Programming language components

• statements: instructions that say what to do
  – compute values, make decisions, repeat sequences of operations
• variables: places to hold data in memory while program is running
  – numbers, text, ...
• syntax: grammar rules for defining legal statements
  – what's grammatically legal? how are things built up from smaller things?
• semantics: what things mean
  – what do they compute?

JavaScript has limitations:

Programming ideas carry over into other languages

easy to do useful things with it

easy to start with

can understand what other web pages are doing (and steal from them)

can use it in your own web pages

many web services rely on JavaScript in browser

all browsers process JavaScript

Why study/use JavaScript?

easy to start

easy to do useful things

• most languages are higher-level and more expressive than the assembly language for the toy machine
  • but it's basically the same idea
  • semantics are more complicated
  • grammar rules are more complicated
  • variables are much richer, more varied, more expressive
  • statements are much richer, more varied, more expressive

• what's grammatically legal? how are things built up from smaller things?
• syntax: grammar rules for defining legal statements
  • numbers, text, ...

• variables: places to hold data in memory while program is running
  • compute values, make decisions, repeat sequences of operations

• statements: instructions that say what to do

Programming language components
Javascript components

• Javascript language
  – statements that tell the computer what to do
  - get user input, display output, set values, do arithmetic, test conditions, repeat groups of statements, …

• libraries, built-in functions
  – pre-fabricated pieces that you don’t have to create yourself
  - math functions, text manipulation

• access to browser and web pages
  – buttons, text areas, images, page contents, …
  - keyboard functions, text manipulation
  - pre-fabricated pieces that you don’t have to create yourself
  - set values, do arithmetic,
  - get user input, display output,
  - statements that tell the computer what to do

• Javascript components

Basic example #1: join 2 names

```html
<html>
  <body>
    <p> name2.html: joins 2 names
      <script language=javascript>
        var firstname, secondname, result
        firstname = prompt("Enter first name")
        secondname = prompt("Enter last name")
        result = firstname + secondname // + means "join" here
        alert("hello, " + result)  // and here
      </script>
    </p>
  </body>
</html>
```
Basic example #2: add 2 numbers

```html
<body>
<script>
var num1, num2, sum
num1 = prompt("Enter first number")
num2 = prompt("Enter second number")
sum = parseInt(num1) + parseInt(num2) // "+" means "add"
alert(sum)
</script>
</body>
</html>
```

There's also `parseFloat(...)` for floating point numbers.

ParseInt(...): converts a sequence of characters into its integer value.

```html
<body>
<script>
var sum = 0
var num
num = prompt("Enter new value, or 0 to end")
while (num != 0) {
    sum = sum + parseInt(num)
    num = prompt("Enter new value, or 0 to end")
}
alert("Sum = " + sum)
</script>
</body>
</html>
```

Adding up numbers: addup.html

Dialog boxes, variables, arithmetic, conversion.
Find the largest number: max.html

• needs an `if` to test whether new number is bigger
• needs another relational operator
• needs a parsing tool to treat input as a number

```javascript
var max = 0
var num

num = prompt("Enter new value, or 0 to end")
while (num != 0) {
    if (parseFloat(num) > max)
        max = num
    num = prompt("Enter new value, or 0 to end")
}
```

```
document.write("Max = " + max)
```
Types, declarations, conversions

• variables have to be declared in a `var` statement
• each variable holds information of a specific type
  – really means that bits are to be interpreted as info of that type
  – internally, 3 and 3.00 and "3.00" are represented differently
  – really means that bits are to be interpreted as info of that type
  – each variable holds information of a specific type
• Javascript usually infers types from context, does conversions automatically

Making decisions and repeating statements

• if-else statement makes decisions
  – the Javascript version of decisions written with `ifzero` and `goto`
  – if-else statement repeats groups of statements

      do this group of statements instead
      } else {
          do this group of statements
      } if (condition is true)

• while statement repeats groups of statements

      do this group of statements
      } while (condition is true)

      Sum = sum + sum
      -- automatically

• parseInt(...) (... if it could have a fractional part
  – `parseInt` (...) if can’t tell from context that string is meant as an
  – integer
  -- `parseFloat` (...) if it could have a fractional part

Sometimes we have to be explicit:

---

Types, declarations, conversions
if-else examples

• can include else-if sections for a series of decisions:

```javascript
var num = prompt("Enter number")
while (num != null) {
  num = parseInt(num)
  if (num > 0) {
    alert(num + " is positive")
  } else if (num < 0) {
    alert(num + " is negative")
  } else {
    alert(num + " is zero")
  }
  num = prompt("Enter number")
}
```

while loop examples

• counting or "indexed" loop:

```javascript
i = 1
while (i <= 10) {
  // do something (maybe using the current value of i)
  i = i + 1
}
```

• "nested" loops:

```javascript
var n = prompt("Enter number")
while (n != null) {
  i = 0
  while (i <= n) {
    document.write("<br>" + i + i*i)
    i = i + 1
  }
  n = prompt("Enter number")
}
```

if-else examples (sign.html)

```javascript
var num = prompt("Enter number")
while (num != null) {
  assert(num + " is zero")
  else if (num + " is negative")
    assert(num + " is positive")
  else if (num + " is positive")
    assert(num + " is zero")
  else {
    assert(num + " is positive")
  }
  num = parseInt(num)
  while (num = prompt("Enter number")
```

• can include else-if sections for a series of decisions:
Functions

- A function is a group of statements that does some computation:
  - The statements are collected into one place and given a name.
  - Other parts of the program can “call” the function.
  - They can give it values to use in its computation (arguments or parameters).
  - The value need not be used.

Javascript provides some useful built-in functions:

You can write your own functions:

- e.g., `prompt`, `alert`, ...

```javascript
function area(r) {
  return 3.14 * r * r
}
```

```javascript
function definition:

```javascript
function name (list of arguments) {
  // the statements of the function
}
```

```
function uses:

```javascript
rad = prompt("Enter radius")

alert("radius = " + rad + ", area = " + area(rad))

alert("area of ring = " + area(1.75) - area(0.6))
```

Function examples
Why use functions?

- Libraries: functions already written for you called/used from other places in a program
- Functions: package a group of statements so they can be repeated/reuse/statement inside/true
- Loops: repeat statements while a condition is true
- Conditions: compare and branch: if-else
- Statements: assignment, input/output, loop, conditional, call
- Assignment: store a new value in a variable

Values

Expressions: operations on variables and constants to produce new values

declarations: specify name (and type) of variables, etc.
variables: places to store data and results during computing

Constants: literal values like 1, 3.14, "Error"

Summary: elements of most programming languages

- Constants: literal values like 1, 3.14, "Error"
- Variables: places to store data and results during computing
- Declarations: specify name (and type) of variables, etc.
- Expressions: operations on variables and constants to produce new values
- Functions: package a group of statements so they can be called/used from other places in a program
- Libraries: functions already written for you
- A good library encourages use of the language

A way to use code written by others long ago and far away

Multiple people can work on the program

Different implementations can interoperate

Implementation details can be changed as long as it still does the same job

Defines an interface

That are separate from each other

Breaks a big job into smaller, manageable pieces

A function collects it into one place

If a computation appears several times in one program

Why use functions?
How Javascript works

• recall the compiler -> assembler -> machine instructions process for Fortran, C, etc.

• Javascript is analogous, but differs significantly in details

• when the browser sees Javascript in a web page (<script> tags)

– passes the Javascript program to a Javascript compiler

• Javascript compiler
  – checks for errors
  – compiles the program into instructions for something like the toy machine, but richer, more complicated, higher level
  – runs a simulator program (like the toy) that interprets these instructions
  – compiles the program into instructions for something like the toy machine, but richer, more complicated, higher level
  – checks for errors
  – often written in C or C++ but can be written in anything

• simulator is often called an "interpreter" or a "virtual machine"

• build on a foundation (rarely start from scratch)
  – spell out precise computational steps in a programming language
  – break into smaller pieces that will work together
  – start with a broad specification

• figure out what to do

What we saw with Javascript or Toy is like reality, but very small

The process of Programming

• what we saw with Javascript or Toy is like reality, but very small

  - Javascript tells browser to do things (pop up dialog box)
  - browser and simulator interact
  - simulator is often called an "interpreter" or a "virtual machine"

  - when an event like click happens, browser tells Javascript ("onclick")

  - Javascript tells browser to do things (e.g., pop up dialog box)

  - run a simulator program (like the toy) that interprets these instructions
  - compiles the program into instructions for something like the toy machine, but richer, more complicated, higher level

  - checks for errors

  - often written in C or C++ but can be written in anything

  - when the browser sees Javascript in a web page (<script> tags)

  - Javascript is analogous, but differs significantly in details

  - for Fortran, C, etc.

  - recall the compiler -> assembler -> machine instructions process

  - it rarely works the first time

  - test to be sure it works, debug if it doesn’t

  - evolve as get a better idea of what to do, or as requirements change
Real-world programming

- programs must evolve to meet changing environments and requirements
- programs are written by teams, managed by managers, and coordinate meetings
- programs are scaled beyond the capabilities of the machine
- programs may be millions of lines of code
- products are measured in terms of lifetimes in years or even decades

... with other systems...

- external criteria for reliability, security, interoperability
- it can use
- constraints on how fast the program must run, how much memory
- schedules and deadlines

big programs need teams, management, coordination, meetings, ...

Typical productivity: 1-10K lines/year/programmer

the same thing, but on a grand scale

Maintenance of old ("legacy") programs is hard

- expertise disappears
- machines and tools and languages become obsolete
- programs must evolve to meet changing environments and requirements