Revision

Course Objectives

1. Appreciate the relationship between most mainstream programming languages
2. Further your programming skills
3. Make learning of new programming languages easier in the future
4. Understand the workings of compilers and interpreters
5. Understand aspects of large-scale software development, and how they can be tackled at the programming language level
6. Enhance communication skills

Workings of Compilers and Interpreters

- Semantics
  - Helps reason about program correctness
  - Indicates what kind of low-level instruction a program executes
- Execution time
  - Certain programming constructs take longer to execute than others
  - Some languages are inherently slower than others
- Memory usage
  - Some primitive operations do not have constant memory usage
  - Garbage collection sometimes adds extra overhead
- Programming platform
  - Understand the difference between language primitive and standard library procedure/method
  - Understand the role of compiler, linker, and loader

Large Scale Software Development

- Cost-effective large-scale software development requires modular development, as well as compliance with a complicated set of rules
- Cooperation requires communication, but communication is costly, and desirable to be minimized
- Module systems implement efficient communication and cooperation between team members
- As for the rules, without an enforcement system, programmers often "stray", leading to increased costs later in the process
- Some languages try to enforce such rules, leading to lower development costs

Communication Skills

- Every discipline has its formative values, developing skills that are outside the profession.
- Study of programming languages makes one pay more attention at what information needs to be communicated, and the format in which this information must be expressed in order to be well understood.

Syllabus

1. Introduction
   - Concepts, classifications, bird's eye view of PL universe
2. Assembly languages, and relationship to C
3. Languages, grammars, regular expressions
4. Data types and expressions
5. Sequential programming; semantics
6. Stateful and non-stateful programming
7. Procedural abstraction; higher order programming
8. Lazy evaluation
9. Types
10. Object oriented programming

Syllabus

11. Exception handling
12. Concurrency
13. Rule-based programming
14. Constraint programming

Programming Paradigms

- Imperative Programming
- Logic Programming
- Functional programming
- Object-oriented programming
- Constraint programming
- Event-driven programming
- Aspect-oriented programming (not covered)
- Orthogonal paradigmatic features
  - Typing
    - Statically/dynamically typed
    - Strongly/weakly typed
  - Strictness: strict/lazy
  - Concurrency: fine/coarse grain

Why C?

- Portable assembly language
- Low overheads compared to real assembly languages
- Most compilers and interpreters for other languages are written in C
- Good background in PL:
  - Relationship of other high level languages to C
  - Relationship of C to assembly languages

Execution of C Programs

- Imperative paradigm
  - Sequential execution of statements
  - Based on the notion of **state**
    - Entire contents of memory accessible to the program
      - global and local variables/procedure arguments
      - dynamically allocated memory
  - Each statement takes the current state to a new state
- Demonstrated in **step-by-step execution** in IDEs/debuggers
  - Visual C++ Express Edition (Windows)
  - Code::Blocks (Windows/Linux)

Systematic Translation Scheme

- Algorithmic procedure for translating a *program skeleton* into an equivalent one.
- Works for all possible programs.
- Can be implemented as a translator or compiler
- Must be specified in enough detail to make the implementation possible.

Assembly Languages

- Means of making machine languages more readable
- Execution unit: *instruction*
  - Very limited amout of computation
- No structured programming
- Programs: large in terms of lines of code
- Different for each architecture
  - Pentium AL $\neq$ MIPS AL
- We abstract AL as a subset of C
  - Interest in low-level programming skill
  - No interest in specific architecture

VAL: Vanilla Assembly Language

- Data:
  - 6 global variables: `int eax, ebx, ecx, edx, esi, edi ;`
  - One global array: `unsigned char M[10000] ;`
  - No local variables!
- Functions:
  - One single function contains all the code
  - Prototype: `void exec(void) ;`

VAL: Vanilla Assembly Language

- Data:
  - 6 global variables: `int eax, ebx, ecx, edx, esi, edi ;`
  - One global array: `unsigned char M[10000] ;`
  - No local variables!
- Functions:
  - One single function contains all the code
  - Prototype: `void exec(void) ;`

Simulate registers

VAL: Vanilla Assembly Language

Simulate registers

- Data:
  - 6 global variables: `int eax, ebx, ecx, edx, esi, edi ;`
  - One global array: `unsigned char M[10000] ;`
  - No local variables!

Simulate memory

- Functions:
  - One single function contains all the code
  - Prototype: `void exec(void) ;`

VAL: Vanilla Assembly Language

Simulate registers

- Data:
  - 6 global variables: `int eax, ebx, ecx, edx, esi, edi ;`
  - One global array: `unsigned char M[10000] ;`
  - No local variables!

Simulate memory

- Functions:
  - One single function contains all the code
  - Prototype: `void exec(void) ;`

Placeholder for code, just to comply with C syntax.

Grammars and Regular Expressions

- **Languages:** how to define
- **Grammars**
  - Specification of languages
  - Language generators
  - Derivations
  - Analysis
  - Language Structure
- **Regular Languages**
  - Regular grammars
  - Deterministic Finite Automata
  - Regular expressions
  - Use of REs in Ruby

Grammars and Regular Expressions

- Programming languages are specified by *grammars*, in a stratified manner.
- The lower level, that of *lexical analysis*, uses *regular grammars*, and their counterpart, *regular expressions*.
  - convert a program into a sequence of *lexemes* — more in tutorial
- The higher level, called *syntactic analysis* uses more sophisticated grammars.
  - capture the structure of the language;
  - use *lexemes* as terminals
  - shall be covered in more detail next time
- *Regular expressions* are a basic data type in Ruby.
  - Ruby can be used to build a *toy lexer*.
  - Examples in the tutorial.

Prolog

Type the following into file `example.pl`

```
parent(john,george).
parent(john,mary).
parent(george,adam).
parent(george,beth).
parent(adam,james).
```

Facts act like a database, and define a relation. They do not contain variables.

Queries:

```
?- consult(example).

?- parent(george,adam).
true.

?- parent(john,X).
X = george ;
X = mary

?- parent(james,X).
false
```

Prolog

Type the following into file `example.pl`

Predicate symbol

```
parent(john,george).
parent(john,mary).
parent(george,adam).
parent(george,beth).
parent(adam,james).
```

Facts act like a database, and define a relation. They do not contain variables.

Queries:

```
?- consult(example).

?- parent(george,adam).
true.

?- parent(john,X).
X = george ;
X = mary

?- parent(james,X).
false
```

Prolog

Ground terms (no vars, but functors allowed).

Type the following into file `example.pl`

```
parent(john,george).
parent(john,mary).
parent(george,adam).
parent(george,beth).
parent(adam,james).
```

Facts act like a database, and define a relation. They do not contain variables.

Queries:

```
?- consult(example).

?- parent(george,adam).
true.

?- parent(john,X).
X = george ;
X = mary

?- parent(james,X).
false
```

Prolog

Typical extension of Prolog programs

Type the following into file `example.pl`

```
parent(john,george).
parent(john,mary).
parent(george,adam).
parent(george,beth).
parent(adam,james).
```

Facts act like a database, and define a relation. They do not contain variables.

Queries:

```
?- consult(example).

?- parent(george,adam).
true.

?- parent(john,X).
X = george ;
X = mary

?- parent(james,X).
false
```

Prolog

Prolog interactive prompt.

Typed by user

Type the following into file `example.pl`

```
parent(john,george).
parent(john,mary).
parent(george,adam).
parent(george,beth).
parent(adam,james).
```

Facts act like a database, and define a relation. They do not contain variables.

Queries:

```
?- consult(example).

?- parent(george,adam).
true.

?- parent(john,X).
X = george ;
X = mary

?- parent(james,X).
false
```

Prolog

Multiple answers for one query

Type the following into file `example.pl`

```
parent(john,george).
parent(john,mary).
parent(george,adam).
parent(george,beth).
parent(adam,james).
```

Answers (interpreter output)

Facts act like a database, and define a relation. They do not contain variables.

Queries:

```
?- consult(example).

?- parent(george,adam).
true.

?- parent(john,X).
X = george ;
X = mary

?- parent(j
false
```

Typed by user to move on to next answer

Prolog Operators

- Infix and functional notations are equivalent

```
?- +(a,b) = a+b.
true.

?- X = +(a,b).
X = a + b.
```

- Expression pattern matching

```
?- X+Y = a+b+c.
X = a+b
Y = c

?- X+Y = a+b*c.
X = a
Y = b*c

?- (a+b)*(X-Y) = Z*(3-4).
Z = a+b
X = 3
Y = 4
```

The `op` Declaration

```
?- X = 1 a 3. % error
?- op(100,yfx,a). % declare a as operator
?- X = 1 a 3. % succes
?- a(b,c) = b a c. % succes
?- a(a,a) = a a a. % success
?- +(+,+) = + + + . % success, note the spaces
?- X a Y = 1 a 2 a 3. % X = 1 a 2, Y= 3
?- X a Y = 1 a (2 a 3). % X = 1, Y = 2 a 3
?- X a Y = 1 a 2 a 3 a 4. % X = 1 a 2 a 3, Y = 4
?- X + Y a Z = 1 + 2 a 3. % X=1, Y=2, Z=3
?- 1+2 a 3 = +(1,a(2,3)). % succes
```

Example: `while` language

```
?- op(950,fx,while).

?- op(949,xfx,do).

?- X = (while x>0 do {x=x-1 ; y = y+x } ).
```

succeeds

```
?- (while B do S) = (while x>0 do {x=x-1 ; y = y+x } ).
```

succeeds with `B = x>0` and

`S = { x=x-1 ; y=y+x}`

```
?- (while B do S) = while(do(B,S)).
```

succeeds

A Simple Programming Language

```
?- op(1099,yf,;).
?- op(960,fx,if).
?- op(959,xfx,then).
?- op(958,xfx,else).
?- op(960,fx,while).
?- op(959,xfx,do).
?- op(960,fx,switch).
?- op(959,xfx,of).
?- op(970,xfx,::).
```

```
?- Code = (
            a = 1 ;
            switch a of {
            0:: { x = 1 ; z = x+1 ;} ;
            2:: { x = 2 ;
                  x = x - 1 ;
                  z = x << 3 ; } ;
            default:: {
                  x = 10 ;
                  y = 5 ;
                  z = 0 ;
                  while ( y > 0 ) do {
                      z = z + x ;
                      y = y - 1 ;
                  } ; } ;
            };
          ), compileHL(Code,Tac).
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \hat{}\ s} \qquad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \qquad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \qquad s \in \{a, \ldots, z\}$$

```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= <subexpr> <term> ['+'|'-']
              | <>
<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::= '(' <expr> ')'
              | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \quad s = s_1 s_2$$

If string s_1 is generated by nonterminal <subexpr>

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{+,-\}$$

$$\frac{}{\texttt{<subexpr>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{*,/\}$$

$$\frac{}{\texttt{<subterm>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \,\hat{}\; s} \quad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \quad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \quad s \in \{a, \ldots, z\}$$

```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= <subexpr> <term> ['+'|'-']
              | <>
<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::= '(' <expr> ')'
              | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \hat{}\ s} \quad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \quad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \quad s \in \{a, \ldots, z\}$$

If string s_1 is generated by nonterminal <subexpr>

And string s_2 is generated by nonterminal <term>

```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= <subexpr> <term> ['+'|'-']
              | <>
<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::= '(' <expr> ')'
              | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \qquad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \qquad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \qquad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \qquad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \qquad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \qquad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \text{\textasciicircum}\, s} \qquad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \qquad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \qquad s \in \{a, \ldots, z\}$$

If string s_1 is generated by nonterminal <subexpr>

And string s_2 is generated by nonterminal <term>

Then string s is generated by nonterminal <expr>

```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= <subexpr> <term> ['+'|'-']
              | <>
<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::= '(' <expr> ')'
              | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \text{\textasciicircum}\, s} \qquad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \qquad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \qquad s \in \{a, \ldots, z\}$$

If string s_1 is generated by nonterminal **<subexpr>**

And string s_2 is generated by nonterminal **<term>**

Then string s is generated by nonterminal **<expr>**

Where s is s_1 concatenated with s_2.

```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= <subexpr> <term> ['+'|'-']
              | <>
<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::= '(' <expr> ')'
              | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \hat{\ }\, s} \qquad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \qquad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \qquad s \in \{a, \ldots, z\}$$

```
<expr>       ::= <subexpr> <term>
<subexpr>    ::= <subexpr> <term> ['+'|'-']
               | <>
<term>       ::= <subterm> <factor>
<subterm>    ::= <subterm> <factor> ['*'|'/']
               | <>
<factor>     ::= <base> <restexp>
<restexp>    ::= '^' <base> <restexp>
               | <>
<base>       ::= '(' <expr> ')'
               | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{+,-\}$$

$$\frac{}{\texttt{<subexpr>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{*,/\}$$

$$\frac{}{\texttt{<subterm>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \widehat{\ }\, s} \qquad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \qquad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \qquad s \in \{a, \ldots, z\}$$

```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= <subexpr> <term> ['+'|'-']
              | <>
<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::= '(' <expr> ')'
              | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \texttt{\^{}}\ s} \qquad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \qquad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \qquad s \in \{a, \ldots, z\}$$

```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= <subexpr> <term> ['+'|'-']
              | <>
<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::= '(' <expr> ')'
              | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash\, \hat{}\ s} \qquad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash\texttt{<>}}$$

etc...

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \qquad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \qquad s \in \{a, \ldots, z\}$$

```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= <subexpr> <term> ['+'|'-']
              | <>
<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::= '(' <expr> ')'
              | a | b | c | d
```

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Piggyback on the syntax analyzer

Building an $\boxed{\text{AST}}$

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Second argument is where the AST is output

Piggyback on the syntax analyzer

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Residual operator

Third argument of some nonterminals

Piggyback on the syntax analyzer

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Convert ASCII code S into atom A.

Piggyback on the syntax analyzer

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).
```

```
build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Piggyback on the syntax analyzer

Tree building:

- If the residual operator is `nil`, just pass the current tree up.
- If the residual operator is not nil, then `L` contains the tree components, which must be assembled into a term.

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Query:

```
1 ?- S="(((a+b)*c/d^e^f-g)^(a*b)+c)*(a+b)",
     expr(S,T), T =.. L, S =.. X.
S = [40, 40, 40, 97, 43, 98, 41, 42, 99|...],
T = (((a+b)*c/d^e^f-g)^ (a*b)+c)* (a+b),
L = [*, ((a+b)*c/d^e^f-g)^ (a*b)+c, a+b],
X = ['.', 40, [40, 40, 97, 43, 98, 41|...]].
```

Syntax Analysis and Semantics

- Reasoning rules are a general formalism for specifying computational mechanisms.
- Syntax analysis can be specified as reasoning rules.
- Prolog rules can easily implement reasoning rules.
- Heuristics need to be employed to make the rules really "computational"
- A syntax analyzer can be easily augmented to produce an AST.

Aggregate Types

- Arrays

- Most langauges provide *records*
  - In C they are called *structures*
  - In object oriented programming they are extended to *objects*

- Unions: specific to C, help save space.

- High-level aggregate datatypes (Python, Ruby):
  - Lists
  - Tuples
  - Sets
  - Dictionaries
  - implemented in libraries for languages without these primitives

Aggregate Types

- Arrays
- Most langauges provide *records*
  - In C they are called *structures*
  - In object oriented programming t
- Unions: specific to C, help save spac
- High-level aggregate datatypes (Pyth
  - Lists
  - Tuples
  - Sets
  - Dictionaries
  - implemented in libraries for languages without these primitives

```
struct struct_name {
  type1 field1 ;
  type2 field2, field3 ;
  type3 field4[10] ;
} var1, var2, * var3 ;
```

Aggregate Types

- Arrays
- Most langauges provide *records*
  - In C they are called *structures*
  - In object oriented programming they are extended to *objects*
- Unions: specific to C, help save space.

```
union union_name {
  type1 field1 ;
  type2 field2, field3 ;
  type3 field4[10] ;
} var1, var2, * var3 ;
```

- High-level aggregate datatypes (Python
  - Lists
  - Tuples
  - Sets
  - Dictionaries
  - implemented in libraries for languages without these primitives

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

f is

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

f is a function

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

f is a function that returns a pointer

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

`f` is a function that returns a pointer to a function

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

f is a function that returns a pointer to a function that returns a pointer

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

f is a function that returns a pointer to a function that returns a pointer to int

Hierarchic Data

Type operators have precedence: `()` and `[]` bind tighter than `*` ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

`a` is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Hierarchic Data

Type operators have precedence: `()` and `[]` bind tighter than `*` ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

a is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Hierarchic Data

Type operators have precedence: `()` and `[]` bind tighter than `*` ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

a is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Hierarchic Data

Type operators have precedence: `()` and `[]` bind tighter than `*` ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

a is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Hierarchic Data

Type operators have precedence: `()` and `[]` bind tighter than `*` ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

`a` is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Hierarchic Data

Type operators have precedence: `()` and `[]` bind tighter than `*` ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

a is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Hierarchic Data

Type operators have precedence: `()` and `[]` bind tighter than `*` ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

a is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Each layer of pointers must be intialized.

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Drop the a to obtain the cast.

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Move one * from right side to the left side

Continue on for each level of pointers

Prolog and Oz Terms

- Terms represent tree-like symbolic data
- They are common to symbolic processing languages: Prolog, Ocaml, Haskell, Scheme
- Allow *pattern-matching*: operation that allows extraction of components of syntactic structures.
- In Prolog, it is not possible to specify that the argument is *limited* to a set of terms
- Typed languages, such as Ocaml and Haskell, allow specification of such restrictions.

```
toString(E1+E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"+",S2,")"],S).
toString(E1-E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"-",S2,")"],S).
toString(E1*E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"*",S2,")"],S).
toString(E1/E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"/",S2,")"],S).
toString(X,[Y])   :- atom(X), char_code(X,Y).
```

Dynamic vs. Static Typing

- Dynamic Typing (Prolog, Python, Ruby, Javascript):
  - Each datum is stored with its type
  - Before each operation, the type is checked
    - If cast is possible, then operation proceeds after cast
    - If cast is not possible, the operation fails with error or exception
    - Prolog predicates may just fail with no error message $\rightarrow$ debugging becomes difficult
  - Less efficient, due to extra tests at execution time

- Static Typing (C, Ocaml, Haskell, Java, C#):
  - Types are inferred at compiled time
  - Data are stored without type
  - If cast is necessary, code for cast is compiled into the executable
  - If cast is not possible, compilation error is issued
  - More restrictive, since inferring types at compile time is weaker than finding out the types directly during execution.
  - More efficient execution, due to lack of type checks at run time.

Terms in Haskell

Data type declaration

```
data Expr = Plus Expr Expr          -- means a + b
          | Minus Expr Expr         -- means a - b
          | Times Expr Expr         -- means a * b
          | Divide Expr Expr        -- means a / b
          | Value String            -- "x", "y", "n", etc.
```

Haskell code

```
toString (Plus left right)   = "(" ++ (toString left) ++ "+" ++ (toString right) ++ ")"
toString (Minus left right)  = "(" ++ (toString left) ++ "-" ++ (toString right) ++ ")"
toString (Times left right)  = "(" ++ (toString left) ++ "*" ++ (toString right) ++ ")"
toString (Divide left right) = "(" ++ (toString left) ++ "/" ++ (toString right) ++ ")"
toString (Value s)           = s
```

Equivalent Prolog code:

```
toString(E1+E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"+",S2,")"],S).
toString(E1-E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"-",S2,")"],S).
toString(E1*E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"*",S2,")"],S).
toString(E1/E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"/",S2,")"],S).
toString(X,[Y])   :- atom(X), char_code(X,Y).
```

Semantics

- Mathematical description of the execution model of a language.
- Essentially a translation mechanism.
- Assumption: we know the description language.
- The semantics helps us learn the new language (for which the semantics is defined).
- Each construct of the new language must be described somehow.
- Generalization: any translation of the new language into an already known language can be construed as semantics.
- Compiler: falls into this category too.

Semantics of Toy Language

- Expressed as reasoning rules
- The context is an *environment*
  - Mapping from variable names to addresses
  - Assume that it already contains all the variables in the programs
  - Later we learn how to build it dynamically
- We assume we can generate new labels on the fly
  - Later we see how we can implement this
- Each rule handles the right hand side of a production in the grammar

Semantics of "While" Statements

$$\frac{\mathcal{E} \vdash [\![\, E_1 \,]\!] = C_1 \qquad \mathcal{E} \vdash [\![\, E_2 \,]\!] = C_2 \qquad \mathcal{E} \vdash [\![\, S \,]\!] = C_3}{\mathcal{E} \vdash [\![\texttt{while(}E_1 \oplus E_2\texttt{)do } S\,]\!] = \begin{array}{ll} '' & \texttt{Lwhile:}\ '' \\ & C_1 \\ & C_2 \\ '' & \texttt{ecx = *(int*)\&M[esp] ; esp += 4 ;} \\ & \texttt{eax = *(int*)\&M[esp] ; esp += 4 ;} \\ & \texttt{if (eax } \oplus \texttt{ ecx) goto Lwhilebody ;} \\ & \texttt{goto Lendwhile;} \\ & \texttt{Lwhilebody:}\ '' \\ & C_3 \\ '' & \texttt{goto Lwhile;} \\ & \texttt{Lendwhile:}\ '' \end{array}} \quad \oplus \in \{<, >, =<, >=, ==, != \}$$

Rule for "While"

```
compile(while B do S,Ein,Eout,Tin,Tout,Lin,Lout) :- !,
        B =.. [O,X,Y], La1 is Lin+1,
        (   O == (\=) -> Otrans = '!=' ; Otrans = O ),
        write('Lwhile'),write(Lin),writeln(':'),
        compileExpr(X,Ein,Ea1,Tin,Ta1),
        compileExpr(Y,Ea1,Ea2,Ta1,Ta2),
        writeln('    ecx = *(int*)&M[esp] ; esp += 4 ;') ,
        writeln('    eax = *(int*)&M[esp] ; esp += 4 ;') ,
        write('    if ( eax '), write(Otrans),
        write(' ecx ) goto Lwhilebody'), write(Lin), writeln(';'),
        write('    goto Lendwhile'),write(Lin),writeln(';'),
        write('Lwhilebody'),write(Lin),writeln(':'),
        compile(S,Ea2,Eout,Ta2,Tout,La1,Lout),
        write('    goto Lwhile'),write(Lin),writeln(';'),
        write('Lendwhile'),write(Lin),writeln(':').
```

"While" Example

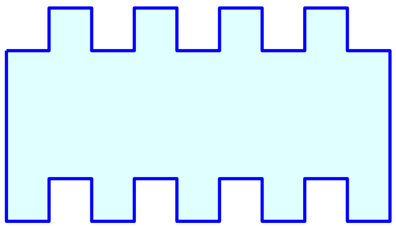
```
?- compile(while x < y do x=x+1,[],Eout,0,Tout,0,_).
Lwhile0:
    ecx = *(int*)&M[0] ; esp -= 4 ; *(int*)&M[esp] = ecx ; // push x
    ecx = *(int*)&M[4] ; esp -= 4 ; *(int*)&M[esp] = ecx ; // push y
    ecx = *(int*)&M[esp] ; esp += 4 ;
    eax = *(int*)&M[esp] ; esp += 4 ;
    if ( eax < ecx ) goto Lwhilebody0;
    goto Lendwhile0;
Lwhilebody0:
    ecx = *(int*)&M[0] ; esp -= 4 ; *(int*)&M[esp] = ecx ; // push x
    esp -= 4 ; *(int*)&M[esp] = 1 ; // push 1
    ecx = *(int*)&M[esp] ; esp += 4 ;
    eax = *(int*)&M[esp] ; esp += 4 ;
    eax += ecx ;
    esp -= 4 ; *(int*)&M[esp] = eax ; // push result of +
    ecx = *(int*)&M[esp] ; esp += 4 ;
    *(int*)&M[0] = ecx ; // pop x
    goto Lwhile0;
Lendwhile0:
Eout = [ (y->4), (x->0)],
Tout = 8.
```

Compositionality

- Small components are combined together to form bigger components.
- The bigger components can be further combined in the same way.
- Similar to *Lego bricks*
- Takes a certain amount of skill to design a compositional architecture

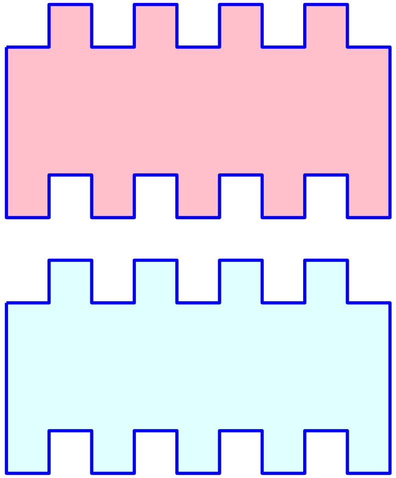
Compositionality

- Small components are combined together to form bigger components.
- The bigger components can be further combined in the same way.
- Similar to *Lego bricks*
- Takes a certain amount of skill to design a compositional architecture



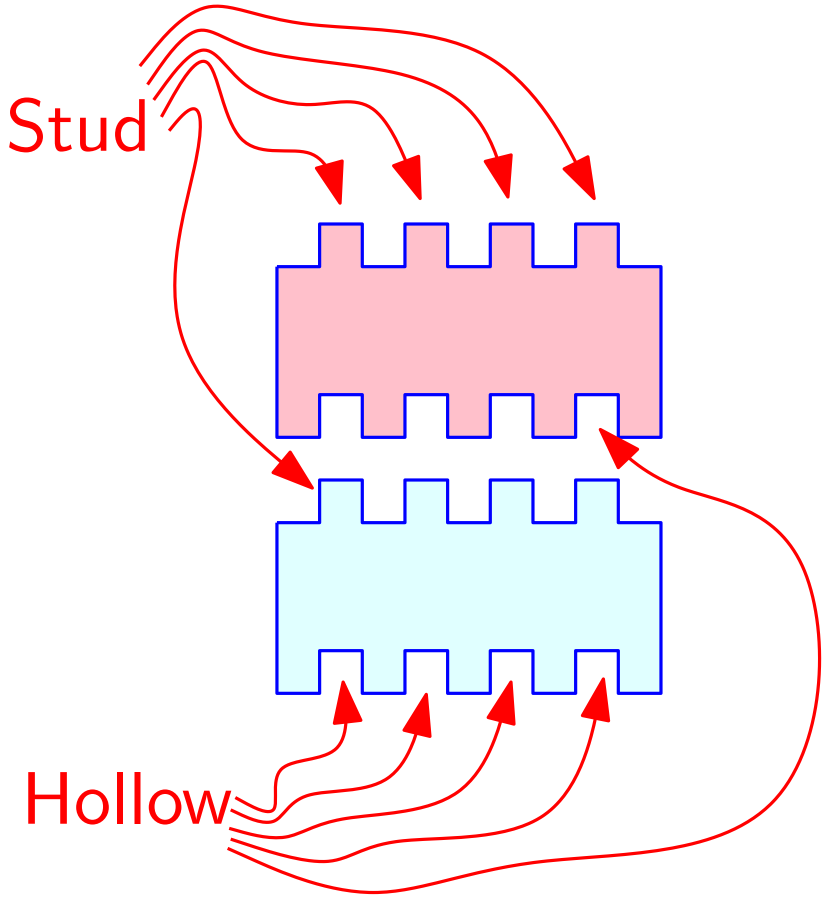
Compositionality

- Small components are combined together to form bigger components.
- The bigger components can be further combined in the same way.
- Similar to *Lego bricks*
- Takes a certain amount of skill to design a compositional architecture



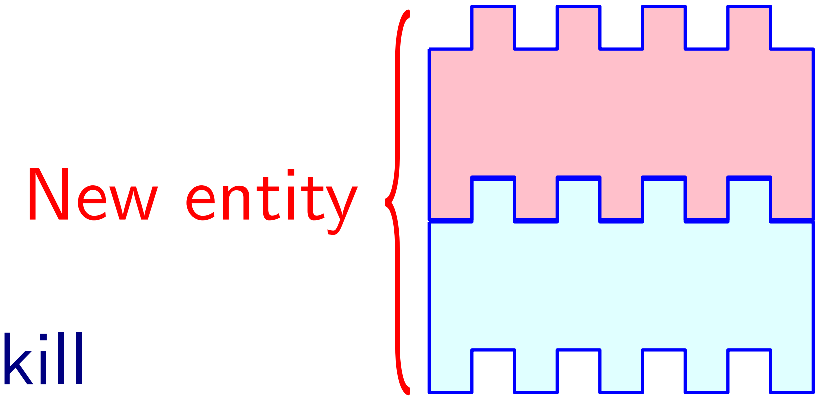
Compositionality

- Small components are combined together to form bigger components.
- The bigger components can be further combined in the same way.
- Similar to *Lego bricks*
- Takes a certain amount of skill to design a compositional architecture



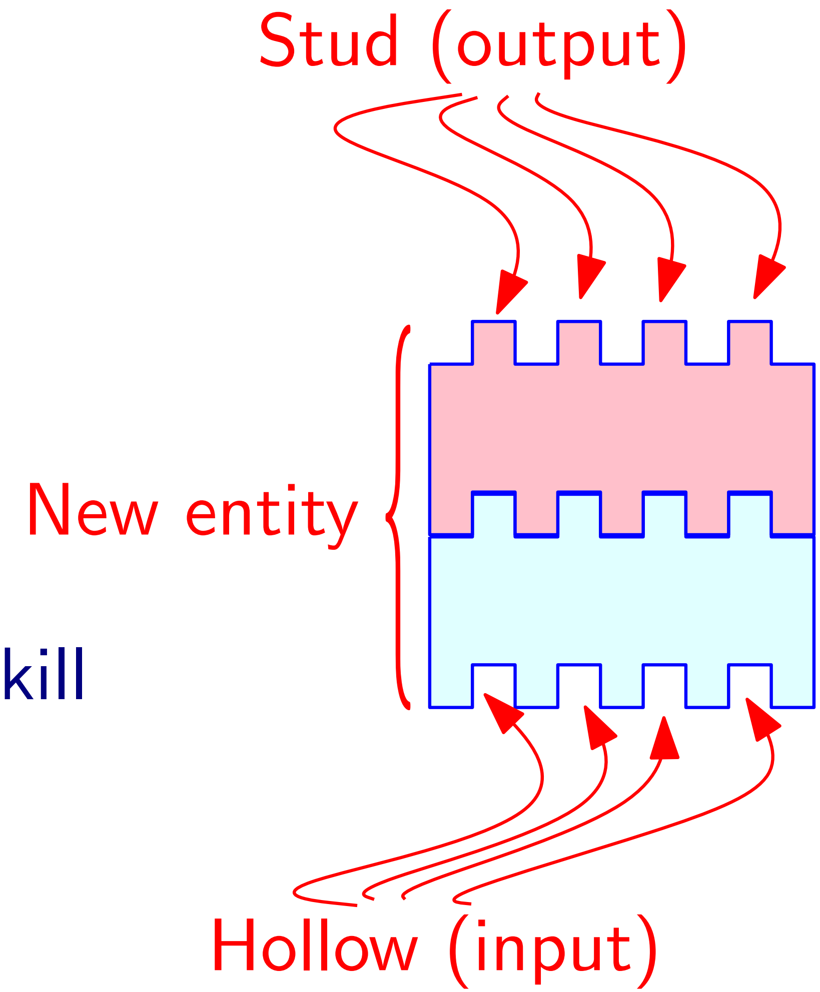
Compositionality

- Small components are combined together to form bigger components.
- The bigger components can be further combined in the same way.
- Similar to *Lego bricks*
- Takes a certain amount of skill to design a compositional architecture



Compositionality

- Small components are combined together to form bigger components.
- The bigger components can be further combined in the same way.
- Similar to *Lego bricks*
- Takes a certain amount of skill to design a compositional architecture



The new entitiy can be further combined!

Semantics

- Semantics is a description of the execution model in a language we supposedly already know
- Compilation is an instance of giving a semantics to a language
- *Compositionality:* important principle that enables the process of defining a semantics
- Prolog is perfectly equipped for writing toy compilers by following semantic rules.

Procedures

- Means of factorizing and reusing code.
- Inspired by mathematical functions.
- Resemble mathematical functional notation.
- Two parts: *definition* and *invocation* (or *call*).
- Definition: has *formal arguments*, that are also used in the body.
- Invocation: has *actual arguments*, to which the formal arguments are *bound* once the procedure is entered.
- May have a *return value*.

Abstraction

- Means of hiding details
- *Abstraction barrier:* defining an interface to a system.
  - Describes a set of operations without details on how the operations are implemented.
  - Allows freedom of changing the implementation later, as long as the high-level operations do not change their behaviour.
- Example: set implementation
  - Operations: union, intersection, difference, etc:
  - Could be implemented as a linked list, or as a bitmap
  - Implementor can change from one implementation to the other, as long as the operations do not change their meaning.

Procedural Abstraction

- Collections of procedures are assembled into *libraries* and *modules*
- We use the library as a *black box*: we learn the interface, and we don't care about the exact implementation.
- The implementor of the library has the freedom to change the implementation as long as the interface stays the same.
- The interface of the library acts as an *abstraction barrier* to the user.
- Devising good abstraction barriers is hard, but the benefit is huge!
  - Makes "using software" much easier than "implementing software"

Procedures as First-Class Values

- *First class value:* entity that:
  - can become the value of a variable
  - can be used as an argument to, or return value from a function
  - can be created as an unnamed value

- Most modern languages allow *functions as first-class values*

- Exceptions:
  - C — only allows pointers to functions as function arguments
  - Prolog — allows dynamic modification of programs by adding and deleting rules, but not the creation of unnamed predicates

- Functions as unnamed entities:
  - Scheme: `(lambda (x) (+ x 1))` — `((lambda (x) (+ x 1)) 5)` $\equiv$ 6
  - Ocaml: `fun x -> x+1` — `(fun x -> x+1) 5` $\equiv$ 6
  - Haskell: `\ x -> x+1` — `(\ x -> x+1) 5` $\equiv$ 6
  - Python: `lambda x: x+1` — `(lambda x: x+1)(5)` $\equiv$ 6

HOP Primitives

- Higher order programming simplifies programming over collections (lists, sets, bags, dictionaries)

- Primitives of higher order programming
  - `map` : apply a function to every element of a collection and create a similar collection of results
  - `fold` : combine all the elements of a collection via an operator
  - `filter` : remove from a collection the elements that do not satisfy a predicate
  - `zip` : create a collection of pairs, each pair being made up of elements of the same rank in two input collections

- They form a very useful *abstraction barrier*

Procedure Implementation

- Parameter passing
  - Actual arguments must be bound to formal arguments
- Return mechanism
  - Upon return from invocation, control must be transferred back to callee
- Local variable allocation
  - Each invocation must have a separate set of local variables, to allow for recursion
- Solution: *Activation Record*

The Language Oz

- Combines Prolog-style unification with higher-order programming techniques specific of functional languages.
- Allows stateful programming in an elegant manner.
- Fine-grain concurrency model that we'll study later.
- No implicit backtracking.
- Terms are still allowed as data
- Arithmetic expressions get evaluated and have value.
- No op declarations.

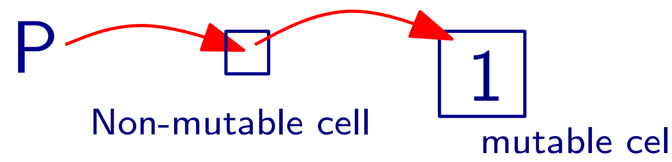
Oz vs. Prolog

- Variables are single assignment, and must have capitalized names.
  - Variables must be declared, and they are initially *unbound*.
- There are no predicates. Procedures can be declared as abstractions, and assigned to variables (consequently, procedure names are capitalized).
- Equality denotes unification.
  - When unification fails, an exception is thrown, rather than trigger backtracking.
- In a procedure, any variable can be used for input or output, like in Prolog.

Stateful Programming in Oz

```
declare
fun {Factorial N}
   P = {NewCell 1}
   proc {Helper N}
      if ( N == 0 )
      then skip
      else
            P := @P * N
            {Helper N-1}
      end
   end
   in
   {Helper N}
   @P
end
{Browse {Factorial 5}}
```

Stateful Programming in Oz




```
declare
fun {Factorial N}
   P = {NewCell 1}
   proc {Helper N}
      if ( N == 0 )
      then skip
      else
            P := @P * N
            {Helper N-1}
      end
   end
   in
   {Helper N}
   @P
end
{Browse {Factorial 5}}
```

create new mutable cell
local function
dereference and assignment to mutable cell
access value of mutable cell
operator evaluated right away
sequential execution
body of function
return value (cell dereference)

Types in Programming

- A type is a collection of computational entities that share some common property.
- There are 3 main uses:
  - Naming and organizing concepts.
  - Making sure that bit sequences in memory are interpreted consistently.
  - Providing information (e.g. size) to the compiler about data manipulated by the program
- Type error: when computational entity is used in an inconsistent manner.

Type Safety

- A PL is type safe is no program is allowed to violate type distinctions.
- More specifically, a data of a given type cannot be "seen" as data of another type
  - *in-situ* casts are not type safe
  - pointer arithmetic is not safe
  - consequently C is not type-safe
- Compile-time vs Run-time type checking
  - Run-time checking: data is paired with its type during execution
    - type consistency is checked before every operation
    - type of data may change during execution
    - overhead incurred
  - Compile-time checking: type consistency is checked at compile time
    - Type is stripped from data during run-time
    - Data cannot change its type during execution
    - No type consistency checks at execution; no overhead

Type Inference

- Type safe languages:
  - Strongly typed: all type consistency can be checked at compile-time; there's no need for run-time checks.
  - Weakly typed: some type consistency checks must be done at run-time

- Some strongly typed languages infer (rather than just check) the types of their data
  - Haskell
  - Ocaml

- Type inference can be viewed as a type of semantics and can be defined via reasoning rules.

Polymorphism and Overloading

- Polymorphism: a symbol may have multiple types simultaneously
- Forms of polymorphism:
  - Parametric polymorphism: function may be applied to any arguments whose types match a type expression involving type variables – Haskell and Ocaml fall into this category.
  - Ad-hoc polymorphism: (also known as overloading): two or more implementations with different types are referred to by the same name
  - Subtype polymorphism: a *subtype* relation is defined between types; an expression with a given type can be used as argument anywhere where a subtype of the current type is expected – Haskell also has this form of polymorphism via *type classes* (not covered).

Haskell

- Functional, strongly typed, polymorphic, lazy (non-strict)
- Named after **Haskell Curry** – pioneer of **lambda calculus**
- Many implementations (some quite efficient), many extensions
- Elegant, theoretically clean
- Very well supported, see `www.haskell.org`

Typing Rules

$$\frac{\Gamma_1, \Delta_1 \vdash e_1 :: T_1 \quad \Gamma_2, \Delta_2 \vdash e_2 :: T_2}{\Gamma_1 \cup \Gamma_2, \Delta_1 \cup \Delta_2 \cup \{T_1 = T_2 \rightarrow T_3\} \vdash (e_1\, e_2) :: T_3} \quad (\text{APP})$$

T_3 is a new type variable

$$\frac{\{x_1 :: T_1\}, \emptyset \vdash x :: T_1 \quad \Gamma \cup \{x :: T_1'\}, \Delta \vdash e :: T_2}{\Gamma, \Delta \cup \{T_1 = T_1'\} \vdash \backslash x \rightarrow e :: T_1 \rightarrow T_2} \quad (\text{ABS})$$

Lazy Evaluation

- Haskell uses *memoized call by name*
- Argument to function is not computed before call; rather it is substituted for the formal argument as an expression.
- Substitution may occur in multiple places; upon the first evaluation, the value of the expression is *memoized* (i.e. stored for later use), and all subsequent references to the expression will access the memoized value, rather than recompute
- An expression that appears as actual argument may never be computed.
- Infinite computations, or exceptional conditions such as division by zero become less dangerous

Purity

- Functions with side effect: when called multiple times with same arguments, returns different results
  - Requires assignment
  - Do not mix well with lazy evaluation, since every expression is evaluated only once – value is memoized, and re-used in subsequent occurrences of same expression.

- Pure function: Function without side-effect.
  - Preferred in a lazy evaluation setting

- Pure language: Language where it is impossible to write functions with side-effects.
  - Usually assignment is removed
  - Haskell is a pure language

Infinite Lists

- Due to lazyness, we can *specify* a list without end
  - Ok as long as we *don't use all the list*
  - Specification is simpler and more elegant as compared to finite lists.
- The list comprehension `[k..]` denotes the infinite list that starts at `k` and contains all the numbers greater than `k` in increasing order.
- Useful only if we only take a finite number of elements in the list
- Using recursion we can define infinite lists containing any series
- Also called **streams**.
- Lead to simple, elegant programs, all due to lazy evaluation

Hamming Numbers

```
hamming = 1 :
          map (2*) hamming
           `merge`
          map (3*) hamming
           `merge`
          map (5*) hamming
  where
  merge (x:xs) (y:ys)
    | x < y = x : xs `merge` (y:ys)
    | x > y = y : (x:xs) `merge` ys
    | otherwise = x : xs `merge` ys
```

Modularity: Basic Concepts

- Component
  - Meaningful program unit
    - Function, data structure, module, ...
- Interface
  - Types and operations defined within a component that are visible outside the component
- Specification
  - Intended behavior of component, expressed as property observable through interface
- Implementation
  - Data structures and functions inside component

Abstract Data Types

- Prominent language development of 1970's
- Main ideas:
  - Separate interface from implementation
    - Example:
      - Sets have empty, insert, union, is_member?, ...
      - Sets implemented as ... linked list ...
  - Use type checking to enforce separation
    - Client program only has access to operations in interface
    - Implementation encapsulated inside ADT construct

Object-oriented programming

- Primary object-oriented language concepts
  - dynamic lookup
  - encapsulation
  - inheritance
  - subtyping
- Program organization
  - Work queue, geometry program, design patterns
- Comparison
  - Objects as closures?

Objects

- An object consists of
  - hidden data
    - instance variables, also called member data
    - hidden functions also possible
  - public operations
    - methods or member functions
    - can also have public variables in some languages

hidden data	
msg_1	$method_1$
. . .	. . .
msg_n	$method_n$

- Object-oriented program:
  - Send messages to objects

Object-Orientation

- Programming methodology
  - organize concepts into objects and classes
  - build extensible systems
- Language concepts
  - dynamic lookup
  - encapsulation
  - subtyping allows extensions of concepts
  - inheritance allows reuse of implementation

Language concepts

- "Dynamic lookup"
  - different code for different object
  - integer "+" different from real "+"
- Encapsulation
  - Implementer of a concept has detailed view
  - User has "abstract" view
  - Encapsulation separates these two views
- Subtyping
- Inheritance

Subtyping and Inheritance

- Interface
  - The external view of an object
- Subtyping
  - Relation between interfaces
- Implementation
  - The internal representation of an object
- Inheritance
  - Relation between implementations

Arrogant Lecturer: C Equivalent

```
struct alecturer {
       void (*say)(struct speaker * self, char* msg) ;
       void (*lecture)(struct lecturer * self, char* msg) ;
       void (*super_say)(struct speaker * self, char* msg) ;
};

void alecturer_say(struct alecturer * self, char * msg){
     char * p = malloc(200) ;
     *p = '\0' ;
     strcat(p,"It is obvious that " ) ;
     strcat(p,msg) ;
     self->super_say(self,p) ;
}

void init_alecturer(struct alecturer *p) {
       init_lecturer(p) ;
       p->super_say = p->say ;
       p->say = alecturer_say ;
}

struct alecturer * make_alecturer() {
       struct alecturer * retVal = malloc(sizeof(struct alecturer));
       init_alecturer(retVal) ;
       return retVal ;
}
```

Exceptions

- Useful for error handling
- Two parts:
  - `try` statement with `catch/finally` clauses
  - `throw/raise` statement: execution jumps to the `catch` clause that can handle the exception
- Without function calls: similar to a labeled `break`
- With function calls: non-local returns

`setjmp/longjmp` in C

- `int setjmp(jmp_buf env)`
  - Sets up the local `jmp_buf` buffer and initializes it for the jump.
  - Saves the program's calling environment in the environment buffer `env`.
  - Direct invocation: `setjmp` returns 0.
  - Return from call to `longjmp`: returns nonzero.
- `void longjmp(jmp_buf env, int value)`
  - Restores context of environment buffer `env`.
  - The value specified by value is passed from `longjmp` to `setjmp`.
  - Program execution continues as if the corresponding invocation of `setjmp` had just returned.
  - `value != 0` -> `setjmp` returns 1 ; otherwise returns value.

Unchecked Exceptions: h

Original code:

```
int h() {
  …
  try {
    …
    E1 e1 ;
    …
    throw e1 ;
    …
    E2 e2 ;
    …
    throw e2 ;
    …
  } catch (E2 e2) {
    …
  } finally {
    …
  }
  …
  return R ;
}
```

C translation

```
int h() {
  …
  if (setjmp(push())) {
    …
    exception.T = E1 ;
    exception.V = e1 ;
    longjmp(pop(),1) ;
    …
    exception.T = E2 ;
    exception.V = e2 ;
    longjmp(pop(),1) ;
    …
    pop();
  } else {
      switch ( exception.T ) {
         case E2 : // handle E2
           …
           exception.T = NOEXCEPTION ;
           goto finally ;
         default:
         finally:
           …
           if ( exception.T != NOEXCEPTION )
             longjmp(pop(),1) ;
      }
  }
  …
  return R ;
}
```

Declarative Concurrent Programming

- What stays the same
  - the result of your program
  - concurrency does not change the result
- What changes
  - programs can compute incrementally
  - incremental input… (such as reading from a network connection) … and incremental processing

Threads

- A **thread** is simply an executing program.
- A program can have more than one thread.
- A thread is created by :

  thread ⟨s⟩ **end**

- Threads compute
  - independently
  - as soon as their statements can be executed
  - interact by binding variables in store

Nondeterminism

- An execution is *nondeterministic* if there is a computation step in which there is a **choice** what to do next
- Nondeterminism appears naturally when there are multiple concurrent states

Declarative concurrency

- *Declarative programming (Reminder)*:
  - the output of a declarative program should be a mathematical function of its input.
- *Functional programming (Reminder)*:
  - the program executes with some input values and when it terminates, it has returned some output values.
- *Data-driven concurrent model*: a **concurrent** program is **declarative** if all executions with a given set of inputs have one of two results:
  - (1) they all do not terminate or
  - (2) they all eventually reach partial termination and give results that are logically equivalent.

Scheduling

- A scheduler is *fair* if it does not starve each runnable thread
  - All runnable threads execute eventually
- Fair scheduling makes it easier to reason about programs
- Otherwise some runnable programs will never get its turn for execution.

Streams

- A most useful technique for declarative concurrent programming to use **streams** to communicate between threads.
- A **stream** is a potentially unbounded list of messages, i.e., it is a list whose tail is an unbound dataflow variable.
- A thread communicating through streams is a kind of “active object”, also called **stream object**.
- A sequence of stream objects each of which feeds the next is called a **pipeline**.
- **Deterministic stream programming**: each stream object always knows for each input where the next message will come from.

Where do we go from here?

- Remember: Languages are just tools
- Essential complementary knowledge:
  - Software Engineering
- Possible next modules:
  - CS4215: PL Implementation
  - CS4212: Compiler Design
  - CS4216: Constraint Logic Programming

Good Luck with your Exams!