

Syntax (continued)

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 \]$

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

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

$AddOp \rightarrow + \ | \ -$

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

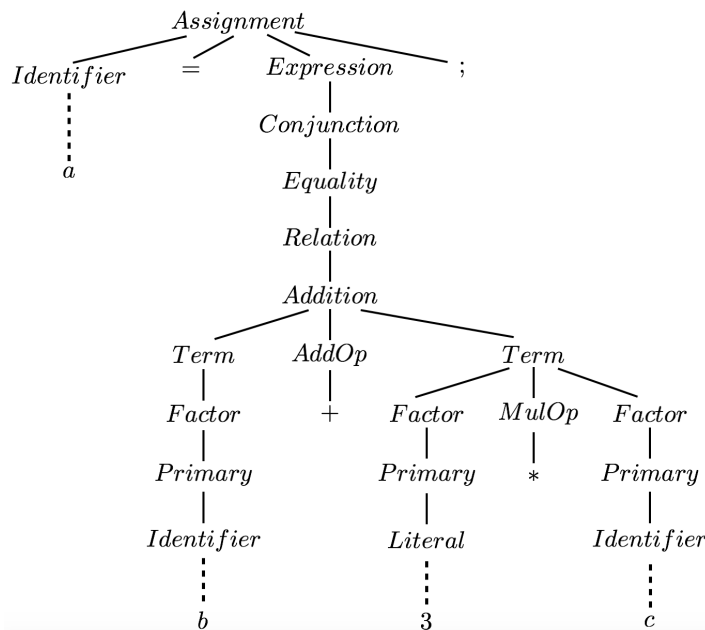
$MulOp \rightarrow * \ | \ / \ | \ \%$

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

$UnaryOp \rightarrow - \ | \ !$

$Primary \rightarrow Identifier \ [\ [Expression] \] \ | \ Literal \ |$

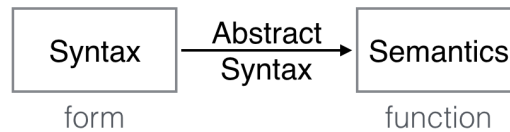
$\ (Expression) \ | \ Type \ (Expression)$



- 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).

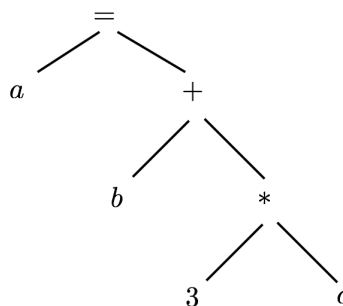
```

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
  
```

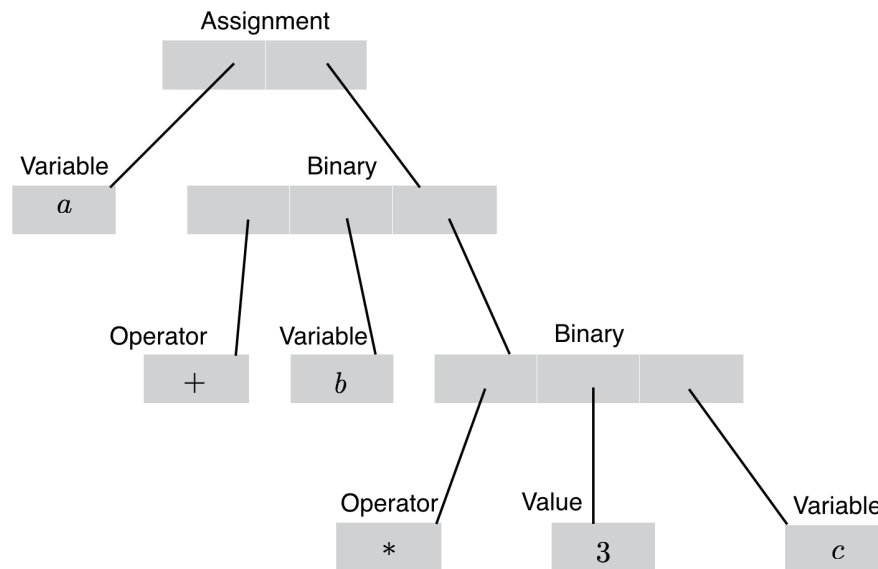
- Abstract syntax definition form
 - Abstract syntax can be defined in the form: $Lhs = Rhs$
 - Lhs : the name of an abstract syntactic class
 - Rhs : defines the class as
 - A list of one or more alternatives
 - A list of essential components that define a member of that class, separated by semicolons (;)
- 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 `b` 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 languages have different semantic rules. For example, the expression `a = b + 3 * c`, the variables `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 a 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.