

Compiler Design and Construction

For:

- ❖ SCII /2019 (July – Sept 2022)
- ❖ Department of Computer Science and Informatics
- ❖ School of Computing and Information Technologies
- ❖ TU-K

By: Salesio M. Kiura, PhD

2

Designing a Simple Compiler

Agenda

- ☐ Grammar and Parse trees
- ☐ Context Free Grammar or Backus-Naur Form
- ☐ Compiler parse trees
- ☐ Derivation during compilation
- ☐ Examples

Grammar and parse tree examples

Consider the following “English” Grammar

A sentence is a noun phrase, a verb, and a noun phrase.

$\langle S \rangle ::= \langle NP \rangle \langle V \rangle \langle NP \rangle$

A noun phrase is an article and a noun.

$\langle NP \rangle ::= \langle A \rangle \langle N \rangle$

A verb is...

$\langle V \rangle ::= \text{loves} \mid \text{hates} \mid \text{eats}$

An article is...

$\langle A \rangle ::= \text{a} \mid \text{the}$

A noun is...

$\langle N \rangle ::= \text{dog} \mid \text{cat} \mid \text{rat}$

How the grammar works

The grammar is a set of rules that say how to build a tree—a *parse tree*

You put $\langle S \rangle$ at the root of the tree

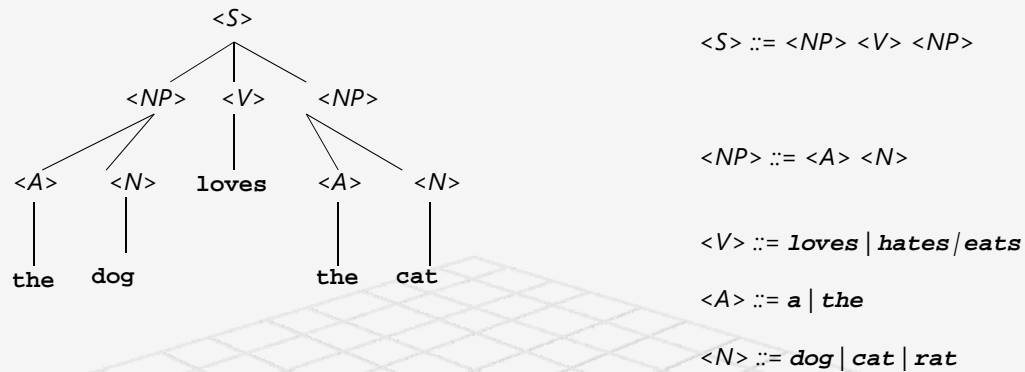
The grammar’s rules say how children can be added at any point in the tree

For instance, the rule

says you can add nodes $\langle NP \rangle$, $\langle V \rangle$, and $\langle NP \rangle$, in that order, as children of $\langle S \rangle$

$\langle S \rangle ::= \langle NP \rangle \langle V \rangle \langle NP \rangle$

A parse tree:



A Programming Language Grammar

$$\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle \mid (\langle \text{exp} \rangle)$$

$$\mid \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$$

- Narrative:
 - An expression can be the sum of two expressions, or the product of two expressions, or a parenthesized sub-expression
 - Or it can be one of the variables **a**, **b** or **c**

Continue
from here
13th July 2022

Exercise

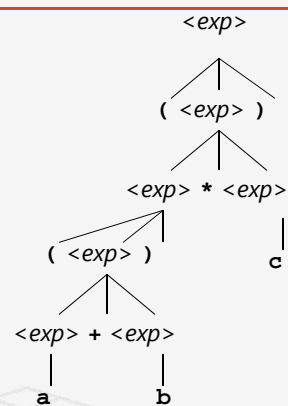
Given a Programming Language Grammar as:

$$\begin{aligned} \langle \text{exp} \rangle ::= & \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle \mid (\langle \text{exp} \rangle) \\ & \mid a \mid b \mid c \end{aligned}$$

- Generate a parse tree for:

- i. $((a+b)*c)$
- ii. $a*(b+c)$
- iii. $a+b+c$
- iv. $(a+b+c)*$

i. $((a+b)*c)$



We do the others

- ii.
- iii.
- iv. ? <Compiler error>

$a*(b+c)$

- A Programming Language Grammar

$$\begin{aligned} \langle \text{exp} \rangle ::= & \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle \mid (\langle \text{exp} \rangle) \\ & \mid a \mid b \mid c \end{aligned}$$

$a+b+c$

- A Programming Language Grammar

$$\begin{aligned} \langle \text{exp} \rangle ::= & \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle \mid (\langle \text{exp} \rangle) \\ & \mid a \mid b \mid c \end{aligned}$$

From above:

1. The above slides demonstrate that a compiler (especially analysis part) is organized around the syntax of a language.
2. The syntax of a programming language describes the **proper form** of its programs. Note the distinction (but obviously related) with the semantics of the language which defines what its programs **mean**; that is, what each program does when it executes.

What is $9-5+2$? is it 2 or 6?

1. We need a notation to specify syntax

Objective of this session!

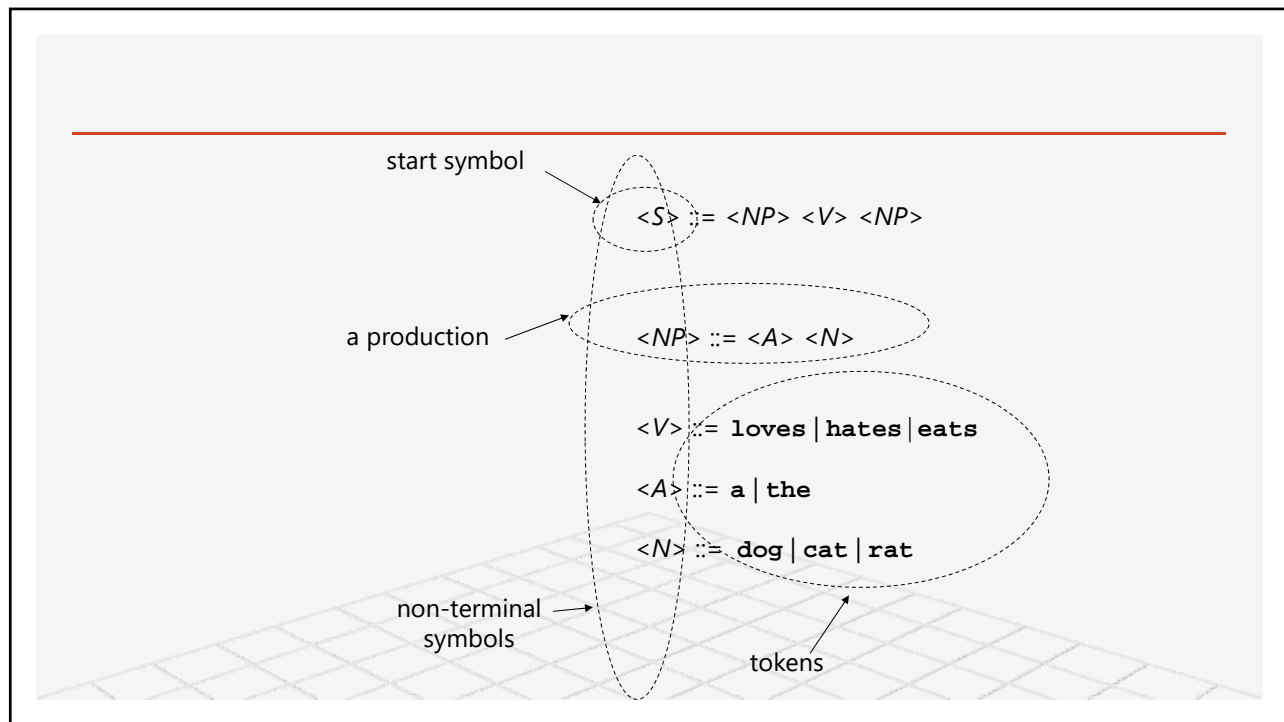
CFG or BNF

Context Free Grammar | Backus-Naur Form

Definition

A context free Grammar is **a set of production rules** that describe (or are used **to generate**) **all possible strings** in a given formal language

Consider the earlier example, we can derive the following:



CFG or BNF

Context Free Grammar | Backus-Naur Form Capabilities:

A CFG or BNF consists of four parts:

1. The set of tokens
2. The set of non-terminal symbols
3. The start symbol
4. The set of productions

- Declare variables
- Initialize variables
- Use variables in a statement: either a **declaration** or an **initialization**

Syntax rules:

```

<statement> ::= <decl> | <initialize>
decl ::= <type> <varName>
type ::= int | char | boolean
varName ::= name | gender
initialize ::= <varName> = <digit>
digit ::= 0 | 1 | 2 | ... 9
  
```

The *tokens* are the smallest units of syntax

Strings of one or more characters of program text

They are atomic: not treated as being composed from smaller parts

The *non-terminal symbols* stand for larger pieces of syntax

They are strings enclosed in angle brackets, as in $\langle NP \rangle$

They are not strings that occur literally in program text

The grammar says how they can be expanded into strings of tokens

The *start symbol* is the particular non-terminal that forms the root of any parse tree for the grammar

The *productions* are the tree-building rules

Each one has a left-hand side, the separator $::=$, and a right-hand side

The left-hand side is a single non-terminal

The right-hand side is a sequence of one or more things, each of which can be either a token or a non-terminal

A production gives one possible way of building a parse tree: it permits the non-terminal symbol on the left-hand side to have the things on the right-hand side, in order, as its children in a parse tree

NB:

When there is more than one production with the same left-hand side, an abbreviated form can be used

The BNF grammar can give the left-hand side, the separator `::=`, and then a list of possible right-hand sides separated by the special symbol `|`

```
<exp> ::= <exp> + <exp> | <exp> * <exp> | ( <exp> )
        | a | b | c
```

Note that there are six productions in this grammar.
It is equivalent to this one:

```
<exp> ::= <exp> + <exp>
<exp> ::= <exp> * <exp>
<exp> ::= ( <exp> )
<exp> ::= a
<exp> ::= b
<exp> ::= c
```

Empty

The special nonterminal `<empty>` is for places where you want the grammar to generate nothing

For example, this grammar defines a typical if-then construct with an optional else part:

```
<if-stmt> ::= if <expr> then <stmt> <else-part>
<else-part> ::= else <stmt> | <empty>
```

Compiler parse trees

To build a parse tree, put the start symbol at the root

Add children to every non-terminal, following any one of the productions for that non-terminal in the grammar

Done when all the leaves are tokens

Read off leaves from left to right—that is the string derived by the tree

Given a string, based on the CFG definition, can generate the parse tree for parsing strings as part of compilation

An example

Consider expressions consisting of digits and plus and minus signs

e.g., strings such as $9-5+2$, $3-1$, or 7

Since a plus or minus sign must appear between two digits, we refer to such expressions as "lists of digits separated by plus or minus signs."

A grammar to describe the syntax of these expressions can take the following form:

Example contd..

list \rightarrow list + digit

list \rightarrow list - digit

list \rightarrow digit

digit: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

These are 4 Productions of the grammar

The bodies of the three productions with nonterminal list as head equivalently can be grouped as:

list \rightarrow list + digit | list - digit | digit

The terminals of the grammar are the symbols :

+ - 0 1 2 3 4 5 6 7 8 9

The nonterminals are **list** and **digit**, with list being the start symbol because its productions are given first

A production is for a nonterminal **if** the nonterminal is the head of the production

A string of terminals is a sequence of zero or more terminals

The string of zero terminals, written as **e**, is called the empty string

Technically, **e** can be a string of zero symbols from any alphabet (where an alphabet represents a collection of symbols)

Derivations (1)

The idea is to form or identify the language defined by a grammar

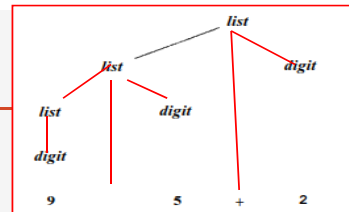
A grammar derives strings by beginning with the start symbol and repeatedly replacing a nonterminal by the body of a production for that nonterminal

The terminal strings that can be derived from the start symbol form the language defined by the grammar

Derivations (2)

For example:

Deduce that $9-5+2$ is a list



list \rightarrow list + digit -----2.1
 list \rightarrow list - digit -----2.2
 list \rightarrow digit -----2.3
 digit: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- 9 is a list by production (2.3), since 9 is a digit.
- $9-5$ is a list by production (2.2), since 9 is a list and 5 is a digit.
- $9-5+2$ is a list by production (2.1), since $9-5$ is a list and 2 is a digit.

Derivations (2)

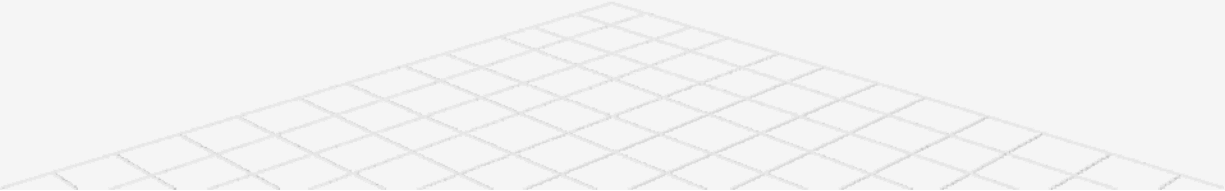
We create a derivation for a call to a method

E.g. **max(a,b)** or **printDetails()**

(describe)

Productions

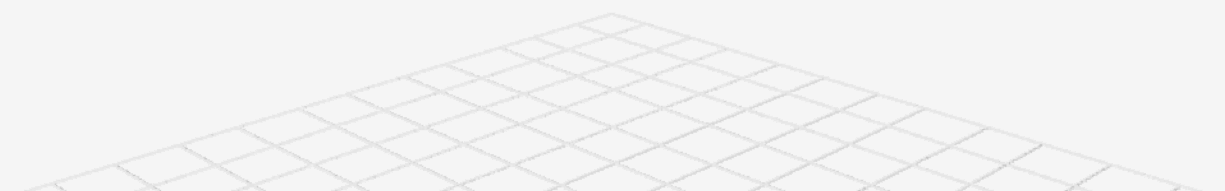
(note the components of a production)



Description [for a method call in java]

A call to a method consists of a method name to identify the method with the parameters enclosed within parentheses, e.g. `max(x,y)` of function `max` with parameters `x` and `y`

Note that for such calls, an empty list of parameters may be found between the terminals `(` and `)`



Sample productions:

```
call -> id ( optparams )  
optparams -> params | e  
params —> params , param | param
```

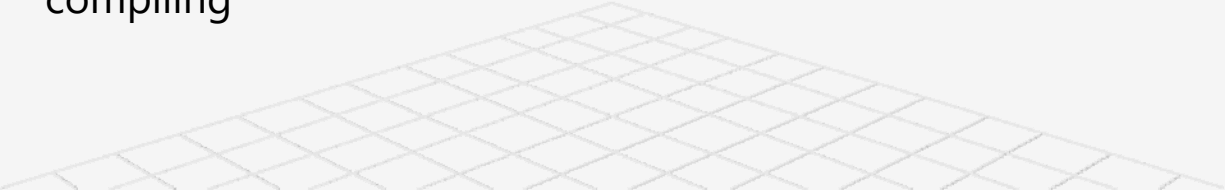
Note that the second possible body for optparams ("optional parameter list") is e, which stands for the empty string of symbols. That is, optparams can be replaced by the empty string, so a call can consist of a function name followed by the two-terminal string `()`. Notice that the productions for params are analogous to those for list in the previous example

We have not shown the productions for param, since parameters are really arbitrary expressions.

Conclusion

Parsing is the problem of taking a string of terminals and figuring out how to derive it from the start symbol of the grammar, and if it cannot be derived from the start symbol of the grammar, then reporting syntax errors within the string.

Parsing is one of the most fundamental problems in all of compiling



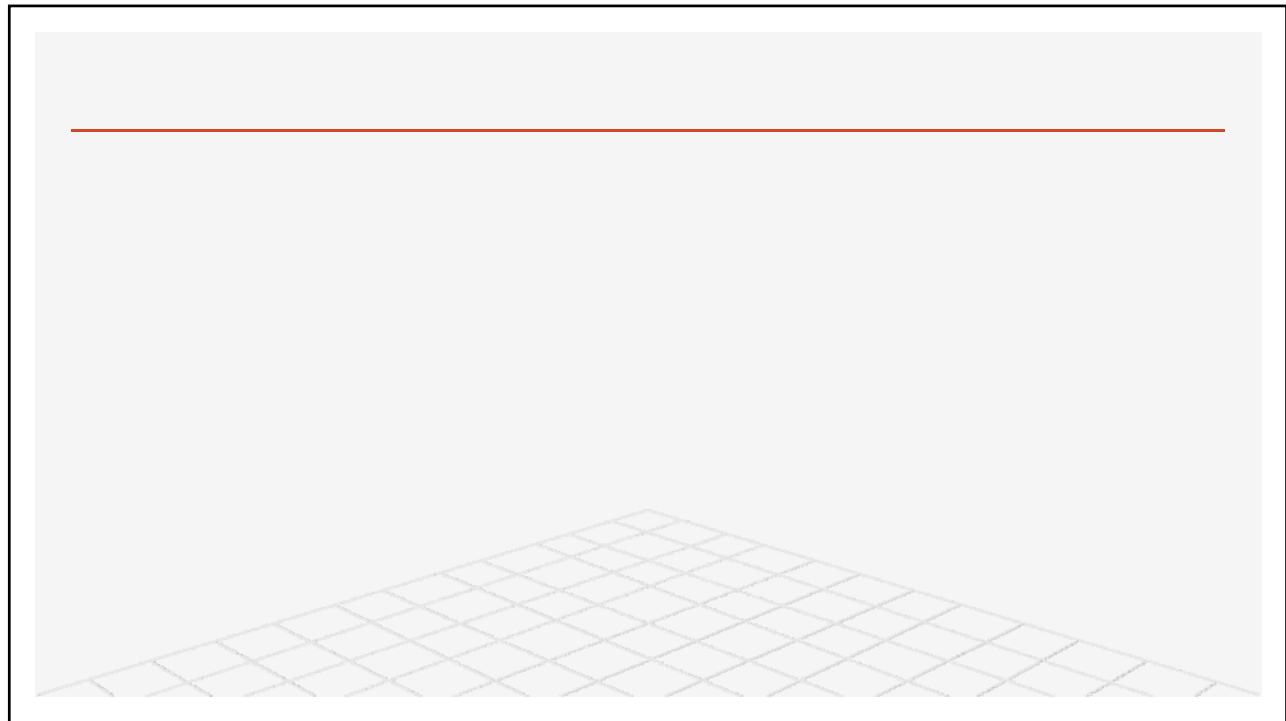
Designing a Simple Compiler (continued...)

Thank You

Reflection: ??

Next →

- Agenda
- Parse trees
- Ambiguity problem
- Associativity of operators
- Precedence of operators
- Example grammar for a valid/legal statement



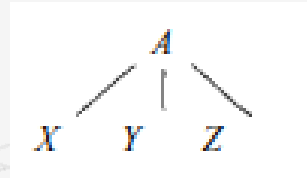
Agenda

- Parse trees
- Ambiguity problem
- Associativity of operators
- Precedence of operators
- Example grammar for a valid/legal statement

Parse Trees

A parse tree pictorially shows how the start symbol of a grammar derives a string in the language

If nonterminal A has a production $A \rightarrow XYZ$, then a parse tree may have an interior node labeled A with three children labeled X , Y , and Z , from left to right



Given a context-free grammar, a parse tree according to the grammar is a tree with the following properties:

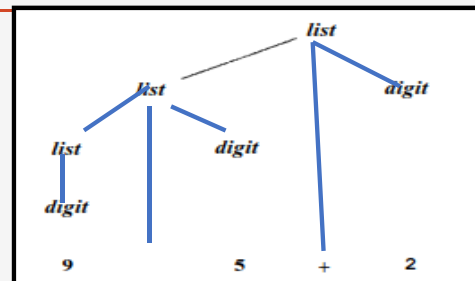
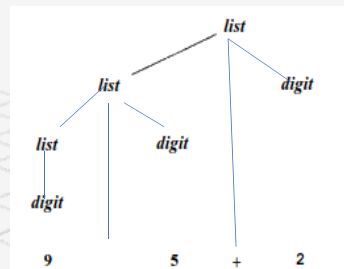
1. The root is labeled by the start symbol.
2. Each leaf is labeled by a terminal or by ϵ (empty string).
3. Each interior node is labeled by a nonterminal.
4. If A is the nonterminal labeling some interior node and X_1, X_2, \dots, X_n are the labels of the children of that node from left to right, then there must be a production $A \rightarrow X_1 X_2 \dots X_n$. Here, X_1, X_2, \dots, X_n each stand for a symbol that is either a terminal or a nonterminal.

As a special case, if $A \rightarrow \epsilon$ is a production, then a node labeled A may have a single child labeled ϵ .

Recall the productions:

```
list -> list + digit -----2.1
list -> list - digit -----2.2
list -> digit -----2.3
digit: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

and consequently the derivation of 9-5+2 illustrated by the parse tree:



NB:

Each node in the tree is labeled by a grammar symbol. An interior node and its children correspond to a production; the interior node corresponds to the head of the production, the children to the body

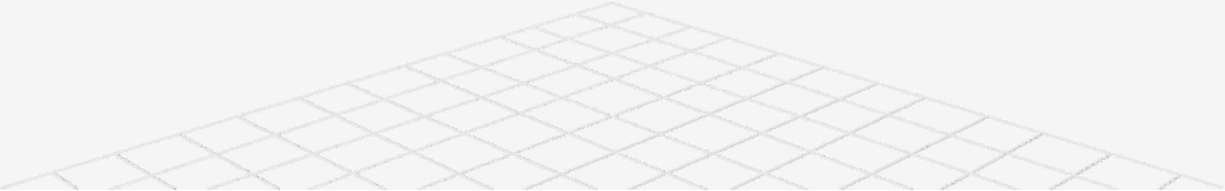
From left to right, the leaves of a parse tree form **the yield of the tree**, which is the string generated or derived from the nonterminal at the root of the parse tree

Use of tree notation for compiler parsing process:

Any tree imparts a natural left-to-right order to its leaves, based on the idea that if X and Y are two children with the same parent, and X is to the left of y, then all descendants of X are to the left of descendants of Y

Another **definition** of the language generated by a grammar is as **the set of strings that can be generated by some parse tree**

The process of finding a parse tree for a given string of terminals is called parsing that string. It will either pass (is a valid string) or fail (is not a valid string in the language)



Ambiguity

Recall the question:

What is 9-5+2 ? is it 2 or 6?

Ex. Generate the parse trees implied by the two possible answers!

How would we make it possible?

Ambiguity problem:

A grammar can have more than one parse tree generating a given string of terminals. Such a grammar is said to be ambiguous

To show that a grammar is ambiguous, all we need to do is find a terminal string that is the yield of more than one parse tree.

Solving Ambiguity of parse trees

Tree structure is not enough – a string with more than one parse tree usually has more than one meaning

We need to design unambiguous grammars for compiling applications, OR use **ambiguous grammars with additional rules to resolve the ambiguities**

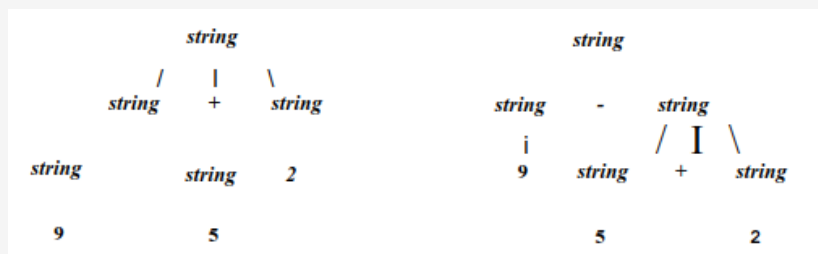


Suppose we used a single nonterminal string and did not distinguish between digits and lists

We could have written the grammar as:

string \rightarrow string + string | string - string | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Now we can have two trees for 9-5+2 correspond to the two ways of parenthesizing the expression: (9-5)+2 and 9 - (5 + 2)



The earlier definition using list, does not permit this interpretation

Associativity of Operators

By convention, $9+5+2$ is equivalent to $(9+5)+2$ and $9-5-2$ is equivalent to $(9-5)-2$

When an operand like 5 has operators to its left and right, conventions are needed for deciding which operator applies to that operand

We say that the operator $+$ associates to the left, because an operand with plus signs on both sides of it belongs to the operator to its left

In most programming languages the four arithmetic operators, addition, subtraction, multiplication, and division are left-associative.

Some common operators such as exponentiation are right-associative

As another example, the assignment operator $=$ in C and its descendants is right associative; that is, the expression $a=b=c$ is treated in the same way as the expression $a=(b=c)$.

Strings like $a=b=c$ with a right-associative operator are generated by the following grammar:

$\text{right} \rightarrow \text{letter} = \text{right} \mid \text{letter}$

$\text{letter} \rightarrow a \mid b \mid \dots \mid z$

contrast between a parse tree for a left-associative operator like $-$ and a parse tree for a right-associative operator like $=$

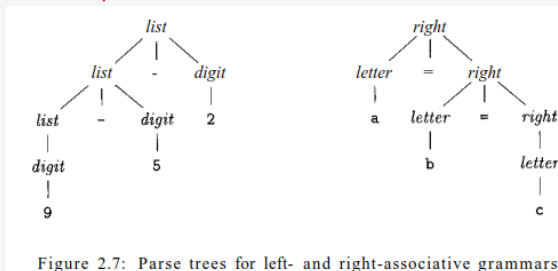


Figure 2.7: Parse trees for left- and right-associative grammars

The parse tree for $9-5-2$ grows down towards the left, whereas the parse tree for $a=b=c$ grows down towards the right.

Reflection

The car knocked the driver who saw the Giraffe overlooking the valley

Are the verbs left or right associative?

Precedence of Operators

What is

$9+5*2$

is it 28 or 19?

The two possible interpretations of this expression are: $(9+5)*2$ or $9+(5*2)$

The associativity rules for + and * apply to occurrences of the same operator, so they do not resolve this ambiguity. Rules defining the relative precedence of operators are needed when more than one kind of operator is present.

We say that * has higher precedence than + if * takes its operands before + does

In ordinary arithmetic, multiplication and division have higher precedence than addition and subtraction

Therefore, 5 is taken by * in both $9+5*2$ and $9*5+2$; i.e., the expressions are equivalent to $9+(5*2)$ and $(9*5)+2$, respectively

A grammar for arithmetic expressions can be constructed from a table showing the associativity and precedence of operators. We start with the four common arithmetic operators and a precedence table, showing the operators in order of increasing precedence. Operators on the same line have the same associativity and precedence:

left-associative: + -

left-associative: * /

We create two non-terminals **expr** and **term** for the two levels of precedence, and an extra non-terminal **factor** for generating basic units in expressions. The basic units in expressions are presently digits and parenthesized expressions

factor \rightarrow digit | (expr)

Now consider the binary operators, * and /, that have the highest precedence. Since these operators associate to the left, the productions are similar to those for lists that associate to the left.

term \rightarrow term * factor
 | term / factor
 | factor

Similarly, expr generates lists of terms separated by the additive operators.

expr \rightarrow expr + term
 | expr - term
 | term

The resulting grammar is therefore

$\text{expr} \rightarrow \text{expr} + \text{term} \mid \text{expr} - \text{term} \mid \text{term}$

$\text{term} \rightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor}$

$\text{factor} \rightarrow \text{digit} \mid (\text{expr})$

NB:

With this grammar, an expression is a list of terms separated by either + or – signs, and a term is a list of factors separated by * or / signs.

Notice that any parenthesized expression is a factor, so with parentheses we can develop expressions that have arbitrarily deep nesting (and arbitrarily deep trees).

Reflection

How does this discussion “relate” to Compiler Design and Construction? We look at example grammars of valid statements

Example: Grammar of valid statements

Keywords allow us to recognize statements, since most statements begin with a keyword or a special character. Exceptions to this rule include assignments and procedure calls.

Consider this grammar that attempts to define legal Java statements :

```
stmt —> id = expression ;
        | if ( expression ) stmt
        | if ( expression ) stmt else stmt
        | while ( expression ) stmt
        | do stmt while ( expression ) ;
        | { stmts }
stmts -> stmts stmt
```

```
public void addPatient(){
    int count=0;
    while (count!=2) {
        Patient access = new Patient();
        access.patientDetails();
        System.out.println("Add new patient.");
        count++;
    }
    options();
}
```

is the highlighted code valid?

In the first production for stmt, the terminal id represents any identifier. The productions for expression are not shown.

The assignment statements specified by the first production are legal in Java, although Java treats = as an assignment operator that can appear within an expression. For example, Java allows `a = b = c`, which this grammar does not.

The nonterminal stmts generates a possibly empty list of statements. The second production for stmts generates the empty list `e`.

The first production generates a possibly empty list of statements followed by a statement.

The placement of semicolons is subtle; they appear at the end of every body that does not end in stmt. This approach prevents the build-up of semicolons after statements such as if- and while-, which end with nested sub-statements.

When the nested substatement is an assignment or a do-while, a semicolon will be generated as part of the substatement.

Assignment / Exercise

1. Given the context-free grammar

$S \rightarrow SS + \mid SS * \mid a$

- i) Show how the string **$aa+a^*$** can be generated by this grammar.
 - ii) Construct a parse tree for this string.
2. Given the grammar below to generate binary strings:

$num \rightarrow \bullet 11 \mid 1001 \mid num0 \mid num num$

- i) Generate 5 possible valid strings in the grammar
- ii) What is special about the equivalent decimal numbers represented by the binary strings?
- iii) Can the number 108 be represented in this grammar?
- iv) Draw the parse tree for the binary string representing 108.

Thank you

For next two chapters read about:

Syntax directed translation

Attributes

Infix and Postfix notations

Translation schemes

Tree traversals

Parsing

Recursive descent

Top-Down parsing

Predictive parsing