Language Description: Syntax

(Based on [Sethi 1996])

Yih-Kuen Tsay

Department of Information Management
National Taiwan University

Language Description

- Clear and complete descriptions of a language are needed by *programmers*, *implementers*, and even language *designers*. Nowadays, a language is typically described by a combination of formal syntax and informal semantics.
- The *syntax* of a language specifies how programs in the language are built up; the *semantics* of the language specifies what programs mean.
- Organization of language descriptions:
  - *Tutorials*
  - *Reference Manuals*
  - *Formal Definitions*

The Two Layers of Syntax

The formal syntax of a programming language usually consists of two layers:

- Lexical Layer
  The lexical syntax of a language corresponds to the spelling of words in English. It governs the formation of *numbers*, *symbols*, *identifiers*, *keywords*, etc.
- Grammar/Syntactic Layer
  The syntax of a language is described by a grammar, in particular a context-free grammar. Notations for writing grammars include BNF, Extended BNF (EBNF), and syntax charts.

Notations for Expressions

- Expressions such as $a + b * c$ have been in use for centuries and were a starting point for the design of programming languages.
- For example,

$$\frac{-b + \sqrt{b^2 - 4 * a * c}}{2 * a}$$

can be written in Fortran as

$$(-b + sqrt(b ** 2 - 4.0 * a * c))/(2.0 * a).$$

Notations for Expressions (cont.)

Programming languages use a mix of notations:

- Prefix Notation (Polish Notation): the operator is written first, followed by its operands, as in $+\ a\ b$.
- Postfix Notation: the operator is written last, preceded by its operands, as in $a\ b\ +$.
- Infix Notation: the operator is written between its operands, as in $a + b$.
- Mixfix Notation: some operations do not fit neatly into the prefix, postfix, and infix classification; one example is:

$$\textbf{if}\ a > b\ \textbf{then}\ a\ \textbf{else}\ b$$

Prefix Notation

- An expression in prefix notation is written as follows:
  - The prefix notation for a constant or variable is the constant or variable itself.
  - The application of a binary operator **op** to subexpressions E_1 and E_2 is written in prefix notation as **op** $E_1\ E_2$.
  - The application of a k-ary operator $\mathbf{op}^k$ to subexpressions E_1, E_2, ..., E_k is written in prefix notation as $\mathbf{op}^k\ E_1\ E_2 \cdots E_k$.
- An advantage of prefix notation is that it is easy to decode (parse) during a left-to-right scan of an expression.
  Examples:
  - $+\ x\ y$ (the sum of x and y)
  - $*\ +\ x\ y\ z$ (the product of $+\ x\ y$ and z)
  - $*\ +\ 20\ 30\ 60\ (= *\ 50\ 60 = 3000)$
  - $*\ 20\ +\ 30\ 60\ (= *\ 20\ 90 = 1800)$

Postfix Notation

- An expression in postfix notation is written as follows:
  - The postfix notation for a constant or variable is the constant or variable itself.
  - The application of a binary operator **op** to subexpressions E_1 and E_2 is written in postfix notation as $E_1\ E_2$ **op**.
  - The application of a k-ary operator $\mathbf{op}^k$ to subexpressions E_1, E_2, ..., E_k is written in postfix notation as $E_1\ E_2 \cdots E_k\ \mathbf{op}^k$.
- An advantage of postfix expressions is that they can be mechanically evaluated with the help of a *stack*.
  Examples:
  - $x\ y\ +$ (the sum of x and y)
  - $x\ y\ +\ z\ *$ (the product of $x\ y\ +$ and z)
  - $20\ 30\ +\ 60\ *\ (= 50\ 60\ *\ = 3000)$
  - $20\ 30\ 60\ +\ *\ (= 20\ 90\ *\ = 1800)$

Infix Notation

- In infix notation, (binary) operators appear between their operands.
- An advantage of infix notation is that it is familiar and hence easy to read.
- Additional concepts, namely *precedence* and *associativity*, needed for resolving ambiguities.
  - Is $a + b * c$ equal to $a + (b * c)$, or $(a + b) * c$?
  - Is $4 - 2 - 1$ equal to $(4 - 2) - 1$, or $4 - (2 - 1)$?
- *Parentheses* may be used to make explicit the intended precedence and associativity.

Infix Notation (cont.)

- Precedence
  - An operator at a higher *precedence level* takes its operands before an operator at a lower precedence level.
  - For example, assuming as usual that the operator $*$ has higher precedence than $+$,

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

- Associativity
  - An operator is *left associative* if subexpressions containing multiple occurrences of the operator are grouped from left to right. For example,

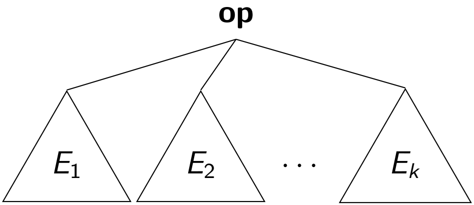
$$4 - 2 - 1 = (4 - 2) - 1 = 2 - 1 = 1.$$

  - An operator is *right associative* if subexpressions containing multiple occurrences of the operator are grouped from right to left. For example,

$$2^{3^4} = 2^{(3^4)} = 2^{81}.$$

Abstract Syntax

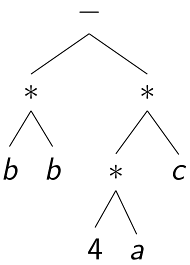
- The *abstract syntax* of a language identifies the meaningful components of each construct in the language.
- The meaningful components of an expression are the operators and their operands in the expression. Their structure can be conveniently represented by a tree, where an operator and its operands are represented by a node and its children (subtrees).



- Trees showing the operator/operand structure of an expression are called *abstract syntax trees*, because they show the syntactic structure of an expression independent of the notation in which the expression was originally written.

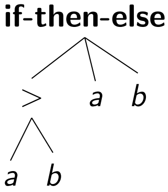
Abstract Syntax (cont.)

An abstract syntax tree for $b * b - 4 * a * c$:



Abstract Syntax (cont.)

An abstract syntax tree for **if** $a > b$ **then** a **else** b:



Lexical Syntax

- Keywords like **if** and symbols like $<=$ are treated as units in a programming language, just as words are treated as units in English.
- The syntax of a programming language is specified in terms of units called *tokens* or *terminals*.
- A *lexical syntax* for a language specifies the correspondence between the written representation of the language and the tokens or terminal in a grammar for the language.
  - Expression: `b * b - 4 * a * c`
  - Token sequence:
    $\mathbf{name}_b * \mathbf{name}_b - \mathbf{number}_4 * \mathbf{name}_a * \mathbf{name}_c$
- Informal description usually suffices for specifying the lexical syntax of a language; real numbers are one possible exception.

Lexical Syntax (cont.)

binary operation	symbol	Pascal	C, C++, Java
less than	$<$	`<`	`<`
less than or equal to	$\leq$	`<=`	`<=`
equal	$=$	`=`	`==`
not equal	$\neq$	`<>`	`! =`
greater than	$>$	`>`	`>`
greater than or equal to	$\geq$	`>=`	`>=`
add	$+$	`+`	`+`
subtract	$-$	`-`	`-`
multiply	$*$	`*`	`*`
divide, for reals	$/$	`/`	`/`
divide, for integers	**div**	`div`	`/`
remainder, for integers	**mod**	`mod`	`%`

Context-Free Grammars

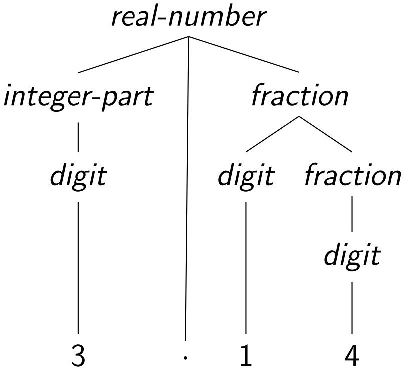
- The *concrete syntax* of a language describes its written representation, including lexical details such as the placement of keywords and punctuation marks.
- Context-free grammars are a formalism for specifying concrete syntax.
- A *context-free grammar*, or simply *grammar*, has four parts:
  - A set of tokens or terminals.
  - A set of nonterminals.
  - A set of productions (production rules) for identifying the components of a construct. Each production has a nonterminal as its left side and a string over the sets of terminals and nonterminals as its right side.
  - A nonterminal chosen as the starting nonterminal.

Context-Free Grammars (cont.)

A CFG in Backus-Naur Form (BNF) for reals:

$$
\begin{array}{rcl}
\langle \text{real-number} \rangle & ::= & \langle \text{integer-part} \rangle . \langle \text{fraction} \rangle \\
\langle \text{integer-part} \rangle & ::= & \langle \text{digit} \rangle \mid \langle \text{integer-part} \rangle \langle \text{digit} \rangle \\
\langle \text{fraction} \rangle & ::= & \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{fraction} \rangle \\
\langle \text{digit} \rangle & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

Parse Trees


real-number
integer-part
fraction
digit
digit
fraction
digit
3
.
1
4

Parse Trees (cont.)

- The productions in a grammar are rules for building strings of tokens.
- A *parse tree* shows how a string can be built:
  - Each leaf is labeled with a terminal or $\langle$*empty*$\rangle$.
  - Each nonleaf node is labeled with a nonterminal.
  - The label of a nonleaf node is the left side of some production and the labels of the children of the node, from left to right, form the right side of that production.
  - The root is labeled with the starting nonterminal.
- A parse tree *generates* the string formed by reading the terminals at its leaves from left to right.

Syntactic Ambiguity

- A grammar for a language is *syntactically ambiguous*, or simply *ambiguous*, if some string in its language has more than one parse tree.
- Programming languages can usually be described by unambiguous grammars.
- If ambiguities exist, they are resolved by establishing conventions that rule out all but one parse tree for each string.
- Example ambiguous grammar:

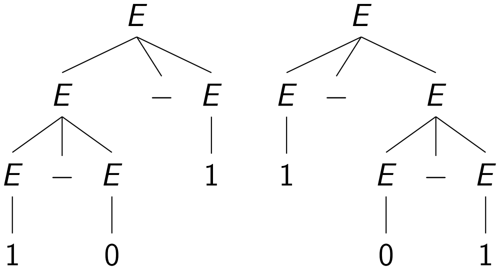
$$E ::= E - E \mid 0 \mid 1$$

The string $1 - 0 - 1$, for instance, has two parse trees.

Syntactic Ambiguity (cont.)

$$E ::= E - E \mid 0 \mid 1$$

Two parse trees for $1 - 0 - 1$:



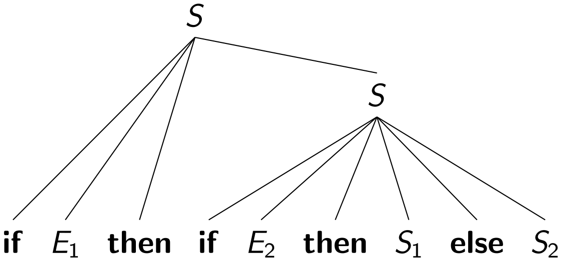
Dangling Else

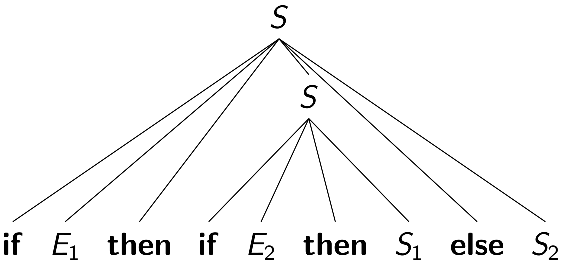
- A well-known example of syntactic ambiguity is the *dangling-else* ambiguity.
- Example ambiguous grammar:

$$S \; ::= \; \textbf{if } E \textbf{ then } S$$
$$S \; ::= \; \textbf{if } E \textbf{ then } S \textbf{ else } S$$

- The string "**if** E_1 **then if** E_2 **then** S_1 **else** S_2 " has two parse trees; the **else** can be matched with either **if**.
- The dangling-else ambiguity is typically resolved by matching an **else** with the nearest unmatched **if**.

Dangling Else (cont.)





Derivations

- A *derivation* consists of a sequence of strings, beginning with the starting nonterminal. Each successive string is obtained by replacing a nonterminal by the right side of one of its productions. A derivation ends with a string consisting entirely of terminals.
- Example:

$$
\begin{array}{rcl}
\textit{real-number} & \Rightarrow & \textit{integer-part} \;.\; \textit{fraction} \\
 & \Rightarrow & \textit{integer-part digit} \;.\; \textit{fraction} \\
 & \Rightarrow & \textit{digit digit} \;.\; \textit{fraction} \\
 & \Rightarrow & 2 \; \textit{digit} \;.\; \textit{fraction} \\
 & \Rightarrow & 2\;1\;.\; \textit{fraction} \\
 & \Rightarrow & 2\;1\;.\; \textit{digit fraction} \\
 & \Rightarrow & 2\;1\;.\;8\; \textit{fraction} \\
 & \Rightarrow & 2\;1\;.\;8\; \textit{digit} \\
 & \Rightarrow & 2\;1\;.\;8\;9
\end{array}
$$

Parse Trees and Abstract Syntax Trees

- A grammar for a language is usually designed to reflect the abstract syntax.
- A well-designed grammar can make it easy to pick out the meaningful components of a construct.
- With a well-designed grammar, parse trees are similar enough to abstract syntax trees that the grammar can be used to organize a language description or a program that exploits the syntax.

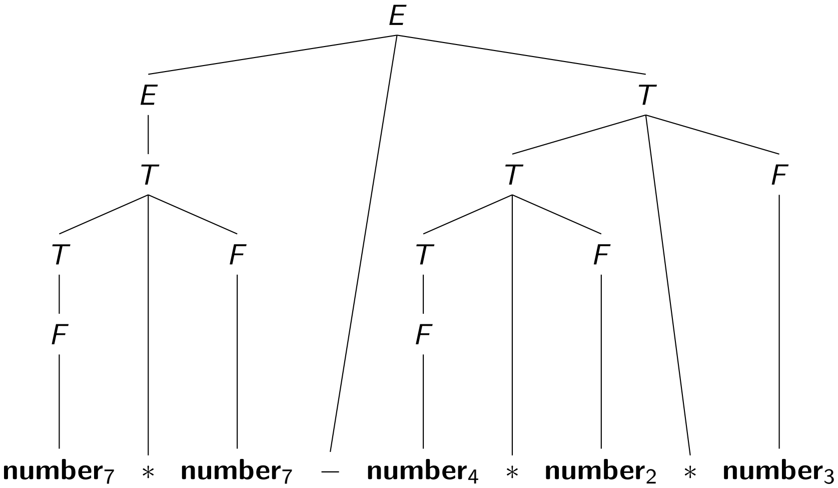
A Grammar for Arithmetic Expressions

$$
\begin{array}{rcl}
E & ::= & E + T \mid E - T \mid T \\
T & ::= & T * F \mid T / F \mid F \\
F & ::= & \textbf{number} \mid \textbf{name} \mid (E)
\end{array}
$$

In BNF,

$$
\begin{array}{rcl}
\langle\text{experssion}\rangle & ::= & \langle\text{experssion}\rangle + \langle\text{term}\rangle \\
 & \mid & \langle\text{experssion}\rangle - \langle\text{term}\rangle \\
 & \mid & \langle\text{term}\rangle \\
\langle\text{term}\rangle & ::= & \langle\text{term}\rangle * \langle\text{factor}\rangle \\
 & \mid & \langle\text{term}\rangle / \langle\text{factor}\rangle \\
 & \mid & \langle\text{factor}\rangle \\
\langle\text{factor}\rangle & ::= & \textbf{number} \mid \textbf{name} \mid (\langle\text{experssion}\rangle)
\end{array}
$$

A Grammar for Arithmetic Expressions (cont.)



Associativity and Precedence

In an increasing order of precedence,

operator	assocaitivity
$:=$	right associative
$+, -$	left associative
$*, /$	left associative

$$
\begin{array}{lcl}
A & ::= & E := A \mid E \\
E & ::= & E + T \mid E - T \mid T \\
T & ::= & T * F \mid T / F \mid F \\
F & ::= & \textbf{number} \mid \textbf{name} \mid (E)
\end{array}
$$

Extended BNF (EBNF)

IM NTU

- Below is an EBNF version of the grammar for arithmetic expressions:

$$\begin{array}{rcl}\langle \textit{expression} \rangle & ::= & \langle \textit{term} \rangle \ \{(`+' \mid `-') \ \langle \textit{term} \rangle\} \\ \langle \textit{term} \rangle & ::= & \langle \textit{factor} \rangle \ \{(`*' \mid `/') \ \langle \textit{factor} \rangle\} \\ \langle \textit{factor} \rangle & ::= & `(' \ \langle \textit{expression} \rangle \ `)' \mid \textbf{name} \mid \textbf{number}\end{array}$$

- Conventions in EBNF:
    - Braces, { and }, represent zero or more repetitions.
    - Brackets, [and], represent an optional construct.
    - A vertical bar, |, represents a choice.
    - Parentheses, (and), are used for grouping.