CS 314 Principles of Programming Languages

Guang Wang

Department of Computer Science

Rutgers University

1. What do you expect to be covered in recitations?
2. I will answer your questions ASAP, normally, in 24 hours.
3. Two parts:

    a. go over(understand the conceptions better)

    b. problem sets/non-graded quizzes(prepare for the midterms and the final)

Why this course?

- Why Learn More than One Programming Language?

Each language encourages thinking about a problem in a particular way.

- Each language provides (slightly) different expressiveness & efficiency.
- ⇒ The language should match the problem

“No free lunch!”

Imperative Paradigm

A program is: A sequence of state-changing actions

Manipulate an abstract machine with:

- **– Variables that name memory locations**
- **– Arithmetic and logical operations**
- **– Evaluate and assign operations**
- **– Explicit control flow statements**

Key operations: *Assignment, Call*

  - **– also *Go To, Go To if 0***
  - **– or *If, While***

Functional Paradigm

A program is: Composition of functions on data

- **Characteristics**
  - **Name values, not memory locations**
  - **Value binding through parameter passing**
  - **Recursion rather than iteration**
- **Key operations: *Function Application, Function Abstraction***
  - **Based on the Lambda Calculus**

Eg: Scheme

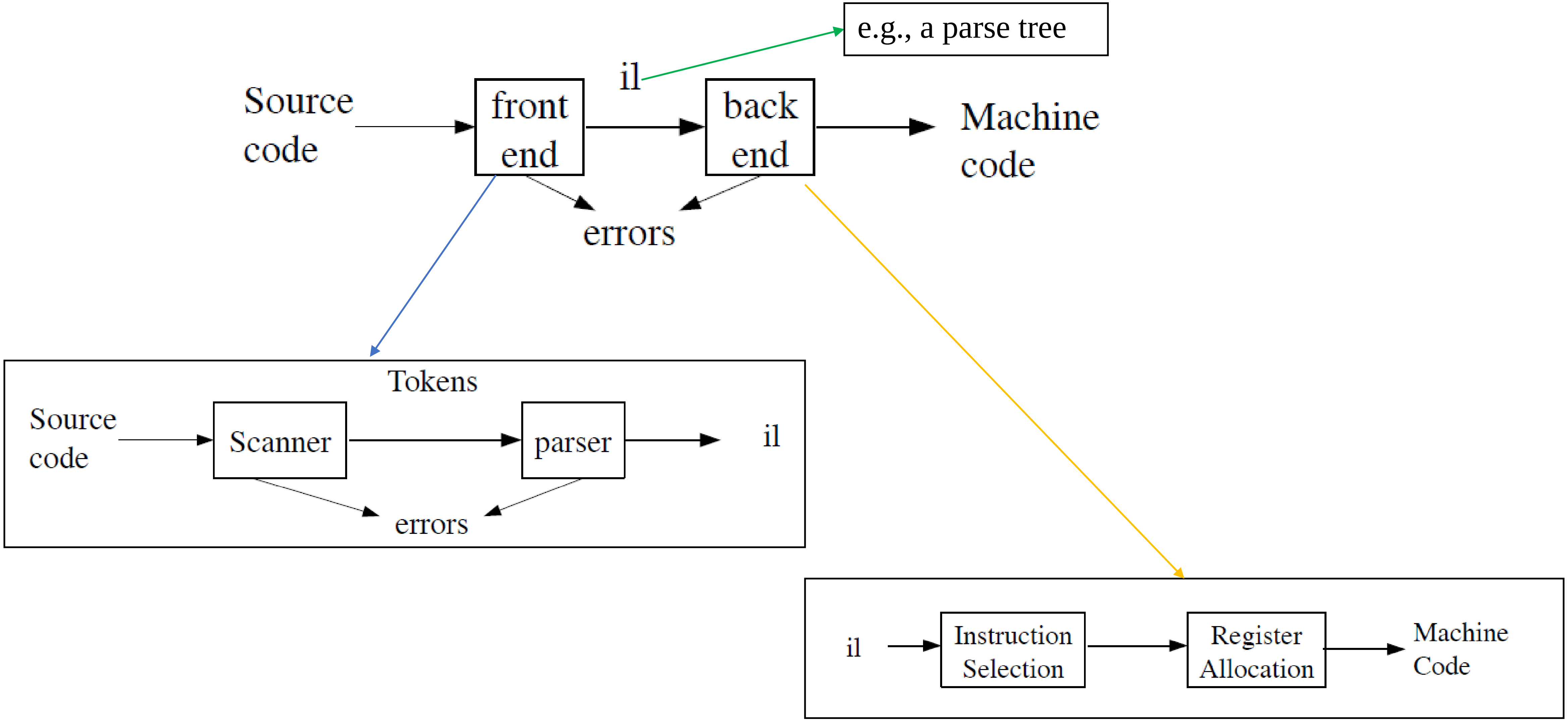
Logic Paradigm

A program is: Formal logic specification of the problem

- **Characteristics**
    - **Programs say *what* properties the solution must have, not *how* to find it**
    - **Solutions are obtained through a specialized form of *theorem-proving***
- **Key operations: *Unification, NonDeterministic Search***
  - **Based on First Order Predicate Logic**

  Eg: Prolog rules

Complier


e.g., a parse tree
il
Source code
front end
back end
Machine code
errors
Tokens
Source code
Scanner
parser
il
errors
il
Instruction Selection
Register Allocation
Machine Code

Scanner

- Maps characters → tokens
  - Tokens: basic unit of syntax
  - E.g., x = x + y becomes a list of tokens:

    <id, x> <operator, assign> <id, x> <operator, + > <id, y>
  - Typical *token* types:

    number, id, operator (e.g., +), keyword (e.g., do, else)

Parser

- **Parse: determine the grammatical structure of a sequence of tokens**
- **Grammar: set of rule like**

  assignment → variable '=' expression ';'
- **Analogy with grammars of human languages.**
  - **e.g., English: Sentence → Subject Verb Object**

Defining a language

- 1. Syntax
    - tokens defined by a *Regular Expression* or *Finite State Automaton*
    - Larger scale structure defined by a *Context Free Grammar*
- 2. Semantics
    - several formal approaches but in practice we just use English to explain the meaning

Formal languages

- Alphabet: a set of symbols
  - {a, b} or {if, while, for}
- String: a sequence of symbols from the alphabet
  - abbab
- Language: a set of strings
  - {ab, abab, ababab, …}
  - Each string: finite length
  - Set: possibly infinite number of strings

Grammar

- A set of Terminal Symbols

  a,b

- A set of Non-terminal Symbols

  MultiAB,AB

- A set of Rules of the form

  Non-terminal => sequence of Symbols

  MultiAB => MultiAB AB

  MultiAB => AB

  AB => a b

Grammar Rule(Production)

A rule is of the form

Left-side => Right-side

***Left-side* is a single Nonterminal,**

***Right-side* is a sequence of Terminals and/or Non-terminals**

The Language of a Grammar

Any string you can produce by the following process:

Initialize a list of symbols to be just the Start Symbol

Repeatedly:

- **Find the first Non-terminal in the list**
- **Find a rule whose Left-hand side is this non-Terminal**
- **Replace the Non-terminal with the Right-hand side of that rule**

Until the list contains only Terminals

The Language of a Grammar

Grammar GAB:

MultiAB => MultiAB AB

MultiAB => AB

AB => a b

Start symbol

Example:

Proof that a b a b a b a b is in L(GAB):

MultiAB

MultiAB AB

MultiAB AB AB

AB AB AB

a b AB AB

a b a b AB

a b a b a b

Derivation in a Grammar

- Example: Write a leftmost derivation of aabb in G2?

G2:

<Stmt> => <A> | <A> <B>

<A> => a | a <A>

<B> => b | <B> b

Answer:

<Stmt> => <A> <B>

=> a<A> <B>

=> aa <B>

=> aa <B>b

=> aabb

Q: How about derivation of aaab in G2? abbb?? baa??

Answer:

<Stmt> => <A> <B>
=> a<A> <B>
=> aa <A> <B>
=> aa a<B>
=> aaab

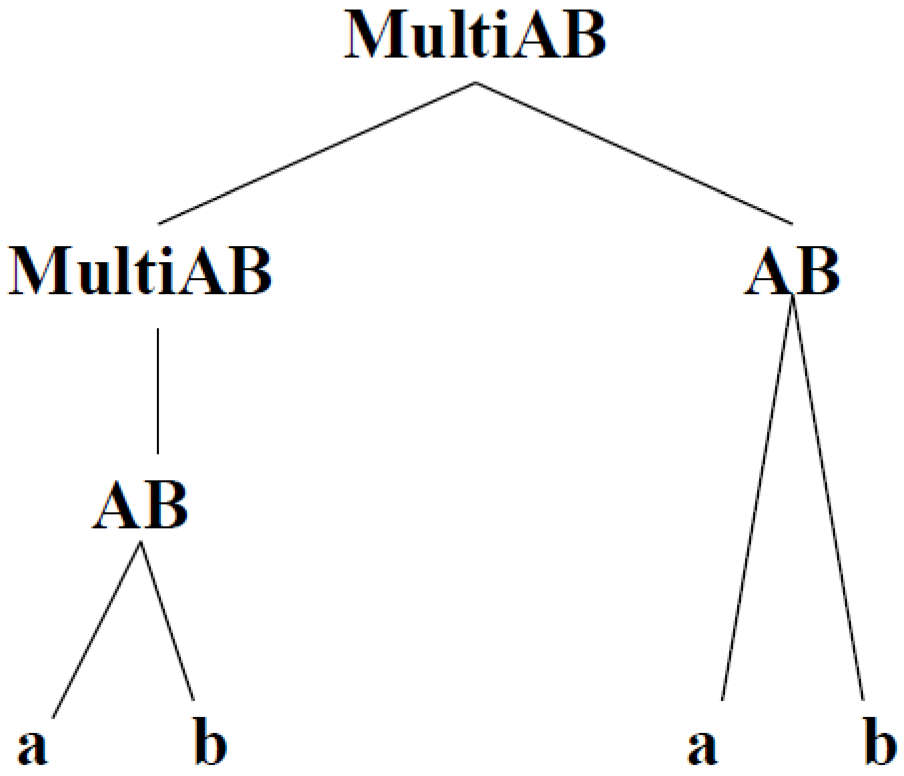
Answer:

<Stmt> => <A> <B>
=> a<B>b
=> a<B> b b
=> abbb

Answer: Can not derivate baa or any strings start with b.

Parse Trees

- **Each internal node is a Non-terminal; its children make up the right-hand side of one of the productions for that Non-terminal.**



Parse Tree

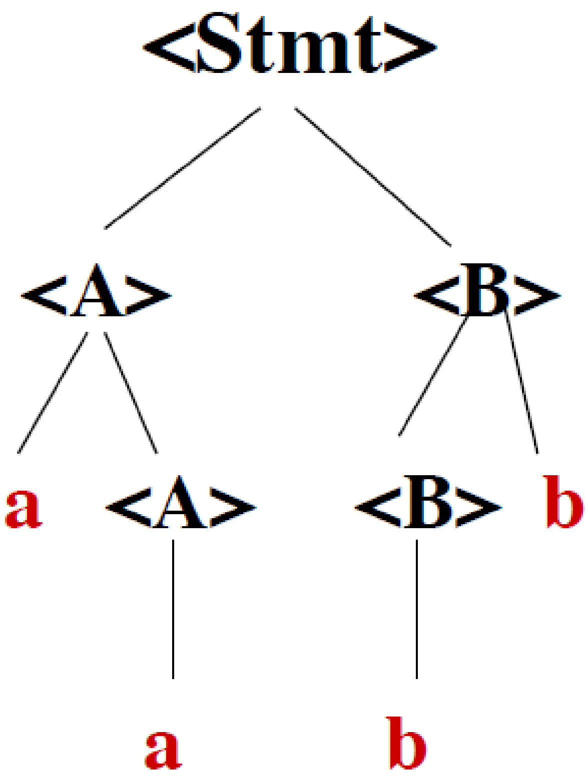
- Example: Draw a parse tree for aabb and aaab in G2?

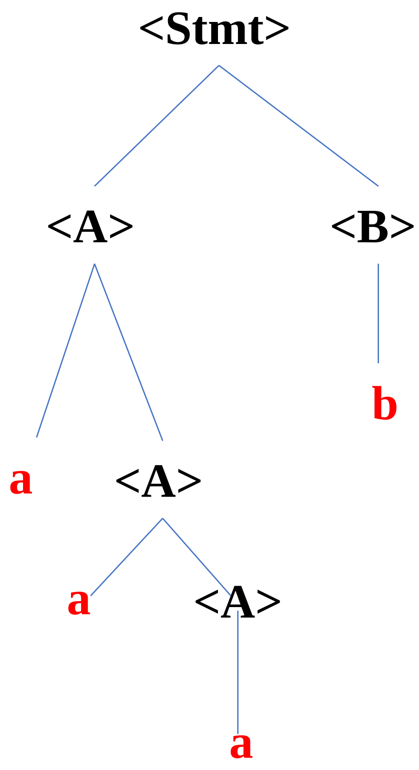
G2:

<Stmt> => <A> | <A> <B>

<A> => a | a <A>

<B> => b | <B> b





Grammars are not Unique

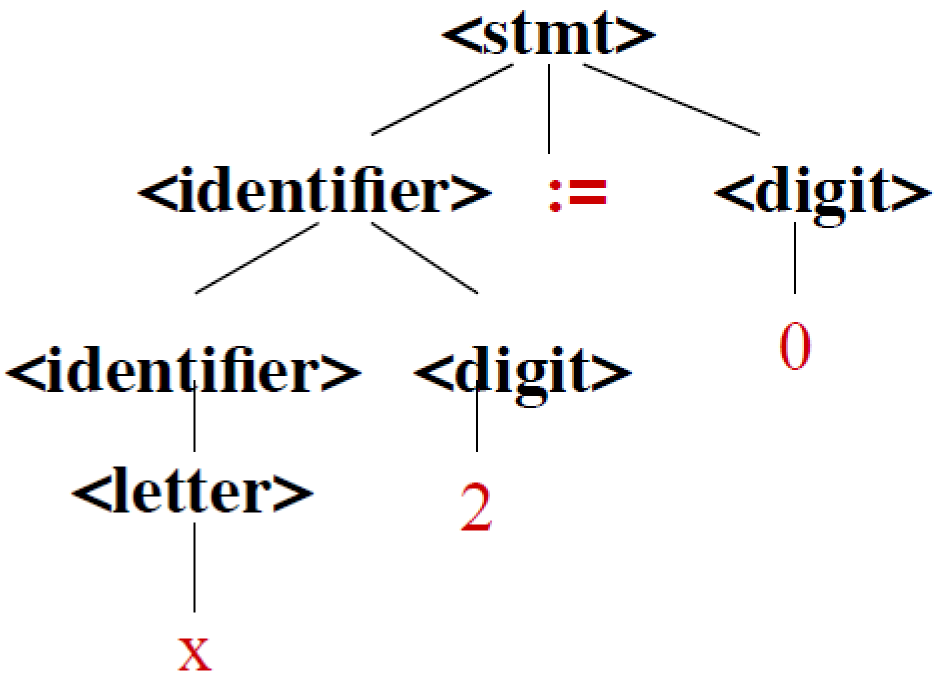
Consider a grammar G:

<stmt> ::= <ident> := <digit>
<ident> ::= <letter>
<ident> ::= <ident> <letter>
<ident> ::= <ident> <digit>
<letter> ::= a | b | c | ... | x | y | z
<digit> ::= 0 | 1 | … | 8 | 9

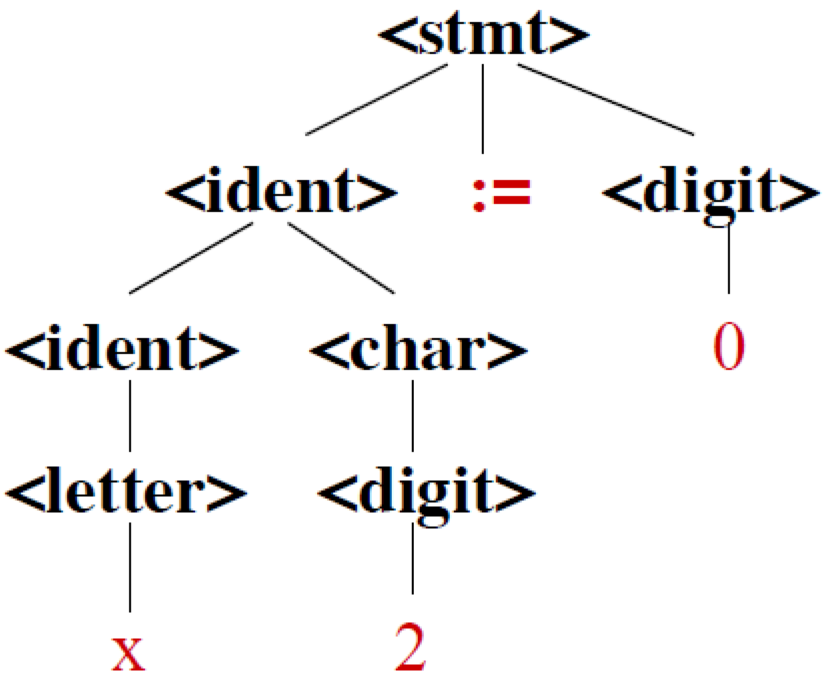
Consider a grammar G':

<stmt> ::= <ident> := <digit>
<ident> ::= <letter>
<ident> ::= <ident> <char>
<char> ::= <letter> | <digit>
<letter> ::= a | b | c | ... | x | y | z
<digit> ::= 0 | 1 | … | 8 | 9

Grammars are not Unique



Parse Tree for G



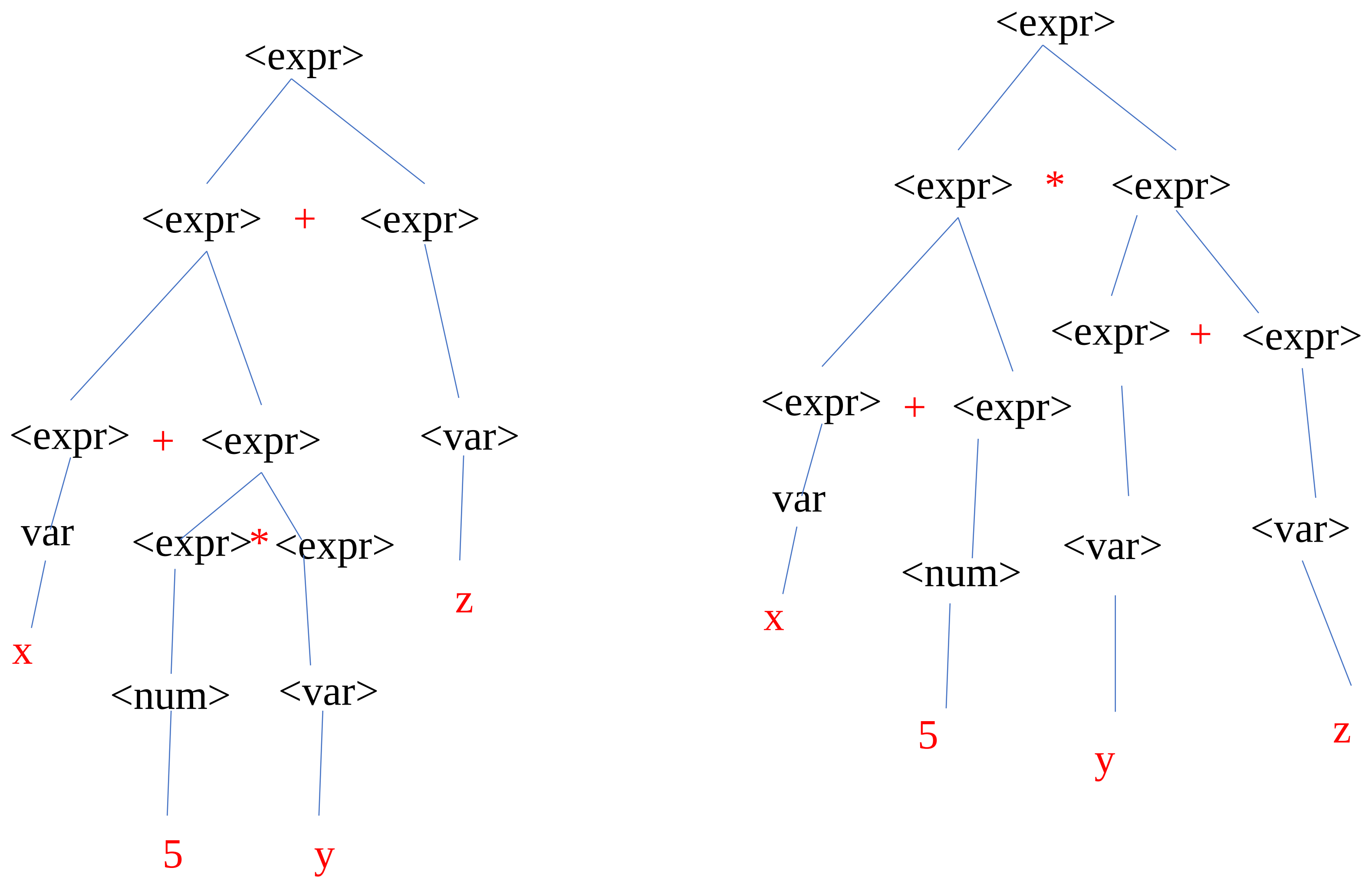
Parse Tree for G’

Arithmetic Expressions

Here is a grammar for arithmetic expressions:

- **<expr> ::= <expr> + <expr> | <expr> - <expr> |**
- **<expr> * <expr> | <expr> / <expr> | <var> | <num>**
- **<var> ::= a | b | c | ... | x | y | z**
- **<num> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

Example: Using this grammar, how would we parse: x + 5 * y + z?



Two different parse trees here

Precedence

Modify the grammar to add precedence:
<expr> ::= <expr> + <expr> | <expr> - <expr> | <term>
<term> ::= <term> * <term> | <term> / <term> | <factor>
<factor> ::= <var> | <num> | (<expr>)
<var> ::= a | b | c | ... | x | y | z
<num> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

E.g.: 3+4*5 means 3+(4*5) rather than (3+4) *5

Associativity

Modify the grammar to add associativity:

<expr> ::= <expr> + <term> | <expr> - <term> |<term>

<term> ::= <term> * <factor> | <term> / <factor> |<factor>

<factor> ::= <var> | <num> | (<expr>)

<var> ::= a | b | c | ... | x | y | z

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

E.g.: 10-4-3 means (10-4)-3 rather than 10-(4-3)

Types of Grammars

***Context Free Grammars*:**

– **Every rule has a single nonterminal on the left-hand side:**

<A> =>…

– **Disallowed: <X> <A> => <X> a**

***Regular Grammars*:**

– **Rules all take the forms:**

<A> => c or <A> => <B> c *(left-linear)*

– **Or rules all take the forms:**

<A> => c or <A> => c <B> *(right-linear)*

– **Disallowed: S => a S b**

– **Cannot generate the language $\{ a^n b^n \mid n = 1,2,3, \ldots \}$**

Types of Grammars

- Context Free Grammars (CFGs) are used to specify the overall structure of a programming language:
  - if/then/else, ...
  - brackets: ( ), { }, begin/end, ...
- Regular Grammars (RGs) are used to specify the structure of tokens:
  - identifiers, numbers, keywords, ...
- RGs are a subset of CFGs

Ambiguous

- An ambiguous string is one that can be parsed into more than one different parse tree.

- Removing ambiguity:

    1: Precedence

    2: Associativity

Regular Expressions

	RE Notation	Language
	an empty RE	{ }
symbol a	a	{a}
null symbol	ε (means empty string)	{“”}
R, S regular exprs	R | S	$L_R \cup L_S$
	a | b (alternation)	*{a, b}*
R, S regular exprs	RS	$L_R L_S$
	ab (concatenation)	*{ab}*
	(a|b)(c|d) (concatenation of alternations)	
		{ac, ad, bc, bd}

Regular Expressions

RE Notation	Language
a	{a}
b	{b}
ab	{ab}
a | b	{a, b}
ab | ac	{ab, ac}
(a | b)(a | c)	{aa, ac, ba, bc}
(abc | ε) d	{abcd, d}

Regular Expressions

RE Notation		Language
R regular expr	R^*	$\{\varepsilon\} \cup L_R \cup L_R L_R \cup L_R L_R L_R \cup \ldots$
	a^*	$\{\varepsilon, a, aa, aaa, \ldots\}$
R regular expr	R^+	$L_R \cup L_R L_R \cup L_R L_R L_R \cup \ldots$
	a^+	$\{a, aa, aaa, \ldots\}$

Note: $\varepsilon a = a \varepsilon = a$

Precedence is + *, concatenation, |

high to low

(all are left associative operators)

Regular Expressions

RE Notation	Language
a*	{ε, a, aa, aaa, …}
ab* = a(b*)	{a, ab, abb, abbb, …}
(ab)*	{ε, ab, abab, ababab, …}
(a | b)*	{ε, a, b, aa, ab, ba, bb, …}
a+	{a, aa, aaa, …}
ab+ = a(b+)	{ab, abb, abbb, …}

Regular Expressions

- Find regex representing the language

Language	Regex
$\{0\}$	
$\{0,1\}$	
$\{0,01\}$	
$\{0, \epsilon\}\{001\}$	
$\{1\}^*\{10\}$	
$\{10,11,1100\}^*$	

Regular Expressions

Language	Regex
{0}	0
{0,1}	0|1
{0,01}	0|01
{0, ϵ}{001}	(0|ϵ)001
{1}*{10}	1*10
{10,11,1100}*	(10|11|1100)*

RE’s for PLs

- **Let *letter* stand for a|b|c|…|a|A|... |Z and *digit* stand for 0|1|2|3|4|5|6|7|8|9**
  - ***letter (letter | digit)**** **is an identifier**
  - ***digit +*** **is an integer constant**
  - ***digit * . digit +*** **is real number. So .5 is, but 5. is not.**

Regular Expressions

Q1: Construct a regular expression for binary numbers of length two

Q2: Construct a regular expression for binary numbers with even length

Q3: Construct a regular expression for floating point numbers that don’t use scientific notation (e.g., 3.5, 0.15, -47.3).

- Answer1: $(1|0)(1|0)$
- Answer2: $((1|0)(1|0))^*$
- Answer3: $(-|\epsilon)(0 - 9)^{+}.(0 - 9)^{+}$

Key Points

1. Given a grammar, knowing how to prove a string in a L(G) or not.

    method 1: derivation

    method 2: parse tree

2. Regular expression

Thanks!!!