Overview of the Compiler

Lexical Analysis

Break input into *"TOKENS"*

Source: `x = y + 1;  /* incr x */ ...`

Tokens: ID, EQUALS, ID, PLUS, INT, SEMI, ...

Overview of the Compiler

Lexical Analysis

Break input into *"TOKENS"*

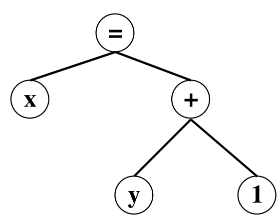
Source: `x = y + 1;  /* incr x */ ...`

Tokens: ID, EQUALS, ID, PLUS, INT, SEMI, ...

Syntax Analysis

Context-Free Grammar

Build a parse tree



Overview of the Compiler

Lexical Analysis

Break input into *"TOKENS"*

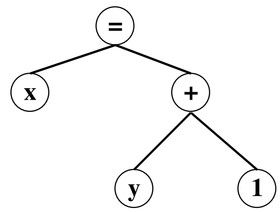
Source: `x = y + 1;  /* incr x */ ...`

Tokens: ID, EQUALS, ID, PLUS, INT, SEMI, ...

Syntax Analysis

Context-Free Grammar

Build a parse tree



Semantic Analysis

Analyze types

Check for "semantic" errors

x = y + true;

Error

Symbol Table

One entry for each identifier

key	type	address
w	bool	50
x	int	54
y	double	58
. . .	. . .	. . .

Symbol Table

One entry for each identifier

Intermediate Code

Not machine specific

```
temp1 := x
temp2 := temp1 + 1
x := temp2
```

key	type	address
w	bool	50
x	int	54
y	double	58
...	...	...

Introduction to Compiling - Part 1

Symbol Table

One entry for each identifier

Intermediate Code

Not machine specific

```
temp1 := x
temp2 := temp1 + 1
x := temp2
```

key	type	address
w	bool	50
x	int	54
y	double	58
...	...	...

Code Optimization

Eliminate redundant data movement

Optimize "goto"s to other "goto" instructions

Symbol Table

One entry for each identifier

key	type	address
w	bool	50
x	int	54
y	double	58
...	...	...

Intermediate Code

Not machine specific

```
temp1 := x
temp2 := temp1 + 1
x := temp2
```

Code Optimization

Eliminate redundant data movement
Optimize "goto"s to other "goto" instructions

Code Generation

Register Assignments
Machine Specific Code

```
mov.w     x,r5
add.w     #1,r5
mov.w     r5,x
```


Symbol Table

One entry for each identifier

key	type	address
w	bool	50
x	int	54
y	double	58
...	...	...

Intermediate Code

Not machine specific

```
temp1 := x
temp2 := temp1 + 1
x := temp2
```

Code Optimization

Eliminate redundant data movement
Optimize "goto"s to other "goto" instructions

Code Generation

Register Assignments
Machine Specific Code

```
mov.w     x,r5
add.w     #1,r5
mov.w     r5,x
```


Error Handling

Can't just abort! ... Find more errors!
Patch things up and keep going
- Lexical Errors
- Syntactic Errors
- Semantic Errors

Lexical Analysis

Token

A "word"

`if` `x` `==` `-123` `then` `...`

Types of tokens

ID	x foo ...
KEYWORD	if while ...
NUMBER	-123 4.0 ...
OPERATOR	+ == ( ) ; ...

"Lexeme"

The characters comprising a token.

`if x = -3.1415e37 then ...`

Lexical Analysis

Token

A "word"

`if` `x` `==` `-123` `then` `...`

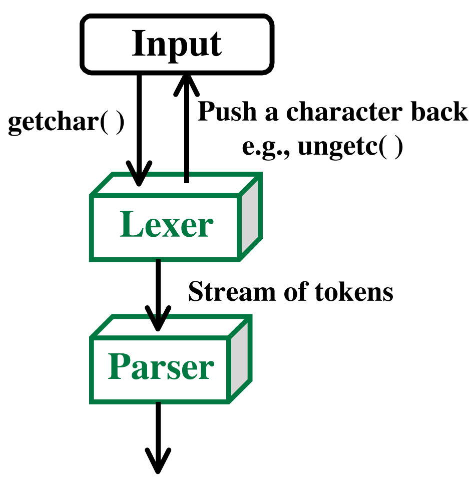
Types of tokens

ID	x foo ...
KEYWORD	if while ...
NUMBER	-123 4.0 ...
OPERATOR	+ == ( ) ; ...

"Lexeme"

The characters comprising a token.

`if x = -3.1415e37 then ...`



The Lexical Analysis Phase

"Lexer"

"Scanner"

"White Space"

Blanks, tabs, newlines

Ignored by lexer

⇒ Not seen by parser

Comments

Identified by lexer

Treated like whitespace

⇒ Not seen by parser

Introduction to Compiling - Part 1

Parser Calls Lexer

To get next token

Lexer returns a single token

"type" (e.g., NUMBER)

"attribute" (e.g., -3.1415)

Example code:

```
#define NUM 1
#define ID 2
...
int getToken () {
      ...
      return NUM;
      ...
}
```

Global Variable:
`attribute`
Side-effect:
Sets this global variable whenever a NUM is seen.

The Token Stream

Source:

```
if ( x == -3.1415 )   /* test x */  then ...
```

Token Stream:

```
< IF >
< LPAREN >
< ID, "x" >
< EQUALS >
< NUM, -3.14150000 >
< RPAREN >
< THEN >
...
```

Token: `< NUM, -3.14150000 >`

Token Type — *Token Attribute, if any*

Option 1:

Lexer writes token stream out to a file.
Parser reads from this file.

More Likely:

Parser calls lexer when it needs a token.
Lexer returns one token at a time.

Types of Token

	Type	Attribute
	ID	Ptr to String / Index into String Table
	INT	Integer value
	REAL	Double value
	CHAR	Char value
	STRING	Ptr to String / Index into String Table
Keywords	WHILE	-
	IF	-
	ELSE	-
	...	
Operators/ Punctuation	PLUS	-
	MINUS	-
	LPAREN	-
	RPAREN	-
	SEMI	-
	...	
	EOF	-

Java Constants

```
static final int
    ID = 1;
    INT = 2;
    REAL = 3;
    CHAR = 4;
    STRING = 5;
    WHILE = 6;
    IF = 7;
     ...
    EOF = 37;
```

Identifiers

A "Regular Expression"

Letter { Letter | Digit }*

"A letter followed by zero or more letters or digits."
[May also want to include underscore: `my_val`]

Identifiers

A "Regular Expression"

Letter { Letter | Digit }*

"A letter followed by zero or more letters or digits."
[May also want to include underscore: `my_val`]

Keywords

Also called:
"Reserved Words"

A fixed set of words:
{ if, while, do, return, else, break, ... }
Look like identifiers, but they are not.

Identifiers:

Letter { Letter | Digit }* - KEYWORDS

The String Table

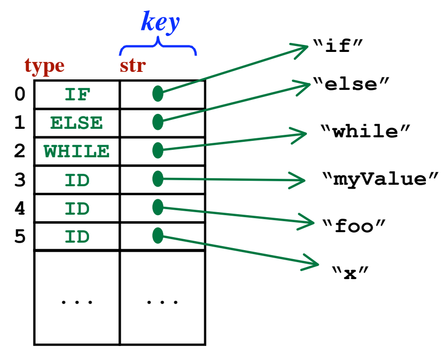
One entry per ID, KEYWORD

Each entry contains
- The lexeme (i.e., the string of characters)
- A type flag (ID, KEYWORD)

Implementation:
Array (see next slide)
Hash Table (faster lookup)

Goal:
Given a new lexeme...
Determine quickly:
- Is it a keyword? Which one?
- Is it an ID we've seen before?

Identify equal IDs so the parser doesn't have to bother with string comparisons.



String Table Operations:

`lookup (String) → int`

Returns index of entry, or -1

`insert (String, type) → int`

Adds a new entry and returns its index

Keywords:

- Initialize the table
- Add keywords

Outline of Lexer

```
function getToken () returns int
  var c: char
      buffer: array of char
  while true
    c = getChar ()
    if c == '\n'
      lineNumber++
    elseIf c is whitespace
      -- nothing
    elseIf isDigit (c)
      Read in zero or more digits.
      attribute = their value
      return INT
    elseIf  isLetter (c)
      Read in more letters and digits,
             placing them in buffer
      p = lookup (buffer)
      if p == -1
        p = insert (buffer, ID)
      endIf
      attribute = p
      return p.type    -- ID, WHILE, IF, ELSE, ...
    elseIf c = '+'
      return PLUS
    ...etc. for other operator symbols...
    else
      Error...
    endIf
  endWhile
endFunction
```


Symbol Table ("Environment")

The same ID may have different meanings in one program.

Example:

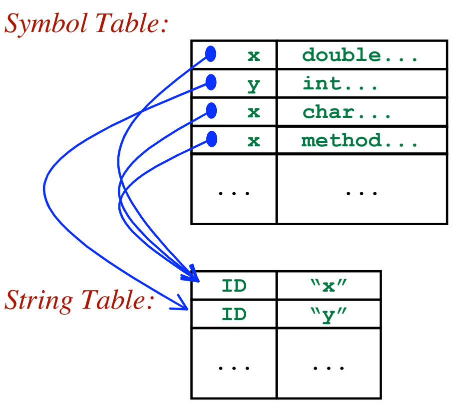
```
void foo (double x) { ... }
...
char x;
...
void x (int i) { ... }
```

Symbol Table ("Environment")

The same ID may have different meanings in one program.

Example:

```
void foo (double x) { ... }
...
char x;
...
void x (int i) { ... }
```



Syntax Analysis

Context-Free Grammar (CFG)

Productions ("Grammar Rules")

Meta Symbols

- **→** "consists of"
- **|** "or", alternatives
- **ε** epsilon, empty string

Non-terminals (Expr, Term, Factor)

Terminals

- **Lexical Tokens (ID, NUMBER)**
- **Literals (+, -, *, /, (,))**

Start Symbol (Expr)

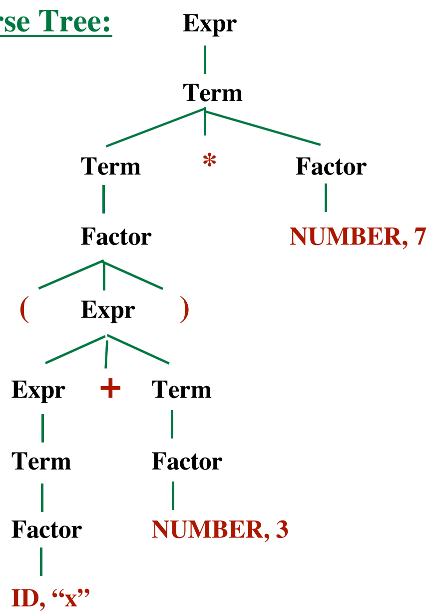
The Classic Expression Grammar

Expr	→ Expr + Term
	→ Expr - Term
	→ Term
Term	→ Term * Factor
	→ Term / Factor
	→ Factor
Factor	→ NUMBER
	→ ID
	→ (Expr)

Parse Trees

Example: `(x + 3) * 7`

Parse Tree:



Internal Nodes

Non-terminals

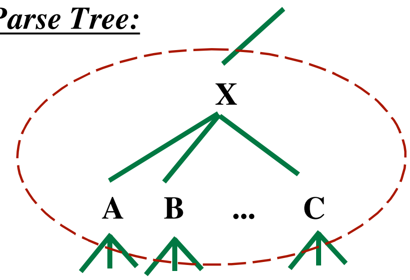
Leaves:

Terminals

A Rule (Production):

X → A B C

Parse Tree:

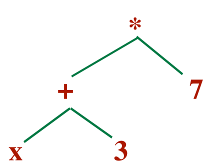


Parse Trees
Match CFG Rules

"Abstract Syntax Trees" (ASTs)
Capture only *essential* info.

Example: `(x + 3) * 7`

Abstract Syntax Tree:



Fewer Nodes, Simpler

Project 1:
- Use Context-Free Grammar to Parse Expression
- Build an Abstract Syntax Tree
- Walk Abstract Syntax Tree to Evaluate Expression

Alternative Right Sides:

```
Expr → Expr + Term
     → Expr - Term
     → Term
```

```
Expr → Expr + Term
     | Expr - Term
     | Term
```

Equivalent

Backus-Naur Notation:

```
Expr →   Expr + Term
```

```
Expr ::= Expr + Term
```

Equivalent

Epsilon

ε The empty string

Example:

A → **b** A | ε

Generates:

?

Epsilon

ε   The empty string

Example:

A → **b** A | ε

Generates:

ε
b
bb
bbb
...

Epsilon

ε   The empty string

Realistic CFG Example

```
Block     → "{" StmtList "}"
StmtList  → Stmt StmtList
          → ε
Stmt      → if Expr then Stmt else Stmt
          → ID ":=" Expr ";"
          → while Expr do Stmt
          → Block
Expr      → ...
```

Ambiguous Grammars

One string ("sentence") has two or more parse trees!

Want To Avoid!

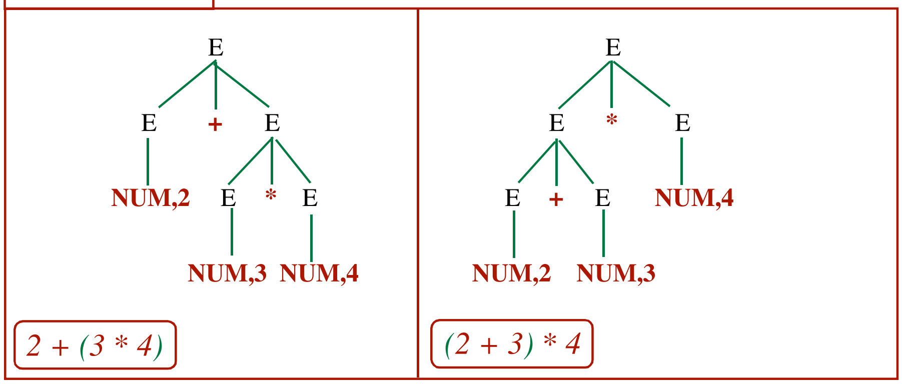
Example:

E → E + E
  → E * E
  → NUM

Source:

2 + 3 * 4

Parse Trees:



Associativity

What does 5 - 4 - 3 mean?

(5 - 4) - 3
5 - (4 - 3)

Precedence

Normal Conventions

Associativity

What does 5 - 4 - 3 mean?
(5 - 4) - 3 ← "Left Associative"
5 - (4 - 3) ← "Right Associative"

Precedence

Normal Conventions

Associativity

What does 5 - 4 - 3 mean?
(5 - 4) - 3 ← "Left Associative"
5 - (4 - 3) ← "Right Associative"
What does x = y = z mean?
(x= y) = z
x = (y = z)

Precedence

Normal Conventions

Associativity

What does 5 - 4 - 3 mean?
(5 - 4) - 3 ← "Left Associative"
5 - (4 - 3) ← "Right Associative"
What does x = y = z mean?
(x= y) = z
x = (y = z)

Precedence

What does 1 + 2 * 3 mean?

Normal Conventions

Associativity

What does 5 - 4 - 3 mean?
(5 - 4) - 3 ← "Left Associative"
5 - (4 - 3) ← "Right Associative"
What does x = y = z mean?
(x= y) = z
x = (y = z)

Precedence

What does 1 + 2 * 3 mean?
(1 + 2) * 3 ← "Plus has higher precedence"
1 + (2 * 3) ← "Multiplication has higher precedence"

Normal Conventions

Associativity

What does 5 - 4 - 3 mean?
(5 - 4) - 3 ← "Left Associative"
5 - (4 - 3) ← "Right Associative"
What does x = y = z mean?
(x= y) = z
x = (y = z)

Precedence

What does 1 + 2 * 3 mean?
(1 + 2) * 3 ← "Plus has higher precedence"
1 + (2 * 3) ← "Multiplication has higher precedence"

Normal Conventions

Left Associative: + - * /
Right Associative: = ^
Precedence:
^ ← highest
* /
+ -
= ← lowest

Assignment Operator:
x = y;
x := y;
Exponentiation Operator:
x = y ^ 3;

Parsing Algorithms

Assume we have a grammar...

"Parser"
Input:
- String of tokens

Output:
- Accept / Reject if errors
- Build a parse tree (or buld AST)
- Execute "semantic actions" (e.g., to check program)
- Print good error messages (when source has errors)

Parsing Algorithms

Assume we have a grammar...

"Parser"

Input:

- String of tokens

Output:

- Accept / Reject if errors
- Build a parse tree (or buld AST)
- Execute "semantic actions" (e.g., to check program)
- Print good error messages (when source has errors)

"Parser Generators"

Input:

- A Grammar

Output:

- A Parser (e.g., a Java code file)

Any context-free grammar can be recognized!!!

A parser can be built.

Any context-free grammar can be recognized!!!
A parser can be built.

Worst-case (Nasty grammars): $\mathbf{O}(N^3)$ time

Any context-free grammar can be recognized!!!
A parser can be built.

Worst-case (Nasty grammars): $\mathbf{O}(N^3)$ time

Typical Programming Languages: $\mathbf{O}(N)$ time ("linear")

Any context-free grammar can be recognized!!!
A parser can be built.

Worst-case (Nasty grammars): $\mathbf{O}(N^3)$ time

Typical Programming Languages: $\mathbf{O}(N)$ time ("linear")

Major Approaches:

- Top-Down Algorithms
  Simpler
  Better Error Messages
- Bottom-Up Algorithms
  Faster
  More General
  More Complex

Recursive Descent Parsing

A "Top-Down" Algorithm

Main Idea:
For each non-terminal, write a routine.

CFG Rule:
$A \rightarrow X\ Y\ Z$

Routine:
```
function ParseA () {
  ParseX ();
  ParseY ();
  ParseZ ();
}
```

The routines may be recursive.

Parse Tree is not constructed
... explicitly.

Invocation of recursive routines = Implicit Parse Tree

Handling Terminals

Global Variable

"Look-ahead" Token

```
var nextToken: TokenType
```

```
function MustHave (t: TokenType) {
  if nextToken == t then
    nextToken = getToken ();
  else
    error;
  endIf
}
```

A CFG Rule: Expr → Term + Expr

```
function ParseExpr () {
  ParseTerm ();
  MustHave (PLUS);
  ParseExpr ();
}
```


Handling Alternatives

Factor → NUM
→ (Expr)
→ ID

```
function ParseFactor () {
  if nextToken == NUM then
    MustHave (NUM);
  elseIf nextToken == '(' then
    MustHave ('(');
    ParseExpr ();
    MustHave (')');
  elseIf nextToken == ID then
    MustHave (ID);
  else
    error "Problems in Factor";
  endIf
}
```

A Problem...

Stmt → "{" StmtList "}"
→ AssignStmt
→ IfStmt
→ WhileStmt

```
function ParseStmt () {
  if nextToken == '{' then
    MustHave ('{');
    ParseStmtList ();
    MustHave ('}');
  else
    ... ? ...      ParseAssignStmt ();
                   ParseIfStmt ();
                   ParseWhileStmt ();
  endIf
}
```

A → X ...
→ Y ...
→ Z ...

Two or more rules start with non-terminals...
Which function to call???

First Sets

Let α be a sequence of terminal and non-terminal symbols.

Consider all strings that α can generate.

Define: FIRST(α) = the set of...
"All tokens that can appear first in such strings"

Example:

Stmt → "{" StmtList "}"
→ AssignStmt
→ IfStmt
→ WhileStmt

FIRST (Stmt) = { "{", ID, IF, WHILE }

Example

A → B C
  → **x**
B → **y** **z**
  → D **w**
C → **u**
D → **v**
  → ε

FIRST (D) = { ? }

Example

A → B C
  → **x**
B → **y** **z**
  → D **w**
C → **u**
D → **v**
  → ε

FIRST (D) = { **v**, ε }
FIRST (C) =

Example

A → B C
→ **x**
B → **y** **z**
→ D **w**
C → **u**
D → **v**
→ ε

FIRST (D) = { **v**, ε }
FIRST (C) = { **u** }
FIRST (D**w**) =

Example

A → B C
→ **x**
B → **y** **z**
→ D **w**
C → **u**
D → **v**
→ ε

FIRST (D) = { **v**, ε }
FIRST (C) = { **u** }
FIRST (D**w**) = { **v**, **w** }
FIRST (B) =

Example

A → B C
→ **x**
B → **y** **z**
→ D **w**
C → **u**
D → **v**
→ ε

FIRST (D) = { **v**, ε }
FIRST (C) = { **u** }
FIRST (D**w**) = { **v**, **w** }
FIRST (B) = { **y**, **v**, **w** }
FIRST (BC) =

Example

A → B C
→ **x**
B → **y** **z**
→ D **w**
C → **u**
D → **v**
→ ε

FIRST (D) = { **v**, ε }
FIRST (C) = { **u** }
FIRST (D**w**) = { **v**, **w** }
FIRST (B) = { **y**, **v**, **w** }
FIRST (BC) = { **y**, **v**, **w** }
FIRST (A) =

Example

A → B C
  → x
B → y z
  → D w
C → u
D → v
  → ε

FIRST (D) = { v, ε }
FIRST (C) = { u }
FIRST (Dw) = { v, w }
FIRST (B) = { y, v, w }
FIRST (BC) = { y, v, w }
FIRST (A) = { x, y, v, w }

For this rule

A → X ...
  → Y ...
  → Z ...

Create this code

```
if nextToken ∈ FIRST (X...) then
  ParseX...();
elseIf nextToken ∈ FIRST (Y...) then
  ParseY...();
elseIf nextToken ∈ FIRST (Z...) then
  ParseZ...();
else
  Error
endIf
```

For this rule

Stmt → "{" StmtList "}"
→ AssignStmt
→ IfStmt
→ WhileStmt

Create this code

```
function ParseStmt () {
  if nextToken == LBRACE then
    Scan a token;
    ParseStmtList ();
    MustHave (RBRACE);
  elseif nextToken == ID then
    ParseAssignStmt ();
  elseIf nextToken == IF then
    ParseIfStmt ();
  elseIf nextToken == WHILE then
    ParseWhileStmt ();
  else
    Error
  endIf
endFunction
```


Introduction to Compiling - Part 1

Problem:

A → X ...
→ Y ...
→ Z ...

What if

$\text{FIRST}(X...) \cap \text{FIRST}(Y...) \neq \emptyset$

Solutions:

- Try to rewrite grammar rules.
- Don't use Recursive-Descent Parsing
  "LR Parsing"

Left Recursion

Grammar Example:

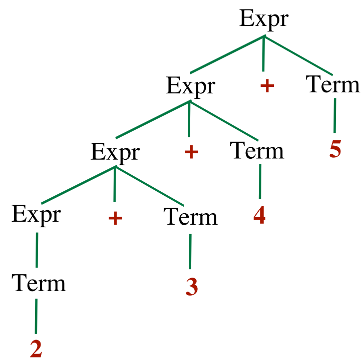
Expr → Expr **+** Term
→ Term

Source:

"2 + 3 + 4 + 5"

Means:

"((2 + 3) + 4) + 5"
Left Associativity



General form:

A → A α | β

β
βα
βαα
βααα
...

Notation:

A, B, C, ... Non-terminal symbols
α, β, γ, ... *Strings* of arbitary symbols

Right Recursion

Grammar Example:

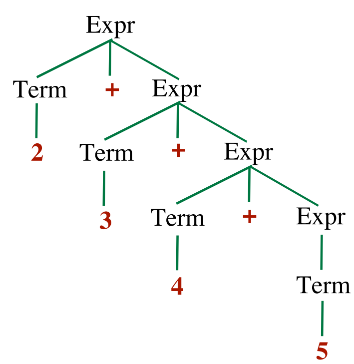
Expr → Term **+** Expr
→ Term

Source:

"2 + 3 + 4 + 5"

Means:

"2 + (3 + (4 + 5))"
Right Associativity



General form:

A → α A | β

β
αβ
ααβ
αααβ
...

Problem: Left Recursive Rules

Before:

Expr → Term + Expr

Now Consider:

Expr → Expr + Term

```
function ParseExpr () {
  ParseExpr ();
  MustHave (PLUS);
  ParseTerm ();
endFunction
```

What is the problem?

Introduction to Compiling - Part 1

Problem: Left Recursive Rules

Before:

Expr → Term + Expr

Now Consider:

Expr → Expr + Term

```
function ParseExpr () {
  ParseExpr ();
  MustHave (PLUS);
  ParseTerm ();
endFunction
```

What is the problem?

Infinite Recursion!!!

More Generally:

A → A ...
  → ...

```
function ParseA () {
  ParseA ();
  ...
endFunction
```

Solution:

- Rewrite rules to eliminate left-recursion.
- Build parser from revised rules.

Example:

Expr → Expr **+** Term
  → Term

```
Term
Term + Term
Term + Term + Term
Term + Term + Term + Term
...
```


More Generally:

A → A ...
  → ...

```
function ParseA () {
  ParseA ();
  ...
endFunction
```

Solution:

- Rewrite rules to eliminate left-recursion.
- Build parser from revised rules.

Example:

Expr → Expr **+** Term
  → Term

⇓

Expr → Term **+** Expr
  → Term

```
Term
Term + Term
Term + Term + Term
Term + Term + Term + Term
...
```

A General Formula for Eliminating Left-Recursion

Given:

$A \rightarrow A\,\alpha \mid \beta$

Notation:

A, B, C, ... Non-terminal symbols
α, β, γ, ... *Strings* of arbitary symbols

Introduce a new non-terminal (say "R").

Rewrite rules:

$A \rightarrow \beta\,R$

$R \rightarrow \alpha\,R \mid \varepsilon$

Example:

Given:

Expr → Expr **+** Term
→ Term

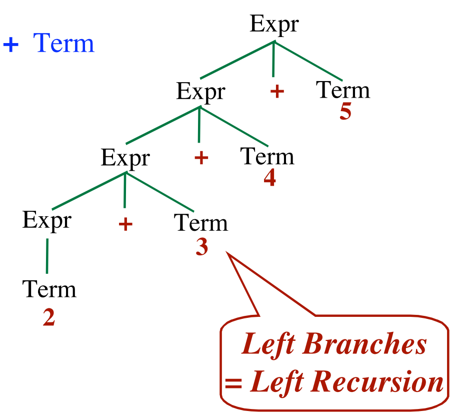
Rewrite as:

Expr → Term R
R → **+** Term R
→ ε

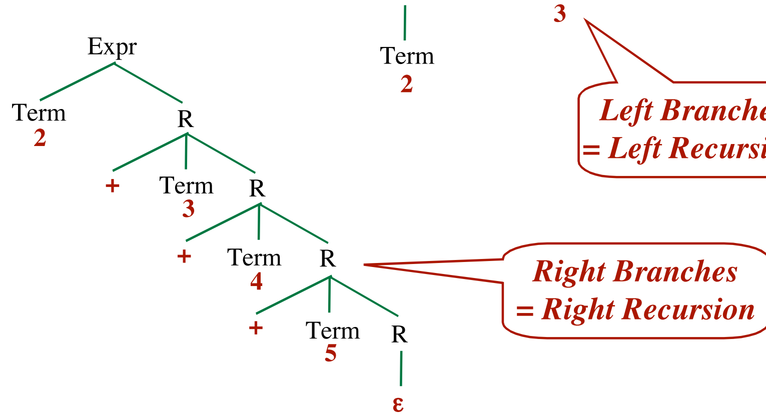
A = Expr
α = **+** Term
β = Term

Example: 2 + 3 + 4 + 5

Expr → Expr **+** Term
→ Term



Expr → Term R
R → **+** Term R
→ ε

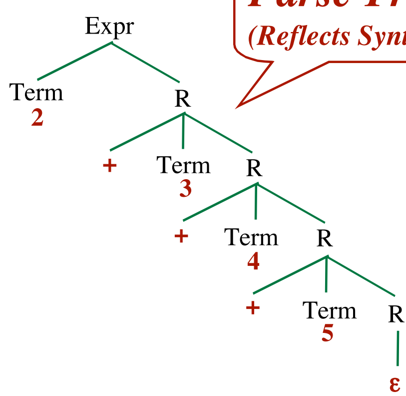


New Grammar Rules:

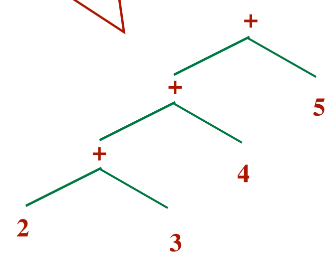
Expr → Term R
R → + Term R
→ ε

Source: "2+3+4+5"

Parse Tree
(Reflects Syntax)



Abstract Syntax Tree
(Reflects Semantic Structure)
((2 + 3) + 4) + 5



Project 1: "E" Language

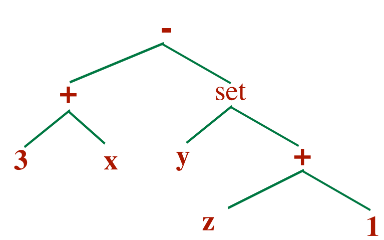
The Grammar:

Expr → Term
→ Term + Term
→ Term - Term
→ **set** ID = Expr
Term → ID
→ NUM
→ "(" Expr ")"

Semantics:
- **Evaluate z+1**
- **Modify y**
- **Use that value in Expr**

Source:
"(3 + x) - (**set** y = z + 1)"

Abstract Syntax Tree:



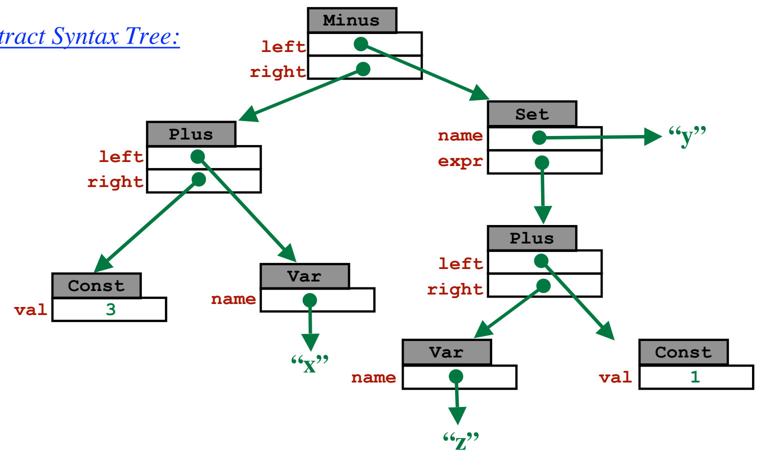
Representation in Java

Classes:

```
Plus
Minus
Set
Var
Const
```

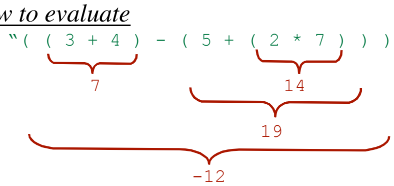
"(3 + x) - (**set** y = z + 1)"

Abstract Syntax Tree:



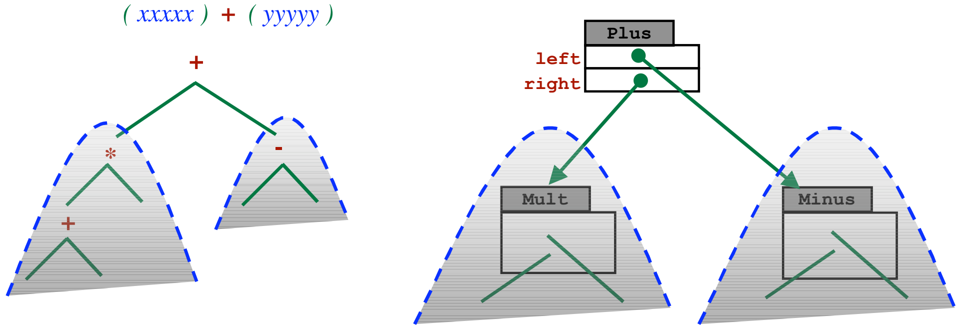
Expression Evaluation

How to evaluate



- *Evaluate sub-expressions first.*
- *Then evaluate the operation.*

(xxxxx) **+** *(yyyyy)*



- A Recursive Method:

`Node . Eval ( ValueEnv ) → IntegerResult`