CS316. 4/2/2014 ①

TinyJ Assignment 1 email*: Do the installation before next Monday.
* With assigned reading and exercises on syntax and method overriding in C++.

Syntax of Programming Languages.

The syntax of programming language is usually specified in terms of

① Units called tokens, such as: while, for, ~~IDENTIFY~~ IDENTIFIER (x, while.for)
Unsigned-int-literal. (72. 47193) ; (>= - (java/C++ tokens).

some tokens (e.g. IDENTIFIER and UNSIGNED-INT-LITERAL in Java/C++) have more than one instance.

Different instances of the same token are semantically different (i.e. have different meanings) but syntax specification written in BNF or some similar notation such as EBNF will not distinguish between them.

Note: It is also possible for the same instance of token to have more than one spelling,

Apple 73.0
ApplE 73.00

A token (like while or for in java/C++) that looks like an identifier but cannot be used as identifiers are called reserved words. [Some languages, such as Fortran and Lisp, has no reserved words, in such Languages words that have special meaning (like IF in Lisp) are called keywords [(defun nil (if (+ if 1)) is a legal lisp function definition] Some author refer to ~~reversed~~ reversed words as keywords.

② Certain chooren language constructs (e.g. <while stmt>, <stmt> or <expression>). In a syntax speciafication of an entire Language (rather just some part of a language) one of these constructs represents acceptable input for a complier or interpreter — e.g. it might be called <program>.

The purpose of a syntax specification is to answer the following 2 questions:

Q1: For each kind of token exactly which sequences of characters constitute a valid instance of that token.

Q2 For each of the language constructs of ①, exactly which sequences of tokens constitute a syntactically valid instance of that language construct?

[Reminder] A syntactically valid $\left(\begin{smallmatrix} x \\ y \end{smallmatrix} = 4 * (y-w)\right)$ instance of a contruct need not be a valid instance — it may contain errors such as ~~understand~~ udercelared ~~inde~~ identifier type ~~mismatch~~ mismath or devided by zero.

Q1 and Q2 are usually answered using context-free grammer notation/BNF or same vanant of this (such as EBNF)

Context-Free grammer Notation is a relatively concise (and yet completely price) notation ~~for specifiy~~ specifying a (possible infinite) set of finite sequences of symbols. The symbols are called the Terminal of the grammer.

In ~~instance~~ answering Q1 (for tokens that have many instances, like IDENTIFIER or Unsigned-float-litera) we can use a grammer whose ~~term~~ terminals are charactor; [Regular expression notation is another way to do this]

In answring Q2, we can ~~use~~ use a grammer whose terminal are tokens

It is also possible to use a single grammar, whose ~~s~~ terminals are character, ~~to~~ answer ~~both~~ both Q1 and Q2.

Context-free grammer notation was ~~intended~~ by Chamsky, who invented a hierarchy of 4 types of grammer for specifying the syntax of English and other similar natural languages. Context-free grammars are Type 2 in Chamsky's Hierarchy.

the idea of a context-free grammer was independently reinvented by Backus a few years later. Backus proposed the use of his grammar notation for specifying the syntax of Algol 60.

the Algo 60 Report (which specified the language Algol 60 in a way that was widdy admired by computer scientists) adopted Backus's proposal.

Naur was the editor of the Algol 60 Report. Backus's notation is now call BNF (Backus Naur Form)

<E> ::= <E> + <T> (BNF)

E → E + T. (not BNF)

Like many authors today(*) we will use BNF to mean some ~~commonly~~ commonly used notation for writing a context-free grammar.

(*) But unlike Sethi: cf Fig 2.6 on P42. and Fig 2.10 on P46.

Sethi would not consider Fig 2.6 to be BNF. But we will consider it to be BNF.

Ex of a Grammar.

UNSIGNED-FLOAT-LITERAL tokens

can be specified in BNF as follows: (assuming they have the form: ~~one or more digits. or~~

$\underbrace{\text{one or more digits}}_{\text{ip (integer part)}}$. $\underbrace{\text{one or more digits}}_{\text{f (fraction)}}$ ← fraction. (Fig 2.3 on P36.)

(1) ufpl ::= ip . f

(2) ip ::= d ip ::= d | ip d. ("d" = "digit").

(3) ip ::= ip d (5)

(4) f ::= d | d f

d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 (15) — undefined constant symbols

The 11 symbols 0, 1, …, 9 are the terminals of this grammar.

The 4 symbols ufpl, ip, f, d are the nonterminals of this grammar.

↓

variables, each of which denotes of finite sequences of terminals.

There are 15 rules called productions ~~each~~ each of the form

N ::= α

N: a single nonterminal

α: a sequence of zero or more terminals and/or nonterminals.

The nonterminal ufpl in the starting nonterminal

This nonterminal denotes the set of sequences that the grammer is intended to define — this set is called the language of / generated by the grammer. Unless otherwise indicated, the starting nonterminal ~~on the left~~ is the nonterminal on the left side of the first production.