CSE 12
Abstract Syntax Trees

- **Compilers and Interpreters**
- **Parse Trees and Abstract Syntax Trees (AST's)**
- **Creating and Evaluating AST's**
- **The Table ADT and Symbol Tables**

Using Algorithms and Data Structures

- Digital computers can do anything that can conceivably be done with information: they are general purpose information processing machines

- You get a computer to do something by programming it to run algorithms that manipulate data structures

- Picking what algorithms and data structures are suitable for a particular application is something every good programmer needs to know

- One important application area is: writing programs that understand programs…

Compilers and interpreters

Distinguish two kinds of program-understanding programs:

- Compilers
  - take a program written in a source language as input and translate it into a machine language, for later execution
- Interpreters
  - take a source language program as input, and execute ("interpret") it immediately, line by line

(You can think of a computer as an interpreter of machine language instructions; the Java Virtual Machine is an interpreter of Java bytecode instructions, which are generated by the Java compiler)

Let's look at how some ideas from CSE 12 (like recursion, and trees) can be used to implement an interpreter for a language

Towards an interpreter

- An interpreter reads a line or statement at a time of a source language text, and then does what that line or statement says to do
- The source language should have clearly defined syntax and semantics rules
    - the syntax rules say what strings legally belong to the source language, and what their structure is
    - the semantics rules say what each legal string in the language means, based on its structure
- Without clearly defined syntax and semantics, writing an interpreter isn't really possible!

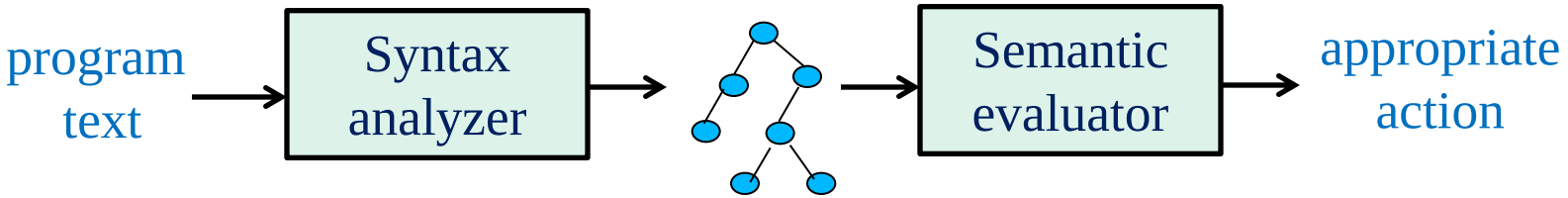
Tasks for an interpreter

- A language interpreter does two things with each line or statement in the input: *syntactic analysis* and *semantic evaluation*
- Syntactic analysis is done in accordance with the syntax rules for the language
  - the output from syntactic analysis is a tree data structure
  - corresponds to "compile time" in a compiled language
- Semantic evaluation is done in accordance with the semantic rules
  - Semantic evaluation takes as input the tree created in the syntactic analysis phase
  - corresponds to "run time" for compiled code
- The result of all this is: doing what the statement says! (For example, computing a value and returning it or printing it out)

Parts of an interpreter

- In designing an interpreter, follow the decomposition into two tasks, and design it to have two parts:

1. A syntactic analysis engine, which takes as input a string, and outputs an appropriate tree structure
2. A semantic evaluation engine, which takes as input that tree, and does what the original input string said to do



Syntax rules and BNF

- The syntax rules for the language can be written down in Backus-Naur Form (BNF) or a similar notation
- A BNF grammar is a list of syntax rules
- Each rule defines one "nonterminal symbol", which appears at the left of a " := " sign in the rule
- Alternative definitions of the nonterminal symbol appear to the right of the " := " sign, separated by " | " signs
- Often the definition of a nonterminal in a BNF grammar is recursive: it defines the nonterminal in terms of itself

Syntax rules and BNF

- The nonterminal symbol defined in rule listed first in the grammar is called the "start" symbol of the grammar
- A symbol not defined by a rule in the grammar is a "terminal symbol", and is usually taken literally
- If a string satisfies the definition of the "start" symbol, it is in the language defined by the BNF grammar; otherwise not
- The process of using the grammar to check to see if a string is in the language is called **parsing** the string

Parsing and derivations

- The process of using the grammar to check to see if a string is in the language defined by the grammar is called **parsing** the string
- One way to parse a string is to try to write down a **derivation** of the target string from the grammar's start symbol:

1) Write down the start symbol. This is the first step in the derivation.
2) If the current step in the derivation consists of only terminal symbols, and is equal to the target string, you have parsed the string; done.
3) Else replace one of the nonterminal symbols in the current step of the derivation with one of its definitions, to produce the next step
4) Go to 2

A BNF grammar for a language

- Here is a BNF grammar for a simple formal language
- The "start" symbol for this grammar is **<A>**
- If a string satisfies the definition of **<A>**, it is in the language defined by this grammar; otherwise not

```
<A> := <B> | <C>
<B> := <ident> = <A>
<C> := <C> + <D> | <C> - <D> | <D>
<D> := <D> * <M> | <D> / <M> | <M>
<M> := <ident> | <const> | (<A>)
<ident> := w | x | y | z
<const> := 0 | 1 | 2 | 3 | 4
```

Example derivations

- A derivation starts with the start symbol, and at each step, replaces one nonterminal symbol by one of the definitions of that nonterminal.
- Here are derivations of the strings `2` and `2 + w + 3` based on that grammar:

```
<A> => <C> => <D> => <M> => <const> => 2
```

```
<A> => <C> => <C> + <D> => <C> + <M> =>
       <C> + <const> => <C> + 3 =>
       <C> + <D> + 3 => <D> + <D> + 3 =>
       <M> + <D> + 3 => <M> + <M> + 3 =>
       <const> + <M> + 3 => <const> + <ident> + 3 =>
       2 + <ident> + 3 =>  2 + w + 3
```

Parsing strings

- Using that grammar, parse these strings of terminal symbols – that is, show a derivation of each of them from the start symbol **<A>**:

w

2 + w

2 + w * 3

(2 + w) * 3

Parsing and parse trees

- For every derivation, there is a corresponding tree: a ***parse tree***
- Each node in the parse tree corresponds to one symbol in the BNF grammar
- Leaves in the parse tree correspond to terminal symbols; internal nodes correspond to nonterminal symbols
- The root of the parse tree is the "start" symbol
- The children of an internal node in the parse tree correspond to the symbols in a definition of the nonterminal symbol corresponding to their parent node
- Reading the leaves of the parse tree left to right gives you the string that has been parsed

Parse trees and derivations

- How to build a parse tree?

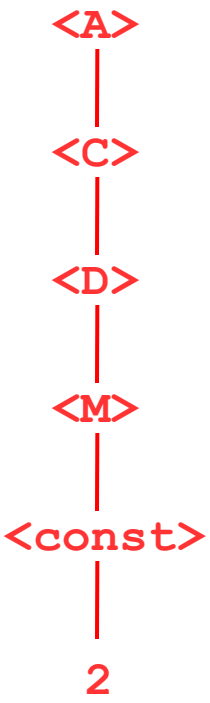
- Given a derivation of a string, you could build a parse tree “top down”:
  - The start symbol is the root of the tree.
  - Children of the root are symbols in the definition of the start symbol used to create the next step in the derivation.
  - In general, children of a node in the parse tree are symbols in the definition of a nonterminal that was used to create a next step in the derivation.

Parse tree facts

- Note that a BNF parse must have a tree structure, because:
  - in a tree, each node except the root has exactly one parent
  - in a BNF grammar, each rule has exactly one nonterminal symbol on the left hand side of the rule (this makes BNF grammars what are called "context free" grammars)
- A BNF grammar may permit several different derivations of the same string, but if the grammar is *unambiguous*, a string will have only one parse tree
- A parse tree can also be built “bottom up”, creating leaves first, and the root last. The algorithm we will consider later does that

Parse tree from a derivation

- A derivation:
  <A> => <C> => <D> => <M> => <const> => 2
- Parse tree corresponding to that derivation:

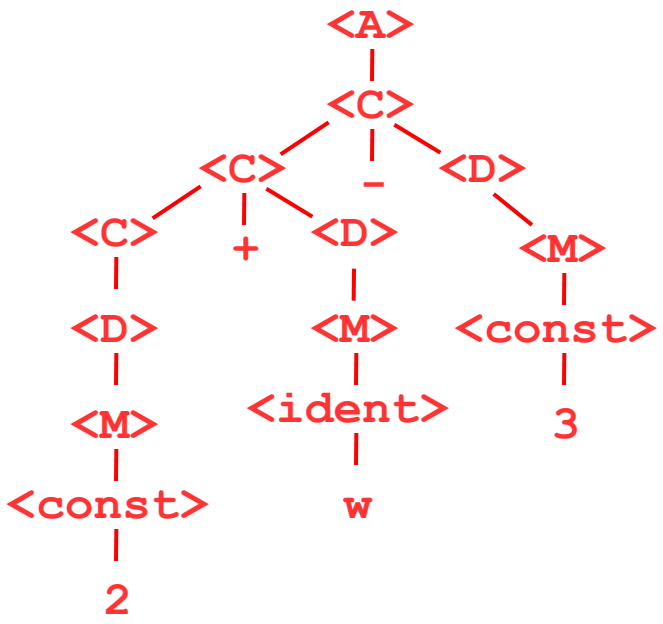


Another parse tree from a derivation

- Another derivation:

<A> => <C> => <C> - <D> => <C> - <M> => <C> - <const> =>
<C> - 3 => <C> + <D> - 3 => <D> + <D> - 3 =>
<M> + <D> - 3 => <M> + <M> - 3 => <const> + <M> - 3 =>
<const> + <ident> - 3 => 2 + <ident> - 3 => 2 + w - 3

- Parse tree corresponding to that derivation:



Constructing parse trees

- Using the example grammar, construct parse trees for these strings:

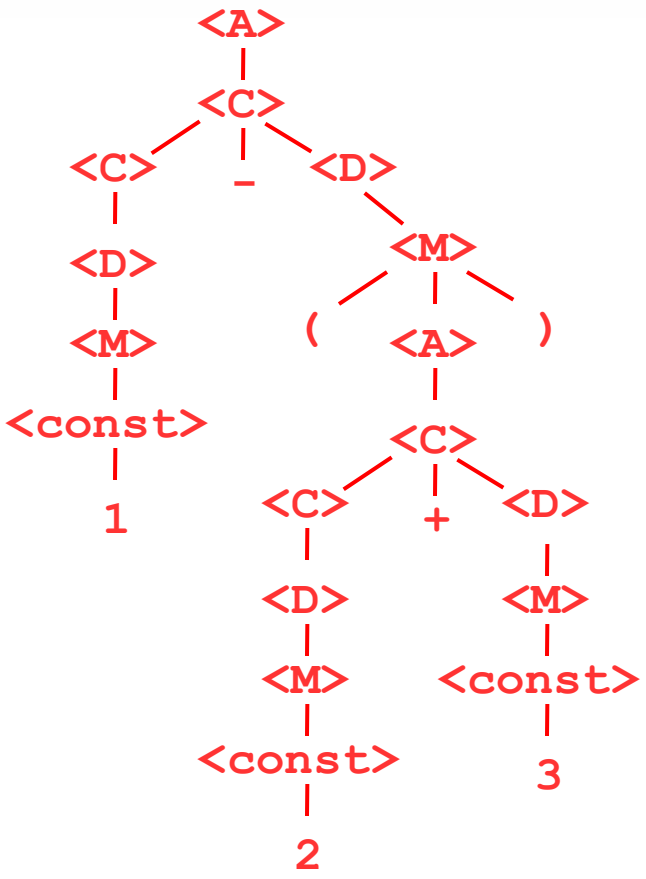
  2 + w * 3 (2 + w) * 3 w = 2

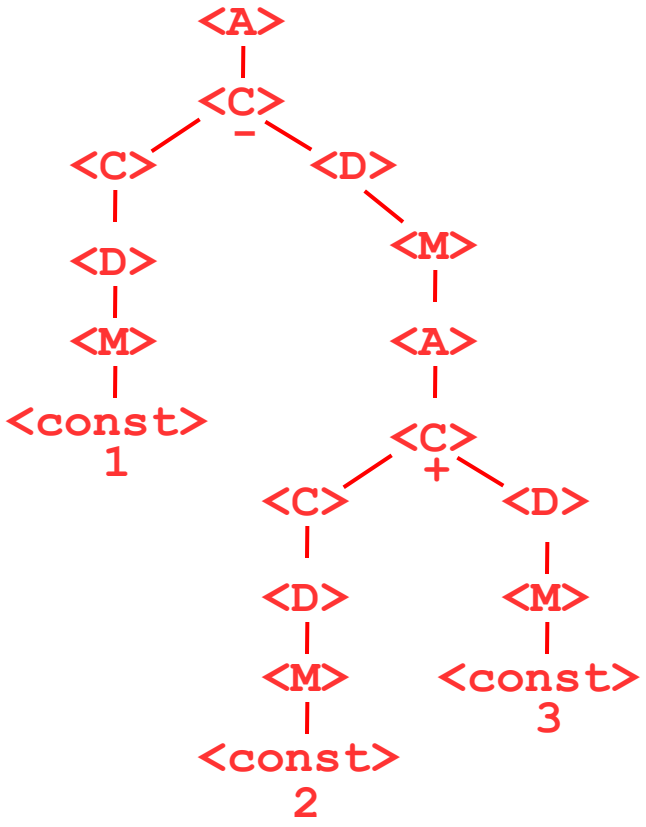
Abstract syntax trees

- A parse tree includes *all* the information in a derivation of a string from the start symbol
    - In a parse tree, the children of a nonterminal symbol node include nodes for *all* of the symbols in the definition of that nonterminal
- But this may be more information than is required for some applications of the tree
- For example: do not need to include nodes for parentheses just used for grouping; information in nodes corresponding to expression operators can be moved into their parent nodes; etc.
- A parse tree with nonessential nodes left out is called an ***Abstract Syntax Tree*** (AST)

Parse tree and AST

- Parse tree and an AST for the string `1 – ( 2 + 3 )`





Syntactic analysis == building an AST

- The input to the syntactic analysis phase of an interpreter is a string; the output is an AST
- So the process of syntactic analysis is the process of constructing an AST
- There are two main issues to address before implementing syntactic analysis in this way:
  - the design of the AST data structure: what type of objects to use for the nodes, etc.
  - what algorithm to use to build the AST

AST data structure design issues

You know how to implement binary trees...

... but AST's raise some additional issues:

- In general, a node in an AST can have more than 2 children (because more than 2 symbols can appear on the right hand side of a rule in the grammar)
- Intuitively, the nodes of an AST are of somewhat different types... a <const> node has a different meaning from a <ident> node, for example
- At the same time, the nodes of an AST are of similar type: they all correspond to part of an input string at some level of analysis

Applying OO design to AST design

In an object-oriented approach, you can easily deal with these design issues:

- Define a separate class corresponding to each different nonterminal symbol in the grammar: nodes in the AST will be instances of these classes
    - lets you make different types of nodes have different possible numbers of children (different number of private instance variables), and different behavior (different methods)
- Make all these classes derived from one base class, or make them all implement one interface
    - permits polymorphism

Abstract syntax tree algorithm design

The easiest way to implement AST construction is to closely follow the BNF grammar definitions

- Each class in the OO design should have a method that "knows how" to parse a string according to the definition of the nonterminal symbol corresponding to that class
  - "Knowing how" to parse a string will rely on parse methods in other classes, that know how to parse strings according to the definitions of their corresponding symbols!
- If the method successfully parses the string, it should return a pointer to an AST node of the appropriate subtype, with its children initialized in the appropriate way
- If the method cannot successfully parse the string, it should return a null pointer to indicate that fact

A piece of an AST construction algorithm

Following this basic design idea, the parse method for nonterminal symbol <A> defined as <A> := <B> | <C> would be a static factory method in a class named A , and could have a structure like this:

```
public static A parse (String s) {
   if (B.parse(s) != null) {
      // the string satisfies def'n of <B>; so,
      // make a new A node, with the result of B.parse(s)
      // as its child, and return it
   } else if (C.parse(s) != null) {
      // the string satisfies def'n of <C>; so,
      // make a new A node, with the result of C.parse(s)
      // as its child, and return it
   } else // the string does not satisfy def'n of <A> !
      return null;
}
```

Semantic rules

- An interpreter first does syntactic analysis, constructing an AST...
- ...then it does semantic evaluation, according to the semantic rules for the language

- To implement semantic evaluation correctly, semantic rules must be clearly stated and followed

- Unfortunately there is no universally accepted notation for specifying semantic rules that is as standard as BNF notation is for specifying syntax rules

- Usually, semantic rules are stated as English or pseudocode, attached to each BNF rule (or equivalently, to each type of node in the AST)

Semantic rules: some examples

<A> := <B> | <C>

The value of <A> is the value of <B> or <C>, as appropriate.

<B> := <ident> = <A>

The value of <B> is the value of the <A> on the rhs of the =. In addition, evaluating a <B> has the side effect of assigning the value of <A> to the variable named by the <ident>.

<C> := <C> + <D>

The value of <C> is the value of <C> plus the value of <D>.

<D> := <D> * <M>

The value of <D> is the value of <D> times the value of <M>.

<ident> := w | x | y | z

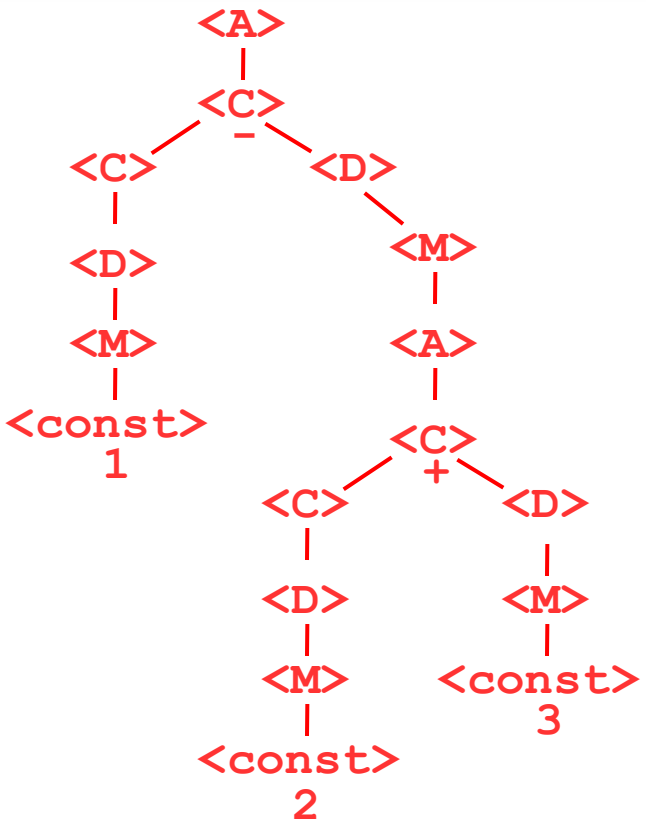
The value of <ident> is the value of the variable named by the identifier. If the variable has not been assigned a value, it is a runtime error.

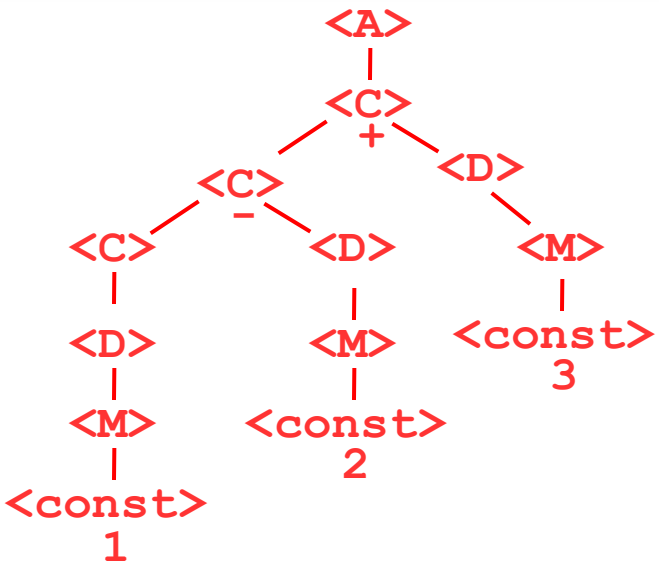
<const> := 0 | 1 | 2 | 3 | 4

The value of <const> is the value of the matching literal constant.

Evaluating an AST

- Evaluate these AST's (for `1 - ( 2 + 3 )` and `1 - 2 + 3`)





Evaluating AST's

Construct AST's for the following expressions, and then evaluate them:

```
2 + 3 * 4
```

```
w = 2
```

```
2 + w * 3
```

Evaluating AST's

- Note that the semantic evaluation phase basically does a postorder traversal of the syntax tree: evaluate the root's children left to right, then return the value of the root
- Note also that the precedence and associativity rules for operators in this language never had to be explicitly stated: they are implicit in the way this grammar is defined
    - "left-recursive" rule gives left-to-right associativity, "right-recursive" rule gives right-to-left associativity
    - symbols "farther from the root" (the start symbol) are evaluated first, giving higher precedence
- Overall, semantic evaluation is very straightforward, once the AST has been constructed

- One additional issue: How to keep track of the values of variables?
- ... answer: use a symbol table, an application of the Map ADT

Map

- The ADT's we have talked about so far have been container structures intended to hold data, sometimes called keys
  - An insert operation inserts a single key value in the structure; a find operation says whether a key value is in the structure; delete removes a key; etc.

- The **Map** ADT has a slightly different emphasis

Map ADT

- A Map ADT is intended to hold *pairs* : each pair consists of a key, together with a related data value
- An insert operation inserts a key-value pair in the table
- A find operation takes a key and returns the value in the key-value pair with that key
- A delete operation takes a key and removes the key-value pair with that key

- The Map ADT is also sometimes called a “Table" or "Dictionary" ADT, or an "associative memory"
  - in the JCF, see the `Map<K,V>` interface

Map ADT

- Domain:
  - a collection of pairs; each pair consists of a key, and related data value
- Operations (typical):
  - *Create* a Map (initially empty)
  - *Insert* a new key-value pair in the Map; if a key-value pair with the same key is already there, update the value part of the pair
  - *Find* the key-value pair in the Map corresponding to a given key; return the value, or null if none
  - *Delete* the key-value pair corresponding to a given key
  - *Traverse* all key-value pairs in the Map

Implementing the Map ADT

- A Map can be implemented in various ways
  - using a list, binary search tree, hashtable (we'll talk about these later), etc., etc.
- In each case:
  - the implementing data structure has to be able to hold key-value pairs
  - the implementing data structure has to be able to do insert, find, and delete operations depending on the key, but storing both the key and its data value
  - the find operation should return the value associated with a given key
- Using Java generics, you can specify the type of the keys, and the type of the values associated with keys

The Map ADT and symbol tables

- The Map ADT is useful in any situation where you want to store, retrieve, and manipulate data based on keys associated with pieces of data

- One important application is a symbol table in a programming language compiler or interpreter

- A symbol table associates identifiers in a program with related data such as whether the identifier is the name of a variable, function, constant, or class; and if it is a variable, what its value is, etc.

Symbol table operations

- Typical symbol table operations include:
  - When a variable is declared, an "insert" operation is done in the table to add an entry saying what type the variable is, and what its initial value is, etc.
  - When a variable is referenced, a "find" operation is done in the symbol table, to see what the variable's value is at the present time
  - When a variable is assigned to, an "insert" or "update" operation is done to change the value associated with the variable
  - When a variable passes out of scope, a "delete" operation is done to remove it and its information from the table

Symbol tables and side effects

- Some expression languages may permit only "pure evaluation": evaluating a node in the parse tree involves no side effects
- However, if the language contains variables and permits assignment of values to variables, the interpreter must correctly handle these side effects, and a symbol table can do that
- The symbol table maintains associations between variables (as keys) and their values
- In a simple expression language, variables are the only symbols, and once introduced they do not go out of scope. So the only symbol table operations needed are:
  - When a variable is evaluated, the symbol table is accessed to find the value associated with the variable
  - When an assignment expression is evaluated, the symbol table is updated to associate the new assigned value with the variable

Next time

- **The Map ADT**
- **Implementations of the Map ADT**
- **Hashing and Hash Tables**
- **Collisions and Collision Resolution Strategies**
- **Hash Functions**
- **Hash Table time costs**

Reading: Gray, Ch 12