Today’s Topics

Team Project

• Introduce this year’s team project

S/SL

• S/SL, the Syntax/Semantic Language
Team Project

• Implement a compiler for a new language called Drift, by modifying the existing PT Pascal compiler.

• You will get your own copy of the PT compiler source code for your team, to change as you wish.

• The project consists of four assignments, changing each of the four phases of the compiler.

• Both your TA and I will give you advice on how to change each phase - you can use the advice, or do it your own way.

• Project is done in teams of 3.

• You must choose your team and sign the contract by next Wednesday.

• Be careful that all team members stay involved in all phases - course material is learned from doing the project!
The Drift Language

- Drift is a new modern compiled language that borrows features from the modern Ruby, Swift and Turing languages (but from our point of view, it is just an extension of PT Pascal)

- Drift changes PT in the following ways:
  - Ruby-style programs and statements
  - Ruby modules (anonymous classes) added
  - Ruby explicit `end` on all statements replaces PT `begin ... end`
  - Turing general `loop` statement replaces PT `repeat ... until`
  - Swift general `switch` statement with `default` replaces PT `case`
  - Swift-style declarations
  - Ruby `elsif` clause added to `if` statements
  - Turing-style `string` type and operations replaces PT `char` type
  - Swift-like `/* ... */` and `//` comments replace PT `{ ... }` and `(* ... *)`

- And for 858 project teams only ...
  - Drift adds Swift-style `functions` as well
• `begin ... end` → Ruby explicit `end` on all statements, no semicolons

```
program foo(input);

procedure bar(a:integer);
begin
  ...
end;

begin
  if a < b then
    begin
      ...
    end
  else
    begin
      while c < d do
        begin
          ...
        end
      end;
end.
```

```
extern input

procedure bar(a:integer)
begin
  ...
end

begin
  if a < b then
    begin
      ...
    end
  else
    begin
      while c < d
        begin
          ...
        end
      end
    end
end.
```
Ruby-style `elsif` clause

```ruby
if a < b then
  begin
    ...
  end
else
  if b < c then
    begin
      ...
    end
  elsif b < c then
    else
      if c < d then
        begin
          ...
        end
      else
        begin
          ...
        end;
  end
else
  begin
    ...
  end
else
end
```

```ruby
if a < b then
  ...
elsif b < c then
  ...
else
  elsif b < c then
    else
      if c < d then
        ...
      else
        ...
  end
else
  else
    begin
      ...
    end
end
```
CISC 458*/858* - Project 2015

• Pascal `case` -> Swift `switch` statement

```pascal
  case x of
    1:
      begin
        ...
        ...
      end
    2:
      begin
        ...
        ...
      end
    3:
      begin
        ...
        ...
      end
  end;

  switch x
    case 1:
      ...
    case 2:
      ...
    case 3:
      ...
  default:
    ...
  end
```
• Pascal `repeat ... until` → Turing generalized `loop`

```
repeat
...
until x=10;
loop
...
break if x==10
...
end
```
CISC 458*/858* - Project 2015

• Turing/Swift-style declarations

**PT Pascal**

```pascal
const a = 1;
b = 2;
c = 3;

type t = integer;

var d: integer;
e: integer;
f: integer;
```

**Drift**

```drift
let a = 1
let a = 2
let a = 3

type t: integer

var d,e,f: integer
```
CISC 458*/858* - Project 2015

• Swift/Ruby-style **funcs** and Ruby **modules** (anonymous classes)

```plaintext
extern input, output
var x: integer

module M
    func c (var a: integer)
        ...
    end

    func d * (var b: integer)
        ...
    end
end

end

d(x) // ok
c(x) // error!
```
CISC 458*/858* - Project 2015

• Turing-style **string** type, Ruby-style assignments

    extern input, output  
    var x, y: string

    x = "foo"  
    y = x + "bar"   // y == "foobar"  
    n = #y         // n == 6  
    x = y :: 3..5   // x == "oba"

• **PT Pascal** has only the **char** type, which is removed from **Drift**
CISC 458*/858* - Project 2015

• Swift-style **functions**

```swift
func inc (x: integer): integer
    return x + 1
end
```

• Returned type can be **integer**, **string**, **boolean** or a subrange

• **PT Pascal** has no valued functions, this is a new functionality!
An Example Drift Program

// Program to conjugate regular French verbs
extern input,output

var infinitive: string
var root: string

loop
    write ("Please give me a regular French 'er' verb "
        + "(end with 'arreter')") writeln

    read (infinitive)

    write ("Thanks, here is the present conjugation") writeln
    root = infinitive :: 1..(#infinitive - 2)

    write ("The root of this verb is '"', root, "'") writeln

    if infinitive :: (#infinitive - 1)..#infinitive == "er" then
        if (root :: 1..1 == "a") or (root :: 1..1 == "e")
            or (root :: 1..1 == "i") or (root :: 1..1 == "o")
            or (root :: 1..1 == "u") then
            write ("J'" + root + "e")
        else
            write ("Je " + root + "e")
    end
An Example Drift Program (cont’d)

```
write ("Tu " + root + "es") writeln
write ("Il ou elle " + root + "e") writeln
writeln

if root :: #root .. #root == "g" then
    write ("Nous " + root + "eons")
else
    write ("Nous " + root + "ons")
end
writeln

write ("Vous " + root + "ez") writeln
write ("Ils ou elles " + root + "ent") writeln
writeln

elsif infinitive :: (#infinitive - 1)..#infinitive == "ir" then
    write ("I'm too tired to do an 'ir' verb") writeln
else
    write ("I don't like the looks of this verb") writeln
end

break if infinitive == "arreter"
end
```
module queue

var qend, qstart : 0 .. qsizem1
varqlength : 0 .. qsize
var contents : array [0 .. qsizem1] of itemtype

func enqueue * (item : itemtype)
    qlength = qlength + 1
    qend = (qend + 1) mod qsize
    contents [qend] = item
end

func dequeue * (var item : itemtype)
    item = contents [qstart]
    qstart = (qstart + 1) mod qsize
    qlength = qlength - 1
end

func empty * (var yes : boolean)
    if qlength == 0 then
        yes = true
    else
        yes = false
    end
end

// initialization
qend = 0; qstart = 1; qlength = 0
end
Another Example Drift Program

// Example queuing program
extern input,output

module spooler
    let qsize = 100
    let qsizem1 = 99
    type itemtype: string
    module queue
        ... // queue module from previous slide
end

func getdata *
    var s: string
    read (s)
    while s != "."
        enqueue (s)
        read (s)
    end
end

func putdata *
    var yes: boolean
    var s: string
    loop
        empty (yes)
        break if yes
        write (s, " ")
    end
    writeln
end
end //spooler
Another Example Drift Program (cont’d)

// The main program, using the modules
write ("Type a few line followed by a line containing only " + "a '.' each time you are asked to spool.") writeln

loop
  write ("Spool input please: ") writeln
  getdata
  writeln
  write ("Thanks – return to continue, ^D to stop") writeln
  break if eof (input)
end

writeln
write ("Unspooled from spool: ") writeln
putdata
CISC 458*/858* - Project Schedule

Team Project

• **Today** - project specification handed out
  
  • **First tutorial** - tomorrow (Thursday) evening at 6:00 pm, Dunning Hall Room 11 (don’t miss it!)

• **Next week:**
  
  • **Team contracts** due in class next Wednesday
  
  • First assignment handed out: **Scanner/Screener** phase
Syntax/Semantic Language

S/SL - Syntax/Semantic Language

• Special purpose language (DSL) specifically designed for implementing compilers

• Designed for specifying multi-pass compilers

• Main language (other than PT Pascal) used in this course - you will have to write S/SL programs in quizzes and on the exam

• S/SL itself is a dataless executable specification language - programs can define only constants (no variables or assignment!)

• All real data manipulation done indirectly in black boxes called semantic mechanisms (implemented elsewhere)
The S/SL Computational Model

Three Components

- The S/SL program (a finite-state control)
- An integer pushdown return stack implementing S/SL rule calls
- Semantic mechanisms - abstract computation modules
The S/SL Processing Model

Pipe-and-Filter Chain

- Input/output stream model designed for joining S/SL programs end-to-end pipe-and-filter style to form multi-pass solutions
- Well suited to compiler implementation
The S/SL Processing Model with Error Streams

Pipe-and-Filter Chain

- Error stream normally does not go on to next pass, rather translated into error messages for user
Semantic Mechanisms

Enforced Information Hiding

- **S/SL** itself is, for all intentions, a *dataless* programming language
- No variables, assignments, or data operations - all data manipulation is *indirectly* carried out in semantic mechanisms
- A small number of named *constants* may be referred to, and the value of the current *input/output/error token*, but *no operations* can be performed other than those provided by *semantic mechanisms*
- Semantic mechanisms augment this dataless world with a set of *black boxes* with buttons (operations) allowing indirect data manipulation
Semantic Mechanisms

- Opaque abstract data types (like static class interfaces)
  - organize semantic operations
  - indirect manipulation of data
  - interface defined in S/SL, but implementation is hidden from the S/SL program
Semantic Mechanisms

Example

• A Symbol Table Mechanism

• From the point of view of the S/SL program, looks like an opaque magic black box

• Internally, has data structures to implement a scoped table of identifiers and their attributes
Summary

Team Project

- This year’s project modifies PT Pascal to be a new modern language called Drift
- Modules, syntax, elsif from Ruby and Swift
- Modern strings from Turing

The S/SL Computational Model

- S/SL is a dataless push-down transducer language well suited to implementing multi-pass compilers
- Semantic mechanisms augment S/SL with black-box data manipulation modules that can be specified, invoked and queried from S/SL programs, but whose implementation is strictly hidden (enforced information hiding)

Next

- S/SL syntax, program structure and operations