbf{b}\mathbf{b}$ . We rename the states for convenience.

$D_{(\mathbf{a} \mathbf{b})^*\mathbf{abb}}$	Input	
State	a	р
S	A	B
A	A	C
B	A	B
C	A	D
D	A	B

We begin by rewriting each of the transitions as follows.

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE



#### Production rules

$$S \longrightarrow aA \mid bB$$

$$A \longrightarrow aA \mid bC$$

$$B \longrightarrow aA \mid bB$$

$$C \longrightarrow aA \mid bD$$

$$D \longrightarrow aA \mid bB \mid \varepsilon$$

and think of each of the symbols S, A, B, C, D as generating symbols and thus producing (rather than consuming strings). For example, the strings abb and aabbabb are generated by the above production rules as follows.

$$S \Rightarrow aA \Rightarrow abC \Rightarrow abbD \Rightarrow abb$$
  
 $S \Rightarrow aA \Rightarrow aaA \Rightarrow aabC \Rightarrow aabbD$   
 $\Rightarrow aabbaA \Rightarrow aabbabC \Rightarrow aabbabD \Rightarrow aabbabb$ 

44

**•** 

**>>** 

Go Back

Full Screen

Close



# Formal languages: Definition, Recognition, Generation

There are three different processes used in dealing with a formal language.

**Definition**: Regular expressions is a formal (functional programming) language used to define or specify a formal language of tokens.

Recognition: Automata are the standard mechanism used to recognize words/phrases of a formal language. An automaton is used to determine whether a given word/phrase is a member of the formal language defined in some other way.

Generation: Grammars are used to define the generation of the words/phrases of a formal language.

PL December 28, 2021

Go Back

Full Screen

Close

#### Non-regular language



Consider the following two languages over an alphabet  $A = \{a, b\}$ .

$$R = \{a^n b^n | n < 100\} 
 P = \{a^n b^n | n > 0\}$$

- ullet may be finitely represented by a regular expression (even though the actual expression is very long).
- $\bullet$  However, P cannot actually be represented by a regular expression
- A regular expression is not powerful enough to represent languages which require parenthesis matching to arbitrary depths.
- All high level programming languages require an underlying language of expressions which require parentheses to be nested and matched to arbitrary depth.

PL December 28, 2021

44

•

**▶**▶

Go Back

Full Screen

Close



 ✓
 Full Screen

 Close
 147 of 721





#### Grammars

**Definition 4.1** A grammar  $G = \langle N, T, P, S \rangle$  consists of

- a set N of nonterminal symbols, or variables,
- a start symbol  $S \in N$ ,
- a set T of terminal symbols or the alphabet,
- a set P of productions or rewrite rules where each rule is of the form  $\alpha \to \beta$  for  $\alpha, \beta \in (N \cup T)^*$

**Definition 4.2** Given a grammar  $G = \langle N, T, P, S \rangle$ , any  $\alpha \in (N \cup T)^*$  is called a sentential form. Any  $x \in T^*$  is called a sentence<sup>a</sup>.

Note. Every sentence is also a sentential form.

44











Full Screen

Close

<sup>&</sup>lt;sup>a</sup>some authors call it a word. However we will reserve the term word to denote the tokens of a programming language.



#### Grammars: Notation

- Upper case roman letters  $(A, B, \dots, X, Y, \text{ etc.})$  denote nonterminals.
- Final upper case roman letters (X,Y,Z) etc.) may also be used as metavariables which denote arbitrary non-terminal symbols of a grammar.
- Initial lower case roman letters (a, b, c etc.) will be used to denote terminal symbols.
- Lower case greek letters ( $\alpha$ ,  $\beta$  etc.) denote sentential forms (or even sentences).
- Final lower case letters  $(u, v, \ldots, x, y, z \text{ etc.})$  denote only sentences.
- In each case the symbols could also be decorated with sub-scripts or super-scripts.

44

**>>** 

PL December 28. 2021

Go Back

Full Screen

Close



#### Context-Free Grammars: Definition

**Definition 4.3** A grammar  $G = \langle N, T, P, S \rangle$  is called context-free if each production is of the form  $X \longrightarrow \alpha$ , where

- $\bullet X \in N$  is a nonterminal and
- $\alpha \in (N \cup T)^*$  is a sentential form.
- The production is terminal if  $\alpha$  is a sentence

Go Back

Full Screen

Close

#### CFG: Example 1



 $G = \langle \{S\}, \{a,b\}, P, S \rangle$ , where  $S \longrightarrow ab$  and  $S \longrightarrow aSb$  are the only productions in P.

**Derivations** look like this:

•

$$S \Rightarrow ab$$

$$S \Rightarrow aSb \Rightarrow aabb$$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$$

•

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb$$

PL December 28, 2021

The first three derivations are complete while the last one is partial

44

**•** 

▶▶

Go B

Full S

EEN

Close



#### **Derivations**

**Definition 4.4** A (partial) derivation (of length  $n \in \mathbb{N}$ ) in a context-free grammar is a finite sequence of the form

$$\alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \alpha_n$$
 (3)

where each  $\alpha_i \in (N \cup T)^*$  ( $0 \le i \le n$ ) is a sentential form where  $\alpha_0 = S$  and  $\alpha_{i+1}$  is obtained by applying a production rule to a non-terminal symbol in  $\alpha_i$  for  $0 \le i < n$ .

**Notation.**  $S \Rightarrow^* \alpha$  denotes that there exists a derivation of  $\alpha$  from S.

**Definition 4.5** The derivation (3) is complete if  $\alpha_n \in T^*$  i.e.  $\alpha_n$  is a sentence. Then  $\alpha_n$  is said to be a sentence generated by the grammar.

 ✓
 Image: Second of the properties of the



### Language Generation

**Definition 4.6** The language generated by a grammar G is the set of sentences that can be generated by G and is denoted  $\mathcal{L}(G)$ .

**Example 4.7**  $\mathcal{L}(G)$ , the language generated by the grammar G is  $\{a^nb^n|n>0\}$ . Prove using induction on the length of derivations.

→ → PL December 28, 2021 Go BACK FULL SCREEN CLOSE 154 OF 721



### Regular Grammars

**Definition 4.8** A production rule of a context-free grammar is

**Right Linear:** *if it is of the form*  $X \longrightarrow a$  *or*  $X \longrightarrow aY$ 

**Left Linear:** if it is of the form  $X \longrightarrow a$  or  $X \longrightarrow Ya$ 

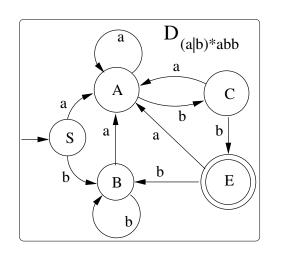
where  $a \in T$  and  $X, Y \in N$ .

**Definition 4.9** A regular grammar is a context-free grammar whose productions are either only right linear or only left linear.

 →
 Image: Second of the properties of the



# DFA to Regular Grammar



$D_{(\mathbf{a} \mathbf{b})^*\mathbf{abb}}$	Input		RLG
State	a	b	Rules
S	A	B	$S \to aA bB$
A	A	C	$A \to aA bC$
B	A	B	$B \to aA bB$
C	A	E	$C \to aA bE b$
E	A	C	$E \to aA bC$

Consider the DFA with the states renamed as shown above. We could easily convert the DFA to a right linear grammar which generates the language accepted by the DFA.

#### CFG: Empty word



 $G = \langle \{S\}, \{a,b\}, P, S \rangle$ , where  $S \longrightarrow SS \mid aSb \mid \varepsilon$  generates all sequences of matching nested parentheses, including the empty word  $\varepsilon$ .

A leftmost derivation might look like this:

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSb \dots$$

A rightmost derivation might look like this:

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow SSS \Rightarrow SaSb \Rightarrow Sab \Rightarrow aSbab \dots$$

Other derivations might look like God alone knows what!

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow SS \Rightarrow \dots$$

Could be quite confusing!

Pl Docombor 28, 2021

Go Back

Full Screen

Close



#### CFG: Derivation trees 1

#### Derivation sequences

- put an artificial order in which productions are fired.
- instead look at trees of derivations in which we may think of productions as being fired in parallel.
- There is then no highlighting in red to determine which copy of a nonterminal was used to get the next member of the sequence.
- ullet Of course, generation of the empty word  $\varepsilon$  must be shown explicitly in the tree.

PL December 28, 2021

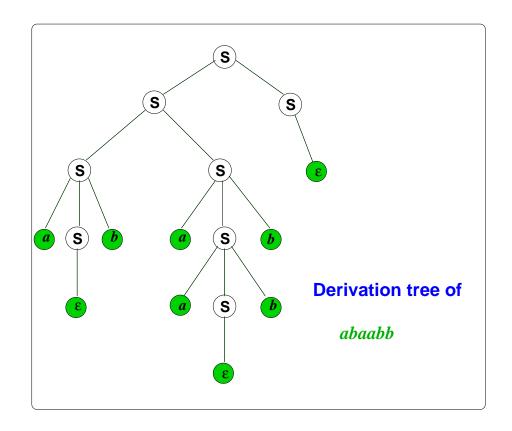
Go Back

Full Screen

Close



#### CFG: Derivation trees 2



∢.

**>>** 

PL December 28, 2021

Go Back

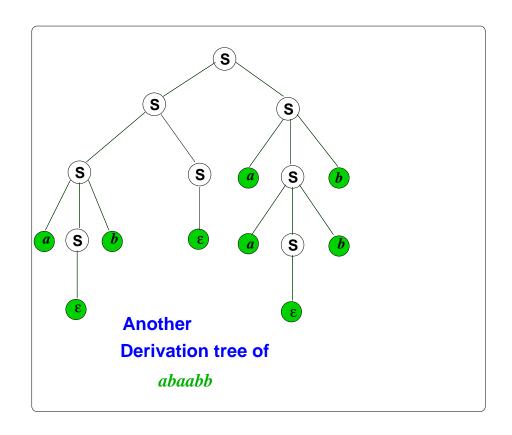
Full Screen

Close



160 of 721

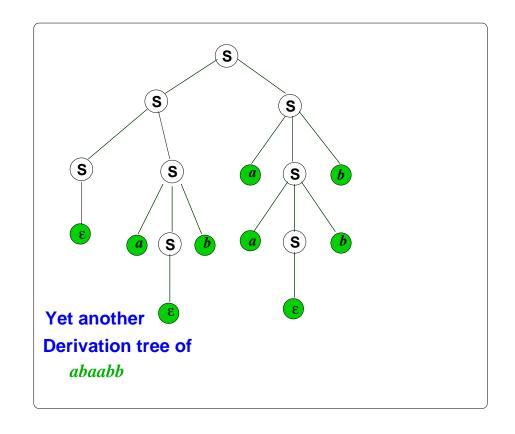
#### CFG: Derivation trees 3







#### CFG: Derivation trees 4



•

**•** 

PL December 28, 2021

Go Back

Full Screen

Close





#### 4.3. Ambiguity

# **Ambiguity Disambiguation**











Full Screen

Close

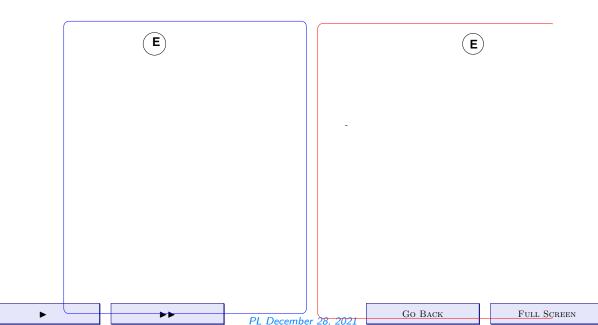
 $G_1 = \langle \{E, I, C\}, \{\mathbf{y}, \mathbf{z}, \mathbf{4}, *, +\}, P_1, \{E\} \rangle$  where  $P_1$  consists of the following productions.

$$E \rightarrow I \mid C \mid E+E \mid E*E$$

$$I \rightarrow \mathbf{y} \mid \mathbf{z}$$

$$C \rightarrow 4$$

Consider the sentence y + 4 \* z.



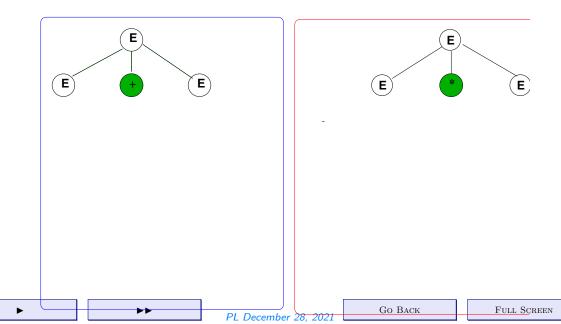
 $G_1 = \langle \{E, I, C\}, \{\mathbf{y}, \mathbf{z}, \mathbf{4}, *, +\}, P_1, \{E\} \rangle$  where  $P_1$  consists of the following productions.

$$E \rightarrow I \mid C \mid E+E \mid E*E$$

$$I \rightarrow \mathbf{y} \mid \mathbf{z}$$

$$C \rightarrow 4$$

Consider the sentence y + 4 \* z.



Close

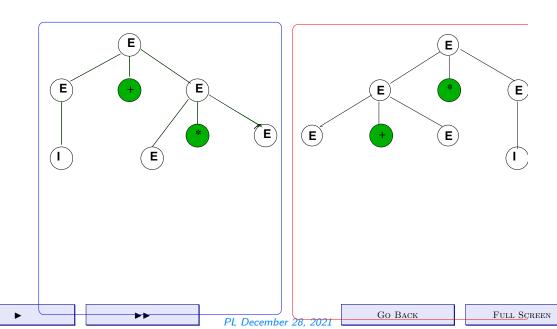
 $G_1 = \langle \{E, I, C\}, \{\mathbf{y}, \mathbf{z}, \mathbf{4}, *, +\}, P_1, \{E\} \rangle$  where  $P_1$  consists of the following productions.

$$E \rightarrow I \mid C \mid E+E \mid E*E$$

$$I \rightarrow \mathbf{y} \mid \mathbf{z}$$

$$C \rightarrow 4$$

Consider the sentence y + 4 \* z.



CLOSE

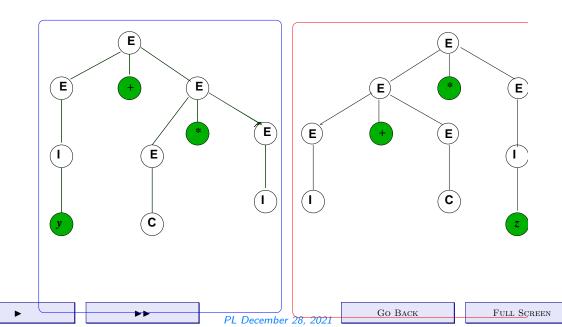
 $G_1 = \langle \{E, I, C\}, \{\mathbf{y}, \mathbf{z}, \mathbf{4}, *, +\}, P_1, \{E\} \rangle$  where  $P_1$  consists of the following productions.

$$E \rightarrow I \mid C \mid E+E \mid E*E$$

$$I \rightarrow \mathbf{y} \mid \mathbf{z}$$

$$C \rightarrow 4$$

Consider the sentence y + 4 \* z.



CLOSE

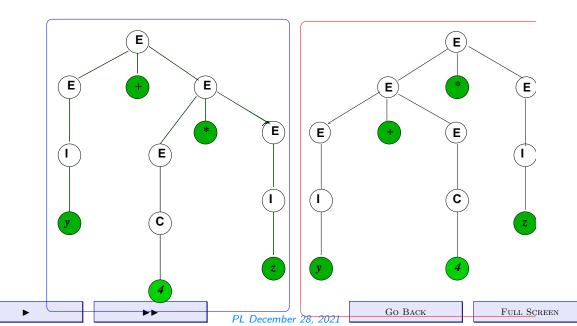
 $G_1 = \langle \{E, I, C\}, \{\mathbf{y}, \mathbf{z}, \mathbf{4}, *, +\}, P_1, \{E\} \rangle$  where  $P_1$  consists of the following productions.

$$E \rightarrow I \mid C \mid E+E \mid E*E$$

$$I \rightarrow \mathbf{y} \mid \mathbf{z}$$

$$C \rightarrow \mathbf{4}$$

Consider the sentence y + 4 \* z.



CLOSE



#### Left-most Derivation 1

Left-most derivation of y+4\*z corresponding to the *first* derivation tree.

$$E \Rightarrow E + E \Rightarrow E + E \Rightarrow I + E \Rightarrow y + E * E \Rightarrow y + E * E \Rightarrow y + C * E \Rightarrow y + 4 * E \Rightarrow y + 4 * I \Rightarrow y + 4 * Z$$

**⋖**∢

**|** 

**▶**▶

PL December 28, 2021

Go Back

Full Screen

Close



#### Left-most Derivation 2

Left-most derivation of y+4\*z corresponding to the second derivation tree.

$$E \Rightarrow E*E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow I+E*E \Rightarrow y+E*E \Rightarrow y+C*E \Rightarrow y+4*E \Rightarrow y+4*$$

**⋖**∢

**•** 

**▶**▶

PL December 28, 2021

Go Back

Full Screen

Close



#### Right-most Derivation 1

Right-most derivation of y+4\*z corresponding to the *first* derivation tree.

**⋖**∢

▶

▶▶

PL December 28, 2021

GO BACK FULL SCREEN

Close



#### Right-most Derivation 2

Right-most derivation of y+4\*z corresponding to the second derivation tree.

$$E \Rightarrow E*E \Rightarrow E*I \Rightarrow E*I \Rightarrow E*z \Rightarrow E+E*z \Rightarrow E+C*z \Rightarrow E+4*z \Rightarrow I+4*z \Rightarrow y+4*z$$

PL December 28, 2021

**⋖**∢

**•** 

**▶**▶

Go Back

Full Screen

Close

172 of 721



# **Characterizing Ambiguity**

The following statements are equivalent.

- A CFG is ambiguous if some sentence it generates has more than one derivation tree
- A CFG is *ambiguous* if there is a some sentence it generates with more than one *left-most derivation*
- A CFG is *ambiguous* if there is a some sentence it generates with more than one *right-most derivation*

 ■
 ■

 PL December 28, 2021
 Go Back
 Full Screen
 Close
 173 of 721

# d\$E

### Ambiguity in CFLs

see Wikipedia

- Some ambiguities result from incorrect grammars, i.e. there may exist a grammar which generates the same language with unique derivation trees.
- There may be some languages which are *inherently ambiguous* i.e. there is no context-free grammar for the language with only unique derivation trees for every sentence of the language.
- Whether a given CFG is ambiguous is *undecidable* i.e. there is no algorithm which can decide whether a given context-free grammar is ambiguous.
- Whether a given context-free language is *inherently ambiguous* is also *unde-cidable* since there is no algorithm which can decide whether any CFG that generates the language is ambiguous.

 ✓
 Image: Second of the properties of the



# Removing ambiguity

There are essentially three ways adopted by programming language designers or compiler designers to remove ambiguity

- Change the language generated by introducing new bracketing tokens, (e.g. new reserved keywords begin...end).
- Introduce new precedence or associativity rules to disambiguate this will invalidate certain derivation trees and may guarantee uniqueness, (e.g. the dangling-else problem see section 4.4).
- Change the grammar of the language (without changing the language generated)

44 +

PL December 28. 2021

Go Back

Full Screen

Close

# Disambiguation



The only way to remove ambiguity (without changing the language generated for a language which is not ambiguous) is to change the grammar by introducing some more non-terminal symbols and changing the production rules  $^a$ . Consider the grammar  $G'_1 = \langle N', \{y, z, 4, *, +\}, P', \{E\} \rangle$  where  $N' = N \cup \{T, F\}$ with the following production rules P'.

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T*F \mid F$$

$$F \rightarrow I \mid C$$

$$I \rightarrow \mathbf{y} \mid \mathbf{z}$$

$$C \rightarrow 4$$

# and compare it with the grammar $G_1$





Go Back

Full Screen

Close

<sup>&</sup>lt;sup>a</sup>However the introduction of fresh non-terminals and rules may introduce new ambiguities, if the designer is not careful!





The left-most derivation of y+4\*z is then as follows.

$$E \Rightarrow E+T \Rightarrow \Rightarrow I+T \Rightarrow I+T \Rightarrow y+T*F \Rightarrow y+T*F \Rightarrow y+F*F \Rightarrow y+C*F \Rightarrow y+4*F \Rightarrow y+4*I \Rightarrow y+1*I \Rightarrow y+1*$$

44

**▶**▶

PL December 28, 2021

Go Back

Full Screen

Close



#### Left-most Derivations

Compare it with the Left-most Derivation 1.

G<sub>1</sub>. 
$$E \Rightarrow E+E \Rightarrow I+E \Rightarrow \mathbf{y}+E \Rightarrow \mathbf{y}+E*E \Rightarrow \mathbf{y}+C*E \Rightarrow \mathbf{y}+4*E \Rightarrow \mathbf{y}+4*I \Rightarrow \mathbf{y}+4*\mathbf{z}$$
  
G'<sub>1</sub>.  $E \Rightarrow E+T \Rightarrow I+T \Rightarrow \mathbf{y}+T \Rightarrow \mathbf{y}+T*F \Rightarrow \mathbf{y}+F*F \Rightarrow \mathbf{y}+C*F \Rightarrow \mathbf{y}+4*F \Rightarrow \mathbf{y}+4*I \Rightarrow \mathbf{y}+4*\mathbf{z}$ 

There is no derivation in  $G'_1$  corresponding to Left-most Derivation 2 (Why not?).

 ✓
 ✓

 Full Screen
 Close

 178 of 721

#### Right-most Derivation 1'

Right-most derivation of y+4\*z corresponding to the *first* derivation tree.

$$E \Rightarrow E+T \Rightarrow \Rightarrow E+T*F \Rightarrow E+T*I \Rightarrow E+T*z \Rightarrow E+C*z \Rightarrow E+4*z \Rightarrow F+4*z \Rightarrow I+4*z \Rightarrow y+4*z$$

Compare it with the Right-most Derivation 1.

There is no derivation corresponding to Right-most Derivation 2. CLOSE



# Disambiguation by Parenthesization

Another method of disambiguating a language is to change the language generated, by introducing suitable bracketing mechanisms.

**Example 4.10** Compare the following fully parenthesized grammar  $G_2$  (which has the extra terminal symbols ( and )) with the grammar  $G_1$  without parentheses

$$E \rightarrow I \mid C \mid (E+E) \mid (E*E)$$

$$I \rightarrow \mathbf{y} \mid \mathbf{z}$$

$$C \rightarrow \mathbf{4}$$

Though unambiguous, the language defined by this grammar is different from that of the original grammar without parentheses.

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 180 OF 721



# Associativity and Precedence

The grammar  $G'_1$  implements

**Precedence.** \* has higher precedence than +.

**Associativity.** \* and + are both left associative operators.

but is parentheses-free, whereas grammar  $G_2$  generates a different language which is unambiguous. We may combine the two with the benefits of both.

PL December 28, 2021





Go Back

Full Screen

Close



# Parenthesization, Associativity and Precedence

**Example 4.11** Compare the following parenthesized grammar  $G_2'$  which combines the benefits of both  $G_1'$  and  $G_2$  (parenthesization wherever required by implementing the bodmas rule).  $G_2' = \langle N', \{y, z, 4, *, +, (,)\}, P_2', \{E\} \rangle$  where  $N' = N \cup \{T, F\}$  with the following production rules  $P_2'$ .

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T*F \mid F$$

$$F \rightarrow I \mid C \mid (E)$$

$$I \rightarrow \mathbf{y} \mid \mathbf{z}$$

$$C \rightarrow \mathbf{4}$$

 →
 Image: Second of the policy of



### 4.4. The "dangling else" problem

- Some programming languages like FORTRAN and Assembly (conditional jumps) have a single **if**...**then** construct. We write it(b, C) to denote **if** b **then** C.
- Some programming languages like ML, OCAML have a single **if**...**then**...**else** construct and we write ite(b, C, C') to denote **if** b **then** C **else** C'.
- Many programming languages have both **if...then** and **if...then...else** constructs which potentially may lead to a dangling-else problem.

The dangling-else problem potentially is an ambiguity associated with a compound construct such as

if 
$$b_1$$
 then if  $b_2$  then  $C_1$  else  $C_2$  (4)

where  $b_1$  and  $b_2$  are boolean expressions and  $C_1$  and  $C_2$  are appropriate constructs (expressions or commands) that are allowed by the language.

The ambiguity arises because the construct (4) may be interpreted as denoting either  $it(b_1, ite(b_2, C_1, C_2))$  or  $ite(b_1, it(b_2, C_1), C_2)$ .







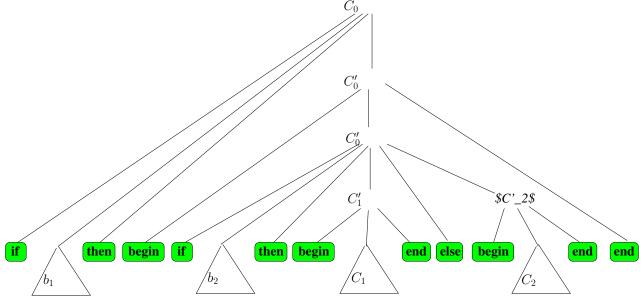


Figure 1:  $it(b_1, ite(b_2, C_1, C_2))$ 

### Disambiguation

1. Disambiguation may be achieved in the language by introducing new bracketing symbols (e.g. **begin...end**) for all constructs of the kind that C belongs to. If the use of these brackets is made mandatory in the language then the construct (4) itself would be syntactically illegal and would have to be replaced by one of the following depending upon the programmer's intention.





Go Back

Full Screen

Close



```
• If the programmer's intention corresponds to it(b_1, ite(b_2, C_1, C_2)) (see the parse tree in figure 1) then
 if b_1 then
    begin
       if b_2 then
         begin
            C_1
         end
       \mathbf{else}
         begin
            C_2
         \mathbf{end}
    end
• If programmer intended ite(b_1, it(b_2, C_1), C_2) (see the parse tree in figure 2).
 if b_1 then
    begin
       if b_2 then
         begin
            C_1
         end
    end
```

PL December 28, 2021

44







else begin  $C_2$  end

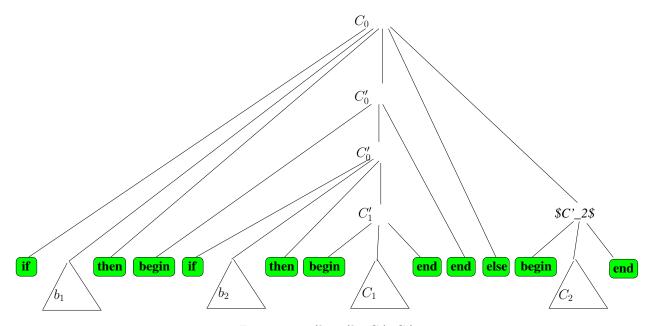


Figure 2:  $ite(b_1, it(b_2, C_1), C_2)$ 

2. While the use of **begin**... **end** is general-purpose enough for all constructs of the kind that C is, it tends to introduce too many tokens in an actual program. Some languages (e.g. Bash) instead introduce a unique *closing* token for

44

◀

•

PL December 28, 2021

Go Back

Full Screen

Close

each construct. That is, the constructs come with pairs of unique opening and closing tokens (e.g. **if...then...fi**, **if...then...else...fi**, **case...esac** etc.) In such a language the constructs corresponding to  $it(b_1, ite(b_2, C_1, C_2))$  would then be written as follows (see also the parse tree in figure 3).

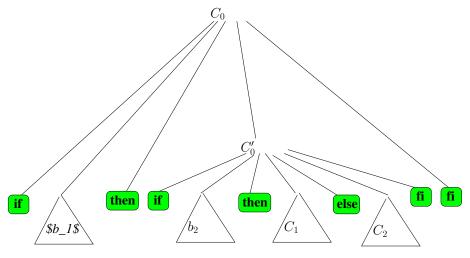


Figure 3:  $it(b_1, ite(b_2, C_1, C_2))$ 

```
egin{array}{ll} 	ext{if } b_1 	ext{ then} \\ 	ext{ } C_1 \\ 	ext{ else} \\ 	ext{ } C_2 \\ 	ext{ fi} \end{array}
```

44

**>>** 

Go Back

Full Screen

Close



while  $ite(b_1, it(b_2, C_1), C_2)$  would be written as (see also the parse tree in figure 4)

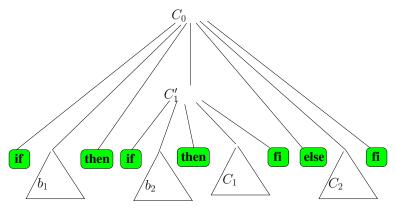


Figure 4:  $it(b_1, ite(b_2, C_1, C_2))$ 

```
\begin{array}{c} \textbf{if} \ b_1 \ \textbf{then} \\ \quad \textbf{if} \ b_2 \ \textbf{then} \\ \quad C_1 \\ \quad \textbf{fi} \\ \textbf{else} \\ \quad C_2 \\ \quad \textbf{fi} \end{array}
```

In general this solution leads to a larger number of reserved words in the language but a smaller number of tokens (produced after scanning) per syntactically valid program as opposed to the previous solution. Languages like C and

**◄**◀

4



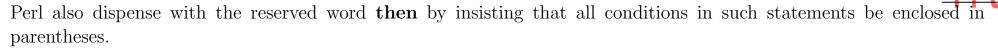
**>>** 

PL December 28, 2021

Go Back

Full Screen

Close



3. Languages like Pascal which use a single bracketing mechanism for command constructs, often try to reduce the number of tokens produced per program by relaxing the mandatory requirement of bracketing, by stipulating that bracketing is required only for compound commands. Thus for atomic commands  $c_1$  and  $c_2$  the ambiguity in

if 
$$b_1$$
 then if  $b_2$  then  $c_1$  else  $c_2$  (5)

is resolved by introducing an associativety rule that each **else** is associated with the nearest enclosing condition. That is the construct (5) is interpreted as referring to  $it(b_1, ite(b_2, C_1, C_2))$ .

4. There are other means of achieving disambiguation of which the most ingenious is the use of white-space indentation in Python to keep it unambiguous. Hence in Python  $it(b_1, ite(b_2, C_1, C_2))$  would be written as

```
if b_1:
   if b_2:
   C_1
   else:
   C_2
```

and  $ite(b_1, it(b_2, C_1), C_2)$  would be written as

if  $b_1$ :

44

**>>** 



if  $b_2$ :  $C_1$ 

else:  $C_2$ 









# Exercise 4.1

- 1. Two context-free grammars are considered equivalent if they generate the same language. Prove that  $G_1$  and  $G'_1$  are equivalent.
- 2. Palindromes. A palindrome is a string that is equal to its reverse i.e. it is the same when read backwards (e.g. aabbaa and abaabaaba are both palindromes). Design a grammar for generating all palindromes over the terminal symbols a and b.
- 3. Matching brackets.
  - (a) Design a context-free grammar to generate sequences of matching brackets when the set of terminals consists of three pairs of brackets  $\{(,),[,],\{,\}\}$ .
  - (b) If your grammar is ambiguous give two rightmost derivations of the same string and draw the two derivation trees. Explain how you will modify the grammar to make it unambiguous.
  - (c) If your grammar is not ambiguous prove that it is not ambiguous.
- 4. Design an unambiguous grammar for the expression language on integers consisting of expressions made up of operators +, -, \*, /, % and the bracketing symbols ( and ), assuming the usual rules of precedence among operators that you have learned in school.

44

PL December 28, 2021

Go Back

Full Screen

Close

- 5. Modify the above grammar to include the exponentiation operator  $\hat{\ }$  which has a higher precedence than the other operators and is right-associative.
- 6. How will you modify the grammar above to include the unary minus operator where the unary minus has a higher precedence than other operators?
- 7. The language specified by a regular expression can also be generated by a context-free grammar.
  - (a) Design a context-free grammar to generate all floating-point numbers allowed by the C language.
  - (b) Design a context-free grammar to generate all numbers in binary form that are not multiples of 4.
  - (c) Write a regular expression to specify all numbers in binary form that are multiples of of 3.

PL December 28, 2021

- 8. Prove that the  $G'_1$  is indeed unambiguous.
- 9. Prove that the grammar of fully parenthesized expressions is unambiguous.
- 10. Explain how the grammar  $G'_1$  implements left associativity and precedence.



 ✓
 Full Screen

 Close
 193 of 721



4.5. Specification of Syntax: Extended Backus-Naur Form

Specification of Syntax: EBNF









PL December 28, 2021

Go Back

Full Screen



The EBNF specification of a programming language is a collection of rules that defines the (context-free) grammar of the language. It specifies the formation rules for the correct grammatical construction of the phrases of the language. In order to reduce the number of rules unambiguously regular expression operators such as alternation, Kleene closure and +-closure are also used. (Con)catenation is represented by juxtaposition. In addition, a period is used to terminate a rule. The rules are written usually in a "top-down fashion".

**Start symbol.** The very first rule gives the productions of the start symbol of the grammar.

Non-terminals. Uses English words or phrases to denote non-terminal symbols. These words or phrases are suggestive of the nature or meaning of the constructs.

# Metasymbols.

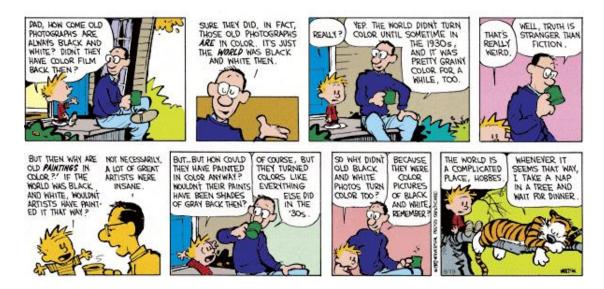
- Sequences of constructs enclosed in "{" and "}" denote zero or more occurrences of the construct (c.f. Kleene closure on regular expressions).
- Sequences of constructs enclosed in "[" and "]" denote that the enclosed constructs are optional i.e. there can be only zero or one occurrence of the sequence.
- Constructs are enclosed in "(" and ")" to group them together.
- " | " separates alternatives.
- "::=" defines the productions of each non-terminal symbol.
- "." terminates the possibly many rewrite rules for a non-terminal.

**⋖**∢





**Terminals.** Terminal symbol strings are sometimes enclosed in double-quotes when written in monochrome (we shall additionally colour-code them).



### Note.

We have chosen to colour-code the EBNF specification in order to clearly separate the colours of the EBNF operators from those of the language that is being specified. Further we have chosen to use different colours for the *Nonterminal* symbols and the *terminal* symbols. In the bad old days when the world was only black-and-white and the only font available was the type-writer font, the <Nonterminal> symbols were usually enclosed in "<>" while the terminal symbols were written directly (optionally enclosed in double-quotes (")).

 →
 →

 PL December 28, 2021

 Go Back
 Full Screen

 Close
 196 of 721



# Balanced Parentheses: CFG

**Example 4.12** A context-free grammar for balanced parentheses (including the empty string) over the terminal alphabet  $\{(,),[,],\{,\}\}$  could be given as  $BP_3 = \langle \{S\},\{(,),[,],\{,\}\},P,\{S\}\rangle$ , where P consists of the productions

$$S \to \epsilon,$$

$$S \to (S)S,$$

$$S \to [S]S,$$

$$S \to \{S\}S$$

 ✓
 ✓

 PL December 28, 2021

 Go Back
 Full Screen

 Close
 197 of 721



# Balanced Parentheses: EBNF

# **Example 4.13** $BP_3$ may be expressed in EBNF as follows:

```
BracketSeq ::= \{Bracket\}.
Bracket
              ::= LeftParen BracketSeq RightParen
                  LeftSqbracket BracketSeq RightSqbracket
                  LeftBrace\ BracketSeg\ RightBrace.
LeftParen ::= "(").
RightParen ::= ")".
LeftSqbracket ::= "["]"
RightSqbracket ::= "]".
LeftBrace ::= "{}".
RightBrace ::= "}".
```

PL December 28, 2021

Go Back

Full Screen

CLOSE

# **EBNF** in **EBNF**



EBNF has its own grammar which is again context-free. Hence EBNF (4.5) may be used to define EBNF in its own syntax as follows:

```
Syntax
                 := \{Production\}.
               ::= NonTerminal "::=" PossibleRewrites ".".
Production
Possible Rewrites ::= Rewrite \{ "|" Rewrite \} .
                 := Symbol \{Symbol\}.
Rewrite
Symbol
           ::= NonTerminal \mid Terminal \mid GroupRewrites.
GroupRewrites ::= "{" PossibleRewrites "}"
                     "[" PossibleRewrites "]" |
                     "(" PossibleRewrites ")".
                 := Letter \{Letter \mid Digit\}.
NonTerminal
                  ::= Character \{Character\}.
Terminal
```

PL December 28, 2021

44



# **EBNF**: Character Set



# The character set used in EBNF is described below.

```
Character
                := Letter \mid Digit \mid SpecialChar
Letter
                ::= UpperCase \mid LowerCase
UpperCase
                                        ``D"\mid ``E"\mid ``F"\mid
                    "R" \mid "S" \mid "T" \mid "U" \mid "V" \mid "W" \mid
                                                             (X')
                                     \mid "d" \mid "e" \mid "f" \mid "g" \mid "h"
LowerCase
               ::= "a" | "b"
                    "i" | "j" | "k" | "l" | "m" | "n" |
                                                         "o" \mid "p" \mid
                    "r" \mid "s" \mid "t" \mid "u" \mid "v" \mid "w" \mid "x" \mid "y"
Digit
                         | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
                                            SpecialChar ::= "!"
```

Go Back



 ✓
 Full Screen

 Close
 201 of 721



### 4.6. The WHILE Programming Language: Syntax

All words written in **bold** font are reserved words and cannot be used as identifiers in any program.

```
Program ::= "program" Identifier "::" Block.
Block
       ::= DeclarationSeq\ CommandSeq\ .
DeclarationSeq ::= \{Declaration\}.
Declaration ::= "var" VariableList": "Type";".
      ::= "int" | "bool".
Type
VariableList ::= Variable\{"," Variable\}.
CommandSeq ::= "{\{}^{"}{Command";}^{"}{\}}"}.
              ::= Variable ":="Expression"
Command
                   "read" Variable
                   "write" IntExpression
                   "if" BoolExpression "then" CommandSeq
                   "else" CommandSeq
                   "endif"
                   "while" BoolExpression "do" CommandSeq
                   "endwh".
```

PL December 28, 2021



```
Expression ::= IntExpression \mid BoolExpression
IntExpression ::= IntExpression AddOp IntTerm | IntTerm .
IntTerm
         ::= IntTerm\ MultOp\ IntFactor\ IntFactor\ .
IntFactor ::= Numeral \mid Variable \mid
                   "("IntExpression")" | "~"IntFactor.
BoolExpression ::= BoolExpression "||" BoolTerm | BoolTerm .
BoolTerm
          ::= BoolTerm "&&" BoolFactor \mid BoolFactor.
          ::= "tt" | "ff" | Variable \mid Comparison |
BoolFactor
                   "("BoolExpression")" | "!"BoolFactor.
               ::= IntExpression RelOp IntExpression.
Comparison
Variable
               := Identifier.
RelOp ::= "<" | "<=" | ">" | ">=" | ">=" | "<>" .
       ::= "+" | "-".
AddOp
MultOp
        ::= "*" | "/" | "\%" .
Identifier ::= Letter\{Letter \mid Digit\}.
Numeral ::= ["+" | ""] Digit\{Digit\}.
```

### Note

- 1. ";" acts as a terminator for both *Declarations* and *Commands*.
- 2. "," acts as a separator in VariableList

44

◀



- 3. Comparison has a higher precedence than BoolTerm and BoolExpression.
- 4. RelOps have lower precedence than any of the integer operations specified in MultOp and AddOp.
- 5. The nonterminals Letter and Digit are as specified earlier in the EBNF character set







# Syntax Diagrams

- EBNF was first used to define the grammar of ALGOL-60 and the syntax was used to design the parser for the language.
- EBNF also has a diagrammatic rendering called syntax diagrams or railroad diagrams. The grammar of SML has been rendered by a set of syntax diagrams.
- Pascal has been defined using both the text-version of EBNF and through syntax diagrams.
- While the text form of EBNF helps in parsing, the diagrammatic rendering is only for the purpose of readability.
- EBNF is a specification language that almost all modern programming languages use to define the grammar of the programming language

44





Full Screen

Close

205 of 721

Go Back



# Syntax Specifications

PL December 28, 2021

- BNF of C
- BNF of Java
- EBNF of Pascal
- Pascal Syntax diagrams
- BNF of Standard ML
- BNF of Datalog
- BNF of Prolog

44





# Syntax of Standard ML

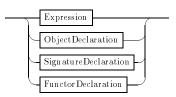
d\$E

Tobias Nipkow and Larry Paulson

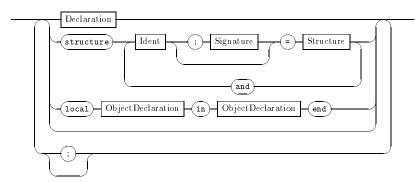
#### PROGRAMS AND MODULES

# Program TopLevelDeclaration ;

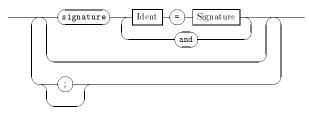
#### TopLevelDeclaration



#### ObjectDeclaration

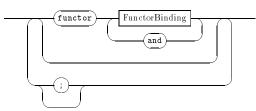


#### Signature Declaration

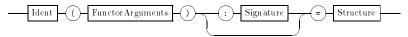




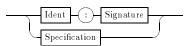
#### Functor Declaration



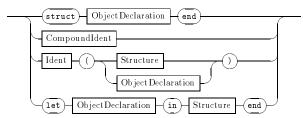
#### Functor Binding



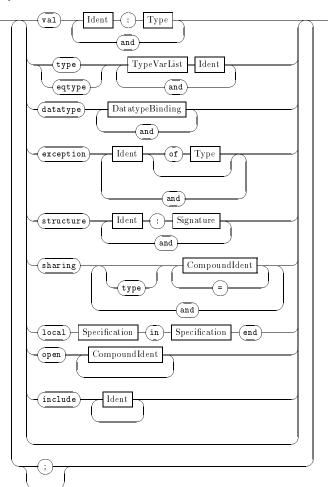
#### Functor Arguments



#### Structure



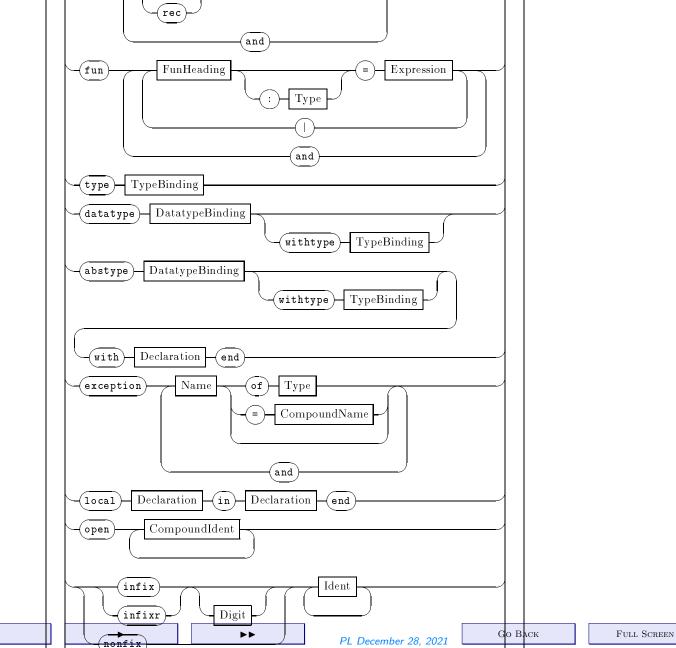
#### Signature





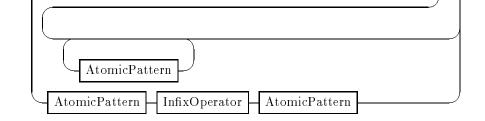
**⋖**◀

PL December 28, 2021



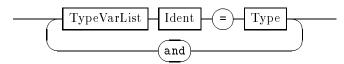


Close

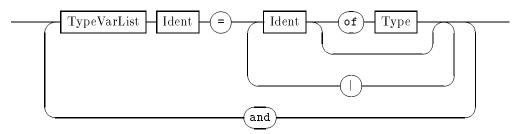




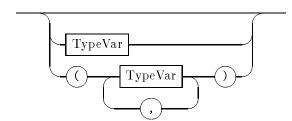
### TypeBinding

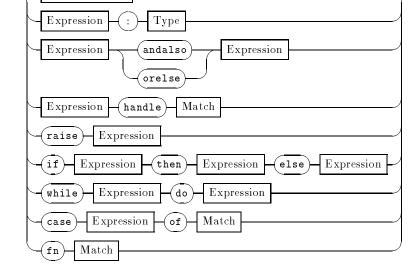


### DatatypeBinding

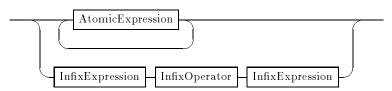


 $Type\ VarList$ 





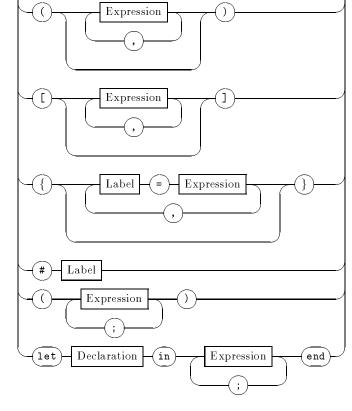
InfixExpression



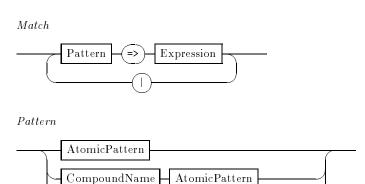


**⋖**◀

PL December 28, 2021



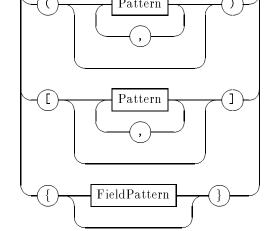




Pattern InfixOperator Pattern

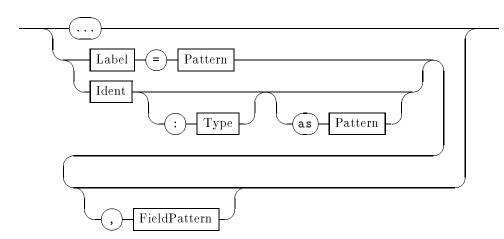
Full Screen

CLOSE 213 OF 721

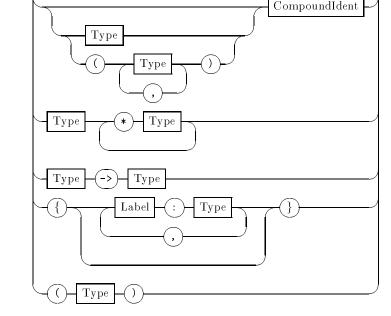




### FieldPattern



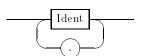
**▶**▶



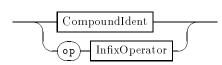


# LEXICAL MATTERS: IDENTIFIERS, CONSTANTS, COMMENTS

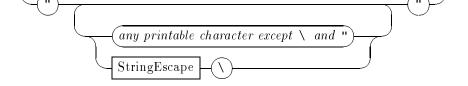
CompoundIdent



Compound Name

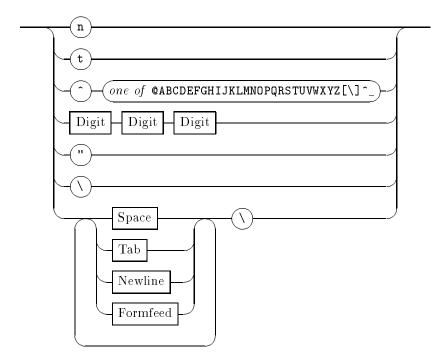


Name



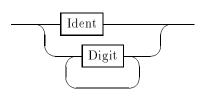


### StringEscape



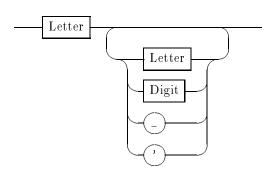
### Numeral



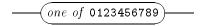




#### Alphanumeric Ident



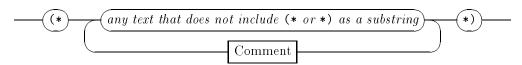
#### Digit



#### Letter

----  $one\ of\ \mathtt{ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz}$ 

#### Comment





#### Exercise 4.2

- 1. Translate all the context-free grammars that we have so far seen into EBNF specifications.
- 2. Specify the language of regular expressions over a non-empty finite alphabet A in EBNF.
- 3. Given a textual EBNF specification write an algorithm to render each non-terminal as a syntax diagram.

PL December 28, 2021

Go Back

Full Screen

Close





#### 4.7. Parsing

## Introduction to Parsing

PL December 28, 2021









Go Back

Full Screen

Close



### Overview of Parsing

### Since

- parsing requires the checking whether a given token stream conforms to the rules of the grammar and
- since a context-free grammar may generate an infinite number of different strings

any parsing method should be guided by the given input (token) string, so that a deterministic strategy may be evolved.

44





Go Back

Full Screen

Close



### Parsing Methods

Two kinds of parsing methods

Top-down parsing Try to generate the given input sentence from the start symbol of the grammar by applying the production rules.

Bottom-up parsing Try to reduce the given input sentence to the start symbol by applying the rules in *reverse* 

In general top-down parsing requires long *look-aheads* in order to do a deterministic guess from the given input token stream. On the other hand bottom-up parsing yields better results and can be automated by software tools.

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 222 OF 721



→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 223 OF 721



→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 224 OF 721



#### 4.8. Recursive Descent Parsing

## Top-down Parsing

- Try to generate the given input sentence from the start symbol of the grammar by applying the production rules.
- Not the most general.
- But most modern high-level programming languages are designed to be efficiently parsed by this method.
- Recursive-descent is the most frequently employed technique when language
   C in which the compiler is written, supports recursion.

44

**•** 

PL December 28, 2021

Go Back

Full Screen

Close



## Recursive Descent Parsing

- Suitable for grammars that are  $LL(1)^a$  parseable.
- A set of (mutually) recursive procedures
- Has a single procedure/function for each non-terminal symbol
- Allows for syntax errors to be pinpointed more accurately than most other parsing methods

PL December 28, 2021

**⋖**∢





Go Back

<sup>&</sup>lt;sup>a</sup>Left-to-right Left-most derivations with 1 look-ahead



### Caveats with RDP: Direct Left Recursion

Any left-recursion in the grammar can lead to infinite recursive calls during which no input token is consumed and there is no return from the recursion. That is, they should not be of the form

$$A \longrightarrow A\alpha$$

This would result in an infinite recursion with no input token consumed.

44

▶

▶▶

PL December 28, 2021

Go Back

Full Screen

Close



### Caveats with RDP: Indirect Left Recursion

• A production cannot even be *indirectly* left recursive. For instance the following is *indirect* left-recursion of cycle length 2.

### Example 4.14

$$\begin{array}{c} A \longrightarrow B\beta \\ B \longrightarrow A\alpha \end{array}$$

where  $\alpha, \beta \in (N \cup T)^*$ .

• In general it should be impossible to have derivation sequences of the form  $A \Rightarrow A_1\alpha_1 \cdots \Rightarrow A_{n-1}\alpha_{n-1} \Rightarrow A\alpha_n$  for nonterminal symbols  $A, A_1, \ldots, A_{n-1}$  for any n > 0.

44

PL December 28, 2021

Go Back

Full Screen

Close

### Caveats with RDP: Left Factoring



For RDP to succeed without backtracking, for each input token and each non-terminal symbol there should be only one rule applicable;

**Example 4.15** A set of productions of the form

$$A \longrightarrow aB\beta \mid aC\gamma$$

where B and C stand for different phrases would lead to non-determinism. The normal practice then would be to left-factor the two productions by introducing a new non-terminal symbol A' and rewrite the rule as

$$\begin{array}{c} A \longrightarrow aA' \\ A' \longrightarrow B\beta \mid C\gamma \end{array}$$

provided B and C generate terminal strings with different first symbols (otherwise more left-factoring needs to be performed).

44





PL December 28, 2021

Go Back

Full Screen

Close



## A Simple Left-recursive Grammar

The following grammar is unambiguous and implements both left-associativity and precedence of operators.  $G = \langle \{E,T,D\}, \{a,b,-,/(,)\}, P,E \rangle$  whose productions are

$$E \rightarrow E - T \mid T$$

$$T \rightarrow T/D \mid D$$

$$D \rightarrow a \mid b \mid (E)$$

PL December 28, 2021







Go Back

FULL SCREEN

Close



### Left Recursion Removal

The grammar G is clearly left-recursive in both the nonterminals E and T and hence is not amenable to recursive-descent parsing.

The grammar may then have to be modified as follows:

$$E \rightarrow TE'$$

$$E' \rightarrow -TE' \mid \varepsilon$$

$$T \rightarrow DT'$$

$$T' \rightarrow /DT' \mid \varepsilon$$

$$D \rightarrow a \mid b \mid (E)$$

Now this grammar is no longer left-recursive and may then be parsed by a recursive-descent parser.

44

**•** 

▶▶

PL December 28, 2021

Go Back

Full Screen

Close



## Recursive Descent Parsing: Determinization

RDP can be deterministic only if

- the input token lookahead uniquely determines the production to be applied.
- We need to define the FIRST symbols that will be generated by each production.
- $\bullet$  In the presence of  $\varepsilon$  productions, symbols that can FOLLOW a given non-terminal symbol also need to be specified.

**⋖**◆





Full Screen

Close



### Nullable

A nonterminal symbol X is **nullable** if it can derive the empty string, i.e.  $X \Rightarrow^* \varepsilon$ . The following algorithm computes nullable(X) for each non-terminal symbol. For convenience nullable is set to false for each terminal symbol in the grammar. NULLABLE(N) is the set of boolean values specifying for each nonterminal symbol whether it is nullable.

**⋖**◆

▶

▶▶

PL December 28, 2021

Go Back

Full Screen

Close



```
Algorithm 4.1
 Nullable (N) \stackrel{df}{=}
       Require: CFG G = \langle N, T, P, S \rangle
       Yields: NULLABLE(N) = \{nullable(X) \mid X \in N\}
    for each a \in T
       do nullable(a) := false;
    for each X \in N
       do nullable(X) := \exists X \to \varepsilon \in P
    repeat
      for each X \to \alpha_1 \dots \alpha_k \in P
              \begin{cases} \textbf{if } \forall i : 1 \leq i \leq k : nullable(\alpha_i) \\ \textbf{then } nullable(X) := \textbf{true} \end{cases}
   until NULLABLE(N) is unchanged
```

44

**▶** 

PL December 28, 2021



### **First**

 $first(\alpha)$  is the set of terminal symbols that can be the first symbol of any string that  $\alpha$  can derive, i.e.  $a \in first(\alpha)$  if and only if there exists a derivation  $\alpha \Rightarrow^* ax$  for any string of terminals x.

### Notice that

- the computation of first requires nullable to be available. Also the first of any terminal symbol is itself.
- also that if  $X \to \alpha Z\beta$  is a production then one cannot ignore the first(Z) in computing first(X) especially if  $\alpha \Rightarrow^* \varepsilon$ . Further if Z is also nullable then  $first(\beta) \subseteq first(X)$ .

44

**∢** 



PL December 28, 2021

Go Back

Full Screen

Close



```
Algorithm 4.2
 First (N \cup T) \stackrel{df}{=}
        Require: CFG G = \langle N, T, P, S \rangle
        Yields: FIRST(N \cup T) = \{first(\alpha) \mid \alpha \in N \cup T\}
     for each a \in T
       do first(a) := \{a\}
     for each X \in N
       do first(X) := \emptyset
     repeat
      for each X \to \alpha_1 \dots \alpha_k \in P
                \begin{cases} \mathbf{if} \ \forall i' : 1 \leq i' < i : \mathbf{nullable}(\alpha_{i'}) \\ \mathbf{then} \ first(X) := first(X) \cup first(\alpha_i) \end{cases}
    until FIRST(N) is unchanged
```

44



PL December 28, 2021



### First And Follow

follow(X) for any nonterminal symbol X is the set of terminal symbols a such that there exists a rightmost derivation of the form

$$S \Rightarrow^* \cdots Xa \cdots \Rightarrow^*$$

i.e. follow(X) is the set of all terminal symbols that can occur immediately to the right of X in a rightmost derivation.

Notice that if there exists a a rightmost derivation of the form

$$S \Rightarrow^* \cdots X\alpha_1 \dots \alpha_k a \cdots \Rightarrow^*$$

such that  $\alpha_1, \ldots, \alpha_k$  are all nullable then again we have

$$S \Rightarrow^* \cdots X \alpha_1 \dots \alpha_k a \dots \Rightarrow^* \cdots X a \dots \Rightarrow^*$$

44



PL December 28, 2021



Full Screen

Close



```
Algorithm 4.3
 Follow (N) \stackrel{df}{=}
        Require: CFG G = \langle N, T, P, S \rangle
        Yields: FOLLOW(N \cup T) = \{follow(\alpha) \mid \alpha \in N \cup T\}
     for each \alpha \in N \cup T
       do follow(\alpha) := \emptyset
     repeat
      for each X \to \alpha_1 \dots \alpha_k \in P
                 for i := 1 to k
                             (if \forall i': i+1 \leq i' \leq k: nullable(\alpha_{i'})
                                 then follow(\alpha_i) := follow(\alpha_i) \cup follow(X)
         do
                     do \langle for j := i + 1 to k
                               do \begin{cases} \mathbf{if} \ \forall i' : i+1 \leq i' < j : nullable(\alpha_{i'}) \\ \mathbf{then} \ follow(\alpha_i) := follow(\alpha_i) \cup first(\alpha_j) \end{cases}
     until FOLLOW(N \cup T) is unchanged
```

44



PL December 28, 2021



## Recursive Descent Parsing: Pragmatics

- In any collection of (possibly) mutually recursive procedures, it is necessary to clearly specify the entry point into the collection and the exits. So we add a new start symbol S and a new end-of-file token EOF (represented by a new terminal symbol \$) with the unique production  $S \to E\$$ .
- Tokens are in upper case and the correspondence between the lexemes and tokens is as follows:

$$ID(a) \leftrightarrow a$$
 ,  $ID(b) \leftrightarrow b$   
 $LPAREN \leftrightarrow ($  ,  $RPAREN \leftrightarrow )$   
 $MINUS \leftrightarrow -$  ,  $DIVIDE \leftrightarrow /$   
 $EOF \leftrightarrow \$$ 

44

▶▶

Go Back

Full Screen

Close



#### A recursive descent parser

```
program Parse;
   var input_token: token;
   function get_token:token;
   begin
      (* lex *)
   end;
   procedure match(expected);
      label 99;
   begin
      if input_token = expected then
      begin
         consume (input_token);
         if input_token <> EOF then input_token := get_token
         else goto 99
      end
      else parse_error
      99:
   end;
```



```
(* The system of mutually recursive procedures begins here *)
procedure Expression; Forward;
procedure Division (* D -> a | b | (E) *)
begin
   case input_token of
  ID: match(ID);
  LPAREN: Expression; match (RPAREN);
   else parse_error
end;
procedure Term_tail (* T' -> /DT' | <epsilon> *)
begin
   case input_token of
  DIVIDE: Division; Term_tail;
  MINUS: Term_tail; (* epsilon production *)
  RPAREN, ID: ; (* skip epsilon production *)
   else parse_error
end;
procedure Term (* T -> DT' *)
begin
   case input_token of
  ID(a), ID(b), LPAREN: Division; Term_tail;
   else parse_error
```

PL December 28, 2021



```
end;
    procedure Expression_tail (* E' -> -TE' | <epsilon> *)
    begin
       case input_token of
      MINUS: Term; Expression_tail;
      RPAREN, ID: ; (* skip epsilon production *)
       else parse_error
    end;
   procedure Expression (* E -> TE' *)
    begin
       case input_token of
       ID(a), ID(b), LPAREN: Term; Expression_tail;
       else parse_error
    end;
begin (* main S -> E$ *)
   input_token := get_token;
   Expression; match (EOF)
end.
```

PL December 28, 2021

Full Screen





#### 4.9. Shift-Reduce Parsing

## Bottom-Up Parsing Strategy

The main problem is to match parentheses of arbitrary nesting depths. This requires a stack  $^a$  data structure to do the parsing so that unbounded nested parentheses and varieties of brackets may be matched.

Our basic parsing strategy is going to be based on a technique called *shift-reduce* parsing.

shift. Refers to moving the next token from the input token stream into a parsing stack.

reduce. Refers to applying a production rule in reverse i.e. given a production  $X \to \alpha$  we reduce any occurrence of  $\alpha$  in the parsing stack to X.

■■ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 244 OF 721

<sup>&</sup>lt;sup>a</sup>In the case of recursive-descent parsing the stack is provided by the recursion facility in the language of implementation.

### Reverse of Right-most Derivations



The result of a Bottom-Up Parsing technique is usually to produce a *reverse* of the right-most derivation of a sentence.

Example For the ambiguous grammar  $G_1$  and corresponding to the right-most derivation 2 we get

$$\mathbf{y} + \mathbf{4} * \mathbf{z} \Leftarrow$$
 $I + \mathbf{4} * \mathbf{z} \Leftarrow$ 
 $E + \mathbf{4} * \mathbf{z} \Leftarrow$ 
 $E + C * \mathbf{z} \Leftarrow$ 
 $E + E * \mathbf{z} \Leftarrow$ 
 $E * I \Leftarrow$ 
 $E * E \Leftarrow$ 
 $E * E \Leftarrow$ 

44

**>** 

PL December 28, 2021

Go Back

Full Screen

Close



## Fully Bracketed Expression

Consider an example of a fully bracketed expression generated by the simple left-recursive grammar defined earlier.

The main questions are

- When to shift and when to reduce?
- If reduce then what production to use?

■■ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 246 OF 721



## Shift-reduce parsing: Invariant

Given a sentence generated by the grammar, at any stage of the parsing, the contents of the stack concatenated with the rest of the input token stream should be a sentential form of a right-most derivation of the sentence.

4

ı

▶▶

2224

Go Back

Full Screen

Close



```
r1. E \longrightarrow E \longrightarrow T

r2 E \longrightarrow T

r3 T \longrightarrow T / D

r4 T \longrightarrow D

r5 D \longrightarrow a \mid b \mid (E)
```

# **Principle:**

Reduce whenever possible. Shift only when reduce is impossible



### Shift













```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D \longrightarrow a | b | ( E)
                                a-(a/b))
Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
```

Shift



```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D \longrightarrow a | b | ( E)
                                   - (a / b))
Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                      a
                                  Shift
```

◀•

•

**>>** 

PL December 28, 2021

Go Back

Full Screen

CLOSE



```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D \longrightarrow a | b | ( E)
                                   - (a / b))
Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                      (a)
                                  Reduce
```

- ■











```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D \longrightarrow a | b | ( E)
                                   - (a / b))
Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                  Reduce
```

**⋖**•







Go Back

Full Screen

CLOSE

252 OF 721



```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D \longrightarrow a | b | ( E)
                                   - (a / b))
Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                  Reduce
```













```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D \longrightarrow a | b | ( E)
                                   - (a / b))
Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                  Shift
```



```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D \longrightarrow a | b | ( E)
                                       (a / b))
Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                   Shift
```

- ■





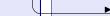






```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D \longrightarrow a | b | ( E)
                                             / b ) )
Principle:
Reduce
whenever possible.
                      (a)
Shift only when
reduce is
impossible
                                   Reduce
```











```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D \longrightarrow a | b | ( E)
                                             / b ) )
Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                   Reduce
```









Go Back

Full Screen

CLOSE

257 OF 721



```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D \longrightarrow a | b | ( E)
                                             / b ) )
Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                   Reduce?
```











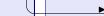


```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D - a | b | ( E)
                                         // b))
Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                Reduce? Shift
```



```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D \longrightarrow a | b | ( E)
                                                b))
Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                   Shift
```















```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D - a | b | ( E)
                                               Principle:
                     (b)
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                Reduce
```













```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D \longrightarrow a | b | ( E)
                                                  Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                   Reduce
```











Full Screen

CLOSE

262 OF 721



```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D - a | b | ( E)
                                              Principle:
                     D
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                Reduce?
```















```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T / D
r4 T → D
r5 D \longrightarrow a | b | ( E)
                                                  Principle:
                       D
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                  No, REDUCE!
```















```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D - a | b | ( E)
                                              Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                Reduce?
```







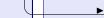






```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D \longrightarrow a | b | ( E)
                                                   Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                   Shift
```











Go Back

Full Screen

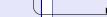
Close

266 of 721



```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D - a | b | ( E)
Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                 Reduce
```













```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D - a | b | ( E)
Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                 Reduce
```













```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D - a | b | ( E)
Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                 Reduce?
```













```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D - a | b | ( E)
Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                No, REDUCE!
```









Go Back

Full Screen

CLOSE

270 of 721



```
r1. E \longrightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D - a | b | ( E)
Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                 Shift
```

- ■









```
r1. E \rightarrow E - T
r2 E → T
r3 T \rightarrow T/D D
r4 T → D
r5 D - a | b | ( E)
Principle:
Reduce
whenever possible.
Shift only when
reduce is
impossible
                                Reduce
```













```
r1. E \longrightarrow E - T

r2 E \longrightarrow T

r3 T \longrightarrow T / D

r4 T \longrightarrow D

r5 D \longrightarrow a \mid b \mid (E)
```

## **Principle:**

Reduce whenever possible. Shift only when reduce is impossible

D

**Reduce** 









```
r1. E \longrightarrow E - T

r2 E \longrightarrow T

r3 T \longrightarrow T / D

r4 T \longrightarrow D

r5 D \longrightarrow a \mid b \mid (E)
```

#### **Principle:**

Reduce whenever possible. Shift only when reduce is impossible

Reduce



```
r1. E \longrightarrow E \longrightarrow T

r2 E \longrightarrow T

r3 T \longrightarrow T / D

r4 T \longrightarrow D

r5 D \longrightarrow a \mid b \mid (E)
```

#### **Principle:**

Reduce whenever possible. Shift only when reduce is impossible

Re

Reduce







#### **Unbracketed Expression**

Consider an example of an unbracketed expression which relies on the precedence rules as defined in the grammar.











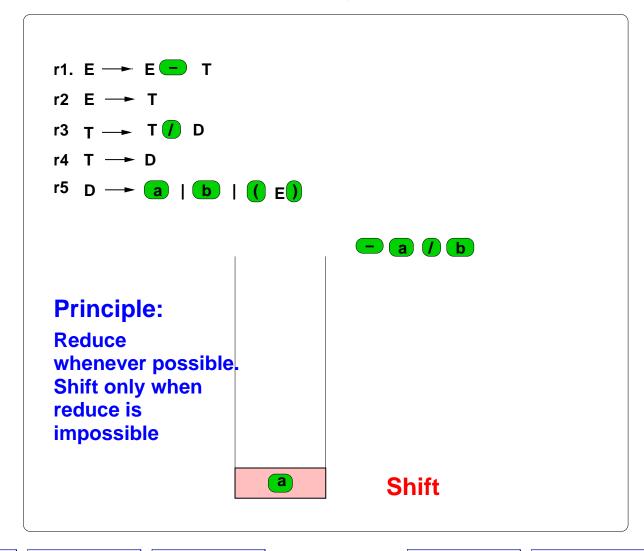


```
r1. E → E - T
r2 E → T
r3 T \longrightarrow T / D
r4 T → D
r5 D \longrightarrow a | b | ( E)
                                a - a / b
```

















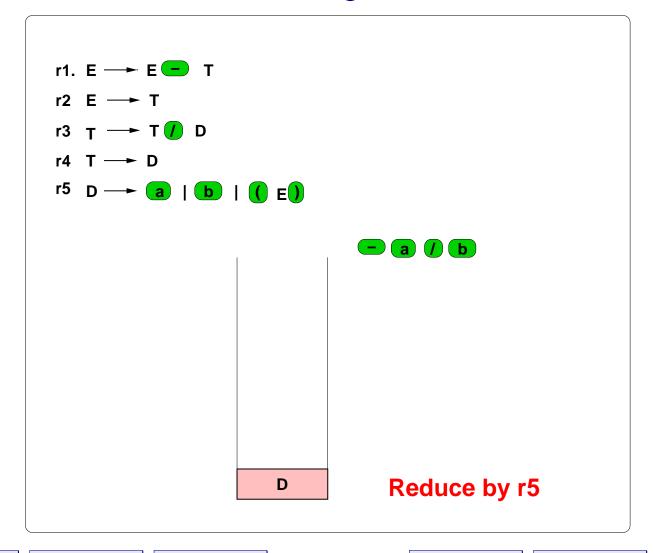
Go Back

Full Screen

Close

278 of 721













Full Screen

279 of 721



```
r1. E \longrightarrow E - T
r2 E → T
r3 T \longrightarrow T / D
r5 D \longrightarrow a | b | ( E)
                                    - a / b
                                        Reduce by r4
```

44









```
r1. E \longrightarrow E - T
r2 E → T
r3 T \longrightarrow T / D
r5 D \longrightarrow a | b | ( E)
                                    - a / b
                                        Reduce by r2
```

44











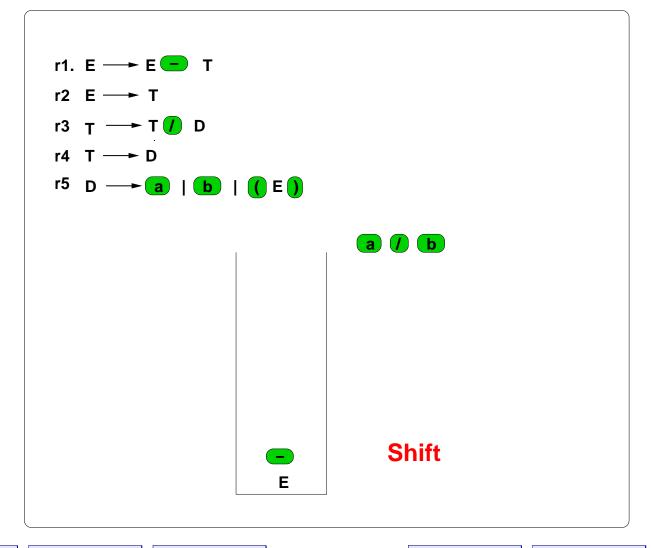
Go Back

Full Screen

Close

281 of 721

















```
r1. E → E - T
r2 E → T
r3 T \longrightarrow T / D
r5 D \longrightarrow a | b | (E)
                                 / b
                                     Shift
                         Ε
```

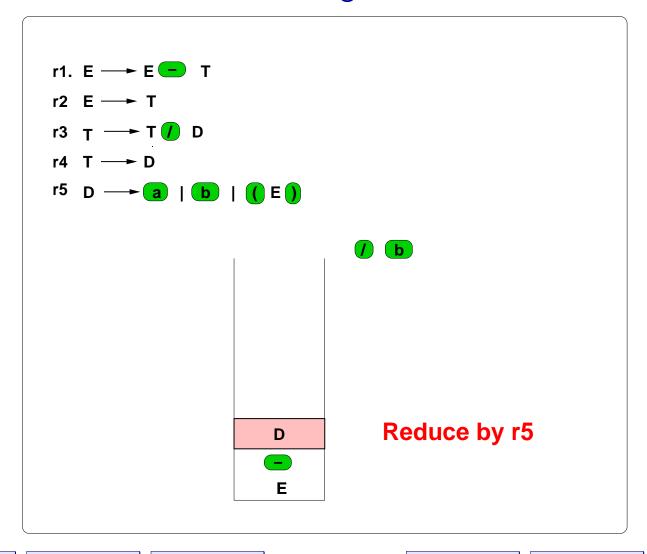










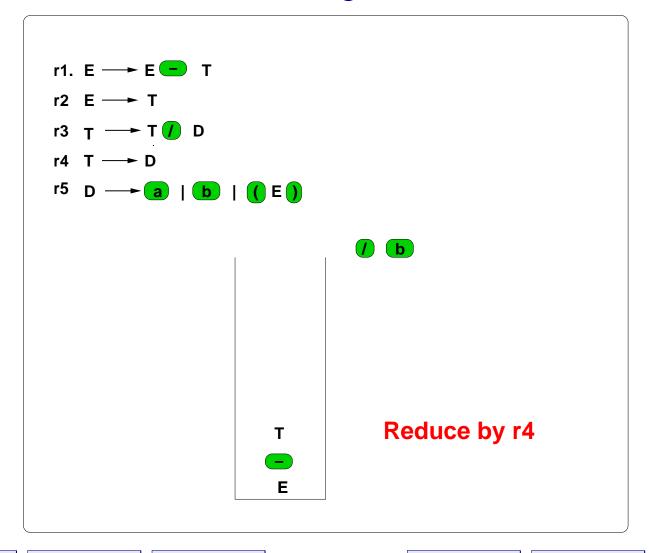


44









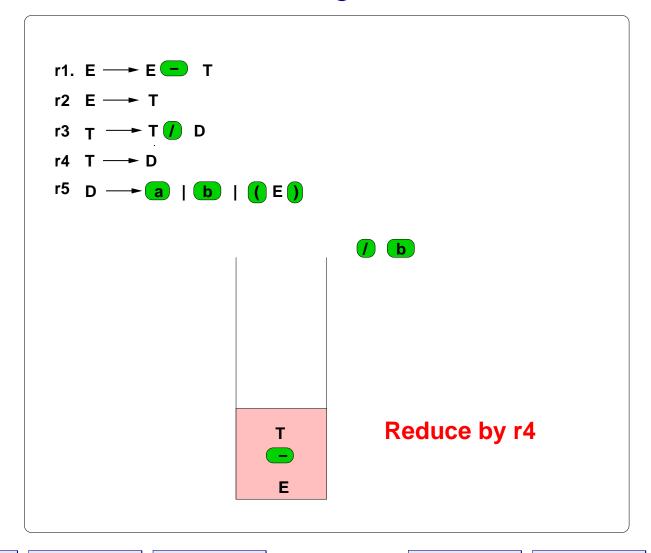










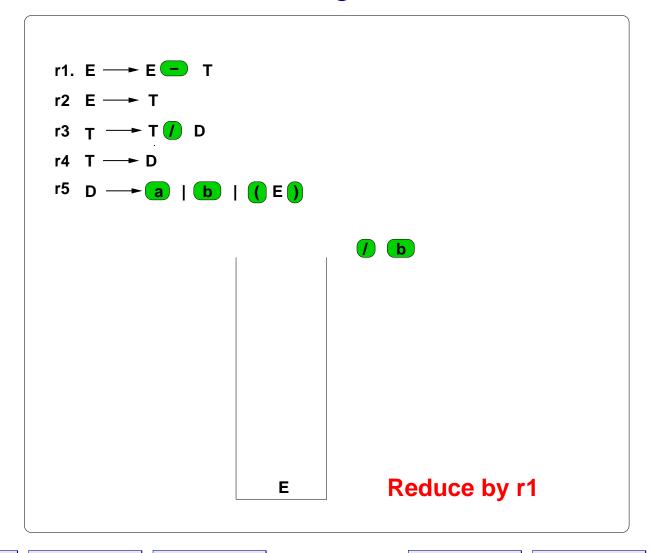


- ◀ ◀









44





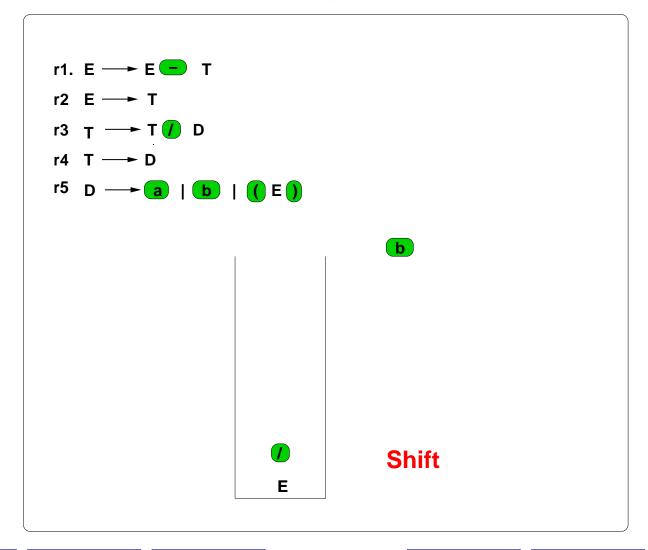






# Parsing: UB10a





44





## Parsing: UB11a



```
r1. E → E - T
r2 E → T
r3 T \longrightarrow T / D
r5 D \longrightarrow a | b | (E)
                                    Shift
                        b
                         Ε
```





## Parsing: UB12a



```
r1. E → E - T
r2 E → T
r3 T \longrightarrow T / D
r5 D \longrightarrow a | b | (E)
                                    Reduce by r5
                        D
                         Ε
```

44









Full Screen

Close

## Parsing: UB13a



```
r1. E → E - T
r2 E → T
r5 D \longrightarrow a | b | (E)
                                Reduce by r4
                      Ε
```

44











Go Back



#### Parsing: UB14a



```
r1. E → E - T
r2 E → T
r3 T \longrightarrow T / D
r5 D \longrightarrow a | b | (E)
                                    Reduce by r2
  Get back!
                        Ε
```





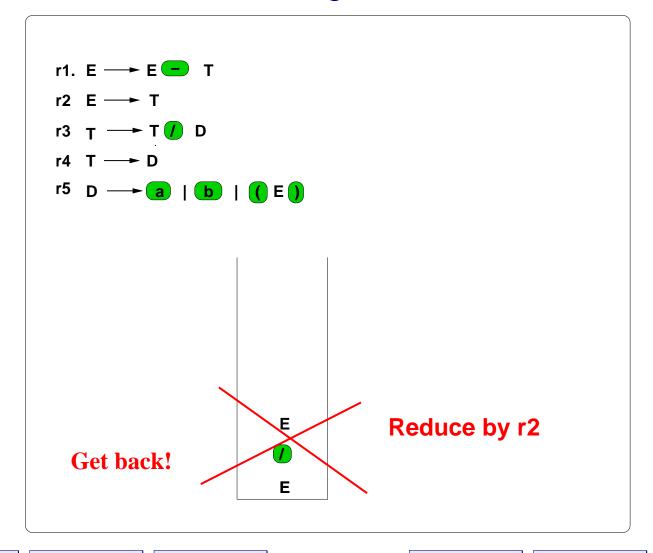






# Parsing: UB14b





44

▶▶

PL December 28, 2021

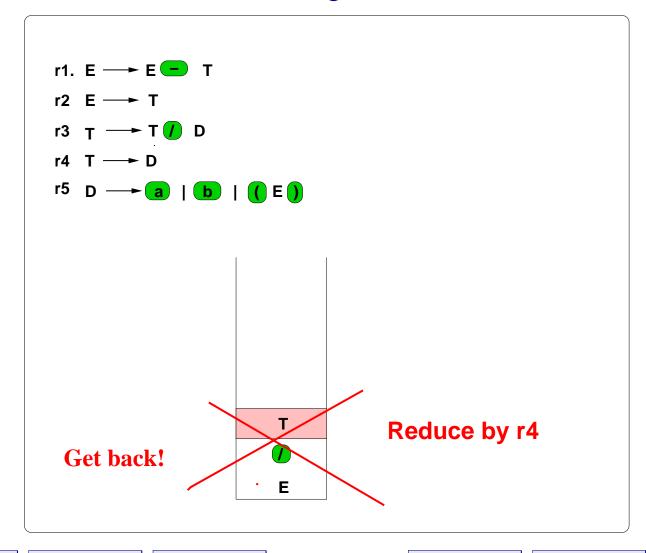
Go Back

Full Screen

Close

# Parsing: UB13b





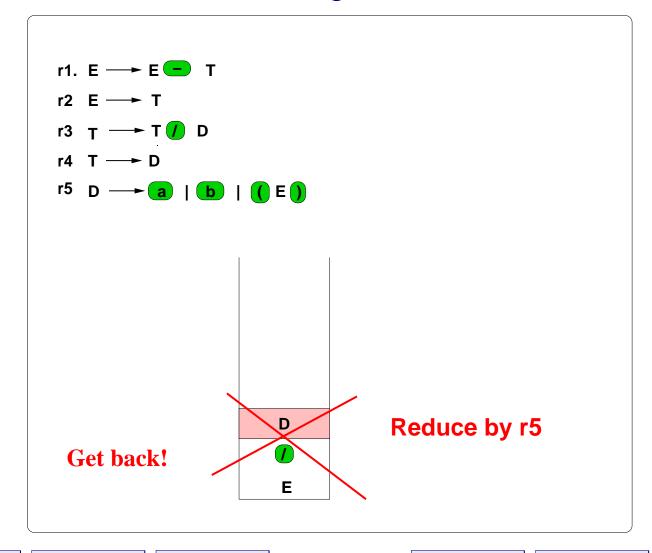






# Parsing: UB12b









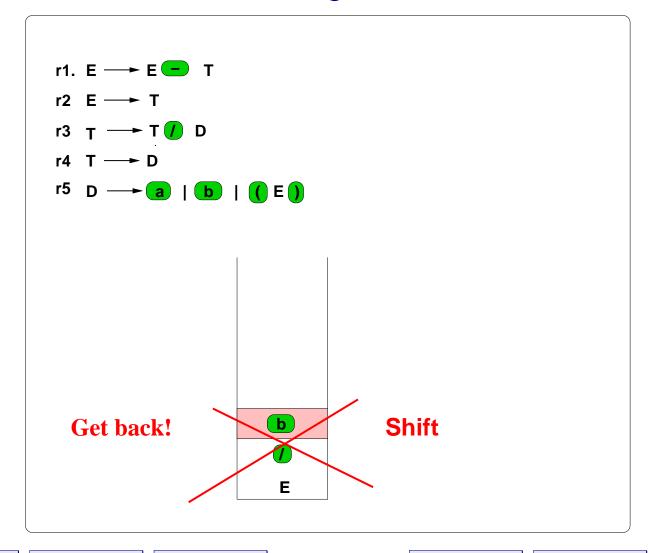






# Parsing: UB11b







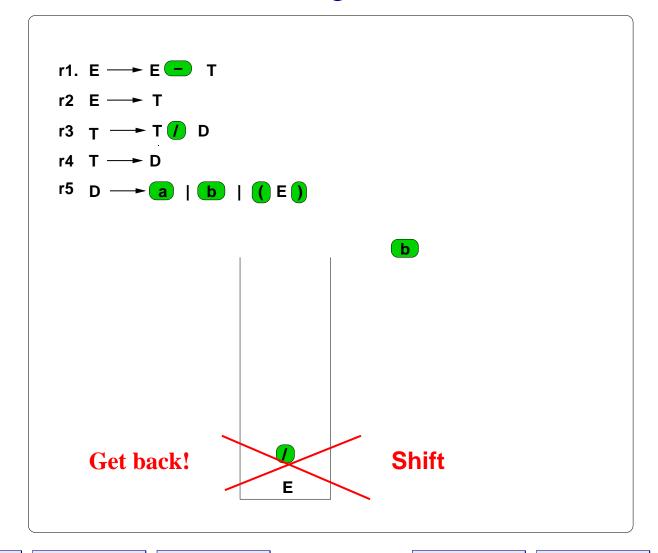






# Parsing: UB10b



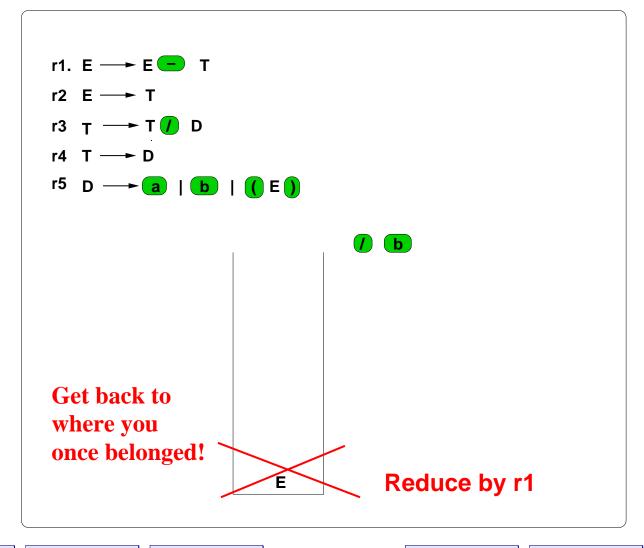








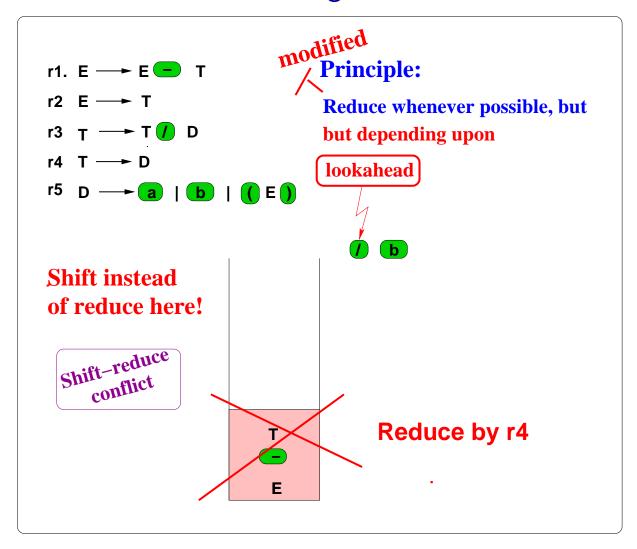












44

▶

▶▶

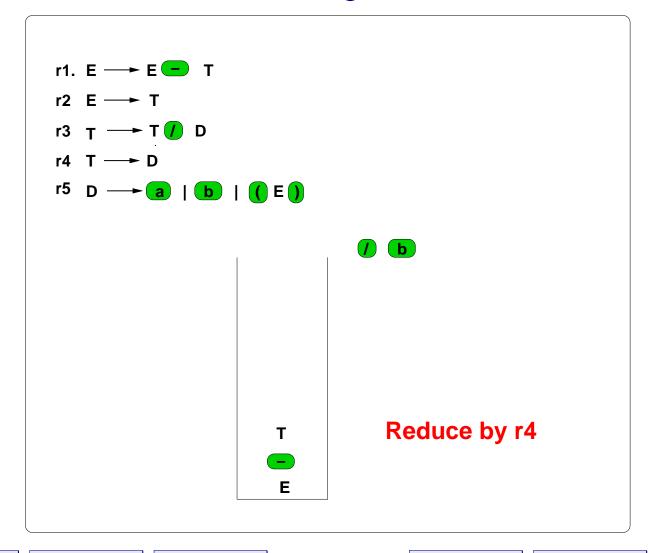
PL December 28, 2021

Go Back

Full Screen

Close





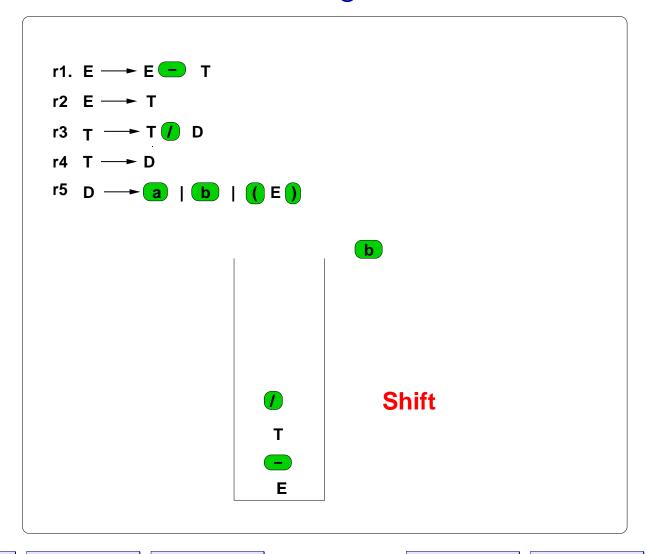
PL December 28, 2021

Go Back

Full Screen

Close













```
r1. E → E - T
r2 E → T
r3 T \longrightarrow T / D
r5 D \longrightarrow a | b | (E)
                        b
                                     Shift
                         Ε
```

44





PL December 28, 2021

Go Back

Full Screen

Close



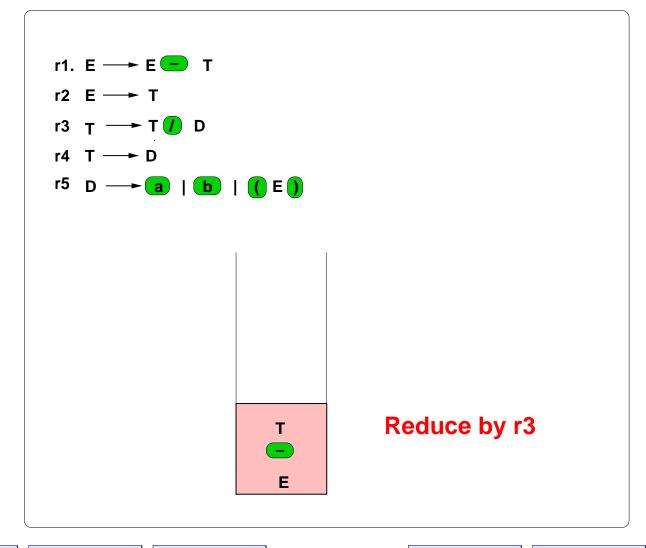
```
r1. E → E - T
r2 E → T
r5 D \longrightarrow a | b | (E)
                                Reduce by r5
                     D
                      Ε
```

















```
r1. E → E - T
r2 E → T
r5 D \longrightarrow a | b | (E)
                                Reduce by r1
                      Ε
```















→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 307 OF 721



#### 4.10. Bottom-Up Parsing

## Bottom-Up Parsing











Go Back

Full Screen

Close



r1.  $E \longrightarrow E - T$ r4 T → D r2 E → T r5 D — a | b | ( E) r3  $T \longrightarrow T / D$ a b

44

**>>** 

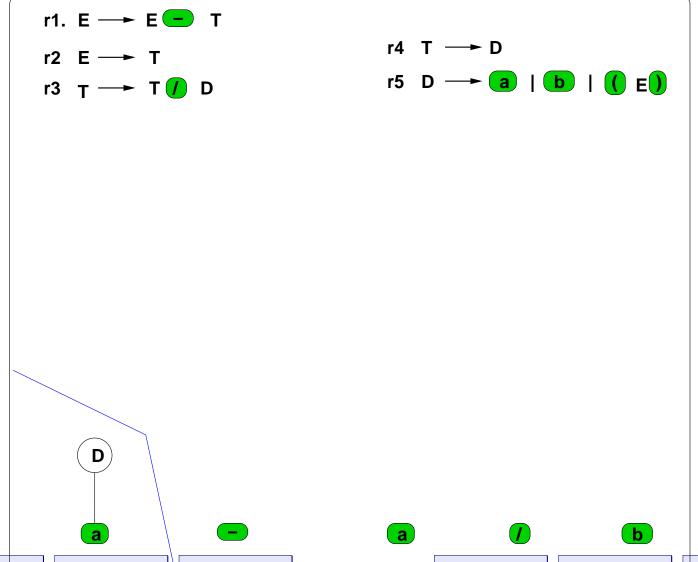
PL December 28, 2021

Go Back

Full Screen

Close





. [ \ ]

**>>** 

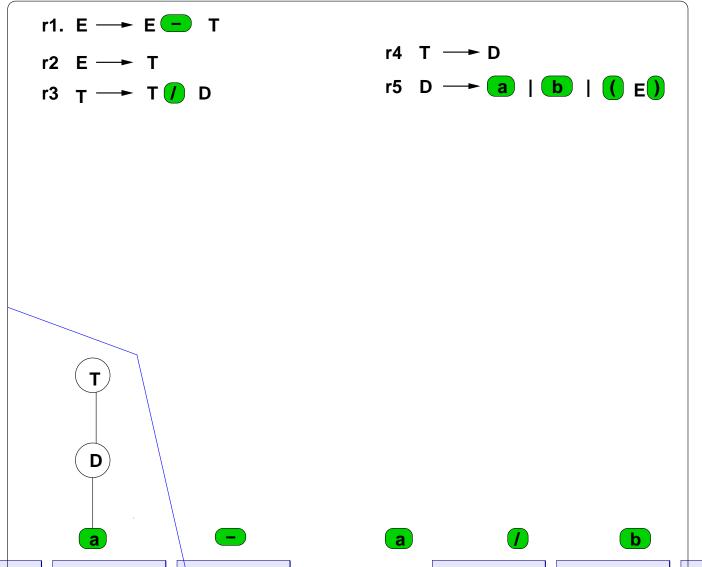
PL December 28, 2021

Go Back

Full Screen

Close





**>** 

**>>** 

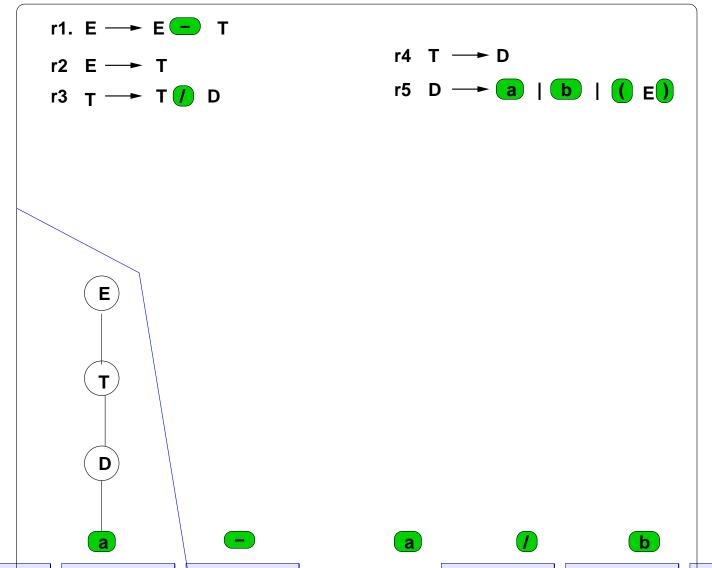
▶▶ PL December 28, 2021

Go Васк

Full Screen

Close





◀ |

•

**>>** 

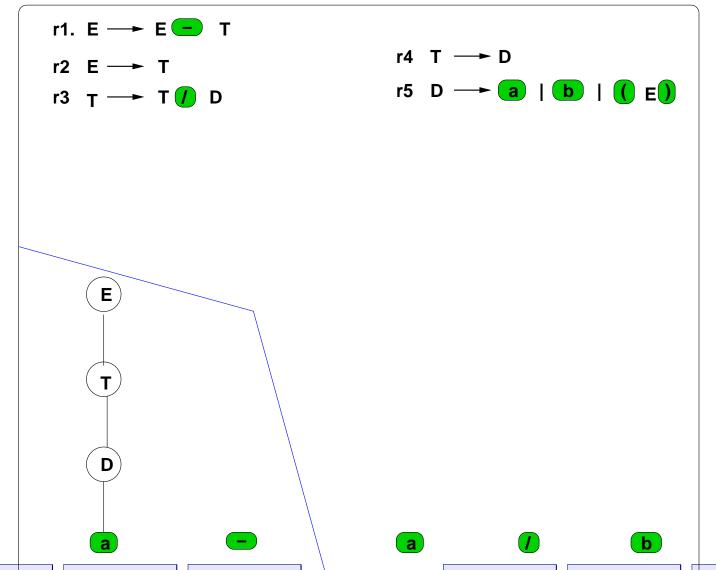
PL December 28, 2021

Go Back

Full Screen

Close





<

**>>** 

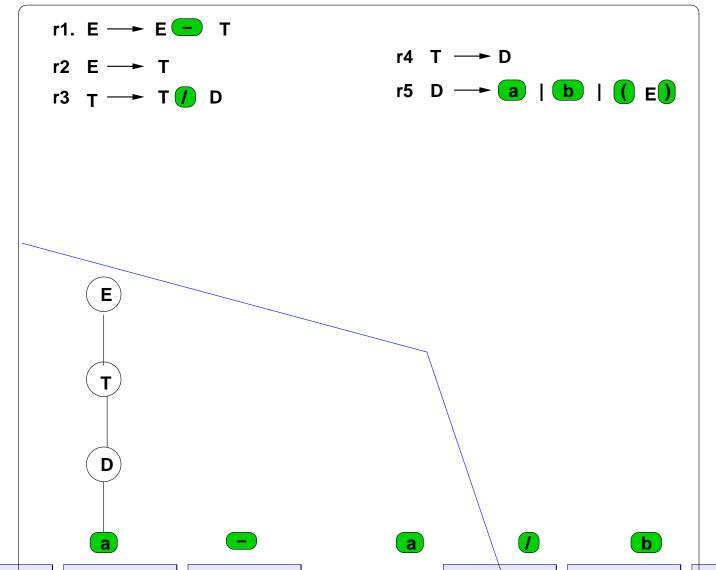
PL December 28, 2021

Go Back

Full Screen

Close





◀

**•** 

**>>** 

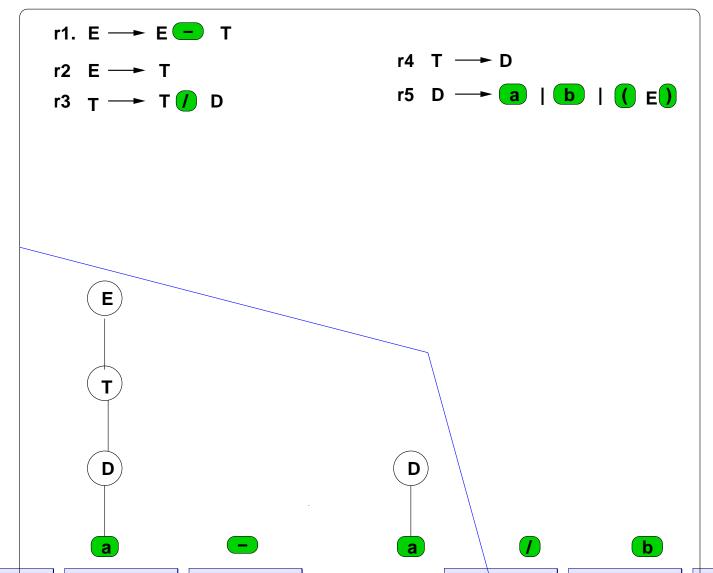
PL December 28, 2021

Go Back

Full Screen

Close





- ◀ •

◀

**•** 

**>>** 

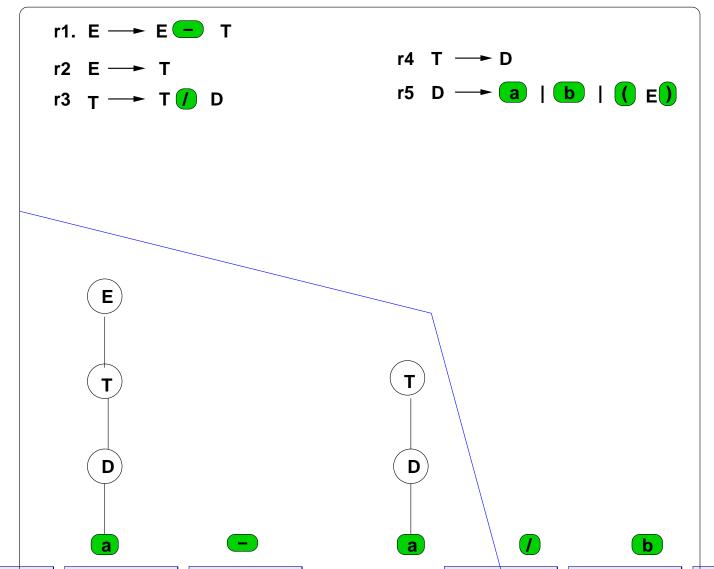
PL December 28, 2021

Go Back

Full Screen

Close





- 4 -

**>>** 

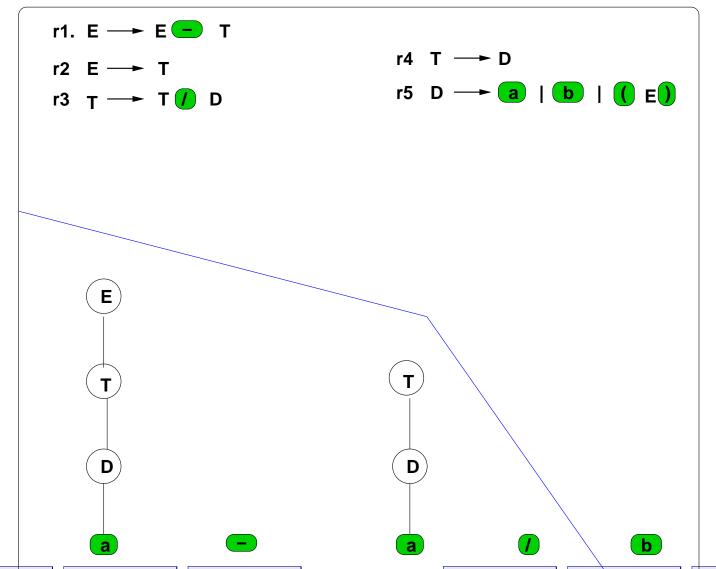
PL December 28, 2021

Go Back

Full Screen

Close





•

**>** 

▶▶

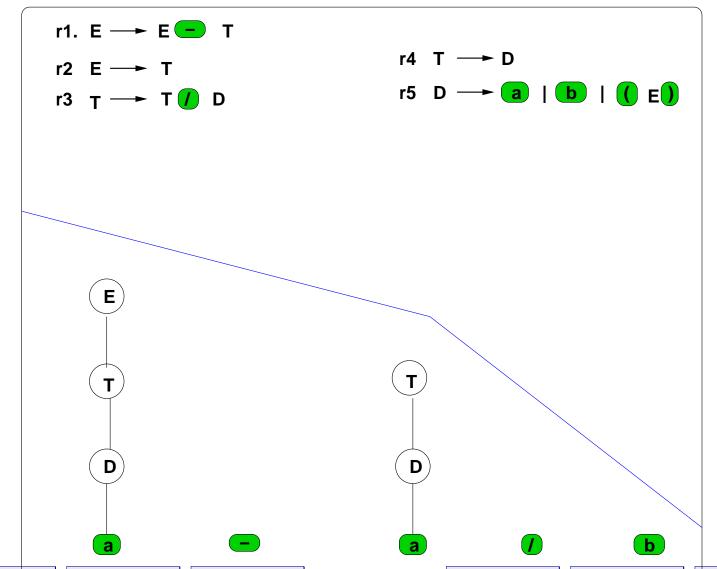
PL December 28, 2021

Go Back

FULL SCREEN

Close





**>** 

**>>** 

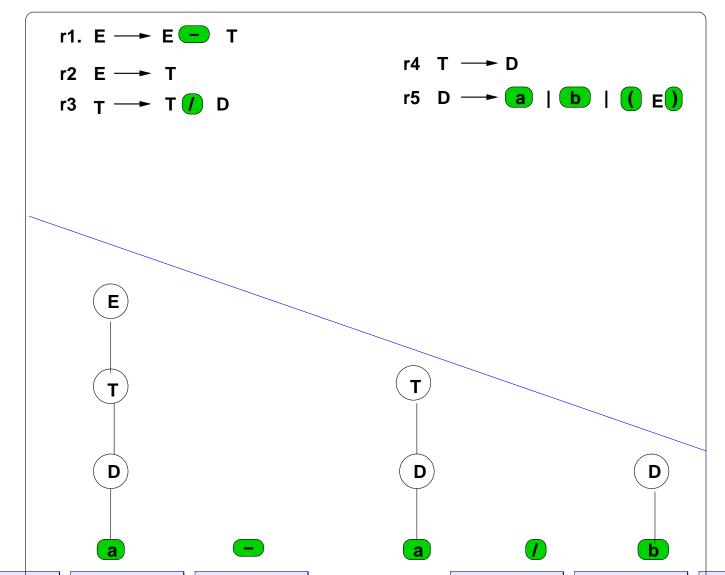
PL December 28, 2021

Go Back

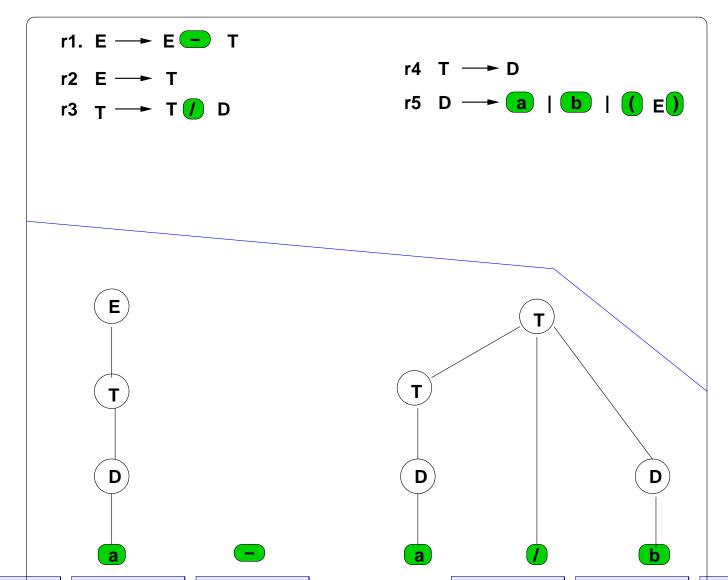
Full Screen

Close

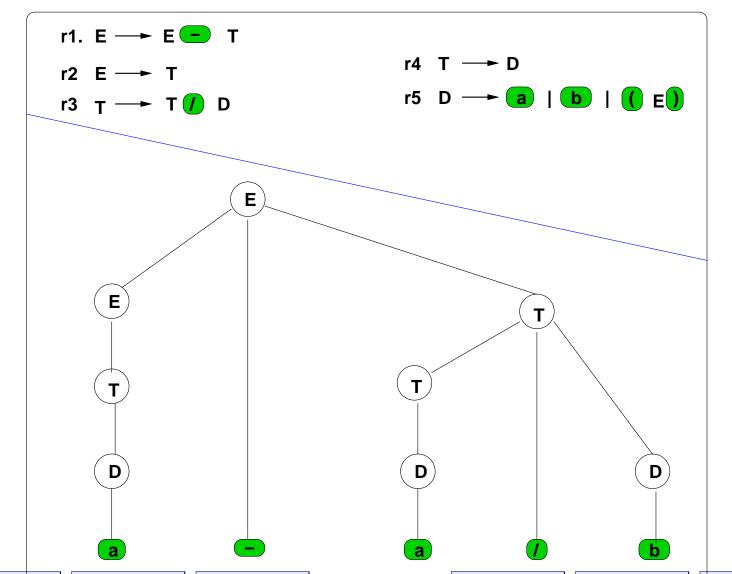












**>** 

▶▶

PL December 28, 2021

Go Back

Full Screen

Close



#### Parsing: Summary: 1

- All high-level languages are designed so that they may be parsed in this fashion with only a single token look-ahead.
- Parsers for a language can be automatically constructed by parser-generators such as Yacc, Bison, ML-Yacc and CUP in the case of Java.
- Shift-reduce conflicts if any, are automatically detected and reported by the parser-generator.
- Shift-reduce conflicts may be avoided by suitably redesigning the context-free grammar.

44



Go Back

FULL SCREEN

Close

322 OF 721



323 of 721

#### Parsing: Summary: 2

- Very often shift-reduce conflicts may occur because of the prefix problem. In such cases many parser-generators resolve the conflict in favour of shifting.
- There is also a possiblility of reduce-reduce conflicts. This usually happens when there is more than one nonterminal symbol to which the contents of the stack may reduce.
- A minor reworking of the grammar to avoid redundant non-terminal symbols will get rid of reduce-reduce conflicts.

The Big Picture

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE



 ✓
 ✓

 PL December 28, 2021

 Go Back

 Full Screen

 Close

 324 of 721





## Parsing Problems 1

The main question in shift-reduce parsing is:

When to *shift* and when to *reduce*?

To answer this question we require

- more information from the input token stream,
- to look at the rest of the input token stream and then take a decision.

But the decision has to be automatic. So the parser requires some rules. Once given the rules we may construct the parser to follow the rules.

44







Go Back

FULL SCREEN

Close



## Parsing Problems 2

But for a very large program it may be impossible to look at all the input before taking a decision. So clearly the parser can look at only a limited amount of the input to take a decision. So The next question:

# How much of the input token stream would the parser require?

Disregarding the very next input token as always available, the length of the extra amount of input required for a shift-reduce decision is called the lookahead.

44

**◄** 



**▶**▶

Go Back

Full Screen

Close



## Parsing Problems 3

Once all the input has been read, the parser should be able to decide

in case of a valid sentence that it should only apply reduction rules and attempt to reach the start symbol of the grammar only through reductions and

in case of an invalid sentence that a grammatical error has occurred in the parsing process

To solve this problem we augment every grammar with a new start symbol S and a new terminal token \$ and augment the grammar with a new special rule. For our previous grammar we have the new rule

$$S \to E$$
\$

44

**∢** 

**>** 

PL December 28, 2021

Go Back

Full Screen

Close



## Augmented Grammar

Consider the following (simplified) augmented grammar with a single binary operator — and parenthesis. We also number the rules.

- 1.  $S \rightarrow E$ \$
- $2. E \rightarrow E-T$
- $3. E \rightarrow T$
- $4. T \rightarrow a$
- 5.  $T \rightarrow (E)$

PL December 28, 2021

Full Screen

Close

# dse

# LR(0) Languages

LR(0) languages are those context-free languages that may be parsed by taking deterministic shift-reduce decisions only based on the contents of the parsing stack and without viewing any lookahead.

- "L" refers to reading the input from left to right,
- "R" refers to the (reverse) of rightmost derivation
- "0" refers to no-lookahead...
- Many simple CFLs are LR(0). But the LR(0) parsing method is too weak for most high-level programming languages.
- But understanding the LR(0) parsing method is most crucial for understanding other more powerful LR-parsing methods which require lookaheads for deterministic shift-reduce decision-making

44

**∢** 

PL December 28. 2021

Go Back

Full Screen

Close



## LR-Parsing Invariant

In any LR-parsing technique the following invariant holds.

For any syntactically valid sentence generated by the augmented grammar, the concatenation of the stack contents with the rest of the input gives a sentential form of a rightmost derivation.

Hence given at any stage of the parsing if  $\alpha \in (N \cup T)^*$  is the contents of the parsing stack and  $x \in T^*$ \$ is the rest of the input that has not yet been read, then  $\alpha x$  is a sentential form of a right-most derivation.

PL December 28, 2021

**⋖**◀

**>** 

.▶

Go Back

Full Screen

Close

# LR(0) Item

An LR(0) item consists of an LR(0) production rule with a special marker on the right hand side of rule.

- The marker is different from any of the terminal or nonterminal symbols of the grammar.
- The marker separates the contents of the stack from the expected form of some prefix of the rest of the input.
- ullet Given a rule  $X \to lpha$ , where X is a nonterminal symbol and lpha is a string consisting of terminal and non-terminal symbols, an LR(0) item is of the form

 $X \to \beta \Lambda \gamma$ 

where 
$$\alpha = \beta \gamma$$
.

• For each rule  $X \to \alpha$ , there are  $|\alpha| + 1$  distinct LR(0) items – one for each position in  $\alpha$ .



## What does an LR(0) item signify?

The LR(0) item

$$X \to \beta_{\blacktriangle} \gamma$$

signifies that at some stage of parsing

- ullet eta is the string (of terminals and nonterminals) on the top of the stack and
- ullet some prefix of the rest of the input can be generated by  $\gamma$

so that whenever  $\beta\gamma$  appears on the stack,  $\beta\gamma$  may be reduced immediately to X.

**⋖**◆

**>** 

▶▶

Go Back

Full Screen

Close



## LR0 Parsing Strategy

The LR0 parsing strategy is to

- 1. construct a DFA whose alphabet is  $N \cup T \cup \{\$\}$
- 2. use the parsing stack to perform reductions at appropriate points

The LR0 parsing table is hence a DFA with 3 kinds of entries.

 $\mathbf{shift}$  i in which a terminal symbol is shifted on to the parsing stack and the DFA moves to state i.

reduce j a reduction using the production rule j is performed  $goto\ k$  Based on the contents of the stack, the DFA moves to state k.

44

▶

▶▶

Go Back

Full Screen

Close



## Favourite Example

## Consider our favourite augmented grammar

- $1. S \rightarrow E$
- $2. E \rightarrow E-T$
- $3. E \rightarrow T$
- $4. T \rightarrow a$
- 5.  $T \rightarrow (E)$

44

I



PL December 28, 2021

Full Screen

Close



#### Rule 1: Items

Rule 1

$$R1. S \rightarrow E$$

has the following three items

$$I1.1 S \rightarrow L\$$$
 $I1.2 S \rightarrow E_{\$}$ 
 $I1.3 S \rightarrow E_{\$}$ 

one for each position on the right hand side of the rule.



Full Screen

CLOSE



#### Rule 2: Items

Rule 2

$$R2. E \rightarrow E-T$$

has the following items

$$I2.1 E \rightarrow E-T$$

$$I2.2 E \rightarrow E - T$$

$$I2.3 E \rightarrow E - T$$

$$I2.4~E \rightarrow E-T_{\blacktriangle}$$

44

**◄** 



▶▶

Go Back

Full Screen

Close



#### Rule 3: Items

Rule 3

 $R3. E \rightarrow T$ 

has just the items

$$I3.1 E \rightarrow AT$$

$$I3.2 E \rightarrow T_{\blacktriangle}$$







#### Rule 4: Items

Rule 4

has the items

$$R4. T \rightarrow a$$

$$\begin{array}{c} I4.1 \ T \rightarrow \underline{\quad a} \\ I4.2 \ T \rightarrow \underline{\quad a} \\ \end{array}$$

44

◀





#### Rule 5: Items

Rule 5

has the items

$$R5. T \rightarrow (E)$$

$$I5.1 T \rightarrow (E)$$

$$I5.2 T \rightarrow (\blacktriangle E)$$

$$I5.3 T \rightarrow (E_{\blacktriangle})$$

$$I5.4 T \rightarrow (E)_{\blacktriangle}$$

**⋖**◀







# Significance of I1.\*

- I1.1  $S \rightarrow E$ \$. Hence
  - 1. The parsing stack is empty and
  - 2. the entire input (which has not been read yet) should be reducible to E followed by the \$.

PL December 28, 2021

- I1.2  $S \rightarrow E_{\blacktriangle}$ \$. Hence
  - $1.\,E$  is the only symbol on the parsing stack and
  - 2. the rest of the input consists of the terminating symbol \$.
- I1.3  $S \rightarrow E$ . Hence
  - 1. There is no input left to be read and
  - 2. the stack contents may be reduced to the start symbol

44

**•** 

▶▶

Go Back

Full Screen

Close

341 of 721



#### DFA States: Initial and Final

- Clearly the *initial* state S1 of the DFA will correspond to item I1.1.
- There should be a state corresponding to item 11.2.
- There should be a goto transition on the nonterminal symbol E from the initial state (corresponding to item 11.1) to the state corresponding to item 11.2.
- The accepting state of the DFA will correspond to item item 11.3.
- There would also be a **shift** transition on \$ from the state corresponding to item 11.2 to the accepting state corresponding to item 11.3.
- There should be a reduce action using rule 1 when the DFA reaches the state corresponding to item 11.3.

44

**◄** 



PL December 28. 2021



Full Screen

Close



#### Input Possibilities: 1

#### Consider item 11.1.

- 1. How will a grammatically valid sentence input reduce to E\$? From the grammar it is obvious that this can happen *only if* the input is of a form such that
  - (a) it can be reduced to E-T (recursively) or
  - (b) it can be reduced to T
- 2. How can the input be reduced to the form T?
- 3. How can the input be reduced to the form E-T?

44

**•** 

▶▶

PL December 28, 2021

Go Back

FULL SCREEN

Close



#### Input Possibilities: 2

#### Consider item 11.1.

- 1. How will a grammatically valid sentence input reduce to E\$?
- 2. How can the input be reduced to the form T?
  - (a) If the enire input consists of only a then it could be reduced to T or
  - (b) If the entire input could be reduced to the form (E) then it could be reduced to T.
- 3. How can the input be reduced to the form E-T?

44

▶▶

PL December 28, 2021

Go Back

FULL SCREEN

Close



#### Input Possibilities: 3

#### Consider item 11.1.

- 1. How will a grammatically valid sentence input reduce to E\$?
- 2. How can the input be reduced to the form T?
- 3. How can the input be reduced to the form E-T?
  - (a) If the entire input could be split into 3 parts  $\alpha$ ,  $\beta$  and  $\gamma$  such that
    - i.  $\alpha$  is a prefix that can be reduced to E, and
    - ii.  $\beta = -$ , and
    - iii.  $\gamma$  is a suffix that can be reduced to T then it could be reduced to E-T

44

**▶** 

▶▶

Go Back

Full Screen

Close

345 of 721



#### Closures of Items

Theoretically each item is a state of a NFA. The above reasoning leads to forming closures of items to obtain DFA states, in a manner similar to the the subset construction. Essentially all NFA states with similar initial behaviours are grouped together to form a single DFA state.

NFA to DFA construction

44

**▶** 

▶▶

Go Back

Full Screen

Close



```
Algorithm 4.4
```

CLOSUREOFITEMS  $(I) \stackrel{df}{=}$ 

Requires: Set  $I \subseteq \mathcal{I}$  of LR(0) items of a CFG with rule set P Ensures: Closure of I for a subset  $I \subseteq \mathcal{I}$  of items repeat for each  $A \to \alpha_{\blacktriangle} X \beta \in I$  do  $\begin{cases} \text{for each } X \to \gamma \in P \\ \text{do } I := I \cup \{X \to {}_{\blacktriangle}\gamma\} \end{cases}$  until no more changes occur in I

44



PL December 28, 2021



## State Changes on Nonterminals

As in the case of the NFA to DFA construction with each state transition we also need to compute closures on the target states.









```
Algorithm 4.5 GoTo (I, X) \stackrel{df}{=}

Requires: I \subseteq \mathcal{I} of LR(0) items of a CFG G = \langle N, T, P, S \rangle, X \in N Ensures: States of the DFA: Each state in the DFA is a closure of items J := \emptyset; for each A \to \alpha_{\blacktriangle} X \beta \in I do J := J \cup \{A \to \alpha X_{\blacktriangle} \beta\}; K := \text{ClosureOfItems}(J); return (K)
```

44



PL December 28, 2021



#### State S1

$$S1 = \text{CLosure}(\{S \to \blacktriangle E\$\})$$

$$= \{S \to \blacktriangle E\$, E \to \blacktriangle E - T, E \to \blacktriangle T,$$

$$T \to \blacktriangle a, T \to \blacktriangle (E)\}$$

$$S1 \xrightarrow{(} \mathbf{Closure}(\{T \to (\blacktriangle E)\}) = S2$$

$$S1 \xrightarrow{E} \mathbf{CLOSURE}(\{S \to E_{\blacktriangle}\$, E \to E_{\blacktriangle} - T\}) = S3$$

$$S1 \xrightarrow{T} \mathbf{CLosure}(\{E \to T_{\blacktriangle}\}) = S7$$

$$S1 \xrightarrow{a} \mathbf{CLosure}(\{T \to a_{\blacktriangle}\}) = S8$$

PL December 28, 2021

44

**•** 

**>>** 

Go Back

Full Screen

Close



#### State S2

$$S2 = \text{Closure}(\{T \to (\blacktriangle E)\})$$

$$= \{T \to \blacktriangle(E), E \to \blacktriangle E - T, E \to \blacktriangle T,$$

$$T \to \blacktriangle a, T \to \blacktriangle(E)\}$$

$$S2 \xrightarrow{(} \mathbf{Closure}(\{T \to (\blacktriangle E)\}) = S2$$

$$S2 \xrightarrow{E} \mathbf{CLOSURE}(\{T \to (E_{\blacktriangle}), E \to E_{\blacktriangle} - T\}) = S9$$

$$S2 \xrightarrow{T} \mathbf{CLosure}(\{E \to T_{\blacktriangle}\}) = S7$$

$$S2 \xrightarrow{a} \mathbf{CLosure}(\{T \to a_{\blacktriangle}\}) = S8$$

44

**•** 

**▶**▶

PL December 28, 2021 Go BACK

Full Screen

Close

$$S3 = \frac{\text{Closure}(\{S \to E_{\blacktriangle}\$, E \to E_{\blacktriangle} - T\})}{= \{S \to E_{\blacktriangle}\$, E \to E_{\blacktriangle} - T\}}$$



However,

$$S3 \xrightarrow{-} \mathbf{Closure}(\{E \to E - \mathbf{A}T\})$$

and

$$\begin{array}{l} \mathbf{CLOSURE}(\{E \rightarrow E - \blacktriangle T\}) \\ = \{E \rightarrow E - \blacktriangle T, T \rightarrow (\blacktriangle E), T \rightarrow \blacktriangle a\} \\ = S4 \end{array}$$

The closures of the other reachable sets of items are themselves.

$$\bullet$$
  $S5 = \{E \rightarrow E - T_{\blacktriangle}\}$ 

$$\bullet S6 = \{S \to E\$_{\blacktriangle}\}$$

$$\bullet S7 = \{E \to T_{\blacktriangle}\}\$$

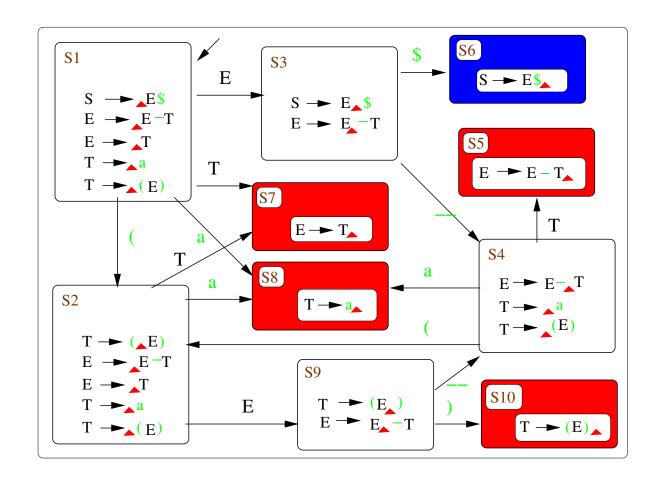
$$\bullet S8 = \{T \to a_{\blacktriangle}\}$$

$$\bullet S9 = \{T \to (E_{\blacktriangle}), E \to E_{\blacktriangle} - T\}$$



# Example: DFA

#### Parsing Table



44

**•** 

**>>** 

PL December 28, 2021

Go Back

Full Screen

Close

# Example: Parsing Table



States			Inpu	t		Nonterminals			
	a			\$		S	$\mid E \mid$	T	
S1	S8	<b>S</b> 2					G3	G7	
S2	S8	<b>S</b> 2					G9	G7	
S3				ACC	S4				
S4	S8	<b>S</b> 2						G5	
S5	R2	R2	R2	R2	R2				
<b>S6</b>	R1	R1	R1	R1	R1				
S7	R3	R3	R3	R3	R3				
S8	R4	R4	R4	R4	R4				
S9			S10		S4				
S10	R5	R5	R5	R5	R5				

Note: All empty entries denote errors

PL December 28, 2021

Go Васк

Full Screen

Close 354 of 721



**Example 4.16** Consider the following simple input viz. a\\$. Here are the parsing steps.



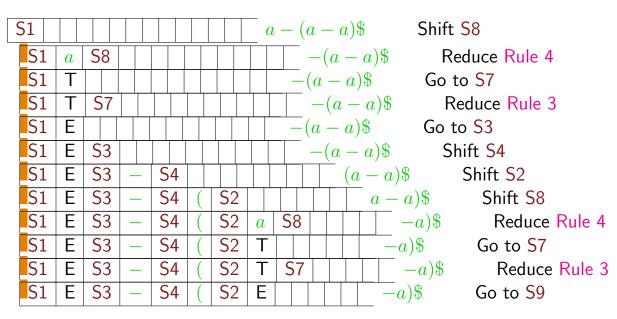
**◄**◀





#### **Example 4.17** Here is a slightly more complex input a - (a - a)\$.

DFA Parsing Table



DFA Parsing Table



DFA Parsing Table

$\begin{array}{ c c c c c c c c c c c c c c c c c c c$														
S1	Е	S3	_	S4	(	S2	Е	S9	_	S4			a)\$	Shift S8
S1	E	S3	_	S4	(	S2	E	S9	_	S4	a	S8	)\$	Reduce Rule 4
S1	E	S3	_	S4	(	S2	E	S9	_	S4	Т		$\overline{}$	Go to S5
S1	E	S3	_	S4	(	S2	E	S9	_	S4	Т	S5	)\$	Reduce Rule 2
S1	E	S3	_	S4	(	S2	Е				)\$		Go to	S9
S1	E	S3	_	S4	(	S2	Е	S9				)\$	Sh	ift S10
S1	E	S3	_	S4	(	S2	Е	S9	)	S10		$\top$	\$	Reduce Rule 5
S1	E	S3	_	S4	Т					\$		Go	to S5	
S1	E	S3	_	S4	Т	S5					\$		Reduce	Rule 2
S1	E							\$		Go to S3				
S1	Е	S3							\$	A	Acce	ept		

PL December 28, 2021

Go Back



#### Exercise 4.3

- 1. Design a LR(0) parser for the grammar of palindromes. Identify whether there are any conflicts in the parsing table.
- 2. Design a LR(0) parser for the grammar of Matching brackets and identify any conflicts.
- 3. Design a context-free grammar for a language on the terminal symbols a and b such that every string has more as than bs. Design a LR(0) parser for this grammar and find all the conflicts, if any.
- 4. Since every regular expression may also be represented by a context-free grammar design an LR(0) parser for comments in C.

44

**▶**▶

Go Back

Full Screen

Close



## CFG = RLG + Bracket Matching

We use the idea that a context-free grammar is essentially a regular grammar with parentheses matching to arbitrary depths. Hence a DFA with some reductions introduced may work.

We modify the grammar to have a special terminal symbol called the end-marker (denoted by \$). Now consider the following simple grammar with a single right-associative binary operator ^ and bracket-matching.

We create a DFA of "items" which also have a special marker called the "cursor" ( $\blacktriangle$ ).

 ✓
 ✓

 Full Screen
 Close

 359 of 721

## LR(0) with Right-Association



#### Consider the following grammar

- $1. S \rightarrow E$ \$
- $2. E \rightarrow P \hat{E}$
- $3. E \rightarrow P$
- $4. P \rightarrow a$
- 5.  $P \rightarrow (E)$

The following items make up the initial state \$1 of the DFA

$$I1.1 S \rightarrow E$$

$$I2.1 E \rightarrow P \hat{E}$$

$$I3.1 E \rightarrow P$$

$$I4.1 P \rightarrow a$$

$$I5.1 P \rightarrow (E)$$

PL December 28, 2021

**⋖**∢

**>** 

**>>** 

Go Back

Full Screen

Close

## Shift-Reduce Conflicts in LR(0)



There is a transition on the nonterminal P to the state S2 which is made up of the following items.

$$I2.2 E \rightarrow P_{\blacktriangle}\hat{}E$$

$$I3.2 E \rightarrow P_{\blacktriangle}$$

Then clearly the LR(0) parser suffers a shift-reduce conflict because

- item 12.2 indicates a shift action,
- item 13.2 produces a reduce action

This in contrast to the parsing table produced earlier where reduce actions took place regardless of the input symbol. Clearly now that principle will have to be modified.

The parsing table in this case would have a shift action if the input in state S2 is a ^ and a reduce action for all other input symbols.

**44 •** 

PL December 28, 2021

Go Back

Full Screen

Close



#### **FOLLOW Sets**

We construct for each non-terminal symbol a set of terminal symbols that can *follow* this non-terminal in any rightmost derivation. In the previous grammar we have

$$follow(E) = \{\$, \}$$
  
 $follow(P) = \{^{\$}\}$ 

Depending upon the input symbol and whether it appears in the FOLLOW set of the non-terminal under question we resolve the shift-reduce conflict. This modification to LR(0) is called Simple LR (SLR) parsing method. However SLR is not powerful enough for many useful grammar constructions that are encountered in many programming languages.

 ■
 ■

 PL December 28, 2021
 Go Back

 Full Screen
 Close

 362 of 721



## Computing FIRST Sets

In order to compute FOLLOW sets we require FIRST sets of sentential forms to be constructed too.

- 1. first  $(a) = \{a\}$  for every terminal symbol a.
- 2.  $\varepsilon \in first(X)$  if  $X \to \varepsilon \in P$ .
- 3. If  $X \to Y_1 Y_2 \cdots Y_k \in P$  then  $first(Y_1) \subseteq first(X)$
- 4. If  $X \to Y_1 Y_2 \cdots Y_k \in P$  then for each i: i < k such that  $Y_1 Y_2 \cdots Y_i \Rightarrow \varepsilon$ ,  $first(Y_{i+1}) \subset first(X)$ .

44

◀

| | |

PL December 28, 2021

Go Back

Full Screen

Close



## Computing FOLLOW Sets

Once FIRST has been computed, computing FOLLOW for each non-terminal symbol is quite easy.

- 1.  $\$ \in follow(S)$  where S is the start symbol of the augmented grammar.
- 2. For each production rule of the form  $A \to \alpha B\beta$ ,  $first(\beta) \{\varepsilon\} \subseteq follow(B)$ .
- 3. For each production rule of the form  $A \to \alpha B\beta$ , if  $\varepsilon \in first(\beta)$  then  $follow(A) \subseteq follow(B)$ .
- 4. For each production of the form  $A \to \alpha B$ ,  $follow(A) \subseteq follow(B)$ .

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 364 OF 721



#### if-then-else vs. if-then

Most programming languages have two separate constructs if-then and if-then-else. We abbreviate the keywords and use the following symbols

Tokens	Symbols
if	i
then	t
else	е
booleans	b
other expressions	a

 →
 Image: Supervision of the property of the property



## if-then-else vs. if-then (Contd.)

and construct the following two augmented grammars  $G_1$  and  $G_2$ .

44





PL December 28, 2021

Go Back

Full Screen

Close



#### Exercise 4.4

- 1. Prove that grammar  $G_2$  is ambiguous.
- 2. Construct the LR(0) parsing tables for both  $G_1$  and  $G_2$  and find all shift-reduce conflicts in the parsing table.
- 3. Construct the FOLLOW sets in each case and try to resolve the conflicts.
- 4. Show that the following augmented grammar <u>cannot</u> be parsed (i.e. there are conflicts that cannot be resolved by FOLLOW sets) either by LR(0) or SLR parsers. (Hint First construct the LR(0) DFA).
  - 1.  $S \rightarrow E$ \$
  - $2. \quad E \quad \rightarrow \quad L = R$
  - 3.  $E \rightarrow R$
  - 4.  $L \rightarrow *R$
  - $5. \quad L \quad \rightarrow \quad a$
  - 6.  $R \rightarrow L$



#### 5. Attributes & Semantic Analysis

#### **Introduction to Semantics**

Context-free grammars (actually EBNF) are used to describe the rules that define the grammatical structure of phrases and sentences in the language. However a manual for a programming language also needs to describe the meaning of each construct in the language both alone and in conjunction with other constructs. This is to enable users of the language to write correct programs and to be able to predict the effect of each construct. Implementors of the language need correct definitions of the meanings to be able to construct correct implementations of the language.

Syntax defines a well-formed program. Semantics defines the meaning of a syntactically correct program. However not all well-formed programs have well defined meanings. Thus semantics also separates meaningful programs from merely syntactically correct ones.

"Meaning" in the case of programming languages often refers to the execution behaviour of the program or the individual constructs. This is useful from an implementation point of view. From a user programmer's point of view It is possible to view a programming language as a precise description mechanism that is independent of execution behaviour and restrict meaning to the "effect" that a program or a construct has on some input (state).

While there are precise means of defining the syntax af the language, most language manuals describe the meanings of the constructs in natural language prose. This unfortunately is not very desirable as natural language tends to be too verbose, imprecise and very often ambiguous. On the other hand, if users and implementors have to be on the same page as regards the behaviour of programs and individual programming constructs a precise and unambiguous definition is required for

**⋖**◀

**◄** 

.

Go Back

Full Screen

Close

this description. Typically a user programmer may misunderstand what a program or a construct will do when executed. Implementors may interpret the meaning differently and hence different implementations of the language may yield different results on the same program.

While there are several formalisms for defining meanings of the constructs of a programming language, they all share the following characteristics in order to maintain a certain uniformity and applicability for any program written in the language

- Meanings should be *syntax-directed* i.e. meanings should be based on the syntactical definition of the language in the sense that it follows the hierarchy of the non-terminals in the grammar. The syntax (grammar) of the language therefore provides the framework for the semantics of the language.
- The meaning should be *compositional* i.e. the meaning of a compound construct should be expressed in terms of the meanings of the individual components in the construct. Hence it is important that the meanings of the meanings of the meanings of the compound constructs may be expressed in terms of the meanings of the individual components of the compound construct.

44





# Attributes & Semantic Analysis











Go Back





#### 5.1. Context-sensitive analysis and Semantics

The Big Picture

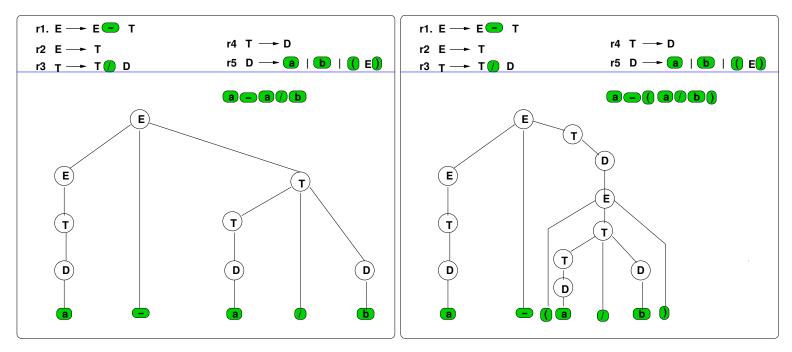


Figure 5: Derivation trees or Concrete parse trees example 5.1

The parser for a context-free grammar transforms the token stream into a derivation tree (which we also call a concrete

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 371 OF 721



parse tree)<sup>3</sup>. What we actually require in order to perform a computation is really an abstract syntax tree.

**Example 5.1** Consider the two sentences a - a/b and a - (a/b) which are both valid sentences generated by the grammar of our favourite example.

The (possibly modified grammar) required for parsing

- treats all tokens uniformly since the phrase structure of the grammar is all-important during the parsing process,
- introduces bracketing and punctuation marks for
  - disambiguation and to override associativity when needed,
  - to facilitate easy parsing

But these symbols do not by themselves carry any semantic<sup>4</sup> information.

- also has many more non-terminal symbols that are required for parsing, but which carry no semantic significance
  - either for the end-user of the language
  - or for the later phases of the compilation process.

Both expressions in example 5.1 have the same meaning (semantics) if we assume that the operations are subtraction and division over integers respectively, and that division has higher precedence than subtraction. But the sentences are

44







Go Back

FULL SCREEN

Close

<sup>&</sup>lt;sup>3</sup>The term parse tree is a much abused term used to refer to anything from a derivation tree to an abstract syntax tree (AST).

<sup>&</sup>lt;sup>4</sup>Semantic analysis is another much abused term, often used by compiler writers to included even merely context-sensitive information.

syntactically different and correspondingly have different parse trees (see fig. 5). Both the expressions may be represented by the following abstract syntax tree (AST) which gives the hierarchical structure of the expression.

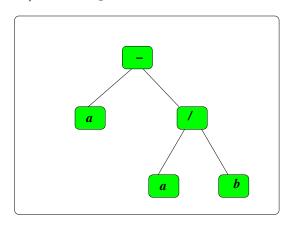


Figure 6: Abstract syntax tree (AST) for the sentences in fig. 5

#### Notice that in figure 6

- Every node in the AST is labelled by a token.
- The AST abstracts away from non-terminals which have significance only for the parsing of the expression and have no semantic significance whatsoever,
- The AST abstracts away from bracketing and punctuation mechanisms and provides a hierarchical structure containing only the *essential* operators and operands.
- The AST clearly distinguishes the operators (based on their arity) from the operands (which are leaves of the AST).

PL December 28, 2021

Go Back

Full Screen

CLOSE



## Context-sensitive Analysis: Preamble

The Big Picture

- Every programming language can be used to program any computable function, assuming of course, it has
  - unbounded memory, and
  - unbounded time
- Context-free grammars are used to specify the phrase structure of a language in a manner that is *free of all context*.

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 374 OF 721

## Semantic Analysis



The Big Picture

- 1. Context-free grammars are not powerful enough to represent all computable functions.
  - **Example 5.2** The language  $\{a^nb^nc^n|n>0\}$  is not context-free but can be generated by a context-sensitive grammar.
- 2. Semantic analysis is an essential step to
  - producing the abstract syntax trees (AST)
  - generating IR-code, since it requires the computation of certain *bits and pieces of information* called attributes (which include information to be entered into the symbol table or useful for error-handling)
  - allocating memory for individual "objects" (variables, constants, structures, arrays etc.)

44

**>** 

PL December 28, 2021

Go Back

Full Screen

Close

## Context-sensitive Analysis: 1



The Big Picture

- There are aspects of a program that cannot be represented/enforced by a context-free grammar definition. Examples include
  - scope and visibility issues with respect to identifiers in a program.
  - type consistency between declaration and use.
  - -correspondence between formal and actual parameters (example 5.2 is an abstraction where  $a^n$  represents a function or procedure declaration with n formal parameters and  $b^n$  and  $c^n$  represent two calls to the same procedure in which the number of actual parameters should equal n).
- Many of these attributes are *context-sensitive* in nature. They need to be computed and if necessary propagated during parsing from wherever they are available.

 ✓
 ✓

 FULL SCREEN
 CLOSE

 376 OF 721



## Context-sensitive Analysis: 2

The Big Picture

- The parser of a programming language provides the *framework* within which the IR-code or even the target code is to be generated.
- The parser also provides a *structuring* mechanism that divides the task of code generation into bits and pieces determined by the individual nonterminals and production rules.
- The parser provides the *framework* from within which the semantic analysis (which includes the bits and pieces of information that are required for code generation) is performed

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 377 OF 721



→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 378 OF 721

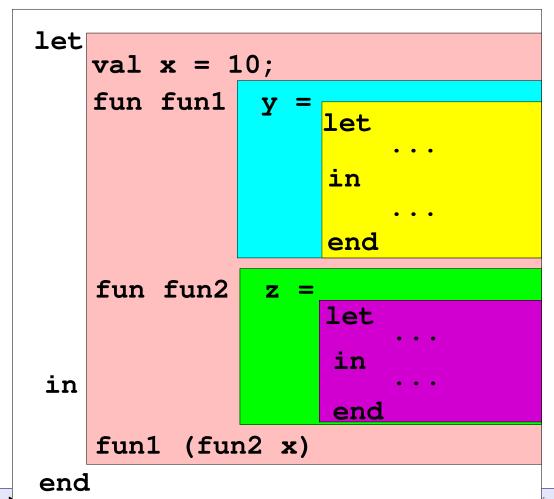


→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 379 OF 721



#### 6. Static Scope Rules

## Disjoint Scopes

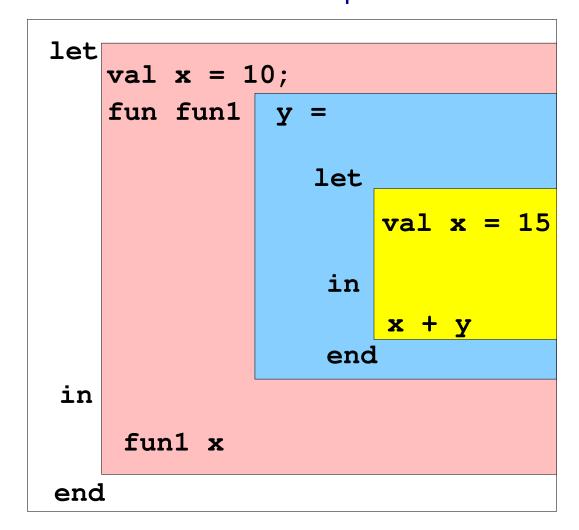


PL December 28, 2021

Close



## **Nested Scopes**



44

**•** 

**>>** 

PL December 28, 2021

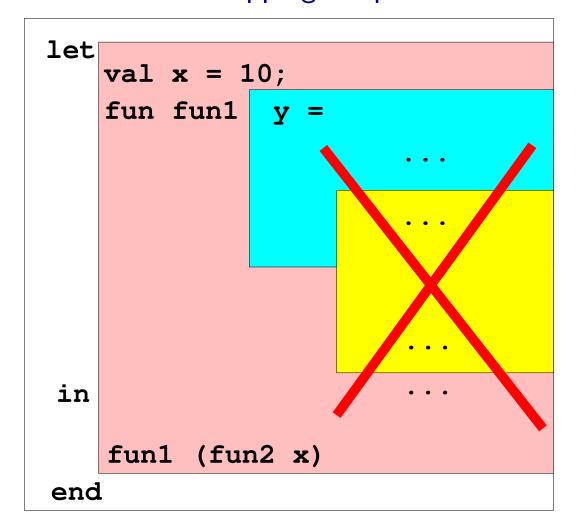
Go Back

Full Screen

Close



# Overlapping Scopes

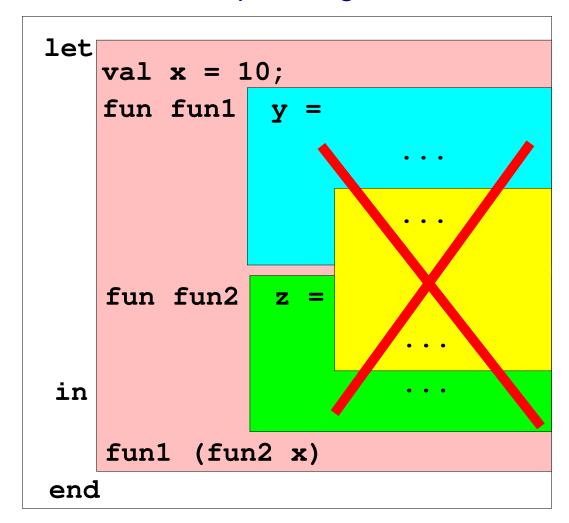


 →
 →
 Full Screen
 Close

 382 of 721



# Spannning



 ✓
 ✓

 PL December 28, 2021

 Go Back

 Full Screen

 Close

 383 of 721



## Scope & Names

- A name may occur either as being defined or as a use of a previously defined name
- The same name may be used to refer to different objects.
- The use of a name refers to the <u>textually</u> most recent definition in the innermost enclosing scope

diagram





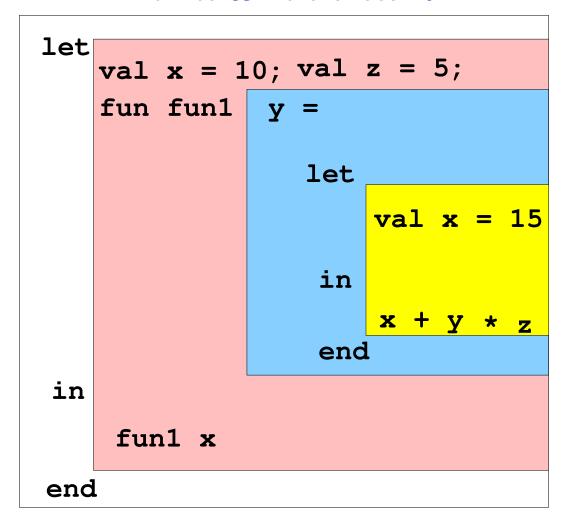


Go Back

Full Screen

Close





Back to Scope & Names

44

**|** 

**>>** 

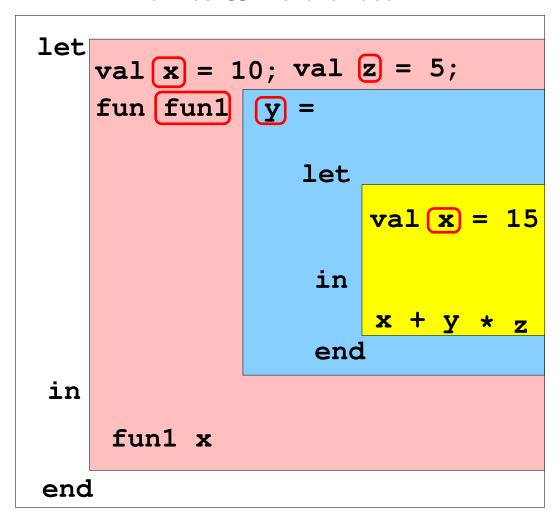
PL December 28, 2021

Go Back

Full Screen

Close





Back to Scope & Names

44

**•** 

**>>** 

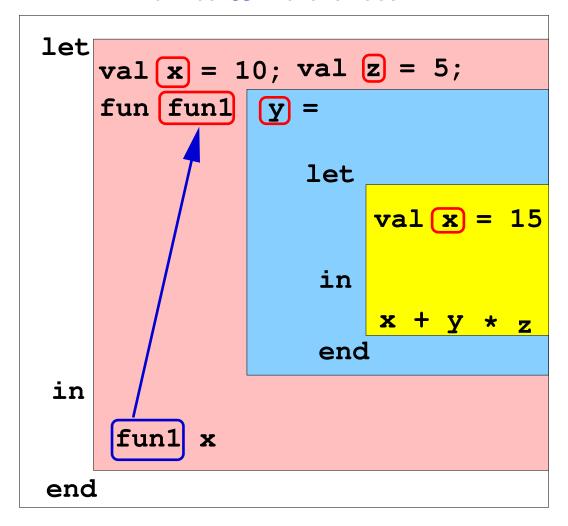
PL December 28, 2021

Go Back

Full Screen

Close





Back to Scope & Names

44

**•** 

**>>** 

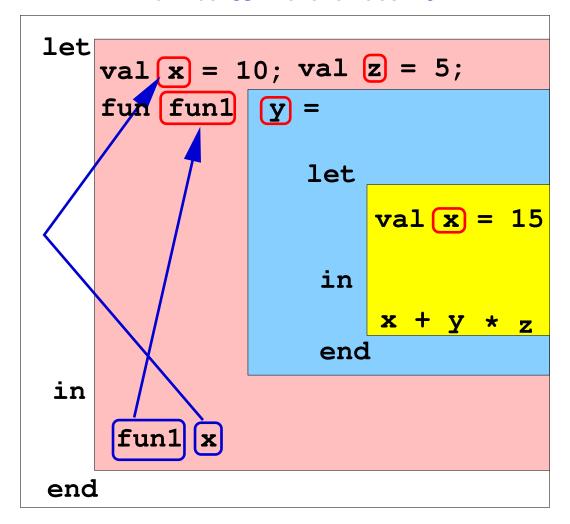
PL December 28, 2021

Go Back

Full Screen

Close





Back to Scope & Names

44

▶▶

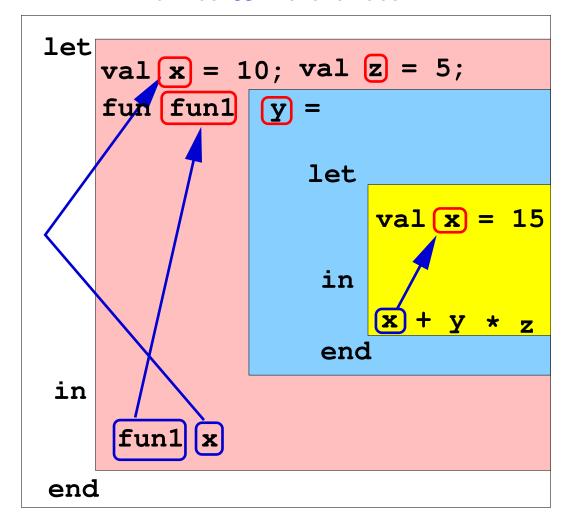
PL December 28, 2021

Go Back

Full Screen

Close





Back to Scope & Names

44

▶▶

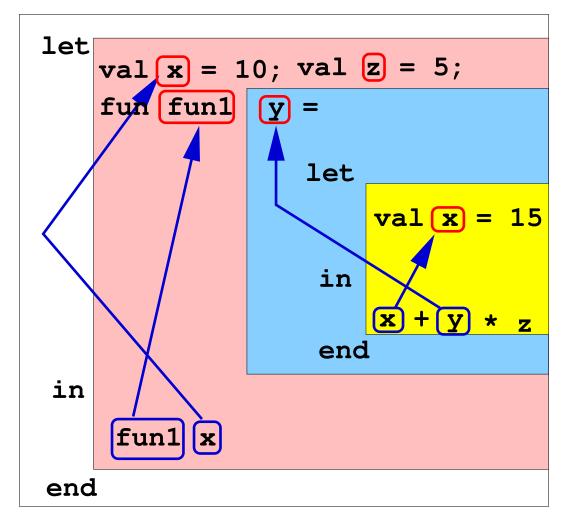
PL December 28, 2021

Go Back

Full Screen

Close





Back to Scope & Names

44

**>** 

**>>** 

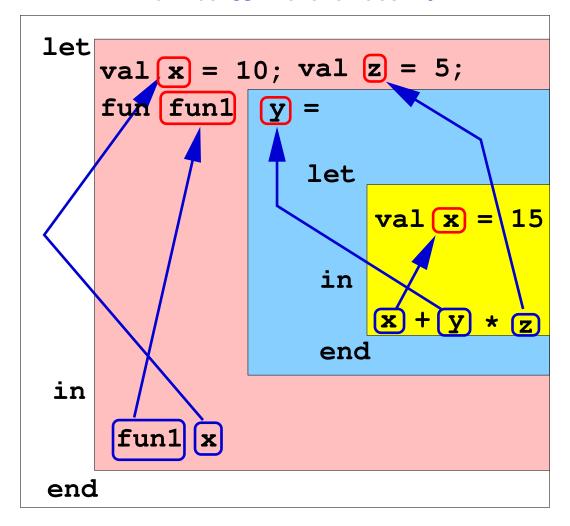
PL December 28, 2021

Go Back

Full Screen

Close





Back to Scope & Names

44

**•** 

**>>** 

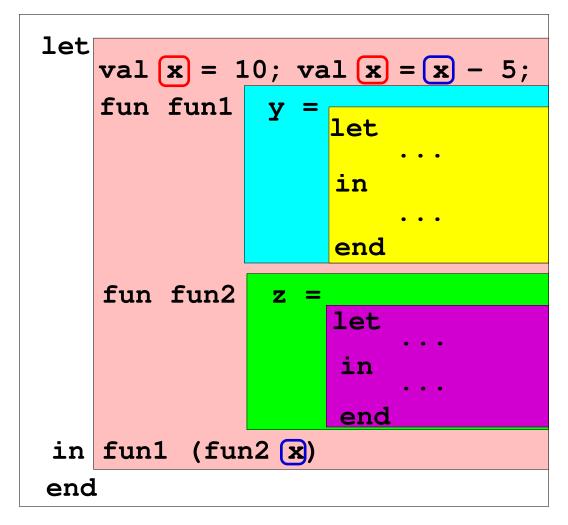
PL December 28, 2021

Go Back

Full Screen

Close





Back to Scope & Names

44

▶▶

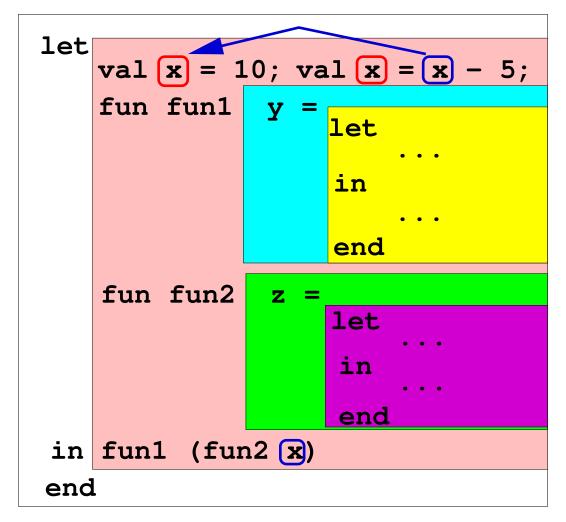
PL December 28, 2021

Go Back

Full Screen

Close





Back to Scope & Names

44

**•** 

**>>** 

PL December 28, 2021

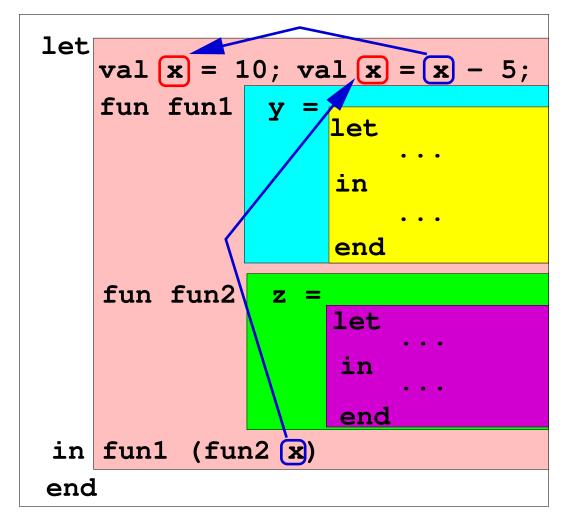
Go Back

Full Screen

Close

393 of 721





Back to Scope & Names

44

•

**>>** 

PL December 28, 2021

Go Back

Full Screen

Close



#### **Definition of Names**

## Definitions are of the form

```
qualifier \underline{name} \dots = body
```

- val name =
- fun  $\underline{name}$  ( argnames ) =
- local definitions in definition end

44

**•** 

**>>** 

Go Back

PL December 28, 2021

CK FULL SCREEN

CLOSE



#### Use of Names

Names are used in expressions. Expressions may occur

- by themselves to be evaluated
- $\bullet$  as the **body** of a definition
- $\bullet$  as the **body** of a let-expression

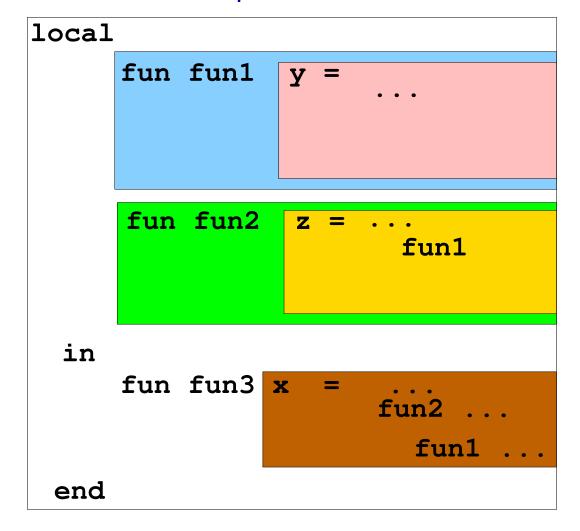
```
let definitions
in expression
end
```

use of local

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 396 OF 721



## Scope & local



4

**•** 

**>>** 

PL December 28, 2021

Go Back

Full Screen

Close



7. Runtime Structure











It is ever so. One of the poets, whose name I cannot recall, has a passage, which I am unable at the moment to remember, in one of his works, which for the time being has slipped my mind, which hits off admirably this age-old situation.

P. G. Wodehouse, The Long Hole in The Golf Omnibus

44

ı

▶▶

Go Back

Full Screen

Close



#### Run-time Environment

Memory for running a program is divided up as follows

Code Segment. This is where the object code of the program resides

Run-time Stack. Required in a *dynamic* memory management technique. Especially required in languages which support recursion. All data whose sizes can be determined *statically* before loading is stored in an appropriate stack-frame (activation record).

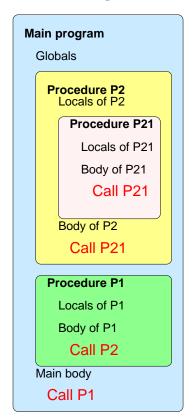
Heap. All data whose sizes are not determined statically and all data that is generated at run-time is stored in the heap.

 ■
 ■

 Image: Close of the propertion of the properties of the properties



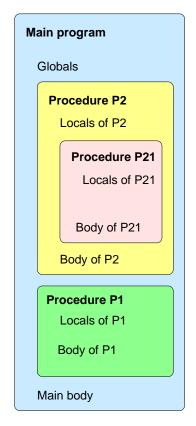
## A Calling Chain



 $\mathsf{Main} \to \mathsf{P1} \to \mathsf{P2} \to \mathsf{P21} \to \mathsf{P21}$ 

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 401 OF 721





Globals

Main









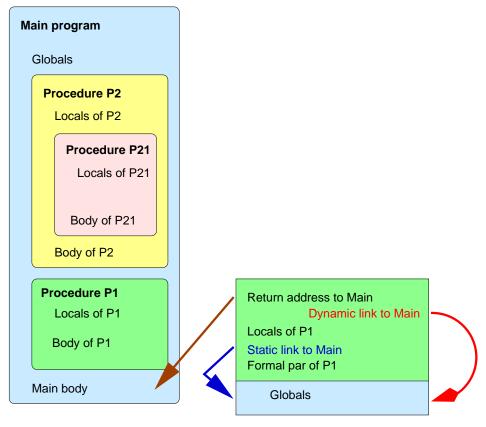






Close

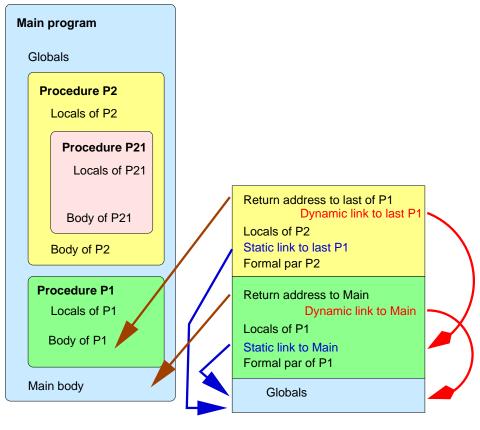




 $\mathsf{Main} \to \mathsf{P1}$ 

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 403 OF 721

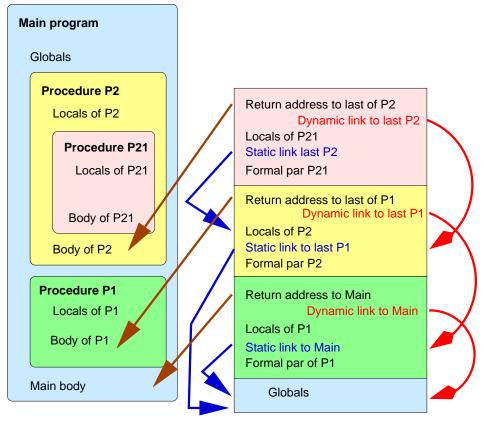




 $\mathsf{Main} \to \mathsf{P1} \to \mathsf{P2}$ 

 →
 Image: Second of the properties of the

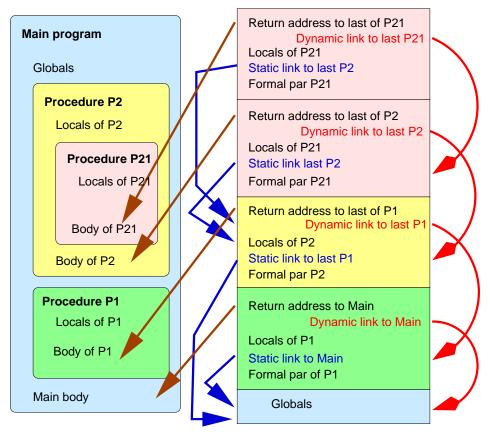




 $\mathsf{Main} \to \mathsf{P1} \to \mathsf{P2} \to \mathsf{P21}$ 

 →
 Image: Second of the properties of the





 $\mathsf{Main} \to \mathsf{P1} \to \mathsf{P2} \to \mathsf{P21} \to \mathsf{P21}$ 

Back to the Big Picture

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 406 OF 721



→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 407 OF 721





#### 8. Abstract Syntax

### Abstract Syntax Trees

The construction of ASTs from concrete parse trees is an example of a transformation that can be performed using a syntax-directed definition that has no side-effects.

Hence we define it using an attribute grammar.

**Definition 8.1** An attribute grammar is a formal way to define semantic rules and context-sensitive aspects of the language. Each production of the grammar is associated with a set of values or semantic rules. These values and semantic rules are collectively referred to as attributes.

 ■
 ■

 Image: Second of the propertion of the properties of the propertie



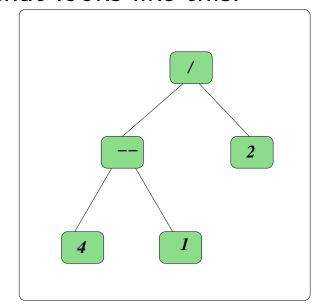
#### Abstract Syntax: 0

$$E \rightarrow E - T \mid T$$

$$T \rightarrow T/F \mid F$$

$$F \rightarrow \mathbf{n} \mid (E)$$

Suppose we want to evaluate an expression (4-1)/2. What we actually want is a tree that looks like this:



44

**•** 

**>>** 

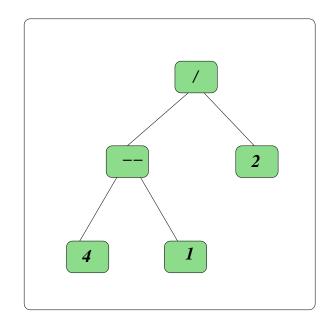
 J
 PL December 28, 2021

Go Back

FULL SCREEN

Close

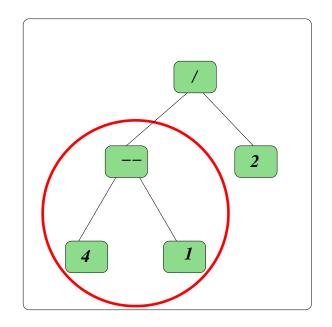




 ✓
 Full Screen

 Close
 411 of 721

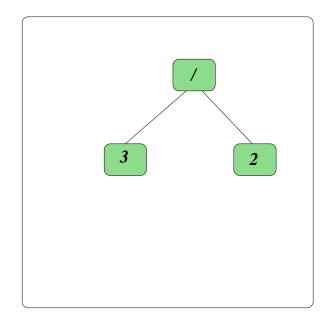




 ✓
 Full Screen

 Close
 412 of 721













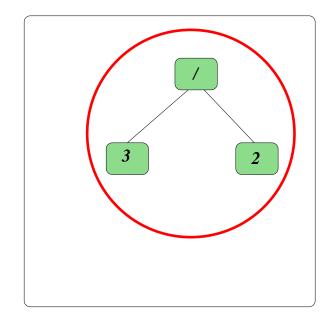






Close





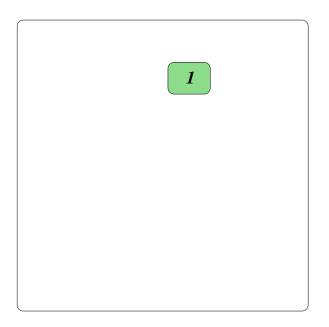












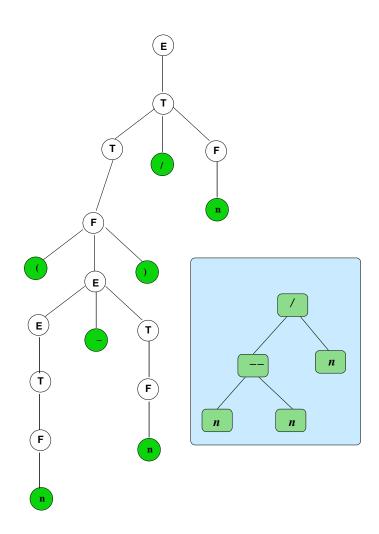
But what we  $actually\ get$  during parsing is a tree that looks like . . .







... *THIS!* 



44

**>** 

**>>** 

PL December 28, 2021

Go Back

Full Screen

Close



### Abstract Syntax

Shift-reduce parsing produces a concrete syntax tree from the rightmost derivation. The syntax tree is concrete in the sense that

- It contains a lot of redundant symbols that are important or useful only during the parsing stage.
  - punctuation marks
  - brackets of various kinds
- It makes no distinction between operators, operands, and punctuation symbols

On the other hand the abstract syntax tree (AST) contains no punctuations and makes a clear distinction between an operand and an operator.

44

▶▶

Go Back

Full Screen

Close

417 OF 721



## Abstract Syntax: Imperative Approach

We use attribute grammar rules to construct the abstract syntax tree (AST) from the parse tree.

But in order to do that we first require two procedures for tree construction.

makeLeaf(literal): Creates a node with label literal and returns a pointer or a reference to it.

makeBinaryNode(opr, opd1, opd2): Creates a node with label opr (with fields which point to opd1 and opd2) and returns a pointer or a reference to the newly created node.

Now we may associate a synthesized attribute called ptr with each terminal and nonterminal symbol which points to the root of the subtree created for it.

 ◄
 Image: Second of the content of the co

## Abstract Syntax Trees: Imperative



$$E_0 \rightarrow E_1 - T \triangleright E_0.ptr := makeBinaryNode(-, E_1.ptr, T.ptr)$$

$$E \rightarrow T \qquad \triangleright E.ptr := T.ptr$$

$$T_0 \rightarrow T_1/F \triangleright T_0.ptr := makeBinaryNode(/, T_1.ptr, F.ptr)$$

$$T \rightarrow F \qquad \triangleright T.ptr := F.ptr$$

$$F \rightarrow (E) \qquad \triangleright F.ptr := E.ptr$$

$$F \rightarrow \mathbf{n} \qquad \triangleright F.ptr := makeLeaf(\mathbf{n}.val)$$

### The Big Picture



PL December 28, 2021

Go Back

Full Screen



## Abstract Syntax: Functional Approach

We use attribute grammar rules to construct the abstract syntax tree (AST) functionally from the parse tree.

But in order to do that we first require two functions/constructors for tree construction.

makeLeaf(literal): Creates a node with label literal and returns the AST.

makeBinaryNode(opr, opd1, opd2): Creates a tree with root label opr (with sub-trees opd1 and opd2).

Now we may associate a synthesized attribute called ast with each terminal and nonterminal symbol which points to the root of the subtree created for it.

 ✓
 ✓

 Full Screen
 Close

 420 of 721

## Abstract Syntax: Functional



$$E_0 \rightarrow E_1 - T \triangleright E_0.ast := makeBinaryNode(-, E_1.ast, T.ast)$$

$$E \rightarrow T \qquad \triangleright E.ast := T.ast$$

$$T_0 \rightarrow T_1/F \triangleright T_0.ast := makeBinaryNode(/, T_1.ast, F.ast)$$

$$T \rightarrow F \qquad \triangleright T.ast := F.ast$$

$$F \rightarrow (E) \qquad \triangleright F.ast := E.ast$$

$$F \rightarrow \mathbf{n} \qquad \triangleright F.ast := makeLeaf(\mathbf{n}.val)$$

### The Big Picture

44











### Abstract Syntax: Alternative Functional

In languages like SML which support algebraic (abstract) datatypes, the functions makeLeaf(literal) and makeBinaryNode(opr, opd1, opd2) may be replaced by the constructors of an appropriate recursively defined datatype AST.

 ✓
 ✓

 Full Screen
 Close

 422 of 721



→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 423 OF 721



→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 424 OF 721



Syntax-Directed Translation

Syntax-directed Translation







PL December 28, 2021

Go Back



#### **Attributes**

An attribute can represent anything we choose e.g.

- a string
- a number (e.g. size of an array or the number of formal parameters of a function)
- a type
- a memory location
- a procedure to be executed
- an error message to be displayed

The value of an attribute at a parse-tree node is defined by the semantic rule associated with the production used at that node.

44

•

**>>** 

PL December 28. 2021

Go Back

Full Screen

CLOSE



## The Structure of a Compiler

- **Divide and conquer.** A large-scale structure and organization of a compiler or translator is defined by the structure of the parser in terms of the individual productions of the context-free grammar that is used in parsing.
- Syntax-directed definitions. The problem of context-sensitive and semantic analysis is split up into the computation of individual attributes and semantic rules in such a way that each production is associated with the (partial) computation of one or more attributes.
- Glue code. Finally it may require some "glue-code" to put together these computations to obtain the final compiler/translator. The glue-code may also be split into some that occurs in the beginning through global declarations/definitions and some which need to be performed in the end.

 ✓
 Full Screen

 Close
 427 of 721

# Syntax-Directed Definitions (SDD)



Syntax-Directed definitions are high-level specifications which specify the evaluation of

- 1. various attributes
- 2. various procedures such as
  - transformations
  - generating code
  - saving information
  - issuing error messages

They hide various implementation details and free the compiler writer from explicitly defining the order in which translation, transformations, and code generation take place.

PL December 28, 2021

44

•

**>>** 

Go Back

Full Screen

Close

428 OF 721



#### Kinds of Attributes

There are two kinds of attributes that one can envisage.

Synthesized attributes A synthesized attribute is one whose value depends upon the values of its immediate children in the concrete parse tree.

A syntax-directed definition that uses only synthesized attributes is called an S-attributed definition. See example

Inherited attributes An inherited attribute is one whose value depends upon the values of the attributes of its parents or siblings in the parse tree.

Inherited attributes are convenient for expressing the dependence of a language construct on the context in which it appears.

 ✓
 Image: Second of the properties of the

### What is Syntax-directed?

- A syntax-directed definition is a generalisation of a context-free grammar in which each grammar symbol has an associated set of attributes, partitioned into two subsets called synthesized and inherited attributes.
- The various attributes are computed by so-called semantic rules associated with each production of the grammar which allows the computation of the various attributes.
- These semantic rules are in general executed during
  - bottom-up (SR) parsing at the stage when a reduction needs to be performed by the given rule and
  - top-down (RDP) parsing in the procedure before the next call or return from the procedure.
- A parse tree showing the various attributes at each node is called an annotated parse tree.

Go Back

Full Screen



#### Forms of SDDs

In a syntax-directed definition, each grammar production rule  $X \to \alpha$  has associated with it a set of semantic rules of the form  $b = f(a_1, \ldots, a_k)$  where  $a_1, \cdots, a_k$  are attributes belonging to X and/or the grammar symbols of  $\alpha$ .

**Definition 9.1** Given a production  $X \to \alpha$ , an attribute a is

**synthesized:** a synthesized attribute of X (denoted X.a) or

**inherited:** an inherited attribute of one of the grammar symbols of  $\alpha$  (denoted B.a if a is an attribute of B).

In each case the attribute a is said to depend upon the attributes  $a_1, \dots, a_k$ .

 →
 Image: Second of the properties of the



#### **Attribute Grammars**

- An attribute grammar is a syntax-directed definition in which the functions in semantic rules can have no side-effects.
- The attribute grammar also specifies how the attributes are propagated through the grammar, by using *graph dependency* between the productions.
- In general different occurrences of the same non-terminal symbol in each production will be distinguished by appropriate subscripts when defining the semantic rules associated with the rule.

The following example illustrates the concept of a syntax-directed definition using synthesized attributes.

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 432 OF 721

Determining the values of arithmetic expressions. Consider a simple attribute val associated with an expression

$$E_0 
ightharpoonup E_1 - T 
ightharpoonup E_0.val := E_1.val - T.val$$
 $E 
ightharpoonup T 
ightharpoonup E.val := T.val$ 
 $T_0 
ightharpoonup T_1/F 
ightharpoonup T_0.val := T_1.val/F.val$ 
 $T 
ightharpoonup F 
ightharpoonup T.val := F.val$ 
 $F 
ightharpoonup (E) 
ightharpoonup F.val := E.val$ 

Note: The attribute n.val is the value of the numeral n computed during

 $F \rightarrow \mathbf{n} \qquad \triangleright F.val := \mathbf{n}.val$ 



### Attributes: Basic Assumptions

- Terminal symbols are assumed to have only synthesized attributes. Their attributes are all supplied by the lexical analyser during scanning.
- The start symbol of the grammar can have *only* synthesized attributes.
- In the case of LR parsing with its special start symbol, the start symbol cannot have any inherited attributes because

PL December 28, 2021

- 1. it does not have any parent nodes in the parse tree and
- 2. it does not occur on the right-hand side of any production.

44

▶▶

Go Back

Full Screen

Close







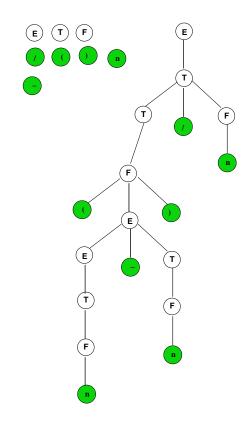
#### 9.1. Synthesized Attributes

### Synthesized Attributes

Evaluating the expression (4-1)/2 generated by the grammar for subtraction and division

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 437 OF 721







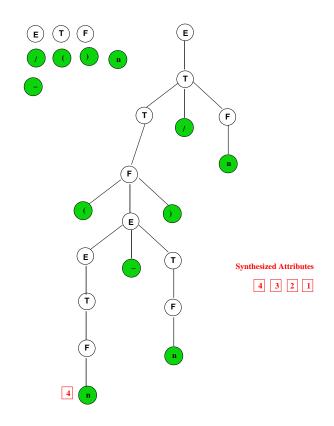














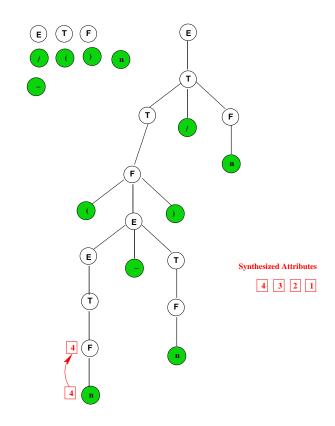




















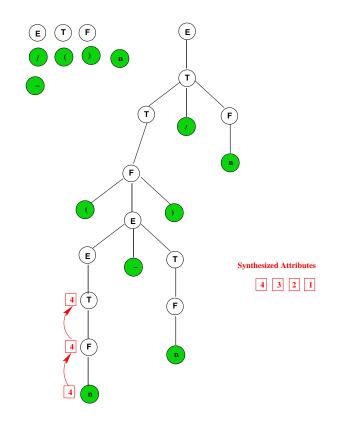






Close











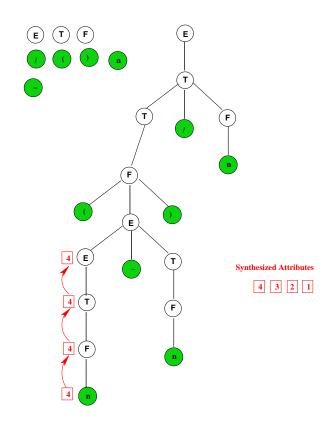














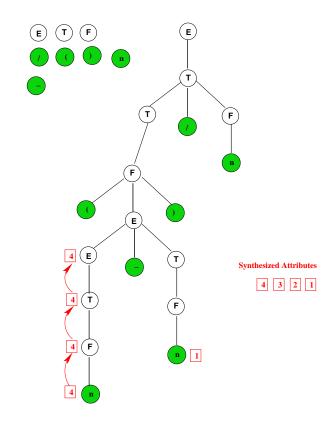


















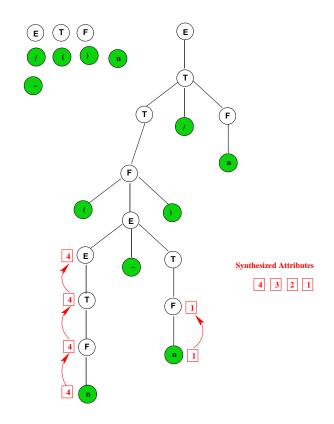




Go Back







PL December 28, 2021







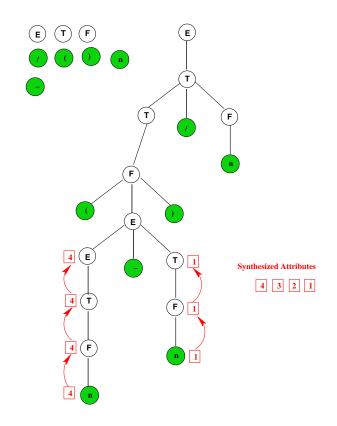




Full Screen

Close





44





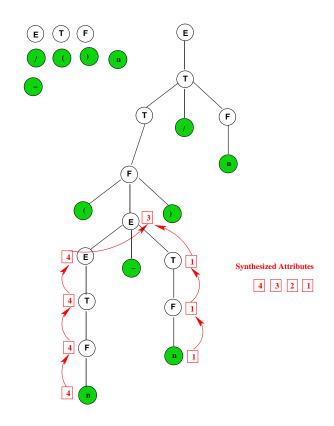






Close













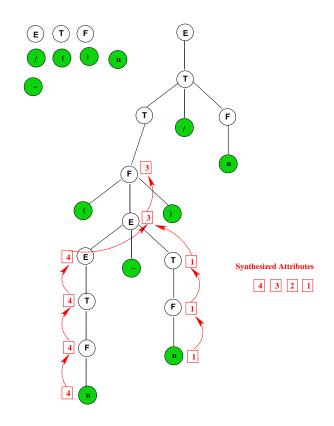




Full Screen

Close













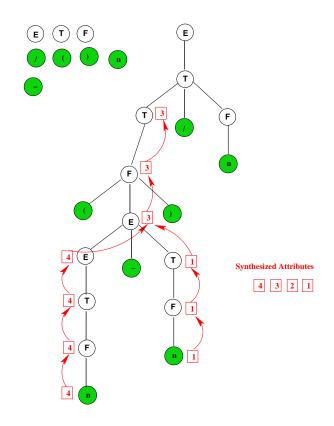


Go Back

Full Screen

Close











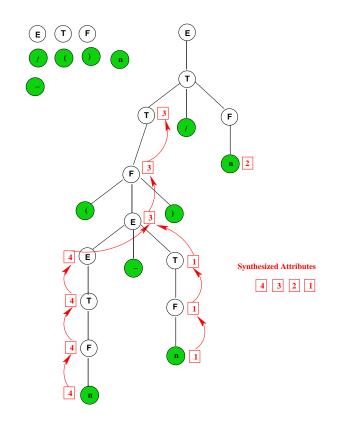












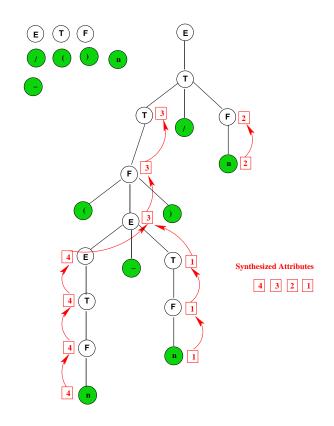
44



















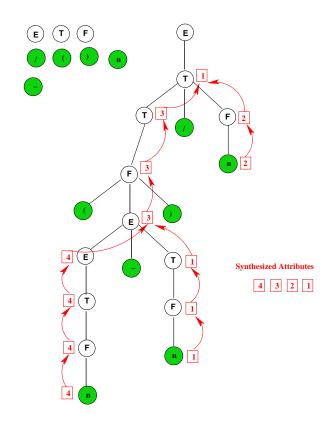






Close











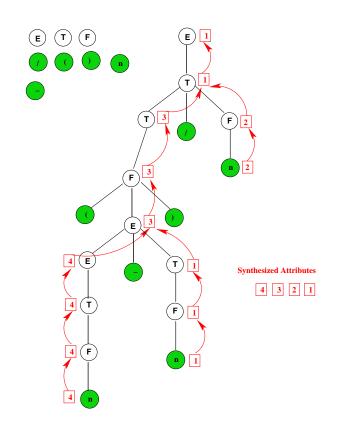
















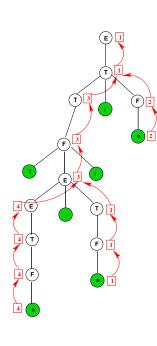








### An Attribute Grammar



$$E_0 \rightarrow E_1 - T \triangleright E_0.val := \underline{sub}(E_1.val, T.val)$$

$$E \rightarrow T \qquad \triangleright E.val := T.val$$

$$T_0 \rightarrow T_1/F \triangleright T_0.val := \underline{div}(T_1.val, F.val)$$

$$T \rightarrow F \qquad \triangleright T.val := F.val$$

$$F \rightarrow (E) \qquad \triangleright F.val := E.val$$

$$F \rightarrow \mathbf{n} \qquad \triangleright F.val := \mathbf{n}.val$$

44

•

**>>** 

PL December 28. 2021

Go Back

Full Screen

Close



454 of 721

### Synthesized Attributes Evaluation: Bottom-up

During bottom-up parsing synthesized attributes are evaluated as follows:

### Bottom-up Parsers

- 1. Keep an attribute value stack along with the parsing stack.
- 2. Just before applying a reduction of the form  $Z \to Y_1 \dots Y_k$  compute the attribute values of Z from the attribute values of  $Y_1, \dots, Y_k$  and place them in the same position on the attribute value stack corresponding to the one where the symbol Z will appear on the parsing stack as a result of the reduction.

→ PL December 28, 2021 Go Back Full Screen Close



### Synthesized Attributes Evaluation: Top-down

During top-down parsing synthesized attributes are evaluated as follows:

**Top-down Parsers** In any production of the form  $Z \to Y_1 \dots Y_k$ , the parser makes recursive calls to procedures corresponding to the symbols  $Y_1 \dots Y_k$ . In each case the attributes of the non-terminal symbols  $Y_1 \dots Y_k$  are computed and returned to the procedure for Z. Compute the synthesized attributes of Z from the attribute values returned from the recursive calls.

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 455 OF 721



→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 456 OF 721



→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 457 OF 721

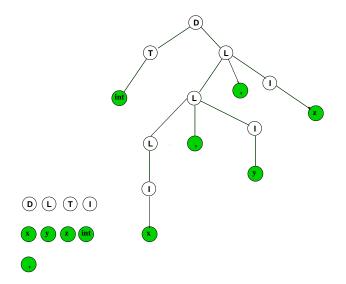


#### 9.2. Inherited Attributes

### Inherited Attributes: 0

C-style declarations generating int x, y, z.

$$D o T L$$
  $T o \mathbf{int} \mid \mathbf{float}$   $L o L, I \mid I$   $I o \mathbf{x} \mid \mathbf{y} \mid \mathbf{z}$ 



PL December 28, 2021

**⋖**∢

**>** 

**>>** 

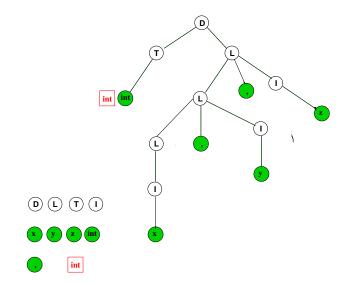
Go Back

Full Screen

Close



C-style declarations generating int x, y, z.



44

**•** 

**>>** 

PL December 28, 2021

Go Back

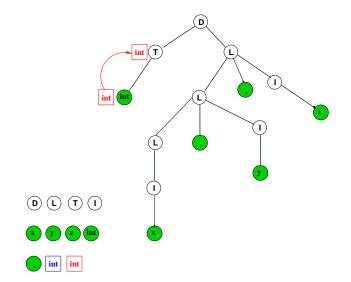
Full Screen

Close

459 of 721



C-style declarations generating int x, y, z.



44

**•** 

**>>** 

PL December 28, 2021

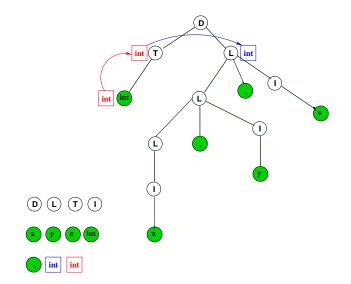
Go Back

Full Screen

Close



C-style declarations generating int x, y, z.



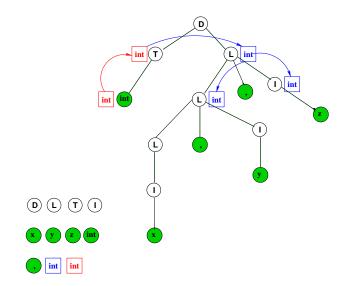
PL December 28, 2021







C-style declarations generating int x, y, z.



44

**•** 

**>>** 

PL December 28, 2021

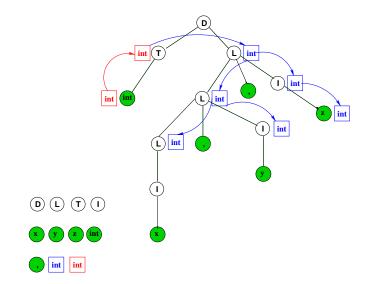
Go Back

FULL SCREEN

Close



C-style declarations generating int x, y, z.



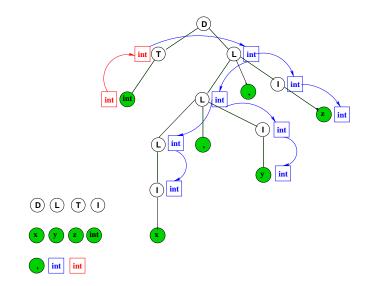
Go Back

Full Screen

Close



C-style declarations generating int x, y, z.



44

•

**>>** 

PL December 28, 2021

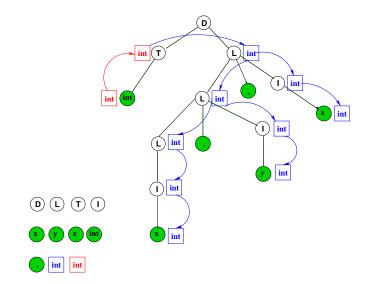
Go Back

Full Screen

Close



C-style declarations generating int x, y, z.



PL December 28, 2021

**⋖**∢

•

▶▶

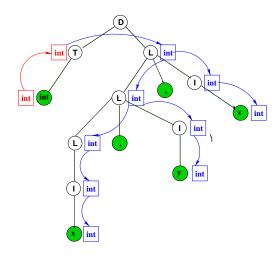
Go Back

Full Screen

Close



### Attribute Grammar: Inherited



$$D \rightarrow TL \quad \triangleright L.in := T.type$$

$$T \rightarrow \mathbf{int} \quad \rhd \ T.type := \mathbf{int}.int$$

$$T \rightarrow \mathbf{float} \triangleright T.type := \mathbf{float}.float$$

$$L_0 \rightarrow L_1, I \triangleright L_1 := L_0.in$$

$$L \rightarrow I \qquad \triangleright I.in := L.in$$

$$I \rightarrow id > id.type := I.in$$

44

,

▶▶

PL December 28, 2021

Go Back

Full Screen

Close



#### L-attributed Definitions

**Definition 9.2** A grammar is L-attributed if for each production of the form  $Y \to X_1 \dots X_k$ , each inherited attribute of the symbol  $X_j$ ,  $1 \le j \le k$  depends only on

- 1. the inherited attributes of the symbol Y and
- 2. the synthesized or inherited attributes of  $X_1, \dots, X_{j-1}$ .

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 467 OF 721



### Why L-attributedness? 1

$$Y \to X_1 \dots X_k$$

Intuitively, if  $X_j.inh$  is an inherited attribute then

• it cannot depend on any synthesized attribute Y.syn of Y because it is possible that the computation of Y.syn requires the value of  $X_j.inh$  leading to circularity in the definition.



#### Why L-attributedness? 2

$$Y \to X_1 \dots X_k$$

Intuitively, if  $X_j.inh$  is an inherited attribute then

• if the value of  $X_j.inh$  depends upon the attributes of one or more of the symbols  $X_{j+1}, \cdots, X_k$  then the computation of  $X_j.inh$  cannot be performed just before the reduction by the rule  $Y \to X_1 \dots X_k$  during parsing. Instead it may have to be postponed till the end of parsing.

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 469 OF 721



#### Why L-attributedness? 3

$$Y \to X_1 \dots X_k$$

Intuitively, if  $X_j.inh$  is an inherited attribute then

• it could depend on the synthesized or inherited attributes of any of the symbols  $X_1 \dots X_{j-1}$  since they would already be available on the attribute value stack.

 ✓
 ✓

 FULL SCREEN
 CLOSE

 470 OF 721



#### Why L-attributedness? 4

$$Y \to X_1 \dots X_k$$

Intuitively, if  $X_j.inh$  is an inherited attribute then

• it could depend upon the inherited attributes of Y because these inherited attributes can be computed from the attributes of the symbols lying below  $X_1$  on the stack, provided these inherited attributes of Y are also L-attributed.

 ■
 ■
 ■
 ■
 PL December 28, 2021
 Go Back
 Full Screen
 Close
 471 of 721

#### A Non L-attributed Definition



Our attribute grammar for C-style declarations is definitely L-attributed. However consider the following grammar for declarations in Pascal and ML.

$$egin{array}{lll} D &
ightarrow L:T &
ightarrow L.in := T.type \ T &
ightarrow {f int} &
ightarrow T.type := {f int.} int \ T &
ightarrow {f real} &
ightarrow T.type := {f real.} real \ L_0 &
ightarrow L_1,I &
ightarrow L_1 := L_0.in \ L &
ightarrow I &
ightarrow I.in := L.in \ I &
ightarrow {f id} &
ightarrow {f id.} type := I.in \ \end{array}$$

In the first semantic rule the symbol L.in is inherited from a symbol to its right viz. T.type and hence is not L-attributed.

44

**•** 

**>>** 

PL December 28. 2021

Go Back

Full Screen

Close



## Evaluating Non-L-attributed Definitions 1

In many languages like ML which allow higher order functions as values, a definition not being L-attributed may not be of serious concern.

But in most other languages it is serious enough to warrant changing the grammar of the language so as to replace inherited attributes by corresponding synthesized ones.

PL December 28, 2021













Close



## **Evaluating Non-L-attributed Definitions 2**

The language of the grammar of Pascal and ML declarations can be generated as follows:

$$D 
ightarrow \mathbf{id}L 
ightharpoonup addtype(\mathbf{id}, L.type) \ L 
ightharpoonup :T.type \ L 
ightharpoonup , \mathbf{id} L 
ightharpoonup L_0.type := L_1.type; \ addtype(\mathbf{id}.L_1.type) \ T 
ightharpoonup \mathbf{int} 
ightharpoonup T.type := \mathbf{int}.int \ T 
ightharpoonup \mathbf{real} 
ightharpoonup T.type := \mathbf{real}.real$$

44

▶▶

Go Back

Full Screen

Close



#### Dependency Graphs

In general, the attributes required to be computed during parsing could be synthesized or inherited and further it is possible that some synthesized attributes of some symbols may depend on the inherited attributes of some other symbols. In such a scenario it is necessary to construct a dependency graph of the attributes of each node of the parse tree.

44



PL December 28, 2021

Go Back

Full Screen

Close



#### Algorithm 9.1

AttributeDependencyGraph  $(T, A) \stackrel{df}{=}$ 

```
Requires: A parse tree T and the list A of attributes Yields: An attribute dependency graph for each node n of T
do \begin{cases} \mathbf{for\ each}\ \text{attribute}\ a \ \text{of node}\ n \end{cases}
do \begin{cases} \mathbf{for\ each}\ \text{attribute}\ n \text{ode}\ n.a \end{cases}
for each node n of T
do \begin{cases} \mathbf{for\ each}\ \text{semantic}\ \text{rule}\ a := f(b_1, \ldots, b_k) \end{cases}
do \begin{cases} \mathbf{for\ } i := 1 \ \mathbf{to}\ k \end{cases}
do \begin{cases} \mathbf{for\ } i := 1 \ \mathbf{to}\ k \end{cases}
do \begin{cases} \mathbf{for\ } a := 1 \ \mathbf{to}\ k \end{cases}
```



→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 477 OF 721

# dse

478 of 721

#### 10. Symbol Table

## Symbol Table

"The name of the song is called 'Haddock's Eyes'."

"Oh, that's the name of the song, is it?" Alice said, trying to feel interested.

"No, you don't understand," the Knight said, looking a little vexed. "That's what the name is called. The name of the song really is, 'The Aged Aged Man'."

Then I ought to have said 'That's what the song is called'?" Alice corrected herself.

"No you oughtn't: that's quite another thing! The song is called 'Ways and Means': but that's only what it's called, you know!"

"Well, what is the song, then?" said Alice, who was by this time completely bewildered.

"I was coming to that", the Knight said. "The song really is 'A-Sitting On a Gate': and the tune's my own invention.

#### Lewis Carroll, Through the Looking-Glass

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE



The Big picture

- The store house of context-sensitive and run-time information about every identifier in the source program.
- All accesses relating to an identifier require to first find the attributes of the identifier from the symbol table
- Usually organized as a hash table provides fast access.
- Compiler-generated temporaries may also be stored in the symbol table

 ■
 ■

 PL December 28, 2021
 Go Back
 Full Screen
 Close
 479 of 721



The Big picture

Attributes stored in a symbol table for each identifier:

- type
- size
- scope/visibility information
- base address
- addresses to location of auxiliary symbol tables (in case of records, procedures, classes)
- address of the location containing the string which actually names the identifier and its length in the string pool

44

▶▶

PL December 28, 2021

Go Back

Full Screen

Close



The Big picture

- A symbol table exists through out the compilation (and run-time for debugging purposes).
- Major operations required of a symbol table:
  - insertion
  - search
  - deletions are purely logical (depending on scope and visibility) and not physical
- Keywords are often stored in the symbol table before the compilation process begins.

PL December 28, 2021

Go Back

Full Screen

Close



The Big picture

Accesses to the symbol table at every stage of the compilation process,

Scanning: Insertion of new identifiers.

Parsing: Access to the symbol table to ensure that an operand exists (declaration before use).

#### Semantic analysis:

- Determination of types of identifiers from declarations
- type checking to ensure that operands are used in type-valid contexts.
- Checking scope, visibility violations.

 →
 Image: Second of the properties of the



The Big picture

IR generation: Memory allocation and relative and address calculation.

Optimization: All memory accesses through symbol table

Target code: Translation of relative addresses to absolute addresses in terms of word length, word boundary etc.

<sup>a</sup>i.e.relative to a base address that is known only at run-time

 ✓
 Image: Second of the properties of the

#### The hash table



Each name is hashed to an index of the hash table whose entry points to a chain of records where each record contains

- a possible link to the next record on the chain (in case of collisions)
- the name of the identifier
- category (e.g. module, procedure, function, block, record, formal parameter etc.)
- scope number of the identifier
- type information
- number of parameters (in case of functions, procedures, modules classes etc.)
- visibility information (derived from qualifiers such as public, private)











# Symbol Table: Scope Stack



Scope rules

- In addition to the hash table a scope stack is maintained for resolving nonlocal references.
- The new scope number is *push*ed onto the scope stack when the compiler enters a new scope and *pop*ped when exiting a scope.
- There could be unnamed scopes too (e.g. unnamed blocks with local declarations, for-loops where the counting variable is local to the loop etc).
- Each (static) scope may be assigned a number in sequential order as it is encountered in the program text starting with 0 assigned for the global scope.
- The scope number of a nested scope is always greater than that of its parent.

Full Screen

CLOSE

485 OF 721





#### 11. Intermediate Representation

Intermediate Representation









. . .

Go Васк

Full Screen

Close

## Intermediate Representation



Intermediate representations are important for reasons of portability i.e. platform (hardware and OS) independence.

- (more or less) independent of specific features of the high-level language. Example. Java byte-code which is the instruction set of the Java Virtual Machine (JVM).
- (more or less) independent of specific features of any particular target architecture (e.g. number of registers, memory size)
  - number of registers
  - memory size
  - word length

Typical Instruction set

44

**>** 

PL December 28, 2021

Go Васк

FULL SCREEN

Close

# IR Properties: Low vs high



1. It is fairly low-level containing instructions common to all target architectures and assembly languages.

How low can you stoop? ...

2. It contains some fairly high-level instructions that are common to most high-level programming languages.

How high can you rise?

- 3. To ensure portability across architectures and OSs. Portability
- 4. To ensure type-safety

  Type safety

Typical Instruction set

44

**>>** 

Go Back

Full Screen

Close



#### IR: Representation?

- No commitment to word boundaries or byte boundaries
- No commitment to representation of
  - —int vs. float,
  - float vs. double,
  - packed vs. unpacked,
  - strings where and how?.

Back to IR Properties

 ✓
 Image: A second of the property of



#### IR: How low can you stoop?

- most arithmetic and logical operations, load and store instructions etc.
- so as to be interpreted easily,
- the interpreter is fairly small,
- execution speeds are high,
- to have fixed length instructions (where each operand position has a specific meaning).

Back to IR Properties

44

•

PL December 28, 2021

Go Back

FULL SCREEN

Close



## IR: How high can you rise?

- typed variables,
- temporary variables instead of registers.
- array-indexing,
- random access to record fields,
- parameter-passing,
- pointers and pointer management
- no limits on memory addresses

Back to IR Properties

**◄**◀

**▶** 

PI Doco

Go Back

Full Screen

Close



#### IR Properties: Portability

- 1. How low can you stoop? . . .
- 2. How high can you rise?
- 3. To ensure portability across architectures and OSs.
  - an unbounded number of variables and memory locations
  - no commitment to Representational Issues
- 4. Type safety

Back to IR Properties

**⋖**◆

**▶** 

PL December 28, 2021

Go Back

FULL SCREEN

Close



#### IR Properties: Type Safety

- 1. How low can you stoop? ...
- 2. How high can you rise?
- 3. Portability
- 4. To ensure type-safety despite the hardware instruction set architectures.
  - Memory locations are also typed according to the data they may contain,
  - No commitment is made regarding word boundaries, and the structure of individual data items.

Back to IR Properties

Typical Instruction set

→ → PL December 28, 2021 Go BACK FULL SCREEN CLOSE 494 OF 721



Three address code: A suite of instructions. Each instruction has at most 3 operands.

- an opcode representing an operation with at most 2 operands
- two operands on which the binary operation is performed
- a target operand, which accumulates the result of the (binary) operation.

If an operation requires less than 3 operands then one or more of the operands is made null.

**◄**◀





Go Back

Full Screen

Close



- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
- Arrays and array-indexing
- Pointer Referencing and Dereferencing

c.f. Java byte-code

44

**▶** 

PL December 28, 2021

Go Back

Full Screen

Close



- Assignments (LOAD-STORE)
  - -x := y bop z, where bop is a binary operation
  - -x := uop y, where uop is a unary operation
  - -x := y, load, store, copy or register transfer
- Jumps (conditional and unconditional)
- Procedures and parameters
- Arrays and array-indexing
- Pointer Referencing and Dereferencing

**∢**∢

**◄** 





- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
  - —goto L Unconditional jump,
  - x relop y goto L Conditional jump, where relop is a relational operator
- Procedures and parameters
- Arrays and array-indexing
- Pointer Referencing and Dereferencing

44



Go Back



PL December 28, 2021

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
  - $-\operatorname{call} p$  n, where n is the number of parameters
  - -return y, return value from a procedures call
  - param x, parameter declaration
- Arrays and array-indexing
- Pointer Referencing and Dereferencing

44

**▶** 

▶▶

Go Back

Full Screen

Close



- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
- Arrays and array-indexing
  - -x := a[i] array indexing for r-value
  - -a[j] := y array indexing for l-value

Note: The two opcodes are different depending on whether l-value or r-value is desired.  $\mathbf{x}$  and  $\mathbf{y}$  are always simple variables

Pointer Referencing and Dereferencing

44

**▶** 

**▶**▶

Go Back

Full Screen

Close



- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
- Arrays and array-indexing
- Pointer Referencing and Dereferencing

```
-x := y - referencing: set x to point to y
```

- -x := \*y dereferencing: copy contents of location pointed to by y into x
- -\*x := y dereferencing: copy <math>r-value of y into the location pointed to by x

PL December 28, 2021

#### **Picture**

**◄**◀

▶▶

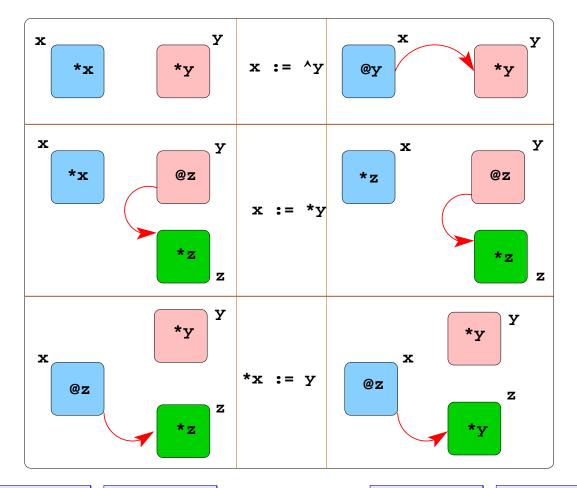
Go Back

Full Screen

Close

# **Pointers**





# d\$E

#### IR: Generation Basics

- Can be generated by recursive traversal of the abstract syntax tree.
- ullet Can be generated by syntax-directed translation as follows: For every non-terminal symbol N in the grammar of the source language there exist two attributes
  - N.place, which denotes the address of a temporary variable where the result of the execution of the generated code is stored
    N.code, which is the actual code segment generated.
- In addition a global counter for the instructions generated is maintained as part of the generation process.
- It is independent of the source language but can express target machine operations without committing to too much detail.

44

**◄** 





PL December 28, 2021

Go Back

Full Screen

Close



#### IR: Infrastructure 1

Given an abstract syntax tree T, with T also denoting its root node.

**T.place** address of temporary variable where result of execution of the T is stored.

newtemp returns a fresh variable name and also installs it in the symbol table along with relevant information

**T.code** the actual sequence of instructions generated for the tree T.

newlabel returns a label to mark an instruction in the generated code which may be the target of a jump.

*emit* emits an instructions (regarded as a string).





Full Screen

CLOSE



#### IR: Infrastructure 2

Colour and font coding of IR code generation process.

- *Green*: Nodes of the Abstract Syntax Tree
- Brown: Intermediate Representation i.e. the language of the "virtual machine"
- *Red*: Variables and data structures of the *language* in which the IR code generator is written
- Blue: Names of relevant procedures used in IR code generation.
- Black: All other stuff.

44





Go Back

Full Screen

Close

505 of 721



## **IR**: Expressions

$$E o id$$
 $E.place := id.place;$ 
 $E.code := emit()$ 
 $E_0 o E_1 - E_2$ 
 $E_0.place := newtemp;$ 
 $E_0.code := E_1.code;$ 
 $E_2.code;$ 
 $emit(E_0.place := E_1.place - E_2.place)$ 

44



# The WHILE Language

Assume there is a language of expressions (with start symbol E) over which the statements are defined. For simplicity assume these are the only constructs of the language.

$S \to id := E$	Assignment
$\mid S; S$	Sequencing
if E then S else Sfi	Conditional
$\bot$ while $E$ do $S$ od	Iteration

PL December 28, 2021

**◄**◀



# IR: Assignment and Sequencing



$$S \longrightarrow id := E$$

$$S.code := E.code \\ emit(id.place := E.place)$$

$$S_0 \longrightarrow S_1; S_2 \triangleright$$

 $emit(S_0.after:)$ 

$$S_0.begin := S_1.begin;$$
 $S_0.after := S_2.after;$ 
 $S_0.code := emit(S_0.begin:)$ 
 $S_1.code$ 
 $S_2.code$ 





## IR: Conditional

```
S_0
          \rightarrow if \ E \ then \ S_1 \ else \ S_2fi
S_0.begin := newlabel;
S_0.after := S_2.after;
S_0.code := emit(S_0.begin:)
              E.code;
              emit(if E.place = 0 goto S_2.begin);
              S_1.code;
              emit(\mathtt{goto}\ S_0.after);
              S_2.code;
              emit(S_0.after:)
```

44

**•** 

PL December 28, 2021

Go Back

FULL SCREEN

Close



#### Selective Evaluation.

Notice that the evaluation/execution of the Conditional is such that only one arm of the conditional is evaluated/executed depending upon the truth value of the condition. This is perfectly consistent with the semantics of the conditional. It is also consistent with the functional semantics of the conditional construct in FL(X). Similar remarks also apply to iteration construct

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 510 OF 721



#### **IR**: Iteration

$$S_0 \longrightarrow while \ E \ do \ S_1 \ od$$
 $S_0.begin := newlabel;$ 
 $S_0.after := newlabel;$ 
 $S_0.code := emit(S_0.begin:)$ 
 $E.code$ 
 $emit(if \ E.place = 0 \ goto \ S_0.after);$ 
 $S_1.code;$ 

defined below.

44

**•** 

▶▶

Go Back

 $emit(\mathtt{goto}\ S_0.begin);$ 

 $emit(S_0.after:)$ 

PL December 28, 2021

Full Screen

Close



#### IR: Generation End

While generating the intermediate representation, it is sometimes necessary to generate jumps into code that has not been generated as yet (hence the address of the label is unknown). This usually happens while processing

- forward jumps
- short-circuit evaluation of boolean expressions

It is usual in such circumstances to either fill up the empty label entries in a second pass over the the code or through a process of *backpatching* (which is the maintenance of lists of jumps to the same instruction number), wherein the blank entries are filled in once the sequence number of the target instruction becomes known.

44

•

**>>** 

Go Back

Full Screen

Close



#### 12. The Pure Untyped Lambda Calculus: Basics

#### 12.1. Motivation for $\lambda$

As Curry points out in his classic work on Combinatory Logic,

Curiously a systematic notation for functions is lacking in ordinary mathematics. The usual notation f(x) does not distinguish between the function itself and the value of this function for an undetermined value of the argument.

Let us consider the nature of functions, higher-order functions (functionals) and the use of naming in mathematics, through some examples.

**Example 12.1** Let  $y = x^2$  be the squaring function on the reals. Here it is commonly understood that x is the "independent" variable and y is the "dependent" variable when we look on it as plotting the function  $f(x) = x^2$  on the x - y axis.

**∢**∢

Go Back

Full Screen

Close

513 OF 721

**Example 12.2** Often a function may be named and written as  $f(x) = x^n$  to indicate that x is the independent variable and n is understood (somehow!) to be some constant. Here f, x and n are all names with different connotations. Similarly in the quadratic polynomial  $ax^2 + bx + c$  it is somehow understood that a, b and c denote constants and that x is the independent variable. Implicitly by using the names like a, b and c we are endeavouring to convey the impression that we consider the class  $\{ax^2 + bx + c \mid a, b, c \in \mathbb{R}\}$  of all quadratic polynomials of the given form.

**Example 12.3** As another example, consider the uni-variate polynomial  $p(x) = x^2 + 2x + 3$ . Is this polynomial the same as  $p(y) = y^2 + 2y + 3$ ? Clearly they cannot be the same since the product p(x).p(y) is a polynomial in two variables whereas p(x).p(x) yields a uni-variate polynomial of degree 4. However, in the case of the function f in example 12.1 it does not matter whether we define the squaring function as  $f(x) = x^2$  or as  $f(y) = y^2$ .

**Example 12.4** The squaring function 12.1 is a continuous and differentiable real-valued function (in the variable x) and its derivative is f'(x) = 2x. Whether we regard f' as the name of a new function or we regard the ' as an operation on f which yields its derivative seems to make no difference.

**Example 12.5** Referring again to the functions f(x) and f'(x) in example 12.4, it is commonly un-

44

PL December 28, 2021

Go Back

Full Screen

Close

derstood that f'(0) refers to the value of the derivative of f at 0 which is also the value the function f'(x) takes at 0. Now let us consider f'(x+1). Going by the commonly understood notion, since f'(x) = 2x, we would have f'(x+1) = 2(x+1). Then for x = 0 we have  $f'(x+1) = f'(0+1) = f'(1) = 2 \times 1 = 2$ . We could also think of it as the function f'(g(0)) where g is the function defined by g(x) = x + 1, then f'(g(0)) = 2g(0) = 2 which yields the same result.

The examples above give us some idea of why there is no systematic notation for functions which distinguishes between a function definition and the application of the same function to some argument. It simply did not matter!

However, this ambiguity in mathematical notation could lead to differing interpretationas and results in the context of mathematical theories involving higher-order functions (or "functionals" as they are often referred to). One common higher order function is the derivative (the differentiation operation) and another is the indefinite integral. Most mathematical texts emphasize the higher-order nature of a function by enclosing their arguments in (square) brackets. Hence if O is a functional which transforms a function f(x) into a function g(x), this fact is usually written O[f(x)] = g(x).

**Example 12.6** Consider the functional E (on continuous real-valued functions of one real variable x)

44

PL December 28, 2021

Go Васк

Full Screen

Close



defined as follows.

$$E[f(x)] = \begin{cases} f'(0) & \text{if } x = 0\\ \frac{f(x) - f(0)}{x} & \text{if } x \neq 0 \end{cases}$$

The main question we ask now is "What does E[f(x+1)] mean?"

It turns out that there are at least two ways of interpreting E[f(x+1)] and unlike the case of example 12.5, the two interpretations actually yield different results!.

1. We may interpret E[f(x+1)] to mean that we first apply the transformation E to the function f(x) and then substitute x+1 for x in the resulting expression. We then have the following.

$$E[f(x)]$$

$$=\begin{cases} f'(0) & \text{if } x = 0\\ \frac{f(x) - f(0)}{x} & \text{if } x \neq 0 \end{cases}$$

$$=\begin{cases} 0 & \text{if } x = 0\\ x & \text{if } x \neq 0 \end{cases}$$

$$= x$$

**⋖**◀

PL December 28, 2021

Go Back

Full Screen

Close

516 OF 721



Since 
$$E[f(x)] = x$$
,  $E[f(x+1)] = x+1$ .

2. Since f(x+1) = f(g(x)) where g(x) = x+1, we may interpret E[f(x+1)] as applying the operator E to the function h(x) = f(g(x)). Hence E[f(x+1)] = E[h(x)] where  $h(x) = f(g(x)) = (x+1)^2 = x^2 + 2x + 1$ . Noting that h'(x) = 2x + 2, h(0) = 1 and h'(0) = 2, we get

$$E[h(x)]$$

$$=\begin{cases} h'(0) & \text{if } x = 0\\ \frac{h(x) - h(0)}{x} & \text{if } x \neq 0 \end{cases}$$

$$=\begin{cases} 2 & \text{if } x = 0\\ x + 2 & \text{if } x \neq 0 \end{cases}$$

$$= x + 2$$

The last example should clearly convince the reader that there is a need to disambiguate between a function definition and its application.



#### 12.2. The $\lambda$ -notation

In function definitions the independent variables are "bound" by a  $\lambda$  which acts as a pre-declaration of the name that is going to be used in the expression that defines a function.

The notation f(x), which is interpreted to refer to "the value of function f at x", will be replaced by (f(x)) to denote an application of a function f to the (known or unknown) value x.

In our notation of the untyped applied  $\lambda$ -calculus the functions and their applications in the examples in subsection 12.1 would be rewritten as follows.

**Squaring** .  $\lambda x[x^2]$  is the squaring function.

**Example 12.2** .  $q \stackrel{df}{=} \lambda \ a \ b \ c \ x[ax^2 + bx + c]$  refers to any quadratic polynomial with coefficients unknown or symbolic. To obtain a particular member of this family such as  $1x^2 + 2x + 3$ , one would have to evaluate  $((q\ 1)\ 2)\ 3)$  which would yield  $\lambda \ x[1x^2 + 2x + 3]$ .

**Example 12.3**.  $p \stackrel{df}{=} \lambda \ x[x^2 + 2x + 3]$ . Then p(x) would be written as  $(p \ x)$  i.e. as the function p applied to the argument x to yield the expression  $x^2 + 2x + 3$ . Likewise p(y) would be  $(p \ y)$  which would yield  $y^2 + 2y + 3$ . The products  $(p \ x).(p \ x)$  and  $(p \ x).(p \ y)$  are indeed different and distinct.

PL December 28, 2021

**Example 12.5** Let us denote the operation of obtaining the derivative of a real-valued function f of one independent variable x by the simple symbol D (instead of the more confusing  $\frac{d}{dx}$ ). Then for any function f, (D f) would yield the derivative. In particular  $(D \lambda x[x^2]) = \lambda x[2x]$  and the value of the derivative at 0 would be obtained by the application  $(\lambda x[2x] 0)$  which would yield 0. Likewise the value of the derivative at x + 1 would be expressed as the application  $(\lambda x[2x] (x + 1))$ . Thus for any function f the value of its derivative at x + 1 is simply the application (D f) (x + 1).

The function g(x) = x + 1 would be defined as  $g \stackrel{df}{=} \lambda \ x[x+1]$  and  $(g \ x) = x + 1$ . Thus the alternative definition of the derivative of f at x + 1 is simply the application  $((D \ f) \ (g \ x))$ .

PL December 28, 2021

**Example 12.6** The two interpretations of the expression E[f(x+1)] are respectively the following.

- 1. ((E f) (x + 1)) and
- 2.  $((E \ h) \ x)$  where  $h \stackrel{df}{=} \lambda \ x[(f \ (g \ x))]$



# Pure Untyped $\lambda$ -Calculus: Syntax

The language  $\Lambda$  of pure untyped  $\lambda$ -terms is the smallest set of terms built up from an infinite set V of variables and closed under the following productions

$$L,M,N:=x$$
 Variable  $\lambda x[L]$  Abstraction  $(L \ M)$  Application

where  $x \in V$ .

- A *Variable* denotes a possible binding in the external environment.
- An Abstraction denotes a function which takes a formal parameter.
- $\bullet$  An Application denotes the application of a function to an actual parameter.



# The language $\Lambda$

- The language  $\Lambda$  is "pure" (as opposed to being "applied") in the sense that it is minimal and symbolic and does not involve any operators other *abstraction* and *application*.
- When used in the context of some algebraic system (e.g. the algebra of integers or reals) it becomes applied. Hence the example of using the  $\lambda$ -notation in the differential calculus is one of an applied  $\lambda$ -calculus.
- It is purely symbolic and no types have been specified which put restrictions on the use of variables in contexts. We will look at typing much later.

 →
 Image: Second of the properties of the



## Free Variables

**Definition 12.7** For any term  $N \in \Lambda$  the set of free variables and the set of all variables are defined by induction on the structure of terms.

N	FV(N)	Var(N)
x	$\{x\}$	$\{x\}$
$\lambda x[L]$	$FV(L) - \{x\}$	$Var(L) \cup \{x\}$
(L M)	$FV(L) \cup FV(M)$	$ig Var(L) \cup Var(M)ig $

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 522 OF 721



## **Bound Variables**

- The set of bound variables BV(N) = Var(N) FV(N).
- The same variable name may be used with different bindings in a single term (e.g.  $(\lambda x[x] \ \lambda x[(x \ y)])$ )
- ullet The brackets "[" and "]" delimit the scope of the bound variable x in the term  $\lambda x[L]$ .
- $\bullet$  The usual rules of static scope apply to  $\lambda$ -terms.

44

▶▶

Go Back

Full Screen

Close

523 OF 721



## Closed Terms and Combinators

## **Definition 12.8**

- $\Lambda_0 \subseteq \Lambda$  is the set of closed  $\lambda$ -terms (i.e. terms with no free variables).
- A  $\lambda$  abstraction with no free variables is called a combinator<sup>a</sup>.

The  $\lambda$ -terms D (see equation (??)) and E (see equation (??)) are combinators.

44 4

**▶**▶

Go Back

Full Screen

Close

<sup>&</sup>lt;sup>a</sup>Combinators represent function definitions



#### **Notational Conventions**

To minimize use of brackets and parentheses unambiguously

- 1.  $\lambda x_1 x_2 \dots x_m[L]$  denotes  $\lambda x_1[\lambda x_2[\dots \lambda x_m[L] \dots]]$  i.e. L is the scope of each of the variables  $x_1, x_2, \dots x_m$ .
- 2.  $(L_1 \ L_2 \ \cdots L_m)$  denotes  $(\cdots (L_1 \ L_2) \ \cdots L_m)$  i.e. application is left-associative.

44

**|** |

▶▶

Go Back

Full Screen

Close

525 OF 721

## Substitution



**Definition 12.9** For any terms L, M and N and any variable x, the substitution of the term N for a variable x is defined as follows:

$$\begin{cases} N/x \} x & \equiv N \\ \{N/x \} y & \equiv y & \text{if } y \not\equiv x \\ \{N/x \} \lambda x[L] & \equiv \lambda x[L] \\ \{N/x \} \lambda y[L] & \equiv \lambda y[\{N/x \} L] & \text{if } y \not\equiv x \text{ and } y \not\in FV(N) \\ \{N/x \} \lambda y[L] & \equiv \lambda z[\{N/x \} \{z/y \} L] & \text{if } y \not\equiv x \text{ and } y \in FV(N) \text{ and } x \text{ is 'fresh'} \\ \{N/x \} (L \ M) & \equiv (\{N/x \} L \ \{N/x \} M) \\ \end{cases}$$

**Lemma 12.10** If L and N are pure  $\lambda$ -terms and x is a variable symbol then  $\{N/x\}L$  is a pure  $\lambda$ -term.

*Proof:* By induction on the structure of the  $\lambda$ -term L.

**QED** 

44

**•** 

**>>** 

PL December 28, 2021

Go Back

Full Screen

Close

526 OF 721



## Notes on Substitution

- ullet In the above definition it is necessary to ensure that the free variables of N continue to remain free after substitution i.e. none of the free variables of N should be "captured" as a result of the substitution.
- The phrase "z is 'fresh' " may be taken to mean  $z \notin FV(N) \cup Var(L)$ .
- $\bullet$   $\Lambda$  is closed under the syntactic operation of substitution.
- ullet Substitution is the only operation required for function application in the pure  $\lambda$ -calculus.

PL December 28, 2021

Go Back

Full Screen

Close



# Compatibility

**Definition 12.11** A binary relation  $\rho \subseteq \Lambda \times \Lambda$  is said to be compatible if  $L \rho M$  implies

- 1. for all variables x,  $\lambda x[L] \rho \lambda x[M]$  and
- 2. for all terms N,  $(L \ N) \ \rho \ (M \ N)$  and  $(N \ L) \ \rho \ (N \ M)$ .

44

**▶**▶

PL December 28, 2021 Go I

Go Back

Full Screen

Close



## Compatible Closure

**Definition 12.12** The compatible closure of a relation  $\rho \subseteq \Lambda \times \Lambda$  is the smallest (under the  $\subseteq$  ordering) relation  $\rho^c \subseteq \Lambda \times \Lambda$  such that

$$\rho \frac{L \rho M}{L \rho^c M}$$

$$\rho \mathbf{Abs} \ \frac{L \ \rho^c \ M}{\lambda x[L] \ \rho^c \ \lambda x[M]}$$

$$ho \mathbf{AppL} \left| \begin{array}{cc} L \rho^c M \\ \hline (L N) \rho^c (M N) \end{array} \right|$$

$$\rho \mathbf{AppR} \quad \frac{L \ \rho^c \ M}{(N \ L) \ \rho^c \ (N \ M)}$$

**⋖**◆

**>** 

▶▶

Go Back

Full Screen

Close



530 of 721

## Compatible Closure: Properties

## Lemma 12.13

- 1.  $\rho^c \supseteq \rho$ .
- 2. The compatible closure of any relation is compatible.
- 3. If  $\rho$  is compatible then  $\rho^c = \rho$ .

# **Example 12.14**

- $1. \equiv_{\alpha}$  is a compatible relation
- 2.  $\rightarrow_{\beta}^{1}$  is by definition a compatible relation.

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE

## $\alpha$ -equivalence



**Definition 12.15** ( $\alpha$ -equivalence)  $\equiv_{\alpha} \subseteq \Lambda \times \Lambda$  is the compatible closure of the relation  $\{(\lambda x[L] \equiv_{\alpha} \lambda y[\{y/x\}L]) \mid y \notin FV(L)\}.$ 

- $\alpha$ -equivalence implies that the the name(s) of the bound (called "independent" in normal mathematics) variable(s) in a function definition is unimportant<sup>a</sup>. Hence  $\lambda x[x^2] \equiv_{\alpha} \lambda y[y^2]^b$ .
- As long as distinct bound variable names do not clash within the same or nested scopes (where they need to be kept visible) one can substitute the other.
- Condition  $y \notin FV(L)$  is necessary to ensure that a "free" y is not captured by the new "bound" variable y.







Go Back

Full Screen

Close

<sup>&</sup>lt;sup>a</sup>it corresponds exactly to uniformly replacing a variable name in a local context in a program by another variable name throughout the block provided there is no clash of variable names.

<sup>&</sup>lt;sup>b</sup>See also ??

<sup>&</sup>lt;sup>c</sup>Whenever they need to be 'collapsed' e.g. when we need the value of f(x,x) as an instance of a function f(x,y), we need to explicitly apply f to the argument pair (x, x).



## Function application: Basic $\beta$ -Reduction

## **Definition 12.16**

- Any (sub-)term of the form  $(\lambda x[L] \ M)$  is called a  $\beta$ -redex
- Basic  $\beta$ -reduction is the relation on  $\Lambda$

$$\rightarrow_{\beta} \stackrel{df}{=} \{ ((\lambda x[L] \ M), \{M/x\}L') \mid L' \equiv_{\alpha} L, L', L, M \in \Lambda \}$$

• It is usually represented by the axiom

$$(\lambda x[L] \ M) \to_{\beta} \{M/x\}L'$$
 (6)

where  $L' \equiv_{\alpha} L$ .

PL December 28, 2021

Go Back

Full Screen

Close

532 OF 721



# Function application: 1-step $\beta$ -Reduction

**Definition 12.17** A 1-step  $\beta$ -reduction  $\rightarrow^1_{\beta}$  is the smallest relation (under the  $\subseteq$  ordering) on  $\Lambda$  such that

$$\beta_1 \frac{L \to_{\beta} M}{L \to_{\beta}^1 M}$$

$$\beta_{\mathbf{1}}\mathbf{Abs} \ \frac{L \to_{\beta}^{1} M}{\lambda x[L] \to_{\beta}^{1} \lambda x[M]}$$

$$\beta_{1}\mathbf{AppL} \frac{L \to_{\beta}^{1} M}{(L \ N) \to_{\beta}^{1} (M \ N)}$$

$$\beta_{1}AppR \frac{L \to_{\beta}^{1} M}{(N L) \to_{\beta}^{1} (N M)}$$

- $\bullet \to_{\beta}^1$  is the compatible closure of basic  $\beta$ -reduction to all contexts.
- $\bullet$  We will often omit the superscript  $^1$  as understood.

44

4

**•** 

**>>** 

Go Back

Full Screen

Close



## Untyped $\lambda$ -Calculus: $\beta$ -Reduction

## **Definition 12.18**

• For all integers  $n \geq 0$ , n-step  $\beta$ -reduction  $\rightarrow_{\beta}^{n}$  is defined by induction on 1-step  $\beta$ -reduction

$$\beta_{\mathbf{n}}\mathbf{Basis} \ \ \overline{L \to_{\beta}^{0} L}$$

$$\beta_{\mathbf{n}}\mathbf{Basis} \ \frac{1}{L \to_{\beta}^{0} L} \qquad \beta_{\mathbf{n}}\mathbf{Induction} \ \frac{L \to_{\beta}^{m} M \to_{\beta}^{1} N}{L \to_{\beta}^{m+1} N} \ (m \ge 0)$$

•  $\beta$ -reduction  $\rightarrow_{\beta}^*$  is the reflexive-transitive closure of 1-step  $\beta$ -reduction. That is,

$$\beta_* \frac{L \to_{\beta}^n M}{L \to_{\beta}^* M} \quad (n \ge 0)$$

Go Back Full Screen Close 534 of 721 PL December 28, 2021



## Computations and Normal Forms

- Loosely speaking, by a normal form we mean a term that cannot be "simplified" further. In some sense it is like a "final answer".
- We use  $\beta$ -reduction as the only way to "compute" final answers by simplification.
- There may be more than one  $\beta$ -redex in a term this may lead to different ways of computing the final answer.

Main Question: Do all the different ways of computing yield the same answer?

 ✓
 Image: Second control of the propertion of the properties of the propertie

### **Function Calls**



Let

$$f = \lambda x[x^2 + 1]$$
$$q = \lambda y[3.y + 2]$$

Consider two different evaluations of the function call (f (g 4))

$$\begin{array}{c|ccccc} Call-by-value & Call-by-name/text \\ (f (g 4)) & (f (g 4)) \\ = (f (3.4 + 2)) & = (g 4)^2 + 1 \\ = (f 14) & = (3.4 + 2)^2 + 1 \\ = (12 + 2)^2 + 1 \\ = 14^2 + 1 & = 14^2 + 1 \\ = 196 + 1 & = 196 + 1 \\ = 197 & = 197 \end{array}$$

44





PL December 28, 2021

Go Back

Full Screen

Close

536 OF 721



# **Function Composition**

- Let  $F\equiv x^2+1$  be an expression involving one independent variable x and let  $f\stackrel{df}{=}\lambda x[F]$
- Let  $G \equiv 3.y + 2$  be an expression involving one independent variable y and let  $g \stackrel{df}{=} \lambda y[G]$ .
- Let  $h = \lambda f[\lambda g[\lambda z[(f(gz))]]$ . Then h is the composition of f and g i.e.  $h = (f \circ g)$ .

The function call (f(ga)) for some value a is (ha) which is exactly ((fog)a). Hence There are at least two different ways of evaluating the composition of functions.

 →
 PL December 28, 2021

 Go Back
 Full Screen

 Close
 537 of 721



# **Evaluating Function Composition**

# Call-by-value.

- 1. First evaluate  $(g \ a) = \{a/y\}G$  yielding a value b.
- 2. Then evaluate  $(f \ b) = \{b/x\}F$  yielding a value c.

## Call-by-text.

- 1. First evaluate  $(f(gy)) = \{(gy)/x\}F = \{G/x\}F$  yielding expression H which contains only y as independent variable. This expression represents a function  $h \stackrel{df}{=} \lambda y[H]$ .
- 2. Evaluate  $(h \ a) = \{a/y\}H$  yielding a value d.

Main Question: Is c = d always?

44

**>>** 

PL December 28. 2021

Go Back

Full Screen

Close



# Untyped $\lambda$ -Calculus: Normalization

## **Definition 12.19**

- A term is called a  $\beta$ -normal form ( $\beta$ -nf) if it has no  $\beta$ -redexes.
- A term is weakly normalising ( $\beta$ -WN) if it can reduce to a  $\beta$ -normal form.
- A term L is strongly normalising ( $\beta$ -SN) if it has no infinite reduction sequence  $L \to_{\beta}^{1} L_{1} \to_{\beta}^{1} L_{2} \to_{\beta}^{1} \cdots$

Intuitively speaking a  $\beta$ -normal form is one that cannot be "reduced" further.

 ✓
 ✓

 PL December 28, 2021
 Go Back
 Full Screen
 Close
 539 of 721



## Some Combinators

# **Example 12.20**

- 1.  $K \stackrel{df}{=} \lambda x \ y[x]$  a projection function.
- 2.  $I \stackrel{df}{=} \lambda x[x]$ , the identity function.
- 3.  $S \stackrel{df}{=} \lambda x \ y \ z[((x \ z) \ (y \ z))]$ , a generalized composition function

4. 
$$\omega \stackrel{df}{=} \lambda x[(x \ x)]$$

are all  $\beta$ -nfs.

 ✓
 ✓

 PL December 28, 2021
 Go Back
 Full Screen
 Close
 540 of 721



# **Examples of Strong Normalization**

# **Example 12.21**

1.  $((K \ \omega) \ \omega)$  is  $strongly\ normalising\ because\ it\ reduces\ to\ the\ normal\ form\ \omega$  in two  $\beta$ -reduction steps.

$$((\mathbf{K} \ \omega) \ \omega) \to_{\beta}^{1} (\lambda y[\omega] \ \omega) \to_{\beta}^{1} \omega$$

2. Consider the term ((S K) K). Its reduction sequences go as follows:

$$((\mathbf{S} \ \mathbf{K}) \ \mathbf{K}) \rightarrow^1_{\beta} \lambda z [((\mathbf{K} \ z) \ (\mathbf{K} \ z))] \rightarrow^1_{\beta} \lambda z [z] \equiv \mathbf{I}$$

44

**>>** 

PL December 28, 2021

Go Back

Full Screen

Close

## **Unnormalized Terms**



# **Example 12.22**

- 1.  $\Omega \stackrel{df}{=} (\omega \ \omega)$  has no  $\beta$ -nf. Hence it is neither weakly nor strongly normalising.
- 2.  $(K (\omega \omega))$  cannot reduce to any normal form because it has no finite reduction sequences. All its reductions are of the form

$$(\mathsf{K}\ (\omega\ \omega)) \to_{\beta}^{1} (\mathsf{K}\ (\omega\ \omega)) \to_{\beta}^{1} (\mathsf{K}\ (\omega\ \omega)) \to_{\beta}^{1} \cdots$$

or at some point it could transform to

$$(\mathsf{K}\ (\omega\ \omega)) \to_{\beta}^{1} \lambda y [(\omega\ \omega)] \to_{\beta}^{1} \lambda y [(\omega\ \omega)] \to_{\beta}^{1} \cdots$$

3.  $((K \ \omega) \ \Omega)$  is weakly normalising because it can reduce to the normal form  $\omega$  but it is not strongly normalising because it also has an infinite reduction sequence

$$((\mathsf{K}\ \omega)\ \Omega) \to_{\beta}^{1} ((\mathsf{K}\ \omega)\ \Omega) \to_{\beta}^{1} \cdots$$
PL December 28, 2021 Go BACK FULL SCREEN

542 OF 721

# dse

# Parameter Passing Mechanisms

- Call-by-name/text defines a Leftmost-outermost-computation, i.e. the leftmost-outermost  $\beta$ -redex is chosen for application.
- Call-by-value defines a Leftmost-innermost-computation, i.e. the leftmost-innermost  $\beta$ -redex is chosen for application.

To study these computation rules with regard to computing  $\beta$ -normal forms we consider the following examples.

## Example 12.23 Let

- $L \to_{\beta}^l P \to_{\beta}$  yield a normal form P in l steps of  $\beta$ -reduction,
- $ullet M o_{eta}^m Q o_{eta}$  yield a normal form Q in m steps and
- $\bullet N \to_{\beta}^n Q \to_{\beta}$  yield a normal form R in n steps

44

**•** 

PL December 28, 2021

Go Back

Full Screen

Close



# Deterministic Computation Mechanisms: K

**Example 12.24** For the term  $((K L) \Omega)$  we have the following reduction sequences.

Call-by-name/text. Choose the leftmost-outermost  $\beta$ -redex

$$\bullet ((\mathsf{K} \ L) \ \Omega) \to_{\beta}^{1} (\lambda y[L] \ \Omega) \to_{\beta}^{1} L \to_{\beta}^{l} P$$

Call-by-value. Choose the leftmost-innermost  $\beta$ -redex

$$\bullet \; ((\mathsf{K} \;\; L) \;\; \Omega) \to_\beta^l \; ((\mathsf{K} \;\; P) \;\; \Omega) \to_\beta^1 \; (\lambda y[P] \;\; \Omega) \to_\beta^* \; (\lambda y[P] \;\; \Omega) \to_\beta^* \; \cdots$$

Here Call-by-value fails to produce the normal form even when it exists.

 →
 →
 December 28, 2021
 Go Back
 Full Screen
 Close
 544 of 721



## Deterministic Computation Mechanisms: S

**Example 12.25** For the term ((S L) M) N) we have the following reduction sequences if (P R) (Q R) is in  $\beta$  normal form.

Call-by-name/text. Choose the leftmost-outermost  $\beta$ -redex

$$\bullet (((\mathsf{S}\ L)\ M)\ N) \to_{\beta}^{3} ((L\ N)\ (M\ N)) \to_{\beta}^{l} ((P\ N)\ (M\ N)) \to_{\beta}^{n} ((P\ R)\ (Q\ N)) \to_{\beta}^{n} ((P\ R)\ (Q\ R))$$

Call-by-value. Choose the leftmost-innermost  $\beta$ -redex

$$\bullet (((\mathsf{S}\ L)\ M)\ N) \to_{\beta}^{l} (((\mathsf{S}\ P)\ M)\ N) \to_{\beta}^{m} (((\mathsf{S}\ P)\ Q)\ N) \to_{\beta}^{n} (((\mathsf{S}\ P)\ Q)\ R) \to_{\beta}^{n} (((\mathsf{S}\ P)\ Q)\ R) \to_{\beta}^{n} ((\mathsf{S}\ P)\ R) \to_{\beta}^{n} (\mathsf{S}\ P) \to_{\beta}^{n} (\mathsf{S}\ P) \to_{\beta}^{n} (\mathsf{S}\ P) \to_{\beta}^{n} (\mathsf{S}\ P) \to_{\beta}^{n} (\mathsf{S$$

Call-by-value takes fewer steps to reduce to the normal form because there is no duplication of the argument N.

 →
 Image: Second control of the propertion of the properties of the propertie



### Contrariwise

**Example 12.26** However consider the term ((K L) M).

Call-by-name/text. In l+2 steps we get the normal form.

$$\bullet ((\mathsf{K} \ L) \ M) \to_\beta^1 (\lambda y[L] \ M) \to_\beta^1 L \to_\beta^l P$$

Call-by-value. We get the normal form in l + m + 2 steps.

$$\bullet ((\mathsf{K} \ L) \ M) \to_\beta^l ((\mathsf{K} \ P) \ M) \to_\beta^m ((\mathsf{K} \ P) \ Q) \to_\beta^1 (\lambda y[P] \ Q) \to_\beta^1 (L \to_\beta^l P)$$

Here Call-by-value takes an extra m steps reducing an argument M that has no influence on the computation!

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 546 OF 721



## Some Morals, Some Practice

## In general,

- Call-by-value (example 12.24) may fail to terminate even if there is a possibility of termination.
- If Call-by-value terminates, then Call-by-name will also terminate. More precisely, if a normal form exists then Call-by-name will definitely find it.
- However, Call-by-value when it does terminate may terminate faster (example 12.25) than Call-by-name/text provided all arguments need to be evaluated in both cases.
- It is also easier to implement Call-by-value rather than Call-by-name under static scope rules in the presence of non-local references.

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 547 OF 721







#### 13. Notions of Reduction

# Notions of Reduction









Go Back

Full Screen

Close



## Reduction

For any function such as  $p = \lambda x[3.x.x + 4.x + 1]$ ,

$$(p \ 2) = 3.2.2 + 4.2 + 1 = 21$$

However there is something asymmetric about the identity,

- While (p-2) deterministically produces 3.2.2 + 4.2 + 1 which in turn
- simplifies deterministically to 21,

44







# Reduction Induced Equality

• It is not possible to deterministically infer that 21 came from (p-2). It would be more accurate to refer to this sequence as a reduction sequence and capture the asymmetry as follows:

$$(p \ 2) \leadsto 3.2.2 + 4.2 + 1 \leadsto 21$$

• And yet they are behaviourally equivalent and mutually substitutable in all contexts (referentially transparent).

44

•

▶▶

Go Back

Full Screen

Close



# Reduction Vs. Equality

- 1. Reduction (specifically  $\beta$ -reduction) captures this asymmetry.
- 2. Since reduction produces behaviourally equal terms we have the following notion of equality.

**⋖**◀







# Untyped $\lambda$ -Calculus: $\beta$ -Equality

**Definition 13.1**  $\beta$ -equality or  $\beta$ -conversion (denoted  $=_{\beta}$ ) is the smallest equivalence relation containing  $\beta$ -reduction ( $\rightarrow_{\beta}^*$ ).

The following are equivalent definitions.

- $1. =_{\beta}$  is the reflexive-symmetric-transitive closure of 1-step  $\beta$ -reduction.
- $2. =_{\beta}$  is the smallest relation defined by the following rules.

**◄**◀

**▶** 

PL December 28, 2021

Go Back

Full Screen

Close

## The Paradoxical Combinator



# **Example 13.2** Consider Curry's paradoxical combinator

$$Y_C \stackrel{df}{=} \lambda f[(C \ C)]$$
 where  $C \stackrel{df}{=} \lambda x[(f \ (x \ x))]$ 

For any term L we have

$$(\mathbf{Y_C} \ L) \rightarrow^1_{\beta} (\lambda x[(L \ (x \ x))] \ \lambda x[(L \ (x \ x))])$$

$$\equiv_{\alpha} (\lambda y[(L \ (y \ y))] \ \lambda x[(L \ (x \ x))])$$

$$\rightarrow^1_{\beta} (L \ (\lambda x[(L \ (x \ x))] \ \lambda x[(L \ (x \ x))]))$$

$$=_{\beta} (L (Y_{C} L))$$

Hence  $(Y_C \ L) =_{\beta} (L \ (Y_C \ L))$ . However  $(L \ (Y_C \ L))$  will never  $\beta$ -reduce to  $(Y_C \ L)$ .

44

PL December 28, 2021

Go Back

Full Screen

Close



#### Recursion and the Y combinator.

Since the lambda calculus only has variables and expressions and there is no place for names themselves (we use names such as K and S for our convenience in discourse, but the language itself allows only (untyped) variables and is meant to define functions anonymously as expressions in the language). In such a situation, recursion poses a problem in the language.

Recursion in most programming languages requires the use of an identifier which names an expression that contains a call to the very name of the function that it is supposed to define. This is at variance with the aim of the lambda calculus wherein the only names belong to variables and even functions may be defined anonymously as mere expressions.

This notion of recursive definitions may be generalised to a system of mutually recursive definitions.

The name of a recursive function, acts as a place holder in the body of the definition (which in turn has the name acting as a place holder for a copy of the body of the definition and so on ad infinitum). However no language can have sentences of infinite length.

The combinator  $Y_C$  helps in providing copies of any lambda term L whenever demanded in a more disciplined fashion. This helps in the modelling of recursive definitions anonymously. What the  $Y_C$  combinator provides is mechanism for recursion "unfolding" which is precisely our understanding of how recursion should work. Hence it is easy to see from  $(Y_C \ L) =_{\beta} (L \ (Y_C \ L))$  that

$$(\mathsf{Y}_\mathsf{C}\ L) =_\beta (L\ (\mathsf{Y}_\mathsf{C}\ L)) =_\beta (L\ (L\ (\mathsf{Y}_\mathsf{C}\ L))) =_\beta (L\ (L\ (\mathsf{Y}_\mathsf{C}\ L))) =_\beta \cdots$$

Many other researchers have defined other combinators which mimic the behaviour of the combinator  $Y_C$ . Of particular

44

◀

▶▶

Go Back

Full Screen

Close

556 OF 721



interest is Turing's combinator  $Y_T \stackrel{df}{=} (T \ T)$  where  $T \stackrel{df}{=} \lambda x \ y[(y \ ((x \ x) \ y))]$ . Notice that

$$\begin{array}{l} (\mathsf{T} \ \, \mathsf{T}) \\ \equiv \ \, (\lambda x \; y[(y \; ((x \; x) \; y))] \; \; \mathsf{T}) \\ \rightarrow^1_\beta \ \, \lambda y[(y \; ((\mathsf{T} \; \mathsf{T}) \; y))] \\ \equiv \ \, \lambda y[(y \; (\mathsf{Y}_\mathsf{T} \; \mathsf{y}))] \end{array}$$

from which, by compatible closure, for any term L we get

$$\begin{array}{l} (\mathsf{Y}_\mathsf{T} \ L) \\ \equiv \ ((\mathsf{T} \ \mathsf{T}) \ L) \\ \to_\beta^* \ (\lambda y [(y \ (\mathsf{Y}_\mathsf{T} \ y))] \ L) \\ \to_\beta^1 \ (L \ (\mathsf{Y}_\mathsf{T} \ L)) \end{array}$$

Thus  $Y_T$  is also a recursion unfolding combinator yielding

$$(Y_T L) =_{\beta} (L (Y_T L)) =_{\beta} (L (L (Y_T L))) =_{\beta} (L (L (Y_T L))) =_{\beta} \cdots$$

PL December 28, 2021





CLOSE



# Compatibility of Beta-reduction and Beta-Equality

**Theorem 13.3**  $\beta$ -reduction  $\rightarrow_{\beta}^*$  and  $\beta$ -equality  $=_{\beta}$  are both compatible relations.



















#### Proof of theorem 13.3

*Proof:*  $(\to_{\beta}^*)$  Assume  $L \to_{\beta}^* M$ . By definition of  $\beta$ -reduction  $L \to_{\beta}^n M$  for some  $n \ge 0$ . The proof proceeds by induction on n

**Basis.** n = 0. Then  $L \equiv M$  and there is nothing to prove.

Induction Hypothesis (IH).

The proof holds for all k,  $0 \le k \le m$  for some  $m \ge 0$ .

**Induction Step.** For n=m+1, let  $L\equiv L_0\to_{\beta}^m L_m\to_{\beta}^1 M$ . Then by the induction hypothesis and the compatibility of  $\to_{\beta}^1$  we have

for all 
$$x \in V$$
,  $\lambda x[L] \to_{\beta}^{m} \lambda x[L_{m}]$ ,  $\lambda x[L_{m}] \to_{\beta}^{1} \lambda x[M]$  By definition of  $\to_{\beta}^{n}$  for all  $N \in \Lambda$ ,  $(L \ N) \to_{\beta}^{m} (L_{m} \ N)$ ,  $(L_{m} \ N) \to_{\beta}^{1} (M \ N)$   $(L \ N) \to_{\beta}^{n} (M \ N)$  for all  $N \in \Lambda$ ,  $(N \ L) \to_{\beta}^{m} (N \ L_{m})$ ,  $(N \ L_{m}) \to_{\beta}^{1} (N \ M)$   $(N \ L) \to_{\beta}^{n} (N \ M)$ 

 $End \ (\rightarrow_{\beta}^*)$ 

 $(=_{\beta})$  Assume  $L =_{\beta} M$ . We proceed by induction on the length of the proof of  $L =_{\beta} M$  using the definition of  $\beta$ -equality.

Basis. n = 1. Then either  $L \equiv M$  or  $L \to_{\beta}^* M$ . The case of reflexivity is trivial and the case of  $L \to_{\beta}^* M$  follows from the previous proof. Induction Hypothesis (IH).

44

**>** 

**>>** 

Go Back

Full Screen

Close



For all terms L and M, such that the proof of  $L =_{\beta} M$  requires less than n steps for n > 1, the compatibility result holds.

Induction Step. Suppose the proof requires n steps and the last step is obtained by use of either  $=_{\beta}$  Symmetry or  $=_{\beta}$  Transitivity on some previous steps.

Case ( $=_{\beta}$  Symmetry). Then the (n-1)-st step proved  $M =_{\beta} L$ . By the induction hypothesis and then by applying  $=_{\beta}$  Symmetry to each case we get

for all variables 
$$x$$
,  $\lambda x[M] =_{\beta} \lambda x[L]$   $\lambda x[L] =_{\beta} \lambda x[M]$  for all terms  $N$ ,  $(M \ N) =_{\beta} (L \ N)$   $(L \ N) =_{\beta} (M \ N)$  for all terms  $N$ ,  $(N \ M) =_{\beta} (N \ L)$   $(N \ M) =_{\beta} (N \ L)$ 

Case ( $=_{\beta}$  Transitivity). Suppose  $L =_{\beta} M$  was inferred in the *n*-th step from two previous steps which proved  $L =_{\beta} P$  and  $P =_{\beta} M$  for some term P. Then again by induction hypothesis and then applying  $=_{\beta}$  Transitivity we get

for all variables 
$$x$$
,  $\lambda x[L] =_{\beta} \lambda x[P]$ ,  $\lambda x[P] =_{\beta} \lambda x[M]$   $\lambda x[L] =_{\beta} \lambda x[M]$  for all terms  $N$ ,  $(L \ N) =_{\beta} (P \ N)$ ,  $(P \ N) =_{\beta} (M \ N)$   $(L \ N) =_{\beta} (M \ N)$  for all terms  $N$ ,  $(N \ L) =_{\beta} (N \ P)$ ,  $(N \ P) =_{\beta} (N \ M)$   $(N \ L) =_{\beta} (N \ P)$ 

 $End (=_{\beta})$ 

QED

44

▶▶

PL December 28, 2021

Go Back

Full Screen

Close

# dse

## Eta reduction

Given any term M and a variable  $x \notin FV(M)$ , the syntax allows us to construct the term  $\lambda x[(M \mid x)]$  such that for every term N we have

$$(\lambda x[(M \ x)] \ N) \to_{\beta}^{1} (M \ N)$$

In other words,

$$\lambda x[(M \ x)] \ N) =_{\beta} (M \ N)$$
 for all terms  $N$ 

We say that the two terms  $\lambda x[(M \ x)]$  and M are extensionally equivalent i.e. they are  $syntactically\ distinct$  but there is no way to distinguish between their behaviours.

So we define basic  $\eta$ -reduction as the relation

$$\lambda x[(L \mid x)] \to_{\eta} L \text{ provided } x \notin FV(L)$$
 (7)

 ✓
 ✓

 PL December 28, 2021
 Go Back
 Full Screen
 Close
 561 of 721



# **Eta-Reduction and Eta-Equality**

The following notions are then defined similar to the corresponding notions for  $\beta$ -reduction.

- 1-step  $\eta$ -reduction  $\to_{\eta}^1$  is the closure of basic  $\eta$ -reduction to all contexts,
- $\bullet \to_{\eta}^{n}$  is defined by induction on 1-step  $\eta$ -reduction
- $\eta$ -reduction  $\to_{\eta}^*$  is the reflexive-transitive closure of 1-step  $\eta$ -reduction.
- $\bullet$  the notions of strong and weak  $\eta$  normal forms  $\eta$ -nf.
- ullet the notion of  $\eta$ -equality or  $\eta$ -conversion denoted by  $=_{\eta}$ .

44

PL December 28. 2021

Go Back

Full Screen

Close



#### Exercise 13.1

- 1. Prove that  $\eta$ -reduction and  $\eta$ -equality are both compatible relations.
- 2. Prove that  $\eta$ -reduction is strongly normalising.
- 3. Define basic  $\beta\eta$ -reduction as the application of either (6) or (7). Now prove that  $\rightarrow_{\beta\eta}^1$ ,  $\rightarrow_{\beta\eta}^*$  and  $=_{\beta\eta}$  are all compatible relations.





Go Back

PL December 28, 2021

Full Screen

Close



 ✓
 ✓

 PL December 28, 2021

 Go Back
 Full Screen

 Close
 564 of 721



#### **Confluence Definitions**

Confluence: Definitions

PL December 28, 2021









Go Back

Full Screen

Close



## Reduction Relations

# **Definition 14.1** For any binary relation $\rho$ on $\Lambda$

- 1.  $\rho^1$  is the compatible closure of  $\rho$
- 2.  $\rho^+$  is the transitive closure of  $\rho^1$
- 3.  $\rho^*$  is the reflexive-transitive-closure of  $\rho^1$  and is a preorder
- 4.  $((\rho^1) \cup (\rho^1)^{-1})^*$  (denoted  $=_{\rho}$ ) is the reflexive-symmetric-transitive closure of  $\rho^1$  and is an equivalence relation.

PL December 28, 2021

5.  $=_{\rho}$  is also called the equivalence generated by  $\rho$ .

44

**•** 

▶▶

Go Back

Full Screen

Close

566 OF 721



## Reduction Relations: Arrow Notation

We will often use  $\longrightarrow$  (suitably decorated) as a reduction relation instead of  $\rho$ . Then

- $\bullet \longrightarrow^1$  denotes the compatible closure of  $\longrightarrow$ ,
- $\bullet \longrightarrow^+$  denotes the transitive closure of  $\longrightarrow$ ,
- $\bullet \longrightarrow^*$  denotes the *reflexive-transitive* closure of  $\longrightarrow$ , and
- $\bullet \stackrel{*}{\longleftrightarrow}$  denotes the *equivalence generated* by  $\longrightarrow$ ,

44





PL December 28, 2021



Full Screen

Close



## The Diamond Property

**Definition 14.2** Let  $\rho$  be any relation on terms.  $\rho$  has the diamond property if for all L, M, N,

$$\begin{array}{cccc}
M & M \\
\rho & \rho \\
L & \Rightarrow \exists P : & \rho \\
\rho & \rho & \rho \\
N & N
\end{array}$$

 ✓
 ✓

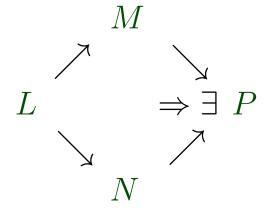
 Full Screen
 Close

 568 of 721



# The Diamond Property: Arrow Notation

We often use a decorated version of the symbol  $\longrightarrow$  for a reduction relation and depict the diamond property as



 →
 Image: Second of the properties of the



## Reduction Relations: Termination

Let  $\longrightarrow$  be a reduction relation,  $\longrightarrow^*$  the least preorder containing  $\longrightarrow$  and  $\stackrel{*}{\longleftrightarrow}$  the least equivalence relation containing  $\longrightarrow^*$ . Then

**Definition 14.3**  $\longrightarrow$  *is* terminating *iff* there is no infinite sequence of the form

$$L_0 \longrightarrow L_1 \longrightarrow \cdots$$

**Lemma 14.4**  $\longrightarrow_{\eta}$  is a terminating reduction relation.

Proof: By induction on the structure of terms.

QED

**∢**∢

▶▶

21

Go Back

Full Screen

Close



#### 14.1. Why confluence?

We are mostly interested in  $\beta$ -reduction which is not guaranteed to terminate. We already know that there are several terms which are only weakly normalising ( $\beta$ -WN). This means that there are several possible reduction sequences, some of which may yield  $\beta$ -normal forms while the others may yield infinite computations. Hence in order to obtain normal forms for such terms we need to schedule the  $\beta$ -reductions carefully to be guaranteed a normal form. The matter would be further complicated if there are multiple unrelated normal forms.

Each  $\beta$ -reduction step may reveal fresh  $\beta$ -redexes. This in turn raises the disquieting possibility that each termination sequence may yield a different  $\beta$ -normal form. If such is indeed the case, then it raises fundamental questions on the use of  $\beta$ -reduction (or function application) as a notion of reduction. If  $\beta$ -reduction is to be considered fundamental to the notion of computation then all  $\beta$ -reduction sequences that terminate in  $\beta$ -nfs must yield the same  $\beta$ -nf upto  $\alpha$ -equivalence.

Hence our interest in the notion of confluence. Since the issue of confluence of  $\beta$ -reduction is rather complicated we approach it in terms of inductively easier notions such as *local confluence*, and *semi-confluence* which finally lead up to *confluence* and the Church-Rosser property.

PL December 28, 2021







Full Screen

571 OF 721



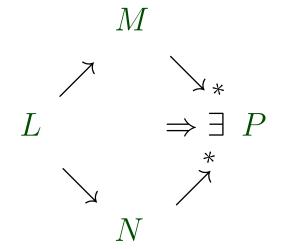
572 of 721

## Reduction: Local Confluence

**Definition 14.5**  $\longrightarrow$  is locally confluent if for all L, M, N,

$$N \longleftarrow L \longrightarrow M \Rightarrow \exists P : N \longrightarrow^* P * \longleftarrow M$$

which we denote by



→ → PL December 28, 2021 Go BACK FULL SCREEN CLOSE

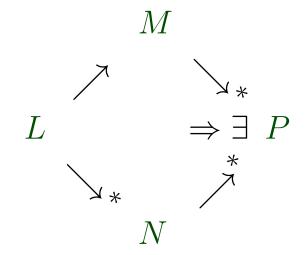


## Reduction: Semi-confluence

**Definition 14.6**  $\longrightarrow$  is semi-confluent if for all L, M, N,

$$N \longleftarrow L \longrightarrow^* M \Rightarrow \exists P : N \longrightarrow^* P * \longleftarrow M$$

which we denote by





## Reduction: Confluence

**Definition 14.7**  $\longrightarrow$  is confluent if for all L, M, N,

$$N * \longleftarrow L \longrightarrow * M \Rightarrow \exists P : N \longrightarrow * P * \longleftarrow M$$

which we denote as

$$\begin{array}{ccc}
M \\
& & \\
& & \\
L & & \Rightarrow \exists P \\
& & \\
& & \\
N
\end{array}$$

Fact 14.8 Any confluent relation is also semi-confluent.





575 of 721

## Reduction: Church-Rosser

**Definition 14.9**  $\longrightarrow$  is Church-Rosser if for all L, M,

$$L \stackrel{*}{\longleftrightarrow} M \Rightarrow \exists P : L \longrightarrow^{*} P \stackrel{*}{\longleftrightarrow} M$$

which we denote by

$$\begin{array}{cccc}
L & \stackrel{*}{\longleftrightarrow} & M \\
\searrow_{*} & & \swarrow \\
\exists P & & \end{array}$$

 →
 →
 December 28, 2021
 Go Back
 Full Screen
 Close



# **Equivalence Characterization**

## Lemma 14.10

- $1. \stackrel{*}{\longleftrightarrow}$  is the least equivalence containing  $\longrightarrow$ .
- $2. \stackrel{*}{\longleftrightarrow}$  is the least equivalence containing  $\longrightarrow^*$ .
- 3.  $L \stackrel{*}{\longleftrightarrow} M$  if and only if there exists a finite sequence  $L \equiv M_0, M_1, \dots M_m \equiv M$ ,  $m \geq 0$  such that for each  $i, 0 \leq i < m$ ,  $M_i \longrightarrow M_{i+1}$  or  $M_{i+1} \longrightarrow M_i$ . We represent this fact more succinctly as

$$L \equiv_{\alpha} M_0 \longrightarrow / \longleftarrow M_1 \longrightarrow / \longleftarrow \cdots \longrightarrow / \longleftarrow M_m \equiv_{\alpha} M \qquad (8)$$

44

**>>** 

PL December 28, 2021

Go Back

FULL SCREEN

Closi







#### Proof of lemma 14.10

#### *Proof:*

- 1. Just prove that  $\stackrel{*}{\longleftrightarrow}$  is a subset of every equivalence that contains  $\longrightarrow$ .
- 2. Use induction on the length of proofs to prove this part
- 3. For the last part it is easy to see that the existence of the "chain equation" (8) implies  $L \stackrel{*}{\longleftrightarrow} M$  by transitivity. For the other part use induction on the length of the proof.

 $\operatorname{QED}$ 

 ✓
 ✓

 PL December 28, 2021

 Go Back
 Full Screen

 Close
 578 of 721





#### 14.2. Confluence: Church-Rosser

# The Church-Rosser Property







Go Back

Full Screen

Close



## Confluence and Church-Rosser

**Lemma 14.11** Every confluent relation is also semi-confluent

**Theorem 14.12** The following statements are equivalent for any reduction relation  $\longrightarrow$ .

- $1. \longrightarrow is Church-Rosser.$
- $2. \longrightarrow is confluent.$

44

**•** 

▶▶

PL December 28, 2021

Go Back

Full Screen

Close



#### Proof of theorem 14.12

*Proof:*  $(1 \Rightarrow 2)$  Assume  $\longrightarrow$  is Church-Rosser and let

$$N * \longleftarrow L \longrightarrow * M$$

Clearly then  $N \stackrel{*}{\longleftrightarrow} M$ . If  $\longrightarrow$  is Church-Rosser then

$$\exists P: N \longrightarrow^* P *\longleftarrow M$$

which implies that it is confluent.

 $(2 \Rightarrow 1)$  Assume  $\longrightarrow$  is confluent and let  $L \stackrel{*}{\longleftrightarrow} M$ . We proceed by induction on the length of the chain (8).

$$L \equiv_{\alpha} M_0 \longrightarrow / \longleftarrow M_1 \longrightarrow / \longleftarrow \cdots \longrightarrow / \longleftarrow M_m \equiv_{\alpha} M$$

**Basis.** m=0. This case is trivial since for any  $P, L \longrightarrow^* P$  iff  $M \longrightarrow^* P$ 

Induction Hypothesis (IH).

The claim is true for all chains of length k,  $0 \le k < m$ .

**Induction Step.** Assume the chain is of length m = k + 1. i.e.

$$L \equiv_{\alpha} M_0 \longrightarrow / \longleftarrow M_1 \longrightarrow / \longleftarrow \cdots \longrightarrow / \longleftarrow M_k \longrightarrow / \longleftarrow M_{k+1} \equiv_{\alpha} M$$

Case  $M_k \longrightarrow M$ . Then by the induction hypothesis and semi-confluence we have

PL December 28, 2021



which proves the claim.

Case  $M_k \leftarrow M$ . Then the claim follows from the induction hypothesis and the following diagram

$$\begin{array}{cccc}
L & \stackrel{*}{\longleftrightarrow} & M_k \longleftarrow M \\
& \searrow_* & \downarrow & \swarrow \\
& \exists P & & \end{array}$$

QED

Lemma 14.13 If a terminating relation is locally confluent then it is semi-confluent.

*Proof:* Assume  $L \longrightarrow M$  and  $L \longrightarrow^* N$ . We need to show that there exists P such that  $M \longrightarrow^* P$  and  $N \longrightarrow^* P$ . We prove this by induction on the length of  $L \longrightarrow^* N$ . If  $L \equiv_{\alpha} N$  then  $P \equiv_{\alpha} M$ , otherwise assume  $L \longrightarrow N_1 \longrightarrow \cdots \longrightarrow N_n = N$  for some n > 0. By the local confluence we have there exists  $P_1$  such that  $M \longrightarrow^* P_1$ . By successively applying the induction hypothesis we get terms  $P_2, \ldots, P_n$  such that  $P_{j-1} \longrightarrow^* P_j$  and  $P_j \longrightarrow^* P_j$  for each  $P_j \cap P_j$  for each  $P_j$ 

QED

From lemma 14.13 and theorem 14.12 we have the following theorem.

Theorem 14.14 If a terminating relation is locally confluent then it is confluent.

44

▶▶

PL December 28. 2021

Go Back

Full Screen

Close



#### *Proof:*

- $\longrightarrow$  on  $\Lambda$  is given to be terminating and locally confluent. We need to show that it is confluent. That is for any L, we are given that
  - 1. there is no infinite sequence of reductions of L, i.e. every maximal sequence of reductions of L is of length n for some n > 0.
  - 2.

$$N_1 \stackrel{1}{\longleftarrow} L \longrightarrow^1 M_1 \Rightarrow \exists P : M_1 \longrightarrow^* P \stackrel{*}{\longleftarrow} N_1$$
 (9)

We need to show for any term L that

$$N^* \leftarrow L \longrightarrow^* M \Rightarrow \exists S : M \longrightarrow^* S^* \leftarrow N$$
 (10)

Let L be any term. Consider the graph  $G(L) = \langle \Gamma(L), \longrightarrow^1 \rangle$  such that  $\Gamma(L) = \{M \mid L \longrightarrow^* M\}$ . Since  $\longrightarrow$  is a terminating reduction

**Fact 14.15** The graph G(L) is acyclic for any term L.

If G(L) is not acyclic, there must be a cycle of length k > 0 such that  $M_0 \longrightarrow^1 M_1 \longrightarrow^1 \cdots \longrightarrow^1 M_{k-1} \longrightarrow^1 M_0$  which implies there is also an infinite reduction sequence of the form  $L \longrightarrow^* M_0 \longrightarrow^k M_0 \longrightarrow^k \cdots$  which is impossible.

Since there are only a finite number of sub-terms of L that may be reduced under  $\longrightarrow$ , for each L there is a maximum number  $p \ge 0$ , which is the length of the longest reduction sequence.

#### Fact 14.16 For every $M \in \Gamma(L)$ ,

- 1. G(M) is a sub-graph of G(L) and
- 2. For every  $M \in \Gamma(L) \{L\}$ , the length of the longest reduction sequence of M is less than p.

44

▶▶

PL December 28, 2021 Go

Go Back

Full Screen

Close



*Proof:* We proceed by induction on p.

**Basis.** p = 0. Then  $\Gamma(L) = \{L\}$  and there are no reductions possible, so it is trivially confluent.

Induction Hypothesis (IH).

For any L whose longest reduction sequence is of length k,  $0 \le k < p$ , property (10) holds.

**Induction Step.** Assume L is a term whose longest reduction sequence is of length p > 0. Also assume  $N \not\leftarrow L \longrightarrow^* M$  i.e.  $\exists m, n \geq 0 : N \not\sim L \longrightarrow^m M$ .

<u>Case m=0</u>. If m=0 then  $M \equiv_{\alpha} L$  and hence  $S \equiv_{\alpha} N$ .

<u>Case n=0</u>. Then  $N \equiv_{\alpha} L$  and we have  $S \equiv_{\alpha} M$ .

Case m, n > 0. Then consider  $M_1$  and  $N_1$  such that

$$N^* \leftarrow N_1 \stackrel{1}{\leftarrow} L \longrightarrow^1 M_1 \longrightarrow^* M$$
 (11)

See figure (7). By (9),  $\exists P: M_1 \longrightarrow^* P^* \longleftarrow N_1$ . Clearly  $M_1, N_1, P \in \Gamma(L) - \{L\}$ . Hence by fact 14.16,  $G(M_1)$ ,  $G(N_1)$  and G(P) are all sub-graphs of G(L) and all their reduction sequences are of length smaller than p. Hence by induction hypothesis, we get

$$P \overset{*}{\longleftarrow} M_1 \longrightarrow^* M \Rightarrow \exists Q : M \longrightarrow^* Q \overset{*}{\longleftarrow} P \tag{12}$$

and

$$N^* \leftarrow N_1 \longrightarrow^* P \Rightarrow \exists R : P \longrightarrow^* R^* \leftarrow N \tag{13}$$

But by (12) and (13) and the induction hypothesis we have

$$R^* \leftarrow P \longrightarrow^* Q \Rightarrow \exists S : Q \longrightarrow^* S^* \leftarrow R$$
 (14)

Combining (14) with (11), (12) and (13) we get

$$N \stackrel{*}{\longleftarrow} L \longrightarrow^* M \Rightarrow \exists S : M \longrightarrow^* S \stackrel{*}{\longleftarrow} N$$
 (15)

**⋖**◀

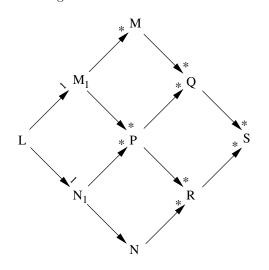
Go Back

Full Screen

Close



Figure 7: Case m > 0 and n > 0



QED

Theorem 14.17 If a terminating relation is locally confluent then it is Church-Rosser.

*Proof:* Follows from theorem 14.14 and theorem 14.12

QED

 ■
 ■
 ■
 December 28, 2021
 Go Back
 Full Screen
 Close
 586 of 721



Close

587 of 721



#### 14.3. The Church-Rosser Property

# The Church-Rosser Property













Full Screen

Close



### Parallel Beta Reduction

**Definition 14.18** The parallel- $\beta$  or  $||\beta|$  reduction is the smallest relation for which the following rules hold.

$$\|\beta_{\mathbf{1}} \xrightarrow{L} \frac{1}{\|\beta\|L}$$

$$|\beta_{\mathbf{1}} \mathbf{App}| \frac{L \longrightarrow_{|\beta}^{1} L', M \longrightarrow_{|\beta}^{1} M'}{(L M) \longrightarrow_{|\beta}^{1} (L' M')}$$

$$|\beta_{1}Abs1| \frac{L \longrightarrow_{|\beta}^{1} L'}{\lambda x[L] \longrightarrow_{|\beta}^{1} \lambda x[L']}$$

$$|\beta_{\mathbf{1}}\mathbf{Abs2}| \frac{L \longrightarrow_{|\beta}^{1} L', M \longrightarrow_{|\beta}^{1} M'}{(\lambda x[L] M) \longrightarrow_{|\beta}^{1} \{M'/x\}L'}$$

 ✓
 ✓

 PL December 28, 2021

 Go Back
 Full Screen

 Close
 589 of 721



## Parallel Beta: The Diamond Property

## Lemma 14.19

$$1. L \longrightarrow_{\beta}^{1} L' \Rightarrow L \longrightarrow_{\parallel \beta}^{1} L'.$$

$$2. L \longrightarrow_{|\beta}^{1} L' \Rightarrow L \longrightarrow_{\beta}^{*} L'.$$

3. The smallest preorder containing  $\longrightarrow_{||\beta}^1$  is  $\longrightarrow_{||\beta}^* = \longrightarrow_{\beta}^*$ .

4. If 
$$L \longrightarrow_{\beta}^{1} L'$$
 and  $M \longrightarrow_{|\beta}^{1} M'$  then  $\{M/x\}L \longrightarrow_{|\beta}^{1} \{M'/x\}L'$ .

Proof: By induction on the structure of terms or by induction on the number of steps in any proof. QED

**Theorem 14.20**  $\longrightarrow_{||\beta|}^{1}$  has the diamond property.

→ PL December 28, 2021 Go Back



#### Proof of theorem 14.20

*Proof:* We need to prove for all L

$$N \xrightarrow[|\beta]{1} L \longrightarrow_{|\beta}^{1} M \Rightarrow \exists P : N \longrightarrow_{|\beta}^{1} P \xrightarrow[|\beta]{1} M$$

We prove this by induction on the structure of L and a case analysis of the rule applied in definition 14.18.

Case 
$$L \equiv x \in V$$
. Then  $L \equiv M \equiv N \equiv P$ .

Before dealing with the other inductive cases we dispose of some trivial sub-cases that arise in some or all of them.

Case  $L \equiv_{\alpha} M$ . Choose  $P \equiv_{\alpha} N$  to complete the diamond.

Case  $L \equiv_{\alpha} N$ . Then choose  $P \equiv_{\alpha} M$ .

Case  $M \equiv_{\alpha} N$ . Then there is nothing to prove.

In the sequel we assume  $N \not\equiv_{\alpha} L \not\equiv_{\alpha} M \not\equiv_{\alpha} N$  and proceed by induction on the structure of L.

Case  $L \equiv \lambda x[L_1]$ . Then clearly M and N were both obtained in proofs whose last step was an application of rule  $||\beta_1 Abs1|$  and so  $M \equiv \lambda x[M_1]$  and  $N \equiv \lambda x[N_1]$  for some  $M_1$  and  $N_1$  respectively and hence  $N_1 \stackrel{1}{||\beta|} L_1 \longrightarrow \stackrel{1}{||\beta|} M_1$ . By the induction hypothesis we have

$$\exists P_1: N_1 \longrightarrow_{\parallel\beta}^1 P_1 \stackrel{1}{\parallel\beta} \longleftarrow M_1$$

Hence by choosing  $P \equiv \lambda x[P_1]$  we obtain the required result.

Case  $L \equiv (L_1 \ L_2)$  and  $L_1$  is not an abstraction.

The rule  $||\beta_1 App|$  is the only rule that must have been applicable in the last step of the proofs of N  $_{||\beta}^1 - L \longrightarrow_{||\beta}^1 M$ . Clearly then there exist  $M_1$ ,  $M_2$ ,  $N_1$ ,  $N_2$  such that  $N_1$   $_{||\beta}^1 - L_1 \longrightarrow_{||\beta}^1 M_1$  and  $N_2$   $_{||\beta}^1 - L_2 \longrightarrow_{||\beta}^1 M_2$ . Again by the induction hypothesis, we have

$$\exists P_1: N_1 \longrightarrow_{\parallel\beta}^1 P_1 \stackrel{1}{\parallel\beta} \longleftarrow M_1$$

44

**▶** 

PL December 28, 2021

Go Back

FULL SCREEN

Close



and

$$\exists P_2: N_2 \longrightarrow_{\parallel\beta}^1 P_2 \stackrel{1}{\parallel} \longleftarrow M_2$$

By choosing  $P \equiv (P_1 \ P_2)$  we obtain the desired result.

Case  $L \equiv (\lambda x[L_1] \ L_2)$ .

Here we have four sub-cases depending upon whether each of M and N were obtained by an application of  $||\beta_1 App|$  or  $||\beta_1 Abs|^2$ . Of these the sub-case when both M and N were obtained by applying  $||\beta_1 App||$  is easy and similar to the previous case. That leaves us with three subscases.

Sub-case: Both M and N were obtained by applying rule  $||\beta_1 Abs2|$ .

Then we have

$$\{N_2/x\}N_1 \equiv N \stackrel{1}{\parallel_{\beta}} \longleftarrow L \equiv (\lambda x[L_1] \quad L_2) \longrightarrow \stackrel{1}{\parallel_{\beta}} M \equiv \{M_2/x\}M_1$$

for some  $M_1$ ,  $M_2$ ,  $N_1$ ,  $N_2$  such that

$$N_1 \stackrel{1}{\parallel_{\beta}} \longleftarrow L_1 \longrightarrow \stackrel{1}{\parallel_{\beta}} M_1$$

and

$$N_2 \stackrel{1}{\parallel_{\beta}} \longleftarrow L_2 \longrightarrow \stackrel{1}{\parallel_{\beta}} M_2$$

By the induction hypothesis

$$\exists P_1: N_1 \longrightarrow_{\parallel\beta}^1 P_1 \stackrel{1}{\parallel} \longleftarrow M_1$$

and

$$\exists P_2: N_2 \longrightarrow_{\parallel\beta}^1 P_2 \stackrel{1}{\parallel} \longleftarrow M_2$$

and the last part of lemma 14.19 we have

$$\exists P \equiv \{P_2/x\}P_1 : N \longrightarrow_{\parallel\beta}^1 P \stackrel{1}{\parallel} \longleftarrow M$$

PL December 28, 2021

completing the proof.

Sub-case: M was obtained by applying rule  $||\beta_1 Abs2|$  and N by  $||\beta_1 App|$ .



Then we have the form

$$(\lambda x[N_1] \ N_2) \equiv N \stackrel{1}{\parallel_{\beta}} \longleftarrow L \equiv (\lambda x[L_1] \ L_2) \longrightarrow^1_{\parallel\beta} M \equiv \{M_2/x\}M_1$$

where again

$$N_1 \stackrel{1}{\parallel_{\beta}} \longleftarrow L_1 \longrightarrow \stackrel{1}{\parallel_{\beta}} M_1$$

and

$$N_2 \stackrel{1}{\parallel_{\beta}} \longleftarrow L_2 \longrightarrow \stackrel{1}{\parallel_{\beta}} M_2$$

By the induction hypothesis

$$\exists P_1: N_1 \longrightarrow_{\parallel\beta}^1 P_1 \stackrel{1}{\parallel} \longleftarrow M_1$$

and

$$\exists P_2: N_2 \longrightarrow_{\parallel\beta}^1 P_2 \stackrel{1}{\parallel} \longleftarrow M_2$$

and finally we have

$$\exists P \equiv \{P_2/x\}P_1 : N \longrightarrow_{\parallel\beta}^1 P \stackrel{1}{\parallel\beta} \longleftarrow M$$

completing the proof.

Sub-case: M was obtained by applying rule  $||\beta_1 App|$  and N by  $||\beta_1 Abs2|$ .

Similar to the previous sub-case.

QED

## Theorem 14.21 $\longrightarrow_{\parallel\beta}^1$ is confluent.

*Proof:* We need to show that for all L, M, N,

$$N \parallel_{B} \stackrel{*}{\longleftarrow} L \longrightarrow_{B} ^{*} M \Rightarrow \exists P : N \longrightarrow_{B} ^{*} P \parallel_{B} \stackrel{*}{\longleftarrow} M$$

44

▶▶

PL December 28, 2021 G

Go Back

Full Screen

Close



We prove this by induction on the length of the sequences

$$L \longrightarrow_{\parallel\beta}^{1} M_{1} \longrightarrow_{\parallel\beta}^{1} M_{2} \longrightarrow_{\parallel\beta}^{1} \cdots \longrightarrow_{\parallel\beta}^{1} M_{m} \equiv M$$

and

$$L \longrightarrow_{\parallel\beta}^{1} N_{1} \longrightarrow_{\parallel\beta}^{1} N_{2} \longrightarrow_{\parallel\beta}^{1} \cdots \longrightarrow_{\parallel\beta}^{1} N_{n} \equiv N$$

where  $m, n \ge 0$ . More specifically we prove this by induction on the pairs of integers (j, i) bounded by (n, m), where (j, i) < (j', i') if and only if either j < j' or (j = j') and i < i'. The interesting cases are those where both m, n > 0. So we repeatedly apply theorem 14.20 to complete the rectangle

$$L \longrightarrow_{||\beta}^{1} M_{1} \longrightarrow_{||\beta}^{1} M_{2} \longrightarrow_{||\beta}^{1} \cdots \longrightarrow_{||\beta}^{1} M_{m} \equiv M$$

$$||\beta\downarrow^{1} \longrightarrow_{||\beta}^{1} P_{11} \longrightarrow_{||\beta}^{1} P_{12} \longrightarrow_{||\beta}^{1} \cdots \longrightarrow_{||\beta}^{1} P_{1m}$$

$$||\beta\downarrow^{1} \longrightarrow_{||\beta}^{1} ||\beta\downarrow^{1} \longrightarrow_{||\beta}^{1} \cdots \longrightarrow_{||\beta}^{1} P_{1m}$$

$$\vdots \qquad \vdots \qquad \vdots \qquad \vdots \qquad \vdots$$

$$||\beta\downarrow^{1} \longrightarrow_{||\beta}^{1} P_{n1} \longrightarrow_{||\beta}^{1} P_{n2} \longrightarrow_{||\beta}^{1} \cdots \longrightarrow_{||\beta}^{1} P_{nm} \equiv P$$

$$QED$$

Corollary 14.22  $\longrightarrow_{\beta}^{1}$  is confluent.

*Proof:* Since  $\longrightarrow_{\beta}^* = \longrightarrow_{\parallel \beta}^*$  it follows from theorem 14.21 that  $\longrightarrow_{\beta}^1$  is confluent.

QED

Corollary 14.23 If a term reduces to a  $\beta$ -normal form then the normal form is unique (upto  $\equiv_{\alpha}$ ).

44

**•** 

**>>** 

Go Back

Full Screen

Close



*Proof:* If  $N_1 \not\leftarrow L \longrightarrow_{\beta}^* N_2$  and both  $N_1 N_2$  are  $\beta$ -nfs, then by the corollary 14.22 they must both be  $\beta$ -reducible to a third element  $N_3$  which is impossible if both  $N_1$  and  $N_2$  are  $\beta$ -nfs. Hence  $\beta$ -nfs are unique whenever they exist. QED

PL December 28, 2021

Corollary 14.24  $\longrightarrow_{\beta}^{1}$  is Church-Rosser.

*Proof:* Follows from corollary 14.22 and theorem 14.12.

QED



Go Back

Full Screen

Close



→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 596 OF 721



An Applied Lambda-Calculus

15.1. FL with recursion

An Applied Lambda-Calculus With Types







Go Back

Full Screen

Close



#### 15.2. Motivation and Organization

In the sequel we will define, by stages a simple higher order programming language.

- **Stage 0.** A simple expression language to represent integers and booleans. Initially we define a representation for integers and booleans purely symbolically (16) as a data type with constant constructors.
- Stage 1. FL(X) a simple expression (functional programming) language with variables that allows expressions to be defined on the two types of data integers and booleans.
  - Static Semantics. By allowing more than one type of data we also show that there is a need for a type-checking discipline since several meaningless constructs may be generated by the grammar. We specify the type-checking (type-inferencing) system for this simple language as the static semantics of the language.
  - **Functional Semantics.** For the well-typed terms we also define the *intended meanings* of these expressions, by defining a functional semantics.
  - Operational (Reduction) Semantics. We show that we can capture the intended meanings of well-typed expressions by a dynamic semantics which specifies symbolically a notion of reduction ( $\delta$ -rules (38) to (47)).
  - Relating Functional and Operational Semantics. The integer values and boolean values are denoted symbolically by  $\delta$ -normal forms. Lemma 15.2 shows that the intended values of integers and boolean values are obtained as

PL December 28, 2021

**◄**◀



- $\delta$ -normal forms and in combination with the property of confluence (see exercise 15.1 problem 3) it follows that all integer and boolean values have unique normal form representations in the expression language.
- Subject Reduction. In problem 15.1.5 we encourage the reader to show that types are preserved under  $\delta$ -reductions (5) i.e. the type of an expression cannot change arbitrarily during reduction (program execution) an important static property that a dynamic semantics should obey.
- Referential Transparency. Further, in problem (6) the reader is encouraged to show that the language enjoys the property of referential transparency viz. that each variable name in an expression may be substituted by its value while preserving the meaning of the expression a dynamic property that any functional programming language should obey.
- Stage 2.  $\Lambda$ +FL(X). However, the language FL(X) lacks the elementary facilities for user-defined functions. Add to that the lack of expressiveness to define even the most common useful integer operations such as addition, subtraction and multiplication. We rectify this by defining  $\lambda$ -abstraction and application to terms of the language. The new extended language  $\Lambda$ +FL(X) allows us to define some (non-recursive) operators and functions over the terms of the language FL(X). BY this addition,  $\beta$ -reduction has been added to the language as well. The language  $\Lambda$ +FL(X) is expressive enough to define some of the common boolean operators and the most common order relations on integers. These have been made possible due to the inclusion of the ternary if-the-else construct(or) ITE and construct(or)s for checking for 0 (IZ) and positive integer values (GTZ).
- Stage 3. The addition of the  $\lambda$ -abstraction and application on top of FL(X) has the drawback that functions and function applications do not have the same status as expressions. To bring function definition and application down to the expression level it is necessary to allow an intermingling of the two. Hence we "flatten" the language to produce a

**⋖**◆

**▶** 

Go Back

Full Screen

Close



genuinely applied  $\lambda$ -calculus with a  $\beta\delta$  reduction mechanism. The result is the language  $\Lambda_{FL}(X)$ .

Stage 4.  $\Lambda_{FL}(X)$  allows the full power of the  $\lambda$ -calculus to be incorporated into the language. Hence it allows higher-order functions as well. However, the power of recursion is not achieved in a type-safe manner because no paradoxical combinator can be made type-safe. Hence even to program some elementary inductive functions like addition, a recursion operator is absolutely required. This yields the language  $\Lambda_{RecFL}(X)$ .

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 600 OF 721

# A Simple Language of Terms: FL0



Let X be an infinite collection of variables (names). Consider the language (actually a collection of abstract syntax trees) of terms  $T_{\Omega}(X)$  defined by the following constructors (along with their intended meanings).  $T_{\Omega}$  denotes the variable-free subset of  $T_{\Omega}(X)$  and is called the set of ground terms.

Construct	Arity	Informal Meaning
Z	0	The number 0
T	0	The truth value ${f true}$
F	0	The truth value false
P	1	The predecessor function on numbers
S	1	The successor function on numbers
ITE	3	The $if$ -then-else construct (on numbers and truth values)
IZ	1	The is-zero predicate on numbers
GTZ	1	The greater-than-zero predicate on numbers

44

**>>** 

PL December 28, 2021

Go Back

Full Screen

Close

## FL(X): Language, Datatype or Instruction Set?



The set of terms  $T_{\Omega}(X)$ , where X is an infinite collection of variable names (that are disjoint from all other symbols in the language) may be defined by the BNF:

$$t := x \in X \mid \mathbf{Z} \mid (\mathbf{P} \ t) \mid (\mathbf{S} \ t) \mid \mathbf{T} \mid \mathbf{F} \mid (\mathbf{ITE} \ \langle t, t_1, t_0 \rangle) \mid (\mathbf{IZ} \ t) \mid (\mathbf{GTZ} \ t)$$

$$\tag{16}$$

- It could be thought of as a user-defined data-type
- It could be thought of as the instruction-set of a particularly simple hardware machine.
- It could be thought of as a simple functional programming language without recursion.
- It is a language with two simple types of data: integers and booleans
- Notice that the constructor (ITE  $\langle t, t_1, t_0 \rangle$ ) is overloaded.

44

**•** 

00 0001

K FULL

CLO

# d\$E

# Extending the language

To make this simple language safe we require

Type-checking: to ensure that arbitrary expressions are not mixed in ways they are not "intended" to be used. For example

- ullet t cannot be a boolean expression in (S t), (P t), (IZ t) and (GTZ t)
- (ITE  $\langle t, t_1, t_0 \rangle$ ) may be used as a conditional expression for both integers and booleans, but t needs to be a boolean and either both  $t_1$  and  $t_0$  are integer expressions or both are boolean expressions.

Functions: To be a useful programming language we need to be able to define functions.

**Recursion**: to be able to define complex functions in a well-typed fashion. Recursion should also be well-typed

44

**•** 

PL December 28. 2021

Go Васк

Full Screen

Close



# Typing FL0 Expressions

We have only two types of objects in FLO – integers and booleans which we represent by <u>int</u> and <u>bool</u> respectively. We then have the following elementary typing annotations for the expressions, which may be obtained by pattern matching.

Basis.

Int.

Bool.

boolCond.

intCond.

Z: int, T: bool, F: bool

 $S: \underline{int} \to \underline{int}, \quad P: \underline{int} \to \underline{int}$ 

 $IZ : \underline{int} \to \underline{bool}, \qquad GTZ : \underline{int} \to \underline{bool}$ 

ITEB :  $\underline{bool} * \underline{bool} * \underline{bool} \rightarrow \underline{bool}$ 

ITEI :  $\underline{bool} * \underline{int} * \underline{int} \rightarrow \underline{int}$ 

44

**|** 

▶▶

Go Back

Full Screen

Close



#### 15.3. Static Semantics of FL(X)

While **dynamic** semantics refers to the run-time behaviour of a program, the **static** semantics refers to all the context-sensitive information about a program that needs to be gathered during the compilation process, to enable the generation of code both for execution as well as error-reporting and handling. Most of this information about variable symbols is stored in the symbol table and is accessed during the code-generation process for both memory allocation and the actual generation of target code.

The purposes of both code-generation and memory allocation aspects are more or less (i.e. except for scope and the absolute addresses of the data-objects during execution) covered by determining the types of the various objects in a program (data objects, functions, procedures etc.). The type of a scalar data item implicitly defines the amount of storage it requires. For example, an integer variable needs perhaps one word of storage and a floating point variable requires two-words of storage, booleans require just a bit (but in the case of byte-addressable or word-addressable machines machines it may be more efficient to assign a byte or word of storage to it). Similarly characters may require a byte of storage and strings require storage that is proportional to their length. All complex data items such as records and arrays being built of the scalar components require correspondingly proportional amounts of storage in the run-time stack. For each of these the compiler creates a so-called data descriptor and stores it in the symbol table and refers to it while generating code. The control units viz. expressions, commands, functions and procedures would require storage (in the code-segment) proportional to the length of the code that is generated for each of them; and the parameters they invoke correspondingly require data

44

**◄** 

PL December 28, 2021

Go Back

Full Screen

Close

descriptors to define the storage requirements for the parameters. Further in the process of compiling a procedure or a function the types of input and output parameters in the definition (declaration) should correspond exactly with the types of the actual parameters in each call (otherwise a compile-time error needs to be generated).

Much of the above process can all be captured by the simple process of assigning types to each data and control unit in a program. Hence most compilers (with static scoping rules) actually perform *static* or *compile-time* type-checking.

#### 15.3.1. Type-checking FL(X) terms

While trying to type FL0 expressions we have had to introduce two new type operators viz. \* and  $\rightarrow$  which allow us to precisely capture the types of expressions involving constructors such as S, P, IZ, GTZ, ITE etc. which we intend to view as functions of appropriate arity on appropriate types of arguments. These type operators will be required for specifying the types of other (user-defined) functions as well. Hence it makes sense for us to define a *formal* language of type expressions (with type variables!) to enable us define types of polymorphic operations (which in the particular case of FL(X) is restricted to overloading the ITE constructor). As we shall see later, this expression language of types may be defined by the grammar

$$\sigma, \tau ::= \underline{\mathtt{int}} \quad \left| \begin{array}{c} \mathtt{bool} \end{array} \right| \quad '\mathtt{a} \in TV \quad \left| \begin{array}{c} (\sigma {
ightarrow} au) \end{array} \right| \quad (\sigma {
ightarrow} au)$$

where  $a \in TV$  is a type variable and all type variables are distinct from program variables in X.

What we have specified earlier are the typing axioms for the *constant* expressions (without variables). For the purpose of typing expressions involving (free) variables we require assumptions to be made about the types of the variables occurring in an expression. In most programming languages these assumptions come from the declarations of variables. For instance,

44

**•** 

▶▶

Go Back

Full Screen

Close

the successor constructor S should be applied only to expressions which yield integer values. Hence for any expression  $t \in FL(X)$ , (S t) would be well-typed only if  $t : \underline{int}$  and further (S t):  $\underline{int}$ . Similarly, given three expressions t,  $t_1$ ,  $t_0$ , the expression (ITE  $\langle t, t_1, t_0 \rangle$ ) type-checks i.e. it is well-typed only if  $t : \underline{bool}$  and the types of  $t_1$  and  $t_0$  are the same – either both  $\underline{bool}$  or both  $\underline{int}$ .

#### 15.3.2. The Typing Rules

In a language of expressions that requires the type of each variable to be declared beforehand, the list of (free) variables and their types may be available as a type environment  $\Gamma$  and the rules that we give are type-checking rules. The rules for type-checking any expression  $t \in T_{\Omega}(X)$  extend the earlier specification by induction on the structure of expressions. More precisely, the earlier specification form the basis of an induction by structure of expressions. The 0-ary constructors and variables form the basis for the structural induction rules and have the following axioms and their types are independent of the type environment. The type-checking rules for expressions form the induction step of the type-checking algorithm and go as follows. These rules also assign types to each individual sub-expression along the way. We begin with the unary constructors and conclude with the conditional operator.

Alternatively, in the absence of declarations, we could derive them as *constraints* on the types of variables (as we shall see later). It is then necessary to also use the concept of type variables as distinct from program variables.

44

▶▶

Go Back

Full Screen

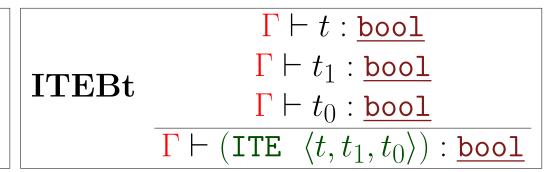
Close

# Static Semantics of FL(X): Type-checking



$$\mathbf{IZt} \ \ \frac{\Gamma \vdash t : \underline{\mathtt{int}}}{\Gamma \vdash (\mathtt{IZ} \ t) : \underline{\mathtt{bool}}} \ \ \boxed{\mathbf{GTZt} \ \ \frac{\Gamma \vdash t : \underline{\mathtt{int}}}{\Gamma \vdash (\mathtt{GTZ} \ t) : \underline{\mathtt{bool}}}}$$

$$\mathbf{ITEIt} \begin{array}{c} \Gamma \vdash t : \underline{\mathtt{bool}} \\ \Gamma \vdash t_1 : \underline{\mathtt{int}} \\ \Gamma \vdash t_0 : \underline{\mathtt{int}} \\ \hline \Gamma \vdash (\mathtt{ITE} \ \langle t, t_1, t_0 \rangle) : \underline{\mathtt{int}} \end{array}$$



**⋖**◀

**•** 

**>>** 

PL December 28, 2021

Go Back

Full Screen

Close



## Well-typed Terms

**Definition 15.1** A term  $t \in T_{\Omega}(X)$  in a type environment  $\Gamma$  is well-typed if there exists a proof of either  $\Gamma \vdash t : \underline{\mathtt{int}}$  or  $\Gamma \vdash t : \underline{\mathtt{bool}}$  (not both).

As we have seen before there are terms that are not well-typed. We consider only the subset  $WT_{\Omega}(X) \subset T_{\Omega}(X)$  while describing the dynamic semantics. While  $T_{\Omega}$  is variable-free subset of  $T_{\Omega}(X)$ ,  $WT_{\Omega} \subset WT_{\Omega}(X)$  is the variable-free subset of  $WT_{\Omega}(X)$ .

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 609 OF 721



# Dynamic Semantics of FL(X)

The dynamic semantics or the run-time behaviour of FL(X) expressions may be specified in several ways.

Functional semantics. The language designer could specify the intended meanings of the constants, constructors and operators in terms that are useful to the user programmer as functions (as an extension of the informal meaning specified earlier), or

Operational semantics The *implementor* of the language could specify the run-time behaviour of expressions through an abstract algorithm.

But any *implementation* should also be consistent with the intended meanings specified by the *designer* 

 ✓
 ✓

 PL December 28, 2021

 Go Back
 Full Screen

 Close
 610 of 721



# Functional Semantics of FL(X):0

Boolean constants. The constructors T and F are interpreted as the boolean constants true and false respectively.

 ${f Zero}$  . Z is interpreted as the *constant* number 0

Positive integers . Each k-fold application, k>0, of the constructor S to Z viz.  $(S \ldots (S Z) \ldots)$  (abbreviated to  $(S^k Z)$  for convenience) is interpreted as the positive integer k.

Negative integers . Similarly, each k-fold application, k>0, of the constructor P to Z viz.  $\underbrace{(P\ \dots (P\ Z)\dots)}_{k-fold}$  (abbreviated as  $(P^k\ Z)$ ) is interpreted as -k.

44

**•** 

PL December 28. 2021

Go Back

Full Screen

Close

# Functional Semantics of FL(X):1



- ullet Let  $\mathbb Z$  and  $\mathbb B$  denote the sets of integers and booleans respectively.
- ullet Each well-typed expression in  $T_\Omega(X)$  denotes either an integer or boolean value depending upon its type.
- Let  $\mathcal{V} = \{v \mid v : X \to (\mathbb{Z} \cup \mathbb{B})\}$  denote the set of all *valuation environments* which associate with each variable a value of the appropriate type (either integer or boolean).
- With the interpretation of the symbols in the language given earlier we associate a meaning function

$$\mathscr{M}:WT_{\Omega}(X)\to (\mathcal{V}\to (\mathbb{Z}\cup\mathbb{B}))$$

such that for each well-typed expression  $t \in WT_{\Omega}(X)$ ,  $\mathscr{M}[t]$  is a function that extends each  $v \in \mathcal{V}$ , inductively on the structure of expressions to a value of the appropriate type.

44

**>>** 

Go Back

Full Screen

CLOSE



(17)

(18)

(19)

(20)

Functional Semantics of FL(X):2

$$\mathscr{M}[x] \ v \stackrel{df}{=} v(x)$$
 $\mathscr{M}[T] \ v \stackrel{df}{=} true$ 
 $\mathscr{M}[F] \ v \stackrel{df}{=} false$ 

$$\mathcal{M}[\mathbf{Z}] \ v \stackrel{df}{=} 0$$

$$\mathcal{M}[(\mathbf{P} \ t)] \ v \stackrel{df}{=} \mathcal{M}[t] \ v - 1$$

$$\mathcal{M}[(\mathbf{S} \ t)] \ v \stackrel{df}{=} \mathcal{M}[t] \ v + 1$$

$$\mathcal{M}[(\mathbf{IZ} \ t)] \ v \stackrel{df}{=} \mathcal{M}[t] \ v = 0$$

$$\mathcal{M}[(\mathsf{GTZ}\ t)]\ v \stackrel{df}{=} \mathcal{M}[t]\ v > 0$$
 
$$\mathcal{M}[(\mathsf{ITE}\ \langle t, t_1, t_0 \rangle)]\ v \stackrel{df}{=} \left\{ \begin{array}{l} \mathcal{M}[t_1]\ v \ \text{if} \ \mathcal{M}[t]\ v \\ \mathcal{M}[t_0]\ v \ \text{if not} \ \mathcal{M}[t]\ v \end{array} \right.$$

$$(24)$$

$$(25)$$





#### 15.4. Equational Reasoning in FL(X)

From the semantics of FL(X) the following identitities are easily derived. We leave the proofs of these identities to the reader. It is also important that some of these identities are used (oriented from left to right) in the definition of the  $\delta$ -rules as rules of reduction (or "simplification") in order to obtain normal forms. In such cases the equality is made asymmetric (left to right).

#### Identities used for simplification

$$(P (S x)) = x$$
 (26)  
 $(S (P x)) = x$  (27)  
 $(ITE (T, x, y)) = x$  (28)  
 $(ITE (F, x, y)) = y$  (29)  
 $(IZ Z) = T$  (30)  
 $(GTZ Z) = F$ 

#### Identities involving normal forms

$$(IZ (S n)) = F, \text{ where } (S n) \text{ is a } \delta \text{-nf}$$

$$(IZ (P n)) = F, \text{ where } (P n) \text{ is a } \delta \text{-nf}$$

$$(GTZ (S n)) = T, \text{ where } (S n) \text{ is a } \delta \text{-nf}$$

$$(GTZ (P n)) = F, \text{ where } (P n) \text{ is a } \delta \text{-nf}$$

$$(32)$$

$$(33)$$

$$(GTZ (P n)) = F, \text{ where } (P n) \text{ is a } \delta \text{-nf}$$

$$(34)$$

Besides the above identities which are actually used in an oriented form for the purpose of computation we may also prove other identities from the functional semantics. Many of these look like they could be included in the rules for computation, but we may not be because of

- the limits of computability in general and
- their inclusion might at times lead to non-determinism and
- in more extreme cases lead to non-termination even though there are deterministic ways to obtain  $\delta$ -normal forms.

**⋖**◆

4

**•** 

**>>** 

PL December 28, 2021

Go Back

Full Screen

Close



However they are useful for reasoning about programs written in the language. For example the following obvious identity

(ITE 
$$\langle b, x, x \rangle$$
) = x, where b is a boolean (36)

(37)

is useful for simplifying a program for human reasoning. However, when included as a  $\delta$ -rule, it greatly complicates the computation when equality of the two arms of the conditional need to be checked (when they are not merely variables but complicated expressions themselves). There is a further complication of defining under what conditions this equality checking needs to be performed.







#### Reduction Semantics

Just as the dynamic behaviour of a  $\lambda$ -term may be described by  $\beta$ -reduction, we may describe the dynamic behaviour of a FL(X) expression through a notion of reduction called  $\delta - reduction$ . It is important that such a notion of reduction produces results (values) that are consistent with the functional semantics. We first begin with the  $\delta$ -normal forms for integers and booleans.

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 616 OF 721



## The Normal forms for Integers

 ${\bf Zero}$ . Z is the unique representation of the number 0 and every integer expression that is equal to 0 must be reducible to Z.

**Positive integers**. Each positive integer k is uniquely represented by the expression  $(S^k \ Z)$  where the super-script k denotes a k-fold application of S.

Negative integers. Each negative integer -k is uniquely represented by the expression  $(P^k \ Z)$  where the super-script k denotes a k-fold application of P.

44

PL December 28. 2021

Go Back

Full Screen

Close



# $\delta$ rules for Integers

$$(P (S x)) \longrightarrow_{\delta} x \tag{38}$$

$$(S (P x)) \longrightarrow_{\delta} x \tag{39}$$

44

**•** 

**▶**▶

Go Back

Full Screen

Close



619 of 721

## $\delta$ Normal Forms for Integers

**Lemma 15.2** The following well-typed (in any type environment  $\Gamma$ ) terms are exactly the  $\delta$ -normal forms in  $WT_{\Omega}$  along with their respective meanings in the functional semantics (in any dynamic environment  $v \in V$ ).

- 1.  $\Gamma \vdash Z : \underline{int} \text{ and } \mathscr{M}[Z] \ v = 0$
- 2.  $\Gamma \vdash T : \underline{bool} \text{ and } \mathscr{M}[T] \ v = true$
- 3.  $\Gamma \vdash F : \underline{bool} \text{ and } \mathscr{M}[F] \ v = false$
- 4. For each positive integer k,  $\Gamma \vdash (S^k \ Z) : \underline{int} \ and \ \mathcal{M}[(S^k \ Z)] \ v = k$
- 5. For each positive integer k,  $\Gamma \vdash (P^k \ Z) : \underline{\mathsf{int}} \text{ and } \mathscr{M}[(P^k \ Z)] \ v = -k$

→ PL December 28, 2021 Go BACK FULL SCREEN



#### $\delta$ Rules for Conditional

Pure Boolean Reductions. The constructs T and F are the normal forms for boolean values.

$$(\mathsf{ITE}\ \langle \mathtt{T}, x, y \rangle) \longrightarrow_{\delta} x \tag{40}$$
 
$$(\mathsf{ITE}\ \langle \mathtt{F}, x, y \rangle) \longrightarrow_{\delta} y \tag{41}$$

44

▶▶

Go Back

Full Screen

Close



#### $\delta$ Rules: Zero Test

#### Testing for zero.

44

**>>** 

PL December 28, 2021

Go Back

Full Screen

Close



# $\delta$ Rules: Positivity

- 4-

▶▶

PL December 28, 2021

Go Back

Full Screen

Close



#### Exercise 15.1

- 1. Find examples of expressions in FLO which have more than one computation.
- 2. Prove that  $\longrightarrow_{\delta}$  is terminating.
- 3. Prove that  $\longrightarrow_{\delta}$  is Church-Rosser.
- 4. The language FL(X) extends FL0 with variables. What are the new  $\delta$ -normal forms in FL(X)?
- 5. Subject reduction. Prove that for any well-typed term  $t \in WT_{\Omega}(X)$ , and  $\alpha \in \{\underline{\mathtt{int}}, \underline{\mathtt{bool}}\}\ if \Gamma \vdash t : \alpha \ and \ t \longrightarrow_{\delta} t' \ then \Gamma \vdash t' : \alpha$ .
- 6. Referential Transparency. Let  $t \in WT_{\Omega}(X)$ ,  $FV(t) = \{x_1, \dots, x_n\}$  and let v be a valuation environment. If  $\{t_1, \dots, t_n\}$  are ground terms such that for each i,  $1 \le i \le n$ ,  $\mathscr{M}[x_i]$   $v = \mathscr{M}[t_i]$  v then prove that

PL December 28, 2021

- (a)  $\mathcal{M}[t] \ v = \mathcal{M}[\{t_1/x_1, \dots, t_n/x_n\}t] \ v \ and$
- (b)  $\{t_1/x_1, \dots, t_n/x_n\}t \longrightarrow_{\delta}^* u \text{ where } \mathscr{M}[u] \ v = \mathscr{M}[t] \ v$

where  $\{t_1/x_1,\ldots,t_n/x_n\}$ t denotes the simultaneous syntactic substitution of every occurrence of variable  $x_i$  by the ground term  $t_i$  for  $1 \le i \le n$ .

44

Go Back

Full Screen

Close



# $\Lambda+FL(X)$ : The Power of Functions

To make the language powerful we require the ability to define functions, both non-recursive and recursive. We define an applied lambda-calculus of lambda terms  $\Lambda_{\Omega}(X)$  over this set of terms as follows:

$$L, M, N ::= t \in T_{\Omega}(X) \mid \lambda x[L] \mid (L M) \tag{48}$$

This is a two-level grammar combining the term grammar (16) with  $\lambda$ -abstraction and  $\lambda$ -application.

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 624 OF 721



## Some Non-recursive Operators

We may "program" the other boolean operations as follows:

NOT 
$$\stackrel{df}{=} \lambda x[\text{ITE } \langle x, F, T \rangle]$$
  
AND  $\stackrel{df}{=} \lambda \langle x, y \rangle[\text{ITE } \langle x, y, F \rangle]$   
OR  $\stackrel{df}{=} \lambda \langle x, y \rangle[\text{ITE } \langle x, T, y \rangle]$ 

We may also "program" the other integer comparison operations as follows:

$$\begin{aligned} &\mathsf{GEZ} \stackrel{df}{=} \lambda x [\mathsf{OR} \ \langle (\mathsf{IZ} \ x), (\mathsf{GTZ} \ x) \rangle] \\ &\mathsf{LTZ} \stackrel{df}{=} \lambda x [\mathsf{NOT} \ (\mathsf{GEZ} \ x)] \\ &\mathsf{LEZ} \stackrel{df}{=} \lambda x [\mathsf{OR} \ \langle (\mathsf{IZ} \ x), (\mathsf{LTZ} \ x) \rangle] \end{aligned}$$

PL December 28, 2021

44

**▶**▶

Go Back

Full Screen

Close



# $\Lambda$ +FL(X): Lack of Higher-order Power?

**Example 15.3** The grammar (48) does not allow us to define expressions such as the following:

- 1. the successor of the result of an application (S  $(L \ M)$ ) where  $(L \ M)$  yields an integer value.
- 2. higher order conditionals e.g.  $\lambda x[(\text{ITE }\langle (L\ x), (M\ x), (N\ x)\rangle)]$  where  $(L\ x)$  yields a boolean value for an argument of the appropriate type.
- 3. In general, it does not allow the constructors to be applied to  $\lambda$ -expressions.

So we extend the language by allowing a free intermixing of  $\lambda$ -terms and terms of the sub-language  $T_{\Omega}(X)$ .

→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 626 OF 721



# $\Lambda_{FL}(X)$ : Higher order functions

We need to *flatten* the grammar of (48) to allow  $\lambda$ -terms also to be used as arguments of the constructors of the term-grammar (16). The language of applied  $\lambda$ -terms (viz.  $\Lambda_{\Omega}(X)$ ) now is defined by the grammar.

PL December 28, 2021

$$L, M, N ::= x \in X \quad | \quad \mathsf{Z} \quad | \quad \mathsf{F}$$

$$\mid (\mathsf{P} \ L) \quad | \quad (\mathsf{S} \ L) \quad | \quad (\mathsf{IZ} \ L) \quad | \quad (\mathsf{GTZ} \ L) \quad | \quad (\mathsf{ITE} \ \langle L, M, N \rangle) \quad | \quad \lambda x[L] \quad | \quad (L \ M) \quad | \quad (\mathsf{M})$$

44

**•** 

▶

Go Back

Full Screen

Close

Unfortunately the result of *flatten*ing the grammar leads to an even larger number of meaningless expressions (in particular, we may be able to generate self-referential ones or ones that may not even be interpretable as functions which yield integer or boolean values.

It is therefore imperative that we define a *type-checking* mechanism to rule out meaningless expressions. As mentioned before, type-checking is not context-free and hence cannot be done through mechanisms such as scanning and parsing and will have to be done separately before any code-generation takes place.

We will in fact, go a step further and design a *type-inferencing* mechanism that will prevent meaningless expressions from being allowed.

Further, given a well-typed expression we need to be able to define a meaning for each expression that is somehow compatible with our intuitive understanding of what  $\lambda$ -expressions involving integer and boolean operations mean. This meaning is defined through an *operational semantics* i.e. a system of transitions on how computation actually takes place for each expression. We define this through a reduction mechanism that is consistent with reduction relations that we have earlier studied for the untyped  $\lambda$ -calculus.

In order for it to be compatible with the notions of reduction in the  $\lambda$ -calculus we require to define a notion of reduction first for expressions that do not involve either  $\lambda$  abstraction or  $\lambda$  application. We refer to this notion of reduction as  $\delta$ -reduction. Furthermore we need to be able to define  $\delta$ -normal forms for these expressions. Since the language is completely symbolic, these normal forms would serve as the final answers obtained in the evaluation of these expressions.

PL December 28, 2021

44

▶▶

Go Back

Full Screen

Close



#### Exercise 15.2

- 1. Prove that the language of (48) is properly contained in the language of (49).
- 2. Give examples of meaningful terms generated by the grammar (49) which cannot be generated by the grammar (48).









Full Screen

Close

#### Recursion in the Applied Lambda-calculus



The full power of a programming language will not be realised without a recursion mechanism. The untyped lambda-calculus has "paradoxical combinators" which behave like recursion operators upto  $=_{\beta}$ .

**Definition 15.4** A combinator Y is called a fixed-point combinator if for every lambda term L, Y satisfies the fixed-point property

$$(Y L) =_{\beta} (L (Y L)) \tag{50}$$

Curry's Y combinator  $(Y_C)$ 

$$Y_C \stackrel{df}{=} \lambda f[(C \ C)]$$
 where  $C \stackrel{df}{=} \lambda x[(f \ (x \ x))]$ 

Turing's Y combinator  $(Y_T)$ 

$$Y_T \stackrel{df}{=} (T \ T) \text{ where } T \stackrel{df}{=} \lambda y \ x[(x \ (y \ y \ x \ ))]$$

PL December 28, 20

Go Back

Full Screen

Close

# d\$E

#### The Paradoxical Combinators

#### **Lemma 15.5** Both $Y_C$ and $Y_T$ satisfy the fixed-point property.

Proof: For each term L we have

$$\begin{array}{l} (\mathsf{Y_c} \ L) \\ \equiv \ (\lambda f[(\mathsf{C} \ \mathsf{C})] \ L) \\ \to_\beta^1 \ (\{L/f\}\mathsf{C} \ \{L/f\}\mathsf{C}) \\ \equiv \ (\lambda x[(L \ (x \ x))] \ \lambda x[(L \ (x \ x))]) \\ =_\beta \ (L \ (\mathsf{Y_c} \ \mathsf{L})) \end{array}$$

Similarly for  $Y_T$  it may be verified that it satisfies the fixed-point property.

$$\begin{array}{c} (Y_T \ L) \\ \equiv \ ((\mathsf{T} \ \mathsf{T}) \ L) \\ \equiv \ ((\lambda y [\lambda x [(x \ ((y \ y) \ x))]] \ \mathsf{T}) \ L) \\ \rightarrow^2_\beta \ (L \ ((\mathsf{T} \ \mathsf{T}) \ L)) \\ =_\beta \ (L \ (\mathsf{Y}_\mathsf{T} \ L)) \end{array}$$

QED

**◄**◀

**>>** 

Go Back

Full Screen

Close



# $\Lambda_{RecFL}(X)$ : Recursion

- But the various Y combinators unfortunately will not satisfy any typing rules that we may define for the language, because they are all "self-applicative" in nature.
- Instead it is more convenient to use the fixed-point property and define a new constructor with a  $\delta$ -rule which satisfies the fixed-point property (definition 50).
- REC is assigned the type  $((\tau \rightarrow \tau) \rightarrow \tau)$  for each type  $\tau$ .

44

**▶**▶

Go Back

Full Screen

Close



$$\Lambda_{RecFL}(X)$$
: Adding Recursion

We extend the language  $\Lambda_{FL}(X)$  with a new constructor

$$L ::= \dots \mid (\mathtt{REC} \ L)$$

and add the fixed point property as a  $\delta$ -rule

$$(\text{REC} \ L) \longrightarrow_{\delta} (L \ (\text{REC} \ L)) \tag{51}$$

 ✓
 ✓

 FULL SCREEN

 CLOSE

 633 OF 721



#### Typing REC

With REC :  $((\tau \rightarrow \tau) \rightarrow \tau)$  and  $L : \tau \rightarrow \tau$  we have that

$$\begin{array}{ccc} (\operatorname{REC} \ L) & : \ \tau \\ (L \ (\operatorname{REC} \ L)) : \ \tau \end{array}$$

#### which

- type-checks (without recourse to self-reference) as a constructor and
- is consistent with our intuition about recursion as a syntactic unfolding operator.

44

▶▶

Go Back

Full Screen

Close



## Recursion Example: Addition

Consider addition on integers as a binary operation to be defined in this language. We use the following properties of addition on the integers to define it by induction on the first argument.

#### Example 15.6

$$x + y = \begin{cases} y & \text{if } x = 0\\ (x - 1) + (y + 1) & \text{if } x > 0\\ (x + 1) + (y - 1) & \text{if } x < 0 \end{cases}$$
 (52)

44



DI D

Go Васк

Full Screen

Close

Using the constructors of  $\Lambda_{RecFL}(X)$  we require that any (curried) definition of addition on numbers should be a solution to the following equation in  $\Lambda_{RecFL}(X)$  for all (integer) expression values of x and y.

$$(plusc \ x \ y) =_{\beta\delta} \text{ITE } \langle (\text{IZ } x), y, \text{ITE } \langle (\text{GTZ } x), (plusc \ (\text{P } x) \ (\text{S } y)), (plusc \ (\text{S } x) \ (\text{P } y)) \rangle \rangle$$
 (53)

Equation (53) may be rewritten using abstraction as follows:

$$plusc =_{\beta\delta} \lambda x [\lambda y [\text{ITE } \langle (\text{IZ } x), y, \text{ITE } \langle (\text{GTZ } x), (plusc \ (\text{P } x) \ (\text{S } y)), (plusc \ (\text{S } x) \ (\text{P } y)) \rangle \rangle]]$$
 (54)

We may think of equation (54) as an equation to be solved in the unknown variable plusc.

Consider the (applied)  $\lambda$ -term obtained from the right-hand-side of equation (54) by simply abstracting the unknown plusc.

$$\mathsf{addc} \stackrel{\mathit{df}}{=} \lambda f[\lambda x \ y[\mathsf{ITE} \ \langle (\mathsf{IZ} \ x), y, \mathsf{ITE} \ \langle (\mathsf{GTZ} \ x), (f \ (\mathsf{P} \ x) \ (\mathsf{S} \ y)), (f \ (\mathsf{S} \ x) \ (\mathsf{P} \ y)) \rangle \rangle]] \tag{55}$$

#### Claim 15.7

$$(REC \ addc) \longrightarrow_{\delta} (addc \ (REC \ addc)) \tag{56}$$

and hence

$$(REC \ addc) =_{\beta\delta} (addc \ (REC \ addc)) \tag{57}$$

44

**>** 

PL Doce

Go Back

Full Screen

Close



Claim 15.8 (REC addc) satisfies exactly the equation (54). That is

$$((REC addc) x y) =_{\beta \delta} ITE \langle (IZ x), y, ITE \langle (GTZ x), ((REC addc) (P x) (S y)), ((REC addc) (S x) (P y)) \rangle \rangle$$
(58)

Hence we may regard (REC addc) where addc is defined by the right-hand-side of definition (55) as the required solution to the equation (53) in which plusc is an unknown.

The abstraction shown in (55) and the claims (15.7) and (15.8) simply go to show that  $M \equiv_{\alpha} \lambda f[\{f/z\}L]$  is a solution to the equation  $z =_{\beta\delta} L$ , whenever such a solution does exist. Further, the claims also show that we may "unfold" the recursion (on demand) by simply performing the substitution  $\{L/z\}L$  for each free occurrence of z within L.

44

**•** 

▶▶

Go Back

Full Screen

Close



#### Exercise 15.3

- 1. Prove that the relation  $\longrightarrow_{\delta}$  is confluent.
- 2. The language FL does not have any operators that take boolean arguments and yields integer values. Define a standard conversion function B2I which maps the value F to Z and T to (S Z).
- 3. Using the combinator add and the other constructs of  $\Lambda_{\Sigma}(X)$  to
  - (a) define the equation for products of numbers in the language.
  - (b) define the multiplication operation mult on integers and prove that it satisfies the equation(s) for products.
- 4. The equation (52) is defined conditionally. However the following is equally valid for all integer values x and y.

$$x + y = (x - 1) + (y + 1) \tag{59}$$

- (a) Follow the steps used in the construction of addc to define a new applied addc' that instead uses equation (59).
- (b) Is (REC addc') =  $_{\beta\delta}$  (addc' (REC addc'))?
- (c) Is addc  $=_{\beta\delta}$  addc'?
- (d) Is (REC addc) =  $_{\beta\delta}$  (REC addc')?
- (e) Computationally speaking (in terms of  $\beta$  and  $\delta$  reductions), what is the difference between addc and addc'?
- 5. The function addc was defined in curried form. Use the pairing function in the untyped  $\lambda$ -calculus, to define
  - (a) addition and multiplication as binary functions independently of the existing functions.
  - (b) the binary 'curry' function which takes a binary function and its arguments and creates a curried version of the binary function.

44

▶▶

Go Back

Full Screen

Close

638 of 721

PL December 28, 2021 GO BACK FUL



 ✓
 Image: Second of the propertion of the properties of the



15.5. Type-checking FL

15.6.  $\Lambda_{RecFL(X)}$  with type rules

# Typing $\Lambda_{RecFL(X)}$ expressions

We have already seen that the simple language FL has

- two kinds of expressions: integer expressions and boolean expressions,
- there are also constructors which take integer expressions as arguments and yield boolean values
- there are also function types which allow various kinds of functions to be defined on boolean expressions and integer expressions.

**∢**∢

•



PL December 28, 2021



# The Need for typing in $\Lambda_{RecFL}(X)$

- A type is an important *attribute* of any variable, constant or expression, since every such object can only be used in certain kinds of expressions.
- ullet Besides the need for type-checking rules on  $T_\Omega(X)$  to prevent illegal constructor operations,
  - rules are necessary to ensure that  $\lambda$ -applications occur only between terms of appropriate types in order to remain meaningful.
  - rules are necessary to ensure that all terms have clearly defined types at compile-time so that there are no run-time type violations.

**◄**◀

1

▶▶

PL December 28, 2021

Go Back

Full Screen

Close

641 of 721

# TL: A Language of Simple Types



Consider the following language of types (in fully parenthesized form) defined over an infinite collection  $'a \in TV$  of type variables, disjoint from the set of variables. We also have two type constants <u>int</u> and <u>bool</u>.

$$\sigma, \tau ::= \underline{\mathsf{int}} \mid \underline{\mathsf{bool}} \mid '\mathsf{a} \in TV \mid (\sigma {*} \tau) \mid (\sigma {\to} \tau)$$

#### Notes.

- <u>int</u> and <u>bool</u> are *type constants*.
- ullet In any type expression au, TVar( au) is the set of type variables
- \* is the product operation on types and
- ullet  $\to$  is the function operator on types.
- We require \* because of the possibility of defining functions of various kinds of arities in  $\Lambda_{\Omega}(X)$ .

44

4

**>>** 

PL December 28, 2021

Go Back

Full Screen

Close



## TL: Precedence and Associativity

PL December 28, 2021

- Precedence. We assume \* has a higher precedence than  $\rightarrow$ .
- Associativity.
  - − \* is left associative whereas
  - $-\rightarrow$  is right associative

44





## Type-inference Rules: Infrastructure

The question of assigning types to complicated expressions which may have variables in them still remains to be addressed.

Type inferencing. Can be done using type assignment rules, by a recursive travel of the abstract syntax tree.

Free variables (names) are already present in the *environment* (symbol table).

Constants and Constructors. May have their types either pre-defined or there may be axioms assigning them types.

Bound variables. May be necessary to introduce "fresh" type variables in the environment.

44







## Type Inferencing: Infrastructure

The elementary typing defined previously ( $\S15.3.1$ ) for the elementary expressions of FL does not suffice

- 1. in the presence of  $\lambda$  abstraction and application, which allow for higher-order functions to be defined
- 2. in the presence of polymorphism, especially when we do not want to unnecessarily decorate expressions with their types.

PL December 28, 2021

**∢**∢





Go Back



Close

#### Type Assignment: Infrastructure



- Assume  $\Gamma$  is the environment<sup>a</sup> (an association list) which may be looked up to determine the types of individual names. For each variable  $x \in X$ ,  $\Gamma(x)$  yields the type of x i.e.  $\Gamma(x) = \sigma$  if  $x : \sigma \in \Gamma$ .
- For each (sub-)expression in FL we define a set C of type constraints of the form  $\sigma = \tau$ , where T is the set of type variables used in C.
- The type constraints are defined by induction on the structure of the expressions in the language FL.
- The expressions of FL could have free variables. The type of the expression would then depend on the types assigned to the free variables. This is a simple kind of polymorphism.
- It may be necessary to generate new type variables as and when required during the process of inferencing and assignment.

<sup>a</sup>usually a part of the symbol table

 ◆◆

 PLI

PL December 28, 2021

Full Screen

Close



## Constraint Typing Relation

**Definition 15.9** For each term  $L \in \Lambda_{\Sigma}(X)$  the constraint typing relation is of the form

$$\Gamma \vdash L : \tau \rhd_T C$$

#### where

- $\Gamma$  is called the context<sup>a</sup> and defines the stack of assumptions<sup>b</sup> that may be needed to assign a type (expression) to the (sub-)expression L.
- ullet au is the type(-expression) assigned to L
- C is the set of constraints
- ullet T is the set of "fresh" type variables used in the (sub-)derivations

<sup>a</sup>usually in the symbol table

44

**•** 



DI D. I CO

Go Back

Full Screen

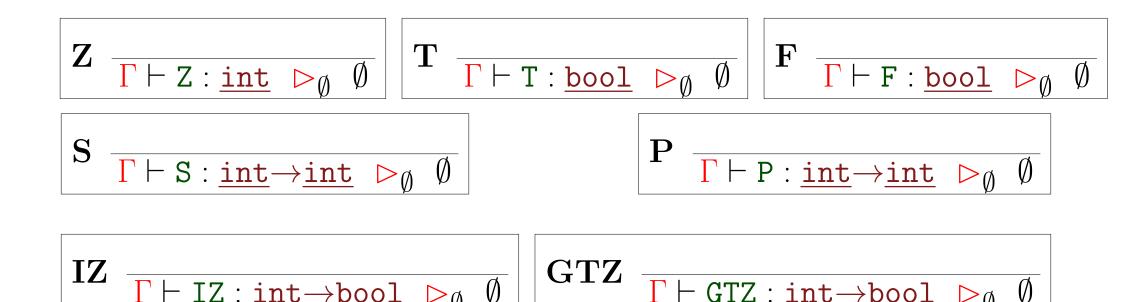
Close

<sup>&</sup>lt;sup>b</sup>including new type variables



## Typing axioms: Basic 1

The following axioms (c.f Typing FL Expressions) may be either predefined or applied during the scanning and parsing phases of the compiler to assign types to the individual tokens and thus create an *initial* type environment  $\Gamma_0$ .



→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 648 OF 721



# Typing axioms: Basic 2

$$\overline{\Gamma \vdash \mathtt{ITE} : \underline{\mathtt{bool}} * \underline{\mathtt{int}} * \underline{\mathtt{int}} \to \underline{\mathtt{int}} \; \triangleright_{\emptyset} \; \emptyset}$$

ITEB 
$$\frac{}{\Gamma \vdash \text{ITE} : \underline{\text{bool}} * \underline{\text{bool}} + \underline{\text{bool}} \rightarrow \underline{\text{bool}} \triangleright_{\emptyset} \emptyset}$$

Notice that the constructor ITE is *overloaded* and actually is two constructors ITEI and ITEB. Which constructor is actually used will depend on the context and the type-inferencing mechanism.

 →
 Image: Second of the properties of the



## Type Rules: Variables and Abstraction

$$\mathbf{Var} \ \ \overline{\Gamma \vdash x : \Gamma(x) \ \triangleright_{\emptyset} \ \emptyset}$$

Abs 
$$\frac{\Gamma, x : \sigma \vdash L : \tau \triangleright_T C}{\Gamma \vdash \lambda x[L] : \sigma \rightarrow \tau \triangleright_T C}$$

PL December 28, 2021

44

**▶** 

Go Back

FULL SCREEN



### Type Rules: Application

#### where

- Condition 1.  $T_1 \cap T_2 = T_1 \cap TVar(\tau) = T_2 \cap TVar(\sigma) = \emptyset$ Condition 2. 'a  $\notin T_1 \cup T_2 \cup TVar(\sigma) \cup TVar(\tau) \cup TVar(C_1) \cup TVar(C_2)$ .
- $\bullet T' = T_1 \cup T_2 \cup \{'a\}$
- $C' = C_1 \cup C_2 \cup \{\sigma = \tau \rightarrow '\mathbf{a}\}\$

44

PL December 28, 2021

Go Back

Full Screen

Close



**Example 15.10** Consider the following simple combinator  $\lambda x[\lambda y[\lambda z[(x \ (y \ z))]]]$  which defines the function composition operator. Since there are three bound variables x, y and z we begin with an initial assumption  $\Gamma = x : 'a, y : 'b, z : 'c$  which assign arbitrary types to the bound variables, represented by the type variables 'a, 'b and 'c respectively. Note however, that since it has no free variables, its type does not depend on the types of any variables. We expect that at the end of the proof there would be no assumptions. Our inference for the type of the combinator then proceeds as follows.

1. 
$$x:'a, y:'b, z:'c \vdash x:'a \triangleright_{\emptyset} \emptyset$$
 (Var)

2. 
$$x:'a, y:'b, z:'c \vdash y:'b \triangleright_{\emptyset} \emptyset$$
 (Var)

3. 
$$x:'a, y:'b, z:'c \vdash z:'c \triangleright_{\emptyset} \emptyset$$
 (Var)

4. 
$$x : 'a, y : 'b, z : 'c \vdash (y z) : 'd \triangleright_{\{'d\}} \{'b = 'c \rightarrow 'd\}$$
 (App)

5. 
$$x : 'a, y : 'b, z : 'c \vdash (x (y z)) : 'e \rhd_{\{'d,'e\}} \{'b = 'c \rightarrow 'd, 'a = 'd \rightarrow 'e\}$$
 (App)

$$6. \ x : '\mathbf{a}, y : '\mathbf{b} \vdash \lambda z[(x \ (y \ z))] : '\mathbf{c} \rightarrow '\mathbf{e} \ \triangleright_{\{'\mathbf{d},'\mathbf{e}\}} \ \{'\mathbf{b} = '\mathbf{c} \rightarrow '\mathbf{d}, '\mathbf{a} = '\mathbf{d} \rightarrow '\mathbf{e}\}$$

$$(\mathbf{Abs})$$

$$7. \ x: \ '\mathbf{a} \vdash \lambda x[\lambda y[\lambda z[(x \ (y \ z))]]]: \ '\mathbf{b} \rightarrow '\mathbf{c} \rightarrow '\mathbf{e} \ \triangleright_{\{'\mathbf{d},'\mathbf{e}\}} \ \{'\mathbf{b} = '\mathbf{c} \rightarrow '\mathbf{d}, \ '\mathbf{a} = '\mathbf{d} \rightarrow '\mathbf{e}\}$$

$$8. \vdash \lambda x[\lambda y[\lambda z[(x \mid (y \mid z))]]] : '\mathtt{a} \to '\mathtt{b} \to '\mathtt{c} \to '\mathtt{e} \quad \triangleright_{\{'\mathtt{d},'\mathtt{e}\}} \quad \{'\mathtt{b} = '\mathtt{c} \to '\mathtt{d}, '\mathtt{a} = '\mathtt{d} \to '\mathtt{e}\}$$

Hence  $\lambda x[\lambda y[\lambda z[(x \ (y \ z))]]]: \ 'a \rightarrow 'b \rightarrow 'c \rightarrow 'e \ subject \ to \ the \ constraints \ given \ by \ \{'b = 'c \rightarrow 'd, 'a = 'd \rightarrow 'e\} \ which \ yields \ \lambda x[\lambda y[\lambda z[(x \ (y \ z))]]]: \ ('d \rightarrow 'e) \rightarrow ('c \rightarrow 'd) \rightarrow 'c \rightarrow 'e$ 

PL December 28, 2021

44

**▶** 

▶▶

Go Back

Full Screen

Close



# Principal Type Schemes

**Definition 15.11** A solution for  $\Gamma \vdash L : \tau \rhd_T C$  is a pair  $\langle S, \sigma \rangle$  where S is a substitution of type variables in  $\tau$  such that  $S(\tau) = \sigma$ .

- ullet The rules yield a *principal type scheme* for each well-typed applied  $\lambda$ -term.
- The term is *ill-typed* if there is no solution that satisfies the constraints.
- Any substitution of the type variables which satisfies the constraints C is an instance of the most general polymorphic type that may be assigned to the term.

44

▶▶

Go Back

FULL SCREEN

Close



#### Exercise 15.4

1. The language has several constructors which behave like functions. Derive the following rules for terms in  $T_{\Omega}(X)$  from the basic typing axioms and the rule App.

$$\mathbf{Sx} \quad \frac{\Gamma \vdash t : \tau \quad \triangleright_T \quad C}{\Gamma \vdash (\mathbf{S} \quad t) : \underline{\mathsf{int}} \quad \triangleright_T \quad C \cup \{\tau = \underline{\mathsf{int}}\}}$$

$$\mathbf{Px} \ \frac{\Gamma \vdash t : \tau \ \triangleright_T \ C}{\Gamma \vdash (\mathsf{P} \ t) : \underline{\mathsf{int}} \ \triangleright_T \ C \cup \{\tau = \underline{\mathsf{int}}\}}$$

$$\boxed{\mathbf{IZx} \ \frac{\Gamma \vdash t : \tau \ \triangleright_T \ C}{\Gamma \vdash (\mathtt{IZ} \ t) : \underline{\mathtt{bool}} \ \triangleright_T \ C \cup \{\tau = \underline{\mathtt{int}}\}}$$

$$\mathbf{GTZx} \ \frac{\Gamma \vdash t : \tau \ \triangleright_T \ C}{\Gamma \vdash (\mathtt{GTZ} \ t) : \underline{\mathtt{bool}} \ \triangleright_T \ C \cup \{\tau = \underline{\mathtt{int}}\}}$$

ITEx 
$$\begin{array}{c|c} \Gamma \vdash t : \sigma & \triangleright_T & C \\ \hline \Gamma \vdash t_1 : \tau & \triangleright_{T_1} & C_1 \\ \hline \Gamma \vdash t_0 : v & \triangleright_{T_0} & C_0 \\ \hline \hline \Gamma \vdash (\mathsf{ITE} \ \langle t, t_1, t_0 \rangle) : \tau & \triangleright_{T'} & C' \end{array} \ (T \cap T_1 = T_1 \cap T_0 = T_0 \cap T = \emptyset)$$

where 
$$T' = T \cup T_1 \cup T_0$$
 and  $C' = C \cup C_1 \cup C_0 \cup \{\sigma = \underline{bool}, \tau = v\}$ 

- 2. Use the rules to define the type of the combinators K and S?
- 3. How would you define a type assignment for the recursive function addc defined by equation (55).
- 4. Prove that the terms,  $\omega = \lambda x[(x \ x)]$  and  $\Omega = (\omega \ \omega)$  are ill-typed.
- 5. Are the following well-typed or ill-typed? Prove your answer.

44

►I

Go Back

Full Screen

Close



- (a) (K S)
- (b) ((K S)  $\omega$ )
- (c) (((S K) K)  $\omega$ )
- $(d) \ (\mathtt{ITE} \ \langle (\mathtt{IZ} \ x), \mathtt{T}, (\mathsf{K} \ \mathsf{x}) \rangle)$









Full Screen

Close



 ✓
 ✓

 PL December 28, 2021

 Go Back
 Full Screen

 Close
 656 of 721



#### An Imperative Language

An Imperative Language











Go Back

Full Screen

Close

# The Concept of State



- Any imperative language indirectly exposes the *memory* (or *store*) to the user for manipulation.
- ullet Memory (or store) is a set Loc of locations used to store the values of variables.
- We define the store to be a (partial) function from Loc to values.  $\sigma: Loc \rightharpoonup (\underline{\mathtt{int}} \cup \underline{\mathtt{bool}})$ .  $Stores = \{\sigma \mid \sigma: Loc \rightharpoonup (\underline{\mathtt{int}} \cup \underline{\mathtt{bool}})\}$  is the set of possible stores.
- Each variable in an imperative program is assigned a location.
- The *environment* is an association  $\gamma$  of variable (names) to locations i.e.  $\gamma: X \to Loc$ .
- The *(dynamic)* state of a program is defined by the pair  $(\gamma, \sigma)$ .

44

4

▶▶

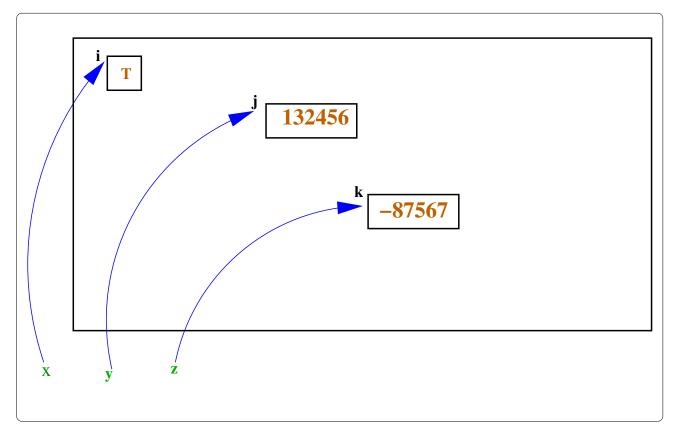
Go Back

Full Screen

Close

#### State: Illustration





- l-values.  $\gamma(x) = \mathbf{i} : \underline{\mathtt{bool}}, \ \gamma(y) = \mathbf{j} : \underline{\mathtt{int}}, \ \gamma(z) = \mathbf{k} : \underline{\mathtt{int}}$
- r-values.  $\sigma(\mathbf{i}) = T$ : bool,  $\sigma(\mathbf{j}) = 132456$ : int,  $\sigma(\mathbf{k}) = -87567$ : int

44

◀

**>>** 

PL December 28, 2021

Go Back

Full Screen

Close



## References in Languages

## ML-like impure functional languages

- have an explicit polymorphic a ref type constructor. Hence x:

  bool ref, y, z: int ref and x is a named reference to the location  $\mathbf{i}$
- have an explicit unary dereferencing operator ! to read the value contained in the location referenced by x, i.e.  $!x = \sigma(\mathbf{i})$ .
- The actual locations however are not directly visible.
- C-like imperative languages are not as fussy as the ML-like languages. C (and C++) even treats locations only as integers and allows integer operations to be preformed on them!

**∢**∢

**▶** 

▶▶

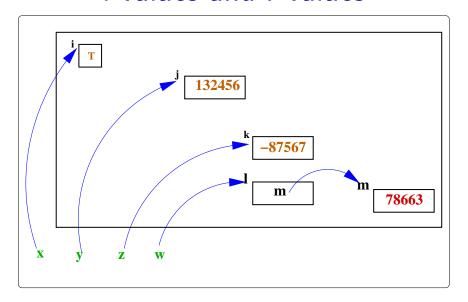
Go Back

Full Screen

Close



#### I-values and r-values



- 1 is the I-value of w i.e  $\gamma(w) = 1 \in Loc$
- ullet m is the r-value of w i.e.  $\sigma(\gamma(w)) = !w = \mathbf{m} \in Loc$
- $\bullet$  m is also an l-value since  $!w: \underline{\mathtt{int}} \ \underline{\mathtt{ref}}$
- $!(!w) = 78663 : \underline{int}$  is the r-value of !w







#### 16.0.1. l-values, r-values, aliasing and indirect addressing

The terms "l-value" (for "left-value") and "r-value" (for "right-value") come from the practice in most imperative languages of writing assignment commands by overloading the variable name to denote both its address  $(\gamma(x))$  in Loc as well as the value  $\sigma(\gamma(x))$  stored in memory. Consider the example,

- x := x + y (Pascal)
- x = x + y (C, C++, Java, Python, Perl)

The occurrence of "x" on the left-hand side of the assignment command denotes a location  $\gamma(x)$  whereas the occurrences of "x" and "y" on the right-hand-side of the assignment denote the values  $\sigma(\gamma(x))$  and  $\sigma(\gamma(y))$  respectively. The term "dereferencing" is used to denote the action of "reading" the value stored in a location.

- This notation for assignment becomes a source of tremendous confusion when locations are also valid values, as in the case of indirect addressing (look at w) and may be manipulated.
- The confusion is further exacerbated when locations are also integers indistinguishable from the integers stored in the locations. The result of dereferencing an integer variable may be one of the following.
  - An invalid location leading to a segmentation fault. For instance, the integer could be negative or larger than any valid memory address.

**⋖**∢

4



PL December 28, 2021

Go Back

FULL SCREEN

Close

- Another valid location with an undefined value or with a value defined previously when the location was assigned to some other variable in a different job. This could lead to puzzling results in the current program.
- Another valid location which is already the address of a variable in the program (leading to an aliasing totally unintended by the programmer). This could also lead to puzzling results in the current program.

Modern impure functional languages (which have strong-typing facilties) usually clearly distinguish between locations and values as different types. Hence every imperative variable represents only an l-value. Its r-value is obtained by applying a dereferencing operation (the prefix operation!). Hence the same assignment command in ML-like languages would be written

```
• x := !x + !y (ML and OCaml)
```

The following interactive ML session illustrates aliasing and the effect on the aliased variables.

```
Standard ML of New Jersey v110.76 [built: Tue Oct 22 14:04:11 2013]
- val u = ref 1;
val u = ref 1 : int ref
- val v = u; (* u and v are aliases for the same location *)
val v = ref 1 : int ref
- v := !v+1;
val it = () : unit
- !u;
```



```
val it = 2 : int
- !v;
val it = 2 : int
- v := !v+1;
val it = () : unit
- !u;.val it = 3 : int
- !v;
val it = 3 : int
-
```

The following ML-session illustrates indirect addressing (and if you get confused, don't come to me, I am confused too; confusion is the price we pay for indiscriminate modification of state).

PL December 28, 2021

```
Standard ML of New Jersey v110.76 [built: Tue Oct 22 14:04:11 2013]
- val x = ref (ref 0);
val x = ref (ref 0) : int ref ref
- val y = !x;
val y = ref 0 : int ref
- val z = ref y;
val z = ref (ref 0) : int ref ref
- y := !y+1;
val it = () : unit
```



```
- !y;
val it = 1 : int
- !z;
val it = ref 1 : int ref
- !(!z);
val it = 1 : int
- !(!x);
val it = 1 : int
```





# Operational Semantics of Expressions

- Consider the language of terms defined FLO. Instead of the  $\delta$ -rules defined earlier, we assume that these terms are evaluated on a hardware which can represent int and bool.
- Assume int is the hardware representation of the integers and bool =  $\{T, F\}$ .
- We assume that every (sub-)expression in the language has been typed with a unique type attribute.
- $\bullet$  We define an expression evaluation relation  $\longrightarrow_e$  such that

$$\longrightarrow_e \subseteq (Stores \times T_{\Omega}(X)) \times (Stores \times (T_{\Omega}(X) \cup int \cup bool))$$

in a given environment  $\gamma$ .

■■ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 666 OF 721





 ✓
 Image: Second of the propertion of the properties of the



#### 16.1. The Operational Semantics of Commands

WHILE: Big-Step Semantics

PL December 28, 2021









Go Back

Full Screen



## The WHILE language

- We initially define a simple language of *commands*.
- ullet The expressions of the language are those of any term algebra  $T_\Omega(X)$ .
- ullet We simply assume there is a well-defined relation  $\longrightarrow_e$  for evaluating expressions in
- We defer defining the relation  $\longrightarrow_e$  for FL0.

**⋖**◆

•

►I

**>** 

Go Back

Full Screen

Close



# State Changes or Side-Effects

- State changes are usually programmed by assignment commands which occur one location at a time.
- In the simple WHILE language side-effects do not occur except by *explicit* assignment commands.

44

**•** 

▶▶

PL December 28. 2021

Go Васк

FULL SCREEN

Close



### Modelling a Side-Effect

Given a store  $\sigma$ , a variable x such that  $\gamma(x) = \ell$  and  $\sigma(\ell) = a$ , the state change effected by the assignment x := b is a new store that is identical to  $\sigma$  except at the location  $\gamma(x)$  which now contains the value b

$$\sigma' = [\gamma(x) \mapsto b]\sigma$$

i.e.

$$\sigma'(\ell) = \begin{cases} \sigma(\ell) & \text{if } \ell \neq \gamma(x) \\ b & \text{otherwise} \end{cases}$$

PL December 28, 2021

**◄**◀



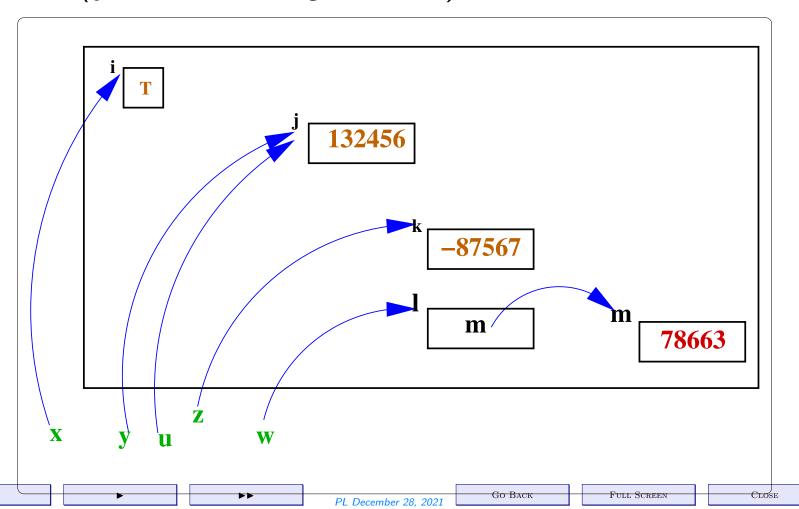
Go Back

Full Screen

Close

#### Aliases

**Definition 16.1** Two (or more) variables are called aliases if they denote the same location (y and u in the figure below).





## The Commands of the WHILE Language

$$c_0, c_1, c ::= \mathbf{skip}$$
 Skip  $x := e$  Assgn  $\{c_0\}$  Block  $\{c_0; c_1\}$  Seq if  $e$  then  $e$  do  $e$  od While

where e is either an integer or boolean expression in the language FL with operational semantics as given before.

PL December 28, 2021

44

**▶** 

▶▶

Go Back

Full Screen

Close

674 of 721

### Operational Semantics: Basic Commands



Skip 
$$\frac{}{\gamma \vdash \langle \sigma, \mathbf{skip} \rangle \longrightarrow_c^1 \sigma}$$

Assgn 
$$\frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_e m}{\gamma \vdash \langle \sigma, x := e \rangle \longrightarrow_c^1 [\gamma(x) \mapsto m] \sigma}$$

#### Notes:

- 1. The Skip rule corresponds to any of the following:
  - a noop
  - the identity function or identity relation on states
  - a command which has no effect on states
- 2. The assignment is the only command in our language which creates a sideeffect (actually changes state)

44







PL December 28, 2021



Full Screen

Close



## Operational Semantics: Blocks

We have defined a block as simply a command enclosed in braces. It is meant to delimit a (new) scope. Later we will see that there could be local declarations as well, in which case the semantics changes slightly to include a new scope

Block 
$$\frac{\gamma \vdash \langle \sigma, c \rangle \longrightarrow_c^1 \sigma'}{\gamma \vdash \langle \sigma, \{c\} \rangle \longrightarrow_c^1 \sigma'}$$

PL December 28, 2021

Go Back

Full Screen

CLOSE



# Operational Semantics: Sequencing

Notice that sequencing is precisely the composition of relations. If the relation  $\longrightarrow_c^1$  is a function (in the case of our language it actually is a function), sequencing would then be a composition of functions

PL December 28, 2021

Full Screen

CLOSE



#### Operational Semantics: Conditionals

$$\begin{array}{c}
\gamma \vdash \langle \sigma, e \rangle \longrightarrow_{e} \mathbf{F}, \\
\gamma \vdash \langle \sigma, c_{0} \rangle \longrightarrow_{c}^{1} \sigma_{0} \\
\hline
\gamma \vdash \langle \sigma, \mathbf{if} \ e \ \mathbf{then} \ c_{1} \ \mathbf{else} \ c_{0} \ \mathbf{fi} \rangle \longrightarrow_{c}^{1} \sigma_{0}
\end{array}$$

$$\begin{array}{c}
\gamma \vdash \langle \sigma, e \rangle \longrightarrow_{e} \mathsf{T}, \\
\gamma \vdash \langle \sigma, c_{1} \rangle \longrightarrow_{c}^{1} \sigma_{1} \\
\hline
\gamma \vdash \langle \sigma, \mathbf{if} \ e \ \mathbf{then} \ c_{1} \ \mathbf{else} \ c_{0} \ \mathbf{fi} \rangle \longrightarrow_{c}^{1} \sigma_{1}
\end{array}$$

#### Selective Evaluation.

Notice again the effect of selective evaluation in the operational semantics of the conditional and again in the semantics of the while loop.

44





PL December 28. 2021



FULL SCREEN

Close



### The While loop

• We use the fact that the while e do c od is really a form of recursion – actually it is a form of "tail recursion". Hence the execution behaviour of while e do c od is exactly that of

if 
$$e$$
 then  $\{c;$ while  $e$  do  $c$  od $\}$  else skip fi (60)

• The following rules may be derived from (60) using the rules for conditional, sequencing and skip (though the number of steps may not exactly correspond).

44

▶▶

Go Back

Full Screen

Close



#### Operational Semantics: While loop

While 0 
$$\frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_e \mathsf{F}}{\gamma \vdash \langle \sigma, \mathsf{while} \ e \ \mathsf{do} \ c \ \mathsf{od} \rangle \longrightarrow_c^1 \ \sigma}$$

While1 
$$\begin{array}{c}
\gamma \vdash \langle \sigma, e \rangle \longrightarrow_{e} \mathsf{T}, \\
\gamma \vdash \langle \sigma, c \rangle \longrightarrow_{c}^{1} \sigma', \\
\gamma \vdash \langle \sigma', \mathsf{while} \ e \ \mathsf{do} \ c \ \mathsf{od} \rangle \longrightarrow_{c}^{1} \sigma'' \\
\hline
\gamma \vdash \langle \sigma, \mathsf{while} \ e \ \mathsf{do} \ c \ \mathsf{od} \rangle \longrightarrow_{c}^{1} \sigma''
\end{array}$$

44

**•** 

▶▶

PL December 28, 2021

Go Back

Full Screen

Close

The effect of side-effects. In the above operational rules we have assumed that expression-evaluation has no side-effects i.e. there are no changes to the state of the program during or as result of expression evaluation. However many programming languages allow side-effects to global or non-local variables during expression evaluation. This does not significantly change the semantics, though it does very significantly change our ability to reason about such programs. In the presence of side-effects during expression-evaluation the semantics of commands would also change as we show in the following modified semantics of commands. In particular we need to carry state information during expression evaluation.

Assgn0 
$$\frac{}{\gamma \vdash \langle \sigma, x := m \rangle \longrightarrow_c^1 [\gamma(x) \mapsto m] \sigma}$$

Assgn1 
$$\frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_e \langle \sigma', e' \rangle}{\gamma \vdash \langle \sigma, x := e \rangle \longrightarrow_c^1 \langle \sigma', x := e' \rangle}$$

$$\begin{array}{|c|c|c|c|c|}\hline & \gamma \vdash \langle \sigma, e \rangle & \longrightarrow_e & \langle \sigma', \mathbf{F} \rangle, \\ \hline \mathbf{Cond0} & & \gamma \vdash \langle \sigma', c_0 \rangle & \longrightarrow_c^1 & \sigma_0 \\ \hline & & \gamma \vdash \langle \sigma, \mathbf{if} \ e \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_0 \ \mathbf{fi} \rangle & \longrightarrow_c^1 & \sigma_0 \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|c|c|}\hline & \gamma \vdash \langle \sigma, e \rangle & \longrightarrow_e & \langle \sigma', \mathsf{T} \rangle, \\ \hline & \gamma \vdash \langle \sigma', c_1 \rangle & \longrightarrow_c^1 & \sigma_1 \\ \hline & \gamma \vdash \langle \sigma, \mathsf{if} \ e \ \mathsf{then} \ c_1 \ \mathsf{else} \ c_0 \ \mathsf{fi} \rangle & \longrightarrow_c^1 & \sigma_1 \\ \hline \end{array}$$

While 0 
$$\frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_e \langle \sigma', \mathsf{F} \rangle}{\gamma \vdash \langle \sigma, \mathsf{while} \ e \ \mathsf{do} \ c \ \mathsf{od} \rangle \longrightarrow_c^1 \ \sigma'}$$

While1 
$$\begin{array}{c} \gamma \vdash \langle \sigma, e \rangle & \longrightarrow_e \langle \sigma', \mathbf{T} \rangle, \\ \gamma \vdash \langle \sigma', c \rangle & \longrightarrow_c^1 \sigma'', \\ \frac{\gamma \vdash \langle \sigma'', \mathbf{while} \ e \ \mathbf{do} \ c \ \mathbf{od} \rangle & \longrightarrow_c^1 \sigma'''}{\gamma \vdash \langle \sigma, \mathbf{while} \ e \ \mathbf{do} \ c \ \mathbf{od} \rangle & \longrightarrow_c^1 \sigma'''} \end{array}$$

The rules of the other constructs viz. skip, blocks and sequencing, remain unchanged.

44

**>** 

**▶**▶

Go Back

Full Screen

Close





#### 16.2. The Semantics of Expressions in FL

# Operational Semantics for FL









Go Back

Full Screen

Close



# Evaluating FL on a machine

- ullet We previously treated FL as simply a data-type and gave  $\delta$ -rules.
- ullet Here we define a deterministic evaluation mechanism  $\longrightarrow_e$  on a more realistic hardware which supports integers and booleans
- The normal forms on this machine would have to be appropriate integer and boolean constants as represented in the machine.

44

▶▶

Go Back

Full Screen

Close



# Operational Semantics: Constants and Variables

Let  $\sigma \in \mathsf{States}$  be any state.

$$T \xrightarrow{\gamma \vdash \langle \sigma, \mathsf{T} \rangle \longrightarrow_e \langle \sigma, \mathsf{T} \rangle}$$

$$\mathbf{F} \xrightarrow{\gamma \vdash \langle \sigma, \mathsf{F} \rangle} \longrightarrow_{e} \langle \sigma, \mathsf{F} \rangle$$

$$\mathbf{Z} \ \ \overline{\gamma \vdash \langle \sigma, \mathsf{Z} \rangle \ \longrightarrow_e \ \langle \sigma, \textcolor{red}{0} \rangle}$$

Go Back

Full Screen

Close



# Operational Semantics: Integer-valued Expressions

$$\mathbf{P} \begin{array}{c|c} \gamma \vdash \langle \sigma, e \rangle & \longrightarrow_e & \langle \sigma, \mathbf{m} \rangle \\ \hline \gamma \vdash \langle \sigma, (\mathbf{P} \ e) \rangle & \longrightarrow_e & \langle \sigma, \mathbf{m} - \mathbf{1} \rangle \end{array} (e, \mathbf{m} : \underline{\mathbf{int}})$$

$$\mathbf{S} \ \frac{\gamma \vdash \langle \sigma, e \rangle \ \longrightarrow_e \ \langle \sigma, \mathbf{m} \rangle}{\gamma \vdash \langle \sigma, (\mathbf{S} \ e) \rangle \ \longrightarrow_e \ \langle \sigma, \mathbf{m} + \mathbf{1} \rangle} \ (e, \mathbf{m} : \underline{\mathtt{int}})$$

PL December 28, 2021

44

ı

▶▶

Go Back

Full Screen

Close



# Operational Semantics: Boolean-valued Expressions

IZ0 
$$\frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_{e} \langle \sigma, \mathbf{m} \rangle}{\gamma \vdash \langle \sigma, (\mathrm{IZ} \ e) \rangle \longrightarrow_{e} \langle \sigma, \mathbf{F} \rangle} (e, \mathbf{m} : \underline{\mathsf{int}}, \mathbf{m} <> \mathsf{0})$$

IZ1 
$$\frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_e \langle \sigma, \mathbf{0} \rangle}{\gamma \vdash \langle \sigma, (\mathsf{IZ} \ e) \rangle \longrightarrow_e \langle \sigma, \mathbf{T} \rangle} (e : \underline{\mathsf{int}})$$

$$\mathbf{GTZ0} \ \frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_{e} \langle \sigma, \mathbf{m} \rangle}{\gamma \vdash \langle \sigma, (\mathsf{GTZ} \ e) \rangle \longrightarrow_{e} \langle \sigma, \mathbf{F} \rangle} \ (e, \mathbf{m} : \underline{\mathsf{int}}, \mathbf{m} <= \mathbf{0})$$

GTZ1 
$$\frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_{e} \langle \sigma, \mathbf{m} \rangle}{\gamma \vdash \langle \sigma, (\text{GTZ } e) \rangle \longrightarrow_{e} \langle \sigma, \mathbf{T} \rangle} (e, \mathbf{m} : \underline{\text{int}}, \mathbf{m} > \mathbf{0})$$

44

►I

PL December 28, 2021

Go Back

Full Screen

Close



### Operational Semantics: Conditional Expressions

ITEI0 
$$\frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_e \langle \sigma, F \rangle}{\gamma \vdash \langle \sigma, (\text{ITE } \langle e, e_1, e_0 \rangle) \rangle \longrightarrow_e \langle \sigma, e_0 \rangle} (e_1, e_0 : \underline{\text{int}})$$

ITEI1 
$$\frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_e \langle \sigma, \mathsf{T} \rangle}{\gamma \vdash \langle \sigma, (\mathsf{ITE} \langle e, e_1, e_0 \rangle) \rangle \longrightarrow_e \langle \sigma, e_1 \rangle} (e_1, e_0 : \underline{\mathsf{int}})$$

ITEB0 
$$\frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_e \langle \sigma, F \rangle}{\gamma \vdash \langle \sigma, (\text{ITE } \langle e, e_1, e_0 \rangle) \rangle \longrightarrow_e \langle \sigma, e_0 \rangle} (e_1, e_0 : \underline{\text{bool}})$$

ITEB1 
$$\frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_e \langle \sigma, \mathsf{T} \rangle}{\gamma \vdash \langle \sigma, (\mathsf{ITE} \langle e, e_1, e_0 \rangle) \rangle \longrightarrow_e \langle \sigma, e_1 \rangle} (e_1, e_0 : \underline{\mathsf{bool}})$$

44



Go Back

Full Screen

Close







### Local Declarations

We introduce declarations through a new syntactic category Decls defined as follows:

$$d_1, d_2, d ::= \underline{\mathtt{int}} \ x \mid \underline{\mathtt{bool}} \ y \mid d_1; d_2$$
 $c ::= \cdots \mid \{d; c\}$ 

- Most languages insist on a "declaration before use" discipline,
- Declarations create "little new environments".
- Need to be careful about whether a variable is at all defined.
- Even if the I-value of a variable is defined, its r-value may not be defined. The rules for variables and assignments then need to be changed to the following.

44

**•** 

PL December 28, 2021

Go Back

Full Screen

Close



### Some changed rules

- ullet We use the symbol ot to denote the undefined.
- ullet We use  $z 
  eq \bot$  to denote that z is well-defined.

$$\mathbf{x'} \xrightarrow{\gamma \vdash \langle \sigma, x \rangle} \longrightarrow_e \langle \sigma, \sigma(\gamma(x)) \rangle (\sigma(\gamma(x)) \neq \bot)$$

**Assgn0'** 
$$\frac{}{\gamma \vdash \langle \sigma, x := m \rangle} \xrightarrow{}_{c} \frac{1}{[\gamma(x) \mapsto m]\sigma} (\gamma(x) \neq \bot)$$

Assgn1' 
$$\frac{\gamma \vdash \langle \sigma, e \rangle \longrightarrow_e \langle \sigma, e' \rangle}{\gamma \vdash \langle \sigma, x := e \rangle \longrightarrow_c^1 \langle \sigma, x := e' \rangle} (\gamma(x) \neq \bot)$$

**⋖**◆

**•** 

PL December 28, 2021

Go Back

Full Screen

Close



### Declarations: Little Environments

The effect of a declaration is to create a little environment which is pushed onto the existing environment. The transition relation

$$\longrightarrow_d \subseteq ((Env \times Stores \times Decls) \times (Env \times Stores))$$

$$\mathbf{int} - \mathbf{x} \ \ \underline{\gamma \vdash \langle \sigma, \mathbf{int} \ x \rangle \longrightarrow_d \langle [x \mapsto l], [l \mapsto \bot] \sigma \rangle} \ \ (l \notin Range(\gamma))$$

$$\mathbf{bool} - \mathbf{x} \ \ \overline{\gamma \vdash \langle \sigma, \mathbf{bool} \ x \rangle \longrightarrow_d \langle [x \mapsto l], [l \mapsto \bot] \sigma \rangle} \ \ (l \notin Range(\gamma))$$

44

◀

PL December 28, 2021

Go Back

Full Screen

Close



# Scope

- The scope of a name begins from its <u>definition</u> and ends where the corresponding scope ends
- Scopes end with definitions of functions
- Scopes end with the keyword end in any let ... in ...end or local in ...end
- Scopes are delimited by brackets "[...]" in (fully-bracketed)  $\lambda$ -abstractions.
- We simply use {} to delimit scope

Go Back

Full Screen

Close



# Scope Rules

- Scopes may be disjoint
- Scopes may be nested one completely within another
- A scope cannot span two disjoint scopes
- Two scopes <u>cannot</u> (partly) overlap

forward









Full Screen

Close

**Example 16.2** local. Consider the following example ML program which uses local declarations in the development of the algorithm to determine whether a positive integer is perfect.

```
local
  exception invalidArg;
  fun ifdivisor3 (n, k) =
    if n <= 0 orelse
       k \le 0 orelse
       n < k
    then raise invalidArg
    else if n \mod k = 0
    then k
    else 0;
  fun sum_div2 (n, 1, u) =
    if n <= 0 orelse
       1 \le 0 orelse
```



```
l \ge n orelse u \le 0 orelse
     u > n
  then raise invalidArg
  else if 1 > u
  then 0
  else ifdivisor3 (n, 1)
     + sum_div2 (n, l+1, u)
fun perfect n =
  if n \le 0
  then raise invalidArg
  else
    let
      val nby2 = n div 2
    in
```

in



```
n = sum_div2 (n, 1, nby2)
         \quad \text{end} \quad
end
```



PL December 28, 2021

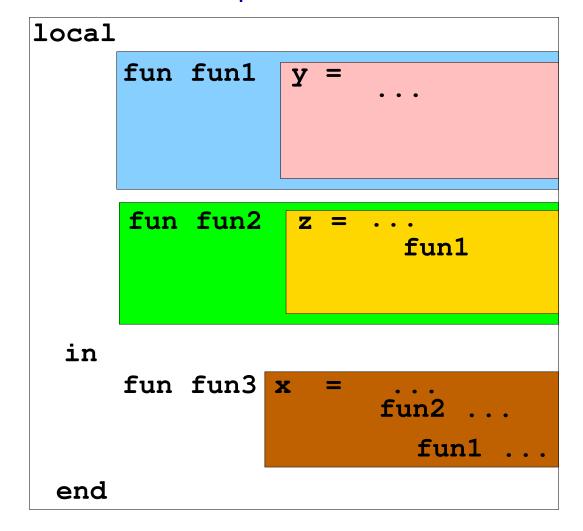
Go Васк

Full Screen

Close



# Scope & local



4.

▶

▶▶

PL December 28, 2021

Go Back

Full Screen

Close



### Execution in the Modified Environment

Once a declaration has been processed a new scope  $\gamma'$  is created in which the new variables are available for use in addition to everything else that was previously present in the environment  $\gamma$  (unless it has been "hidden" by the use of the same name in the new scope).  $\gamma'$  is pushed onto  $\gamma$  to create a new environment  $\gamma[\gamma']$ . For any variable x,

$$\gamma[\gamma'](x) = \begin{cases} \gamma'(x) & \text{if } x \in Dom(\gamma') \\ \gamma(x) & \text{if } x \in Dom(\gamma) - Dom(\gamma') \\ \bot & otherwise \end{cases}$$

$$\mathbf{D} - \mathbf{Seq} \quad \begin{array}{c} \gamma \vdash \langle \sigma, d_1 \rangle \longrightarrow_d \langle \gamma_1, \sigma_1 \rangle \\ \gamma[\gamma_1] \vdash \langle \sigma_1, d_2 \rangle \longrightarrow_d \langle \gamma_2, \sigma_2 \rangle \\ \hline \gamma \vdash \langle \sigma, d_1; d_2 \rangle \longrightarrow_d \langle \gamma_1[\gamma_2], \sigma_2 \rangle \end{array}$$

44

▶

PL December 28. 2021

Go Back

Full Screen

Close

# Semantics of Anonymous Blocks



Block 
$$\begin{array}{c} \gamma \vdash \langle \sigma, d \rangle \longrightarrow_d^* \langle \gamma', \sigma' \rangle \\ \gamma[\gamma'] \vdash \langle \sigma', c \rangle \longrightarrow_c^* \sigma'' \\ \hline \gamma \vdash \langle \sigma, \{d; c\} \rangle \longrightarrow_c \sigma'' \upharpoonright Dom(\sigma) \end{array}$$

### Note.

- Note the use of the multi-step transitions on both declarations and commands
- We have given up on single-step movements, since taking these "big"-steps in the semantics is more convenient and less cumbersome
- ullet Note that the "little" environment  $\gamma'$  which was produced by the declaration d is no longer present on exiting the block.
- On exiting the block the domain of the state returns to  $Dom(\sigma)$ , shedding the new locations that were created for the "little" environment.

44

**>>** 

PL December 28, 2021

Go Back

Full Screen

Close





→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 703 OF 721



### Parameterless Subroutines: Named Blocks

The introduction of named blocks allows transfer of control from more control points than in the case of unnamed blocks.

$$d_1, d_2, d ::= \cdots \mid \underline{\operatorname{sub}} P = c$$
 $c ::= \cdots \mid P$ 

- ullet The scope rules remain the same. All names in c refer to the most recent definition in the innermost enclosing scope of the current scope.
- ullet c may refer to variables that are visible in the static scope of P.

44

**>>** 

Go Back

Full Screen

Close

### What does a procedure name represent?



- An anonymous block transforms a store  $\sigma$  to another store  $\sigma'$ .
- Each procedure name stands for a piece of code which effectively transforms the store.
- Unlike an anonymous block which has a fixed position in the code, a named procedure may be called from several points (representing many different states).
- Each procedure represents a "state transformer".
- However under static scope rules, the environment in which a procedure executes remains fixed though the store may vary.
- Our environment, in addition to having locations should also be able to associate names with state transformers.

$$Proc_0 = Stores \rightarrow Stores$$

$$Env = \{ \gamma \mid \gamma : X \rightarrow (Loc + Proc_0) \}$$

Each procedure declaration  $\underline{\operatorname{sub}}\ P = c$  modifies the environment  $\gamma$  by associating the procedure name P with an entity called a procedure closure  $\operatorname{proc}0(c,\underline{\gamma})$  which represents the body of the procedure and the environment in which it is to be executed.

$$\boxed{ \begin{array}{c} \mathbf{DSub0} \ \hline \gamma \vdash \langle \sigma, \underline{\mathtt{sub}} \ P = c \rangle \longrightarrow_d \langle [P \mapsto proc0(c, \gamma)] \gamma, \sigma \rangle \end{array} }$$

CSub0 
$$\frac{\gamma_1 \vdash \langle \sigma, c \rangle \longrightarrow_c^* \sigma'}{\gamma \vdash \langle \sigma, P \rangle \longrightarrow_c \langle [P \mapsto proc0(c, \gamma)] \gamma, \sigma' \rangle} \quad (\gamma(P) = proc0(c, \gamma_1))$$

If P is recursive then we modify the last rule to

CrecSub0 
$$\frac{\gamma_2 \vdash \langle \sigma, c \rangle \longrightarrow_c^* \sigma'}{\gamma \vdash \langle \sigma, P \rangle \longrightarrow_c \langle [P \mapsto proc0(c, \gamma)] \gamma, \sigma' \rangle} \quad (\gamma(P) = proc0(c, \gamma_1))$$
 where  $\gamma_2 = [P \mapsto \gamma(P)] \gamma_1$ .

<sup>Ч</sup>A generalization to mutual recursion with many such procedures is the principle.



### Subroutines with Value Parameters

We consider the case of only a single parameter for simplicity.

$$d_1, d_2, d ::= \cdots \mid \underline{\operatorname{sub}} \ P(\underline{\operatorname{t}} \ x) = c \mid \underline{\operatorname{sub}} \ P(\underline{\operatorname{bool}} \ x) = c$$

$$c ::= \cdots \mid P(e)$$

$$Proc_0 = Stores \to Stores$$

$$Proc_v = (Stores \times (\underline{\operatorname{int}} \cup \underline{\operatorname{bool}})) \to Stores$$

$$Proc = Proc_0 + Proc_v$$

$$Env = \{\gamma \mid \gamma : X \to (Loc + Proc)\}$$

**⋖**◆

PL December 28, 2021

Go Back

Full Screen

Close

# Semantics of Call-by-value

$$\mathbf{DSubv} \ \ \overline{\gamma \vdash \langle \sigma, \underline{\mathtt{sub}} \ P(\underline{\mathtt{t}} \ x) = c \rangle \longrightarrow_d \langle [P \mapsto proc_v(\underline{\mathtt{t}} \ x, c, \gamma)] \gamma, \sigma \rangle}$$

where  $t \in \{int, bool\}$ 

$$\begin{array}{c} \gamma \vdash \langle \sigma, e \rangle \longrightarrow_{e}^{*} v \\ \underline{\gamma_{2} \vdash \langle [l \mapsto v] \sigma, c \rangle \longrightarrow_{c}^{*} \sigma'} \\ \overline{\gamma \vdash \langle \sigma, P(e) \rangle \longrightarrow_{c} \sigma' \upharpoonright Dom(\sigma)} \end{array} (\gamma(P) = proc_{v}(\underline{\mathbf{t}} \ x, c, \gamma_{1})) \\ \end{array}$$

### where

- $\bullet \gamma_2 = [x \mapsto l][P \mapsto \gamma(P)]\gamma_1$
- $l \notin Range(\gamma) \cup Dom(\sigma)$  and
- $\bullet \gamma(P) = proc_v(\mathbf{t} \ x, c, \gamma_1).$

Go Back

Full Screen

Close

### Subroutines with Reference Parameters

The Call-by-value parameter passing mechanism requires the evaluation of an expression for the value parameter to be passed to the procedure. It requires in addition the allocation of a location to store the value of the actual expression. This strategy while quite efficient for scalar variables is too expensive when the parameters are large structures such as arrays and records. In these case it is more usual to pass merely only a *reference* to the parameter and ensure that all modifications to any component of the formal parameter are instantaneously reflected also in the actual parameter.

We consider the case of a single reference parameter for simplicity. We consider the case of only a single parameter for simplicity.

$$d_1, d_2, d := \cdots \mid \underline{\text{sub}} \ P(\underline{\text{ref t}} \ x) = c \mid \underline{\text{sub}} \ P(\underline{\text{ref bool}} \ x) = c$$
 $c := \cdots \mid P(x)$ 

Notice that unlike the case of value parameters, the actual parameter in the

<sup>--</sup>calling code can only pass a-variable that is already present in its environment 121

We augment the definition of Proc to include a new entity viz.  $Proc_r$ . then have



$$Proc_0 = Stores \rightarrow Stores$$
 $Proc_v = (Stores \times (\underline{int} \cup \underline{bool})) \rightarrow Stores$ 
 $Proc_r = (Stores \times Loc) \rightarrow Stores$ 
 $Proc = Proc_0 + Proc_v + Proc_r$ 
 $Env = \{\gamma \mid \gamma : X \rightarrow (Loc + Proc)\}$ 

$$\mathbf{DSubr} \ \ \overline{\gamma \vdash \langle \sigma, \underline{\mathtt{sub}} \ P(\underline{\mathtt{t}} \ x) = c \rangle \longrightarrow_d \langle [P \mapsto proc_r(\underline{\mathtt{t}} \ x, c, \gamma)] \gamma, \sigma \rangle}$$

where  $t \in \{int, bool\}$ 

CrecSubr 
$$\frac{\gamma_2 \vdash \langle [\sigma, c \rangle \longrightarrow_c^* \sigma'}{\gamma \vdash \langle \sigma, P(y) \rangle \longrightarrow_c \sigma'} (\gamma(P) = proc_r(\underline{\mathbf{t}} \ x, c, \gamma_1))$$

where

 $\bullet \gamma_2 = [x \mapsto \gamma(y)][P \mapsto \gamma(P)]\gamma_1,$ 

G

Go Back I

Full Screen

Close



→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 711 OF 721





#### 17. Logic Programming and Prolog

# Logic Programming

A program is a theory (in some logic) and computation is deduction from the theory.

J. A. Robinson

44

**>>** 

PL December 28, 2021

Go Back

Full Screen

Close



### FOL: Reversing the Arrow

Let

$$\phi \leftarrow \psi \stackrel{df}{=} \psi \rightarrow \phi$$

Consider any clause  $C = \{\pi_1, \dots, \pi_p\} \cup \{\neg \nu_1, \dots, \neg \nu_n\}$  where  $\pi_i$ ,  $1 \le i \le p$ are positive literals and  $\neg \nu_j$ ,  $1 \leq j \leq n$  are the negative literals. Since a clause in FOL with free variables represents the universal closure of the disjunction of its literals, we have

PL December 28, 2021

Go Back

Full Screen

Close



### **FOL Arrow Reversal**

$$C \Leftrightarrow \vec{\forall} [(\bigvee \pi_i) \lor (\bigvee \neg \nu_j)]$$

$$1 \leq i \leq p \qquad 1 \leq j \leq n$$

$$\Leftrightarrow \vec{\forall} [(\bigvee \pi_i) \lor \neg (\bigwedge \nu_j)]$$

$$1 \leq i \leq p \qquad 1 \leq j \leq n$$

$$\Leftrightarrow \vec{\forall} [(\bigwedge \nu_j) \to (\bigvee \pi_i)]$$

$$1 \leq j \leq n \qquad 1 \leq i \leq p$$

$$\equiv \vec{\forall} [(\bigvee \pi_i) \leftarrow (\bigwedge \nu_j)]$$

$$1 \leq i \leq p \qquad 1 \leq j \leq n$$

$$df = \pi_1, \dots, \pi_p \leftarrow \nu_1, \dots, \nu_n$$

44

•

▶▶

PL December 28, 2021 Go BACK

Full Screen

Close



### FOL: Horn Clauses

### **Definition 17.1: Horn clauses**

Given a clause

$$C \stackrel{df}{=} \pi_1, \dots, \pi_p \leftarrow \nu_1, \dots, \nu_n$$

- Then C is a Horn clause if  $0 \le p \le 1$ .
- C is called a
  - -program clause or rule clause if p = 1,
  - -fact or unit clause if p=1 and n=0,
  - -goal clause or query if p = 0,
- $\bullet$  Each  $\nu_i$  is called a sub-goal of the goal clause.

**⋖**∢

1

▶▶

Go Back

Full Screen

Close



# FOL: Program or Rule Clause

$$P \stackrel{df}{=} \pi \leftarrow \nu_1, \dots, \nu_n$$

$$\equiv \vec{\forall} [\pi \vee (\bigvee \neg \nu_j)]$$

$$1 \le j \le n$$

$$\equiv \vec{\forall} [\pi \vee \neg (\bigwedge \nu_j)]$$

$$1 < j < n$$

PL December 28, 2021

and is read as " $\pi$  if  $\nu_1$  and  $\nu_2$  and ...and  $\nu_n$ ".

44

**•** 

▶▶

Go Back

Full Screen

Close



### FOL Facts: Unit Clauses

$$F \stackrel{df}{=} \pi$$

$$\equiv \vec{\forall}[\pi]$$

PL December 28, 2021







### FOL: Goal clauses

### Given a goal clause

$$G \stackrel{df}{=} \leftarrow \nu_1, \dots, \nu_n$$

$$\Leftrightarrow \vec{\forall} [\neg \nu_1 \lor \dots \lor \neg \nu_n]$$

$$\Leftrightarrow \neg \vec{\exists} [\nu_1 \land \dots \land \nu_n]$$

If  $\vec{y} = FV(\nu_1 \wedge \ldots \wedge \nu_n)$  then the goal is to prove that there exists an assignment to  $\vec{y}$  which makes  $\nu_1 \wedge \ldots \wedge \nu_n$  true.

44

**>** 

▶▶

Go Back

Full Screen

Close



# First-order Logic Programs

# **Definition 17.2: First-order Logic programs**

A First-order logic program is a finite set of Horn clauses, i.e. it is a set of rules  $P = \{h^1, \dots, h^k\}$ ,  $k \geq 0$  with  $h^l \equiv \pi^l \leftarrow \nu_1^l, \dots, \nu_{n_l}^l$ , for  $0 \leq l \leq k$ .  $\pi^l$  is called the **head** of the rule and  $\nu_1^l, \dots, \nu_{n_l}^l$  is the **body** of the rule.

Given a logic program P and a goal clause  $G = \{\nu_1, \dots, \nu_n\}$  the basic idea is to show that

 $P \cup \{G\}$  is unsatisfiable  $\Leftrightarrow \vec{\exists} [\nu_1 \wedge \cdots \wedge \nu_n]$  is a logical consequence of P

**∢∢** 

**•** 

**▶**▶

Go Back

FULL SCREEN

CLOSE



Effectively showing  $P \models \vec{\exists} [\nu_1 \land \cdots \land \nu_n]$  implies that we need to find values for the variables  $X = \bigcup_{j=1}^n FV(\nu_j)$  which ensure

that  $P \cup \{G\}$  is unsatisfiable. By Herbrand's theorem this reduces to the problem of finding substitutions of ground terms in the Herbrand base for variables in such a manner as to ensure unsatisfiability of  $P \cup \{G\}$ . This substitution is also called a *correct answer substitution*.

We may regard a logic program therefore as a set of postulates of a family of models (represented by a Herbrand model) and any correct answer substitution that may be derived (through resolution refutation) as a proof of the Goal as a logical consequence of the postulates. Since resolution refutation is sound and complete we effectively show  $P \vdash_{\mathscr{R}} \exists [\nu_1 \land \cdots \land \nu_n]$  where  $\vdash_{\mathscr{R}}$  denotes a proof by resolution refutation.

A propositional logic program is one in which there are no variables either in P or in the goal clause G and the execution is a pure application of the rule Res0 to obtain a contradiction.

44

**▶** 

PL December 28, 2021

Go Back

Full Screen

Close





SURE THEY DID, IN FACT, THOSE OLD PHOTOGRAPHS ARE IN COLOR. IT'S JUST THE AVORLE WAS BLACK AND WHITE THEN.















→ PL December 28, 2021 Go BACK FULL SCREEN CLOSE 722 OF 721



### Prolog: EBNF1

```
cprogram> ::= <clause list> <query> | <query>
cate list> ::= cate> |
                cate list> , <predicate>
cate> ::= <atom> | <atom> ( <term list> )
<term list> ::= <term> | <term list> , <term>
<term> ::= <numeral> | <atom> | <variable> | <structure>
<structure> ::= <atom> ( <term list> )
<query> ::= ?- redicate list>.
```

44

.

Go Back

FULL SCREEN

Close

723 of 721

### Prolog: EBNF2



724 OF 721

```
<atom> ::= <small atom> | ' <string> '
<small atom> ::= <lowercase letter> |
                 <small atom> <character>
<variable> ::= <uppercase letter> | <variable> <character>
<lowercase letter> ::= a | b | c | ... | x | y | z
<uppercase letter> ::= A | B | C | ... | X | Y | Z | _
<numeral> ::= <digit> | <numeral> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<character> ::= <lowercase letter> | <uppercase letter> |
                <digit> | <special>
<special> ::= + | - | * | / | \ | ^ | ~ | : | . | ? | |
<string> ::= <character> | <string> <character>
```

PL December 28, 2021

Go Back

Full Screen

44



### Algorithm

#### Algorithm 17.1

```
INTERPRETO (P,G) \stackrel{df}{=}

\begin{cases}
\mathbf{requires:} & \text{A propositional logic program } P \text{ and propositional goal } G \\
goals := \{G\} \\
\mathbf{while} & \text{goals} \neq \emptyset \\
\begin{cases}
\text{Choose some goal } A \in goals \\
\text{Choose a clause } A' \leftarrow B_1, \dots, B_k \in P : A \equiv A' \\
\mathbf{if } A' \text{ does not exist} \\
\mathbf{then exit} \\
\mathbf{else} & goals := (goals - \{A\}) \cup \{B_1, \dots, B_k\} \\
\mathbf{if } & goals = \emptyset \\
\mathbf{then return } & (yes) \\
\mathbf{else return } & (no) \\
\mathbf{ensures:} & yes \text{ if } P \vdash G \text{ else } no
\end{cases}
```



PL December 28, 2021



### Algorithm

```
Algorithm 17.2
```

```
INTERPRET1 (P,G) \stackrel{df}{=}
       requires: A logic program P and goal G
    Standardize variables apart in P \cup \{G\}
    goalStack := emptyStack
    \theta := \mathbf{1}
    push(goalStack, \theta G)
    while \neg empty(goalStack)
              \begin{cases} A := pop(goalStack) \\ \mathbf{if} \ \exists A' \leftarrow B_1, \dots, B_k \in P : unifiable(A, A') \end{cases}
                 then \begin{cases} \tau := \text{UNIFY } (A, A') // algorithm ?? \\ \theta := \tau \circ \theta \end{cases}
       do
                  else exit
               if k > 0
                  then push(goalStack, \theta B_k, \dots, \theta B_1)
    \mathbf{if} \ empty(goalStack)
       then return (\theta)
       else return (no)
       ensures: if P \vdash G then \theta else no
```

PL December 28, 2021