

Syntax

Syntax in English vs. in Programming Languages

- A sentence in English can be composed of an adjective followed by a noun followed by a verb followed by an adverb.
- There are many other forms of sentences in English. But for now, we just use this simple form.

sentence → *adjectivenounverbadverb*

adjective → *good|delicious|green|beautiful*

noun → *pens|rice|sun|ideas*

verb → *eats|sleep|write|jumps*

adverb → *deeply|well|furiously|calmly*

Greenideassleepfuriously.

- Good, delicious, green, and beautiful are adjective; Pen, rice, sun, and ideas are nouns; verb can be chosen from eat, sleep, write, and jump; Adverb can be selected from deeply, well, furiously, and calmly.
- Here we define a simple grammar of sentences in English. Using this grammar, we can compose a sentence like “Green ideas sleep furiously.” This sentence is grammatically correct, but does not make any sense.
- In [programming languages](#), we can think [syntax as the grammar](#) in English, and [semantics are the meaning](#) in English.
- The [syntax](#) of a programming language is [defined by a grammar](#).
- The [syntax](#) of a programming language is a [precise description](#) of [all its grammatically correct programs](#).

Categories of grammars

- Noam Chomsky defined [four categories](#) of grammars: [regular](#), [context-free](#), [context-sensitive](#), and [unrestricted](#).
- In all of these grammars, the process of [analyzing a string](#) involves [applying a single rule to a single location](#) in [serial order](#).
- There are other grammars that do not follow Chomsky’s hierarchy, such as L-systems, because they [apply one or more rules in parallel to all instances of a symbol](#) in the string. However, L-systems are not suited for compilation of a program; they are intended as grammars to generate an output string.

- Chomsky's grammar categories are intended to support **deconstruction** of a **string into nonterminal symbols** to **evaluate** whether it is a **correct** program and enable **deduction** of its **semantics**.
- Regular** - corresponds to a standard **finite-state automata**; its rules are *all of the form* $A \rightarrow \omega B$ or *all of the form* $A \rightarrow B\omega$, where A and B are **single nonterminal symbols**, and ω is a **valid string of terminal symbols**. In both cases, B can be null if ω is not null. The **ordering of B and ω** **determine** whether the language is a **right regular** grammar or a **left regular** grammar.
- Context-free** - corresponds to a **pushdown automata**; its rules are of the form $A \rightarrow \omega$, where A is a **single nonterminal symbol** and ω is a **valid string** and can be *mix of terminal and nonterminal symbols, including A* . Context-free grammars allow **recursive syntactic** structures.
- Context-sensitive** - has rules of the $\alpha \rightarrow \beta$, where both α and β can be **strings containing terminals and nonterminal symbols**; the **length of β** must be **greater than or equal to α** so that the *program cannot shrink*. A context-sensitive grammar is **undecidable**, in the sense that we *cannot necessarily decide if a program is valid* given the grammar.
- Unrestricted** - **identical to a context-sensitive** grammar, but **removes the length restriction** on β .
- Programming languages use **context-free** grammars because they are the **most powerful** grammars that are **decidable**.
 - A regular grammar, for example, cannot represent the need for opening and closing brackets in an expression. (cannot match any level of nested balanced parentheses)
 - An arbitrary context-sensitive grammar is unlikely to be compilable in reasonable time and may not have an unambiguous interpretation.

Context-free Grammar

- LL(k)**
 - Most programming languages are, in fact, part of a **subset of context-free** grammars called LL(k) grammars, which means the **grammar can be parsed in a single pass** through the code **by looking ahead k symbols**.
 - LL(k) **parses** the input **from Left to right**, performing the **Leftmost derivation** of the string.
 - Not all context-free grammars are LL(k) grammars, but all **LL(k) grammars** can be **converted into a parse tree** in a linear time in a single pass.
 - An LL(k) grammar **cannot contain left recursion** (e.g. $A \rightarrow A + B$)
 - An LL(k) parser begins with the start, or program symbol and **builds the parse tree from the top down**, ending with the leaves as rules that convert to the terminal symbols of input string.
- LR(k)**
 - An **alternative to LL(k)** grammars. A **subset of context-free** grammars that **allow left recursion, but not right recursion** (e.g. $A \rightarrow B + A$).
 - An LR(k) parser **looks ahead k symbols**, **parses** the input **from Left to right**, performing the **Rightmost derivation** of the string, and starts with the leaves of the parse tree and **builds up the tree from the bottom**.

- Most production compilers, such as gcc, are recursive descent LL(1) compilers.
- A context-free grammar consists of **four components**:
 - **T** : A set of **terminal** symbols, which defines the **alphabet** of the language. It includes **keyword** strings and the set of **legal characters** for creating symbols.
 - **N** : A set of **nonterminal** symbols, which defines the **abstract concepts** in the programming language, such as **expression** or **functions**.
 - **P** : A set of **productions**, which defines the **relationships** between **nonterminal** symbols and **terminal** symbols.
 - **S** : A **start symbol**, which identifies the **highest level concept** of the language.
- A production in a context-free grammar has the form

$$A \rightarrow \omega$$

where the capital A is a single symbol from the set of nonterminal symbols N , ω is a sequence of symbols from the union of the set of terminal symbols T and the set of nonterminal symbols N ($T \cup N$). The length of the sequence, ω , is not less than one ($|\omega| \geq 1$).

- **Grammars are used to define two steps of compilation (or interpretation)**
 - **Lexical Analysis**: separates the stream of characters into separate tokens (e.g. string literals, identifiers such as function and variable names, numbers, and operators).
 - **Syntax Analysis**: determines whether or not the tokens form a legal program. This step creates the Abstract Syntax Tree, which is a helpful representation of how the grammar rules determine it is a legal program
- A **syntactically correct** program **cannot necessarily** be converted into an **executable** program. There may still be a semantic error, for example, suppose you have a function named `my_func` that has 2 parameters, and the code has the line


```
b = my_func( 1, 2, 3 )
```

 This is syntactically correct, but there are too many arguments, so the program should not be compiled (at least for a programming language like C or Java - Python may let this go until runtime).

BNF

- Most programming languages are formally described using variations of a notation called **Backus-Naur Form [BNF]**.
- A BNF grammar is a **context-free grammar**.
- The **terminal** and **nonterminal** symbols are **disjoint**.
- **Terminal** symbols are generally limited to those that can be **entered on a keyboard**. **Non-terminal** symbols represent **organizational concepts** in a programming language.
- The start symbol for a programming language is the highest level concept in the language, such as Program. Some languages permit libraries or packages to sit at the top level.
- Metasymbols used by BNF:
 - Vertical bar **|** : **or**
 - Dots **...** : **a series of values**

- Right arrow $\rightarrow$: **imply**
- Example: Use BNF describe integer formally.
 - An integer is a sequence of digits. To describe integer, we need to define digit first.
 - A digit can be any number between 0 and 9. So, we define digit as

$$\text{Digit} \rightarrow 0 \mid \dots \mid 9$$

- An integer can be a single digit or an integer followed by a digit. So, we defined integer as

$$\text{Integer} \rightarrow \text{Digit} \mid \text{IntegerDigit} \text{ (left recursion, leftmost of } \omega \text{ is Integer)}$$

$$\text{Integer} \rightarrow \text{Digit} \mid \text{DigitInteger} \text{ (right recursion, rightmost of } \omega \text{ is Integer)}$$

- What are T, N, P, S in this example?
 - T is 0 ... 9
 - N is Digit and Integer
 - P are the two products
 - S is Integer.

BNF Derivation

- To **determine** if a string is **valid** according to a grammar.
- A derivation is **a series of replacements** defined by the productions.
- LL(k) grammars
 - LL(k) grammars may contain right recursion, e.g. $L \rightarrow D L \mid D$, but not left recursion
 - **Leftmost derivation**: In each step of a derivation, apply production rule to the **leftmost** symbol.
 - **Top-down approach**
 - The derivation **begins** from the **start symbol**.
 - The production rules **iteratively replace** nonterminal symbols with nonterminal and terminal symbols **until** there are **no more nonterminal** symbols in the string.
 - If the **terminal symbols match** the string, the string is **valid**.
- LR(k) grammars
 - LR(k) grammars may contain left recursion, e.g. $L \rightarrow L D \mid D$, but not right recursion
 - **Rightmost derivation**: In each step of a derivation, apply production rule to the **rightmost** symbol. We use a rightmost derivation if the grammar is LR(k) (which means it)
 - **Bottom-up approach**
 - Replace all terminal symbols with nonterminal symbols and then continue to replace nonterminal symbols until the only remaining symbol is the start symbol.
 - **Start from** the **terminal** symbols (leaves) and work **up towards the start symbol** (root).

- The procedure for determining if the string is legal (according to the given grammar) involves a stack (Note this doesn't tell you how to build the tree.)
 - while there are more tokens
 - Check to see if the tokens currently on the stack are the appropriate right-hand-side for a rule.
 - If so, then pop those items off the stack and push the left-hand-side symbol onto the stack (this is called a **reduce** operation)
 - If not, then push the next token onto the stack

- **Example1:** Given the formal definition of integer written in an LL grammar, check whether 31 is a valid integer. Since it is LL, we use the top-down approach.

$$\begin{aligned} Integer &\rightarrow Digit|DigitInteger \\ Digit &\rightarrow 0| \dots |9 \end{aligned}$$

- Topdown approach

$$\begin{aligned} Integer &\Rightarrow DigitInteger \\ \square &\Rightarrow 3Integer \\ \square &\Rightarrow 3Digit \\ \square &\Rightarrow 31 \end{aligned}$$

- **Example2:** Given the formal definition of integer written in an LR grammar, check whether 31 is a valid integer. Since it is an LR grammar, we use the bottom-up approach. I will be drawing a stack sideways (bottom on left, top on right)

$$\begin{aligned} Integer &\rightarrow Digit|IntegerDigit \\ Digit &\rightarrow 0| \dots |9 \end{aligned}$$

- Bottom-up approach
- Push 3, stack is [3]
- Reduce 3 to *Digit*, stack is [*Digit*]
- Reduce *Digit* to *Integer*, stack is [*Integer*]
- Push 1, stack is [*Integer*, 1]
- Reduce 1 to *Digit*, stack is [*Integer*, *Digit*]
- Reduce *Integer Digit* to *Digit*, stack is [*Integer*]

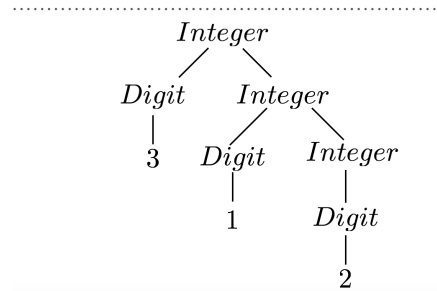
$$\begin{aligned} 312 &\Rightarrow Digit\ 1\ 2 \\ &\Rightarrow Integer\ 1\ 2 \\ &\Rightarrow Integer\ Digit\ 2 \\ &\Rightarrow Integer\ 2 \\ &\Rightarrow Integer\ Digit \\ &\Rightarrow Integer \end{aligned}$$

Parse Tree

- A **graphical form** of derivation. The root of the tree (drawn at the top) is the start symbol.
- Each derivation step corresponds to a new subtree. The parent of each subtree is a nonterminal and its children are the non-terminals and terminals for the right-hand side of the appropriate rule (and if there is more than one option on the right-hand side, then just the appropriate option is represented, e.g. an Integer node may have both a Digit and an Integer node for children, or it may have just a Digit as an only child).
- The leaves of the tree are terminals.
- Reading the leaves of the tree from left to right should be the same as reading the original string.
- Example: Draw the parse tree for 312 by using the following rules [top-down, leftmost]

$$Integer \rightarrow Digit | DigitInteger$$

$$Digit \rightarrow 0 | \dots | 9$$

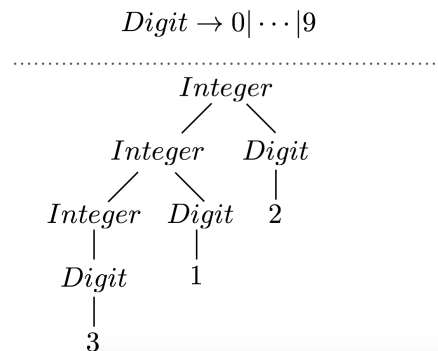


- The above derivation is representative of a recursive descent (**LL(k)**) parser that starts with the top of parse tree.
- Notice that as we read the tree from top to bottom, the left-most digit (3) is resolved “first” (it is higher up on the tree). Also notice how the right-recursion appears as the chain of Integer non-terminals going from the root via the rightmost children.

- Example: Draw the parse tree for 312 by using the following rule

$$Integer \rightarrow Digit | IntegerDigit$$

parse tree for 312 by using [bottom-up, rightmost]



- The above derivation is representative of a [LR\(k\)](#) parser that starts with the leftmost leaf of the parse tree.
- Notice that the 3, 1, and 2 appear in the correct order (as we read the leaves from left to right), and that if we read the tree from top to bottom, the rightmost digit is resolved “first” (it is higher up in the tree) (and that is why we refer to this as a rightmost derivation even though we went left to right when we constructed it). Also notice how the left-recursion appears as the chain of Integer non-terminals going from the root via the leftmost children.
- For shortish examples like this, I draw the tree as I am manipulating the stack. I just draw the tree from left to right and add connections to parents nodes as I go. I then check to make sure it makes sense.
- In both top-down and bottom-up derivations, the parser has to be smart about selecting the appropriate production rule. When you are doing it by hand, you can basically scan ahead to make sure you are using the rule that will allow you to finish the derivation (i.e. if there is a + sign, then use the E+T rule rather than the T rule).
- But in these examples we have no reason to prefer one parse tree for another (well, that isn’t quite true because we may prefer the top-down approach to the bottom-up approach, but I mean that the end-product doesn’t make us have a preference).
- Let’s look at parse trees that aren’t just part of the number-parsing. Let’s look in a more complicated context - parsing mathematical expressions. Then we will care which parse tree we have.

 EBNF

- To **increase the clarity and brevity** of syntax descriptions, **Extended BNF (EBNF)** was introduced.
- Based on BNF, EBNF includes more metasympols:
 - Curly braces $\{\}$: include the enclosed symbols **0 or more times**
 - Parentheses $()$: include the enclosed symbols **1 or more times**
 - Square brackets $[]$: indicate an **optional sequence of symbols**
- Example: Describe the integer formally in EBNF

$$Integer \rightarrow Digit\{Digit\}$$

- The above production cannot avoid the situation where an integer with multiple digits starts with 0. How to rewrite the production to avoid the situation?

$$\begin{aligned} Digit &\rightarrow 0 \mid \dots \mid 9 \\ nonzeroDigit &\rightarrow 1 \mid \dots \mid 9 \\ Integer &\rightarrow nonzeroDigit\{Digit\} \end{aligned}$$

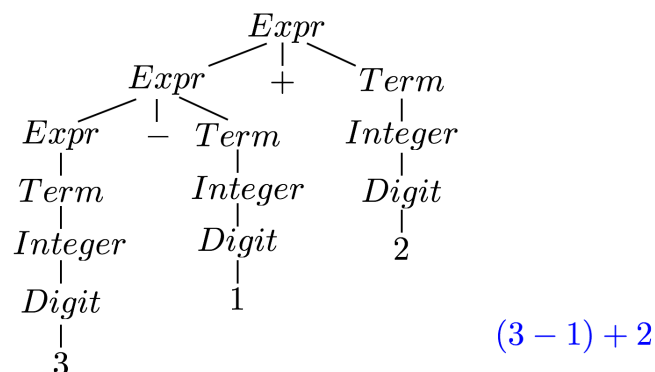
or

$$\begin{aligned} zero &\rightarrow 0 \\ nonzeroDigit &\rightarrow 1 \mid \dots \mid 9 \\ Integer &\rightarrow zero \mid nonzeroDigit\{zero \mid nonzeroDigit\} \end{aligned}$$

 Different Parse Trees Lead to Different Expressions

- The order of non-terminals on the right side is important to generating parse trees.
- Example 1:** for 3 - 1 + 2
rules $Expr \rightarrow Expr + Term \mid Expr - Term \mid Term$
 $Term \rightarrow Integer$
 $Integer \rightarrow Digit\{Digit\}$
 $Digit \rightarrow 0 \mid \dots \mid 9$

Draw a parse tree using the following



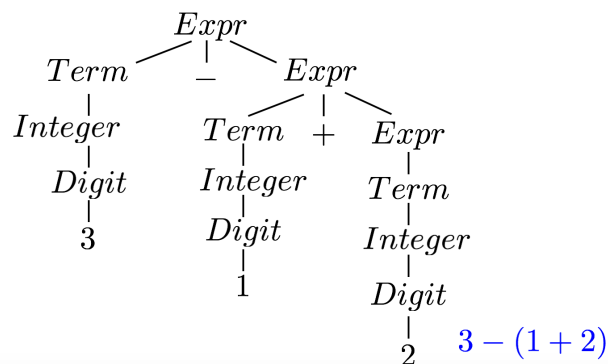
- **Example 2:** Draw a parse tree for $3 - 1 + 2$ using the following rules

$$Expr \rightarrow Term + Expr | Term - Expr | Term$$

$$Term \rightarrow Integer$$

$$Integer \rightarrow Digit \{ Digit \}$$

$$Digit \rightarrow 0 | \dots | 9$$



- In the first $3 - 1 + 2$ example, the result is interpreted as $(3 - 1) + 2$ (+ is higher than -). In the second $3 - 1 + 3$ example, the result is interpreted as $3 - (1 + 2)$. (different associativities indicated by the different placement of parentheses)
- Maintaining the order of operations is essential in programming languages.
- In order to specify hierarchies of operations, grammars can become extremely complex

Flexible & Ambiguous Grammars

- A grammar is ambiguous if its language contains at least one string with two or more distinct parse trees.
- Sometime we might want to use an ambiguous grammar to simplify the number of rules required.
- Ambiguities in grammars are generally resolved using additional rules.
 - For example, if we have a table of precedence and a default left-to-right ordering of operators of equal precedence, then we can resolve any ambiguities that arise.
- Dangling else
 - Another common ambiguity in language syntax.
 - When an if statement is contained inside an if statement, which if statement does a

ifStatement $\rightarrow$ *if (Expression)Statement* | *if (Expression)Statement else Statement*

Statement $\rightarrow$ *Assignment* | *ifStatement* | *Block*

Block $\rightarrow$ { *Statement* { *Statement* } }

subsequent else belong to?

- Consider the following code snippet.
- The second else could match with either if condition. Only by inserting brackets could the interpretation be unambiguous.
- Solution of C
 - The arbitrary rule, included in the description of the language, that an else clause is associated with the textually nearest if statement in any ambiguous case. The actual output is “there”.

```
#include <stdio.h>

int main (int args, char *argv[]) {
    int a = -1;
    int b = 1;

    if (a < 0)
        if (b < 0)
            printf("here\n");
    else
        printf("there\n");
}
```

- Solution of Java
 - Clearly defined in grammar to address the unambiguous.
 - It is not permitted that an if statement without an else clause as the single statement after an if.

```
public class Ambiguity {  
    public static void main (String args[]) {  
        int a = -1;  
        int b = 1;  
  
        if (a < 0)  
            if (b < 0)  
                System.out.println("here");  
        else  
            System.out.println("there");  
    }  
}
```

- The following code snippet, for example, will not do what the tabbing implies.
- The actual output is “there”, since Java consider the else branch belongs to the second if statement.
- Solution of Python
 - Require nested if statements to be indented. The actual output is “”.

Clite (handout)

- We use a small language, Clite, in this course to learn the concept of programming languages.
- Clite is a subset of C.
- The order of operators, UnaryOp > MulOp > AddOp > RelOp > EquOp > && > ||
- {} and [] in bold are EBNF meta-symbols.
- What are the T, N, P, S in Clite concrete syntax?
 - T (A ~ Z, a ~ z, 0 ~ 9, ASCIIChar, true, false, int, bool, float, char, main, if, while, operators), N, P, S (Program)

Concrete Syntax of Clite (from Tucker and Noonan, 2007).

Program → int main () { *Declarations Statements* }
Declarations → { *Declaration* }
Declaration → *Type Identifier* [[*Integer*]] { , *Identifier* [[*Integer*]] }
Type → int | bool | float | char
Statements → { *Statement* }
Statement → ; | *Block* | *Assignment* | *IfStatement* | *WhileStatement*
Block → { *Statements* }
Assignment → *Identifier* [[*Expression*]] = *Expression*;
IfStatement → if (*Expression*) *Statement* [else *Statement*]
WhileStatement → while (*Expression*) *Statement*

Expression → *Conjunction* { || *Conjunction* }
Conjunction → *Equality* { && *Equality* }
Equality → *Relation* [*EquOp Relation*]
EquOp → == | !=
Relation → *Addition* [*RelOp Addition*]
RelOp → < | < = | > | > =
Addition → *Term* { *AddOp Term* }
AddOp → + | -
Term → *Factor* { *MulOp Factor* }
MulOp → * | / | %
Factor → [*UnaryOp*] *Primary*
UnaryOp → - | !
Primary → *Identifier* [[*Expression*]] | *Literal* |
 (*Expression*) | *Type* (*Expression*)

Identifier → *Letter* { *Letter* | *Digit* }
Letter → a | b | ... | z | A | B | ... | Z
Digit → 0 | ... | 9
Literal → *Integer* | *Boolean* | *Float* | *Char*
Integer → *Digit* { *Digit* }
Boolean → true | false
Float → *Integer* . *Integer*
Char → ' *ASCIIChar* '

Concrete Syntax and Concrete Syntax Tree

- Example: Draw a parse tree for $a = b + 3 * c$; by using Clite Concrete Syntax

Concrete grammar for *Assignment* and *Expression* (Tucker and Noonan, 2007).

$Assignment \rightarrow Identifier \ [\ [Expression] \] = Expression;$

$Expression \rightarrow Conjunction \ \{ \ || Conjunction \}$

$Conjunction \rightarrow Equality \ \{ \ \&\& Equality \}$

$Equality \rightarrow Relation \ [\ EquOp Relation \]$

$EquOp \rightarrow == \ | \ !=$

$Relation \rightarrow Addition \ [\ RelOp Addition \]$

$RelOp \rightarrow < \ | \ < = \ | \ > \ | \ > =$

$Addition \rightarrow Term \ \{ \ AddOp Term \}$

$AddOp \rightarrow + \ | \ -$

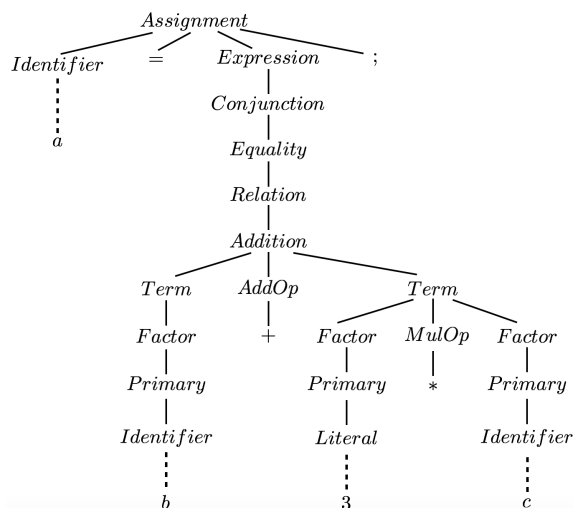
$Term \rightarrow Factor \ \{ \ MulOp Factor \}$

$MulOp \rightarrow * \ | \ / \ | \ \%$

$Factor \rightarrow [\ UnaryOp \] \ Primary$

$UnaryOp \rightarrow - \ | \ !$

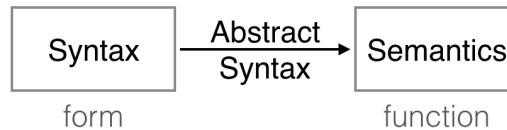
$Primary \rightarrow Identifier \ [\ [Expression] \] \ | \ Literal \ | \ (Expression) \ | \ Type \ (Expression)$



- Observations
 - concrete syntax results in **large parse trees** with long chains of non-terminals that lead to a single terminal
 - **many non-terminal symbols** used to describe **precedence** of operators
 - the use of non-terminal symbols results in **unique parse tree**, which is essential for making language **unambiguous**

Abstract Syntax (handout)

- Purpose: **bridge the syntax to the semantics**



- A notation that allows parser to remove nonessential symbols and generate a tree that contains only the essential elements of the computation.
 - A **simplified version** of a parse tree that **removes non-terminals that do not contribute to the semantic meaning** of the program.
- Abstract syntax is written slightly differently from EBNF syntactic description because the abstract syntax **does not care about how things are written, only about describing the essential relationships** between components.
- Clite Abstract Syntax

Abstract Syntax of Clite (from Tucker and Noonan, 2007).

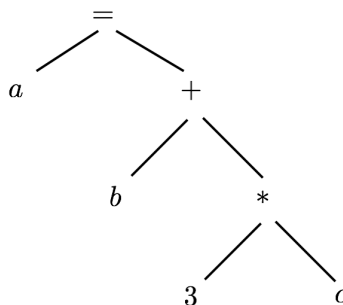
```

Program = Declarations decpart; Statements body;
Declarations = Declaration*
Declaration = VariableDecl | ArrayDecl
VariableDecl = Variable v; Type t
ArrayDecl = Variable v; Type t; Integer size
Type = int | bool | float | char
Statements = Statement*
Statement = Skip | Block | Assignment | Conditional | Loop
Skip =
Block = Statements
Conditional = Expression test; Statement thenbranch, elsebranch
Loop = Expression test; Statement body
Assignment = VariableRef target; Expression source
Expression = VariableRef | Value | Binary | Unary
VariableRef = Variable | ArrayRef
Binary = Operator op; Expression term1, term2
Unary = UnaryOp op; Expression term
Operator = BooleanOp | RelationalOp | ArithmeticOp
BooleanOp = && | ||
RelationalOp = == | != | < | <= | > | >=
ArithmeticOp = + | - | * | /
UnaryOp = ! | -
Variable = String id
ArrayRef = String id; Expression index
Value = IntValue | BoolValue | FloatValue | CharValue
IntValue = Integer intValue
BoolValue = Boolean boolValue
FloatValue = Float floatValue
CharValue = Char charValue
  
```

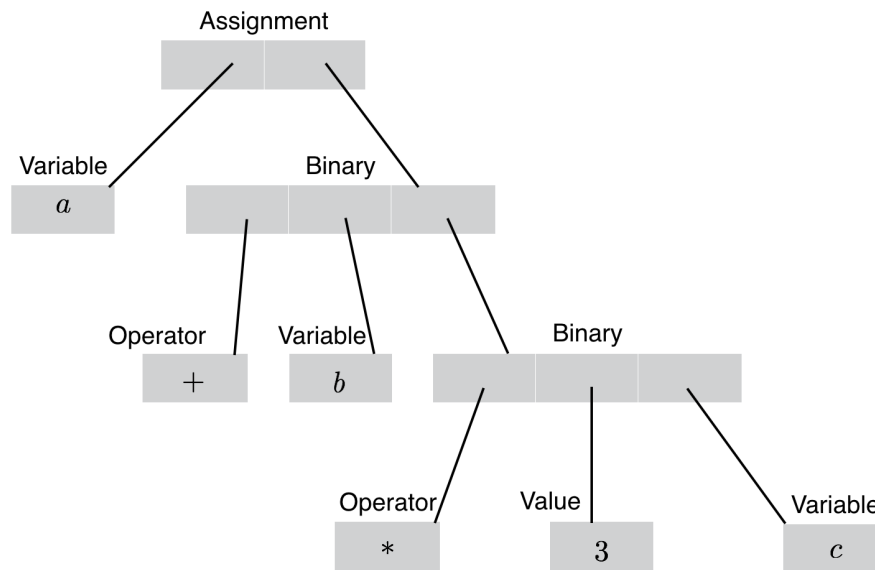
- Abstract syntax definition form
 - Abstract syntax can be defined in the form: $Lhs = Rhs$
 - Lhs : the name of an abstract syntactic class
 - Rhs : defines the class as
 - A list of one or more alternatives
 - A list of essential components that define a member of that class, separated by semicolons (;)
- The essential of assignment is nothing more than move value from one place to another. The abstract syntax for an Assignment defines the type that needs to be on the left-hand side and the type that needs to be on the right-hand side of an assignment operator.
- All of the operator terms are combined together into the Binary and Unary categories based on their functionality rather than precedence.
- Precedence is handled by the concrete syntax tree, and once the unambiguous parse tree exists, we can reduce it to simpler form by replacing it with the abstract syntax
- Properties of abstract grammar that make it a more useful representation for analysis than the concrete grammar:
 - The non-terminals in the grammar are more directly connected to their semantic meaning than in a concrete grammar, with differences based primarily on function rather than precedence or spatial relationship.
 - This make it possible to start linking code to non-terminals, as their definition is functional rather than syntactic.
 - The organization of the grammar explicitly links inputs and outputs of functional operations, discarding syntactic formalities.
 - This makes it easy to internally represent the functionality, as we can represent each non-terminal by a structure that holds a reference to its inputs and output(s) and operator.

Simplify Concrete Syntax Tree and Convert to Abstract Syntax Tree

- Discard all separator or terminating symbols
- Discard all nonterminals that are trivial root. A trivial root is a symbol with only one subtree
- Replace all remaining nonterminals with operators which are leaf or one of their immediate subtrees



- When drawing an abstract syntax tree, we can use box to represent each node, with a spot for each input, output, and operator.



Characteristics of Abstract Syntax Tree

- Characteristics
 - There are **no keywords or terminator** symbols
 - There is **no explicit indication of operator precedence** in the syntax. Precedence is completely defined by the shape of the tree.
 - The tree explicitly represents the internal structure of each expression.
- The **lack of keywords and precedence** means an abstract syntax representation of an expression should be **common across any programming languages** that have the ability to implement the operation. The meaning of above abstract syntax tree should be general across languages. The computer should multiply the contents of a variable with identity c by the integer 3, add the result to the contents of variable with identity d and assign the result to a variable with identity a. At this essence, the expression does three of the four things a computer can do: store data, move data, and manipulate data. (What's the fourth? Adjust control flow based on data)
- The **transformation between a concrete parse tree and an abstract parse tree** is not trivial, but also not complex. It is straightforward to **automate and represents the second stage** in compilation. Once complete, the **abstract parse tree** becomes the **basis for code generation**. The abstract parse tree is **not completely language independent**, however, because different language have different semantic rules. For example, the expression $a = b + 3 * c$, the variable a, b and c could all be integers, which in most languages is the basis. However, in Python, all three variables could be strings and it would be perfectly valid expression. Furthermore, in Python, only b and c need to be defined prior to the expression, which in C or Java there must be some kind of prior declaration and typing of the three variables.

