Principles of Programming Languages
Lecture 14

Wael Aboulsaadat

wael@cs.toronto.edu

http://portal.utoronto.ca/

Acknowledgment: parts of these slides are based on material by Diane Horton & Eric Joanis @ UoT

References: Scheme by Dybvig; PL Concepts and Constructs by Sethi; Concepts of PL by Sebesta; ML for the Working Prog. By Paulson; Prog. in Prolog by Clocksin and Mellish; PL Pragmatics by Scott

ML: exceptions

- **How to handle an exception?**
  - Syntax

```
<expression>
handle  <exception_1> => <exception-handler_1>
|       <exception_2> => <exception-handler_2>
|       ….
|       <exception_n> => <exception-handler_n>
```

  - If no exceptions are raised, then return the value of **<expression>**
  - If **<exception$_i$>** is raised then return the value of **<exception-handler$_i$>**
    - Only the first matching exception is considered.

- **Example:** N! / (M! (N-M)!)

```
- exception Negative of int;
- exception TooBig   of int;
- fun comb (N,M) =
      if N < 0 then raise Negative(N)
        else if M < 0 then raise Negative(M)
              else if M > N then raise TooBig(M)
                  else
                    fact(N) div (fact(M) * fact(N-M));
  > val comb = fn : int * int -> int
```

```
-  fun mycomb (N,M) =
       comb(N,M)
       handle Negative(X)  => ~1
       |         TooBig(M)  => 0;

> val mycomb = fn : int * int -> int
- mycomb(11,8);
> val it = 165 : int
- mycombt(~5,123);
> val it = ~1 : int
```

ML: exceptions & scopes

- **Suppose f calls g calls h, and h raises an exception:** *g handler is used*

f
<f-expression> → (calls g)
handle
<exception$_i$> =>
<exception-handler$_f$>

g
<g-expression> → (calls h)
handle
<exception$_i$> =>
<exception-handler$_g$>

h
……
……
raise <exception$_i$>
……

- **Example:**

```
- exception e1;
- exception e2;
- exception e3;
- fun  h(1) = raise e1
  |    h(2) = raise e2
  |    h(3) = raise e3
  |    h(_) = "ok";
- fun g(N) =  h(N)
      handle e2 => "error g2"
      |          e3 => "error g3";
- fun f(N) =   g(N)
      handle e1 => "error f1"
      |          e2 => "error f2";
```

```
- f(4);
> val it = "ok" : string
- f(3);
> val it = "error g3" : string
- f(2);
> val it = "error g2" : string
- f(1);
> val it = "error f1" : string
- f(0);
> val it = "ok" : string
```

ML: structures

- **Syntax**

```
structure <structure-name> =
struct
        (* exceptions, definitions, functions… *)
end
```

```
structure Mapping =
struct
    fun insert(key,value,[]) = [(key,value)]
    |    insert(key,value,(key1,value1)::rest) =
            if key = key1 then
                (key,value)::rest
            else
              (key1,value1)::insert(key,value,rest);

    fun lookup(key,(key1,value1)::rest) =
        if key = key1 then
            value1
        else
            lookup(key,rest);
end;
```


ML: structures – cont'd

- **Structure access:**
  - Using long identifier
    - E.g.
      ```
      - Mapping.insert(538,"languages",[]);
      > val it = [(538,"languages")] : (int * string) list

      - Mapping.lookup(538,[(538,"languages"),(540,"courses")]);
      > val it = "languages" : string
      ```
  - Using open function
    - E.g.
      ```
      - open Mapping;
      - lookup(538,[(538,"languages"),(540,"courses")]);
       > val it = "languages" : string
      ```

ML: structures – cont’d

- **Properties**
  - It is legal to define one structure within another
  - If a structure has been defined within another structure, then its components can be accessed by an extension of the long identifier principle (x.y.z…)
  - A structure may be opened within another to achieve greater modularity. However, this may lead to name redefinition problems
  - There is no equality defined over structures.

ML: signatures

- **Syntax**

```
signature <signature-name> =
sig
        (* definitions *)
end;
```

- **Example:**

```
- signature OBJ_sig =
 sig
        type OBJECT
        val grow : OBJECT -> OBJECT
        val shrink: OBJECT -> OBJECT
 end;
```

ML: signatures

- **Signatures & Structures:**

```
- signature OBJ_sig =
  sig
      type OBJECT
      val grow : OBJECT -> OBJECT
      val shrink: OBJECT -> OBJECT
  end;

- structure INT_struct : OBJ_sig =
  struct
      type OBJECT = int
      fun grow n    = n + 1
      fun shrink n  = n –1
  end;
```

- **Benefits of using signature:**
  – Separation of specification from implementation decisions
  – Ability to provide programmers with different views of source code

- ***If a structure <u>implements</u> a signature, then this structure is said to be <u>constrained</u> by this signature.***

ML: signatures

- **Rules of signatures**
  - Rule 1: name matching
  - Rule 2: type matching
  - Rule 3: privacy
    - Any definition within a constrained structure that is not matched within its signature is private.
      - Such definition cannot be referenced by long identifier nor is it is made available if the structure is opened

```
signature FOO =
  sig
        val talkToMe : unit -> int
  end;
  structure Foo2 : FOO =
  struct
        val bar = 42
        fun talkToMe () = bar
        fun hidden() =  (* more code *)
  end;
```

ML: signatures

- **Properties**
  - There is no equality defined for signatures.
  - They are top-level objects, and cannot be defined within another object; furthermore *(unlike structures)* they cannot be nested.
  - The keyword include can be used to save writing long signatures by incorporating the contents of existing signatures within a new definition:
    - E.g.

```
signature NUM_sig    =
sig
    include OBJ_sig
    val Int_to_OBJ : int   -> OBJECT
    val Real_to_OBJ: real -> OBJECT
end
```

Part 2: Language Design

Language Specification: syntax vs. semantics

- **Syntax**
  - The structural rules of a language that determine the *form* of a program written in the language
  - Examples:
    - In C, variable names can be followed by two adjacent + symbols (Index++)
    - In Java, the main method must be defined as public static void main(…)
    - In C++/C, the if statement is written as if(<expression>) <block> else <block>
- **Semantics**
  - The *meaning* of the various language constructs in the context of a given program
  - Examples:
    - In C ‘j = Index++;’ means “increment Index after assigning its value to j”
    - In Java, defining a main method in a class means you can start the program by invoking that class from the command line.
    - In C++/C, the if statement means a selection construct that allows programmer to express one of two possible execution paths depending on some condition.

Language Specification: syntax vs. semantics

Fortran

```
        SUM = 0
        DO 11 K=1,N
        SUM = SUM + 2 * K
11      CONTINUE
```

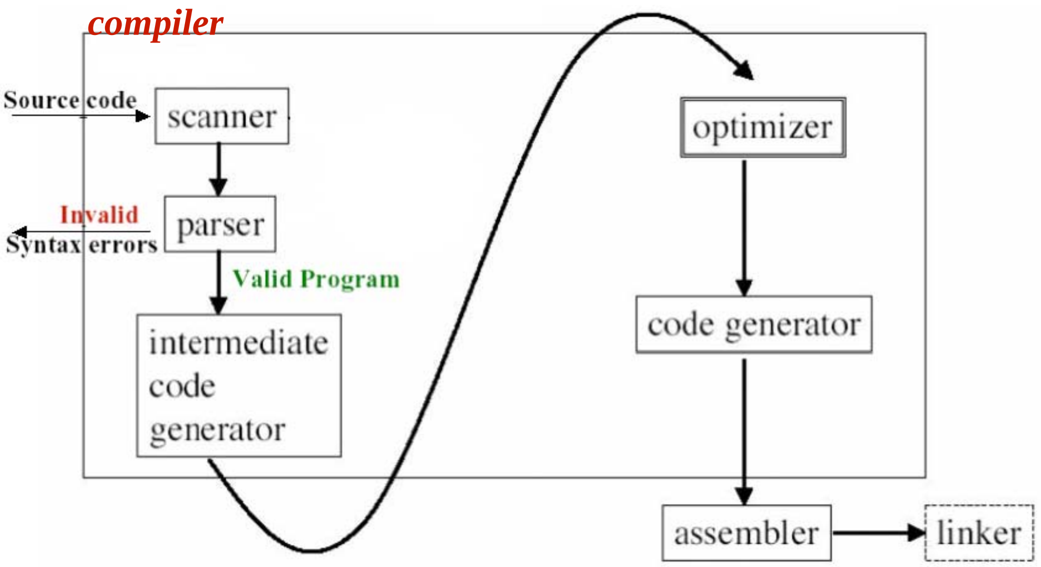
C (Java/Javascript)

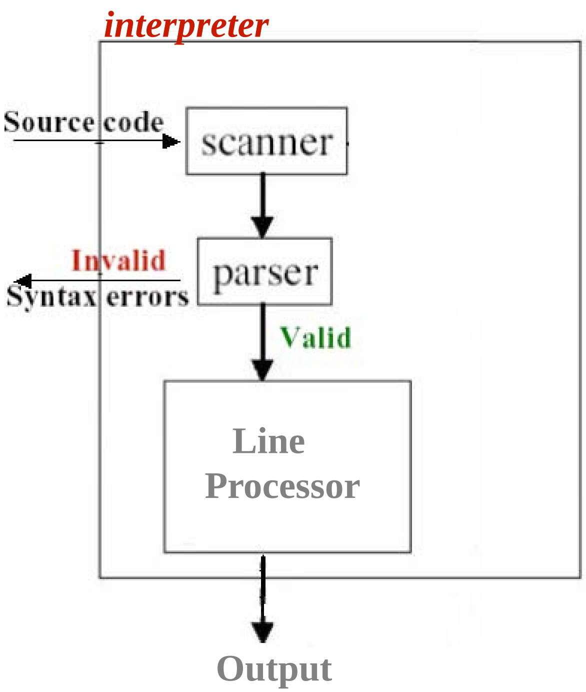
```
sum = 0;
for (k=1; k <= n; ++k)
   sum += 2*k;
```

Pascal

```
sum := 0;
for k:= 1 to n do
   sum := sum + 2 * k;
```

Language Specification: compilation vs. interpretation





Language Specification: Scanner & Parser

- **Scanner**
  - Divides program into sentences and tokens. Checks identifier format.

token1

↓

```
sum = 0;
for (k=1; k <= n; ++k)
   sum += 2*k;
```

```
sum = 0;for (k=1; k <= n; ++k) sum += 2*k;
```

Language Specification: Scanner & Parser

- **Scanner**
  - Divides program into sentences and tokens. Checks identifier format.

```
sum = 0;
for (k=1; k <= n; ++k)
   sum += 2*k;
```

token2

↓

```
sum = 0;for (k=1; k <= n; ++k) sum += 2*k;
```

Language Specification: Scanner & Parser

- **Scanner**
  - Divides program into sentences and tokens. Checks identifier format.

```
sum = 0;
for (k=1; k <= n; ++k)
   sum += 2*k;
```

token3

```
sum = 0;for (k=1; k <= n; ++k) sum += 2*k;
```

Language Specification: Scanner & Parser

- **Scanner**
  - Divides program into sentences and tokens. Checks identifier format.

```
sum = 0;
for (k=1; k <= n; ++k)
   sum += 2*k;
```

token4

```
sum = 0;for (k=1; k <= n; ++k) sum += 2*k;
```

Language Specification: Scanner & Parser

- **Scanner**
  - Divides program into sentences and tokens. Checks identifier format.

```
sum = 0;
for (k=1; k <= n; ++k)
   sum += 2*k;
```

```
sum = 0;for (k=1; k <= n; ++k) sum += 2*k;
```

Language Specification: Scanner & Parser

- **Scanner**
  - Divides program into sentences and tokens. Checks identifier format.

```
sum = 0;
for (k=1; k <= n; ++k)
   sum += 2*k;
```

```
sum = 0;for (k=1; k <= n; ++k) sum += 2*k;
```

- **Parser**
  - Decides if the program is written according to language specification

Language Specification: Scanner & Parser

- **Scanner**
  - Divides program into sentences and tokens. Checks identifier format.

```
sum = 0;
for (k=1; k <= n; ++k)
   sum += 2*k;
```

```
sum = 0;for (k=1; k <= n; ++k) sum += 2*k;
```

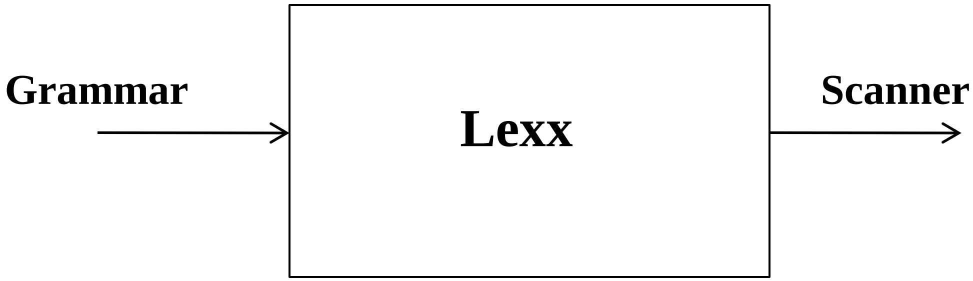
- **Parser**
  - Decides if the program is written according to language specification

```
sum = 0;for (k=1; k <= n; ++k) sum += 2*k;
```

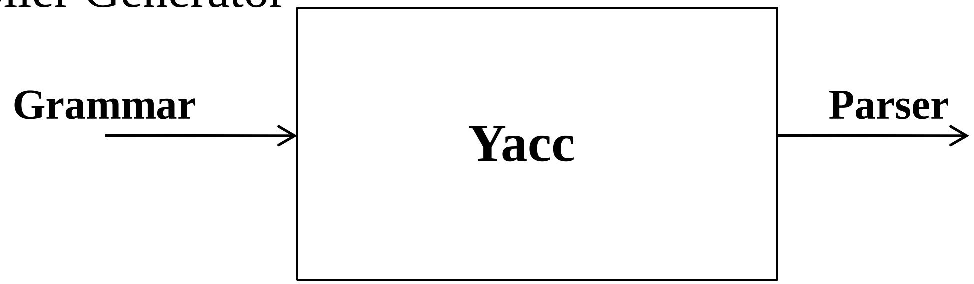
What this is?
Is this a valid assignment
Yes → Cool, let's move forward
No → what can it be?... Cant figure out! → programmer error!

Language Specification: Lexx & Yacc

- **Lexx**
  - Lexical Analyzer
  - Scanner Generator



- **Yacc**
  - Yet Another Compiler Compiler
  - Compiler Generator



Grammar: introduction

- **Grammar:**
  - A Grammar is a formalism that describes which sequence of terminals are meaningful in a PL. Formally, it is defined as a quadruple (*N*, *T*, *P*, *S*) where:
    - *N* is the set of symbols called *Nonterminals*
    - *T* is the set of symbols called *Terminals*
    - *P* is the set of *productions*
    - *S* subsetof N is the nonterminal called the *starting symbol*
  - Example:

    G = (N,T,P,S) where N = {S} , T = {a,b},

    P={S → aS, S → bS, S → }

- **Production:**
  - A ***production*** is a rule of the form X → Y where X is a string of symbols (*terminals or nonterminals*) containing at least one nonterminal, and Y is a string of symbols (*terminals or nonterminals*)

Grammar: context free

- **A *context free grammar* (CFG) is a grammar in which |X| = 1, i.e. X is a single nonterminal**
  - LHS: 1 nonterminal
  - RHS: a sequence of terminals and nonterminals
  - E.g.
    - S → ab (CFG)
    - SA → ab (non CFG)

- **CFG is sufficient to describe most of the constructs in programming languages**

- **Programming languages describable by CFG are recognizable by push down automata (*analogues to FSA with a stack*)**

Language Specification : example

- **Consider the 'language' of noun phrases**

  It was a sunny day.

  We had a picnic in a lovely secluded park.

- **A *grammar* for simple noun phrases:**

  noun-phrase → *adjective-list* noun

  adjective-list → adjective adjective*

  ** Indicate zero or more times*

Language Specification : example derivation

- **It was a <u>sunny day</u>.**

noun-phrase → *adjective-list* noun

adjective-list → adjective adjective*

Language Specification : example derivation

- **It was a <u>sunny day</u>.**

noun-phrase → *adjective-list* day

adjective-list → adjective adjective*

Language Specification : example derivation

- It was a sunny day.

noun-phrase → *adjective-list* day

adjective-list → adjective adjective*

Language Specification : example derivation

- **It was a <u>sunny day</u>.**

noun-phrase → *adjective-list* day

adjective-list → adjective adjective*

Language Specification : example derivation

- **It was a <u>sunny day</u>.**

noun-phrase → *adjective-list* day

adjective-list → sunny adjective*

Language Specification : example derivation

- **It was a sunny day.**

noun-phrase → *adjective-list* day

adjective-list → sunny adjective*

Language Specification : example derivation

↓

- **It was a <u>sunny day</u>.**

noun-phrase → *adjective-list* day

adjective-list → sunny

↑

Language Specification : example derivation

- It was a sunny day.

noun-phrase → sunny day

adjective-list → sunny

Language Specification : example derivation

- **It was a <u>sunny day</u>.**

noun-phrase → *adjective-list* noun

adjective-list → adjective adjective*

Language Specification : example derivation

- **It was a <u>sunny day</u>.**

noun-phrase → *adjective-list* noun

adjective-list → adjective adjective*

Language Specification : example derivation

↓

- **It was a <u>sunny day</u>.**

noun-phrase → *adjective-list* noun

adjective-list → sunny adjective*

↑

Language Specification : example derivation

- **It was a <u>sunny day</u>.**

noun-phrase → *adjective-list* noun

adjective-list → sunny adjective*

Language Specification : example derivation

- **It was a <u>sunny day</u>.**

noun-phrase → *adjective-list* noun

adjective-list → sunny adjective*

Language Specification : example derivation

- **It was a sunny day.**

$$\textit{noun-phrase} \rightarrow \textit{adjective-list}\ \text{noun}$$
$$\textit{adjective-list} \rightarrow \text{sunny}$$

Language Specification : example derivation

- **It was a <u>sunny day</u>.**

noun-phrase → sunny noun

adjective-list → sunny

Language Specification : example derivation

↓

- **It was a <u>sunny day</u>.**

↓

noun-phrase → sunny day

adjective-list → sunny