Overview

- Organization
- Course overview
- Introduction to programming concepts
Organization

Organizational

- I need some feedback
  - Tutorials/exercises
  - Assignment 1

- How does the reading go
  - Chapter 1
Reading Suggestions

- Chapter 2
  - Sections 2.1 – 2.3 [careful]
  - Section 2.4 – 2.5 [browse]
  - Section 2.6 [careful]

- And of course the handouts!

Summary So Far

- We know about functions
  - recursive functions
  - how to compose them
  - touched on higher order functions

- We know about partial values
  - bound and unbound variables (single assignment, dataflow)
  - numbers and atoms
  - tuples, lists, records
  - unification

- We know (a bit) about a declarative programming model
  - functions of partial values
Questions?

- Now is the time to ask!

Overview

- We are finishing “Introduction to programming concepts”
  - procedures
  - local declarations
  - translating programs to kernel language

- We are starting with computation model of declarative programming
Towards the Model

This is the outlook section

Confusion

- By now you should feel uneasy and slightly embarrassed (maybe even confused)
- We haven’t explained how computation actually proceeds
- No, you are fine? Wait and see…
Another Length

\[
\text{fun } \{L \ Xs \ N\} \\
\text{ case } Xs \\
\text{ of } \text{nil} \text{ then } N \\
[\ ] \ X|Xr \text{ then } \{L \ Xr \ N+1\} \\
\text{ end} \\
\text{ end} \\
\text{fun } \{\text{Length } Xs\} \\
\{L \ Xs \ 0\} \\
\text{ end}
\]

Comparison

- This length is six-times faster then our first one!
  - hey, it has one argument more!
  - so what
  - what could be the difference
  - and what is more: it takes considerable less memory!
  - actually, it runs in constant memory!

- Our model will answer
  - intuition: even though recursive it executes like a loop
There Is No Free Lunch!

- Before we can answer the questions we have to make the language small
  - sort out what is primitive: kernel language
  - what can be expressed

- Kernel language
  - based on procedures
  - no functions

What Is a Procedure?

- It does not return a value
  - Java: methods with `void` as return type

- But how to return a value anyway?
  - Idea: use an unbound variable
  - Why: we can supply value later (before return)
  - Aha: so that's why we have been dwelling on this!
Our First Procedure: Sum

```
proc {Sum Xs N}
    case Xs
    of nil then N=0
        [] X|Xr then N=X+{Sum Xr}
    end
end
```

- Hey, we call `Sum` as if it was a function
  - that's okay. It is just syntax
  - we'll sort that out next week

Being More Primitive

```
proc {Sum Xs N}
    case Xs
    of nil then N=0
        [] X|Xr then
            local M in {Sum Xr M} N=X+M end
    end
end
```

- Local declaration of variables
- Needed to fully base kernel language on procedures
What is Computation Model

- Formal language
  - Syntax
- Semantics
  - How sentences of the language are executed on (an abstract) machine
- Precise model
  - Allows reasoning about program correctness
  - Allows reasoning program’s time complexity
  - Allows reasoning about program’s space complexity

Towards Computation Model

- Step One: Make the language small
  - Transform the language of function on partial values to a small kernel language

- Kernel language
  - procedures: no functions
  - records: no tuple syntax
  - local declarations: no nested calls

Statements and Expressions

- Expressions describe computations that return a value
- Statements just describe computations
  - Transforms the state of a store (single assignment)
- Kernel language
  - The only expressions allowed: value construction for primitive data types
  - Otherwise only statements

What Is a Procedure?

- It does not return a value
  - Java: methods with `void` as return type
- But how to return a value anyway?
  - Idea: use an unbound variable
  - Why: we can supply its value after we have computed it!
  - Aha: so that’s why we have been dwelling on this!
Our First Procedure: Sum

```plaintext
proc {Sum Xs N}
    case Xs
    of nil then N=0
    [] X|Xr then N=X+{Sum Xr}
    end
end

Hey, we call Sum as if it was a function
  that's okay. It is just syntax
```

Being More Primitive

```plaintext
proc {Sum Xs N}
    case Xs
    of nil then N=0
    [] X|Xr then
        local M in {Sum Xr M} N=X+M end
    end
end

Local declaration of variables
Needed to fully base kernel language on procedures
```
Local Declarations

\texttt{local X in ... end}

- Introduces the variable identifier \( X \)
  - visible between \texttt{in} and \texttt{end}
  - called scope of the variable
  - also scope of the declaration

- Creates a new store variable
- Links identifier to store variable
  - also uses an environment
  - more on this later

Abbreviations for Declarations

- Kernel language
  - just one variable introduced
  - no direct assignment

- Programming language
  - several variables
  - variables can be also assigned (initialized) when introduced
Transforming Declarations

Multiple Variables

\[
\begin{align*}
\text{local } X, Y \text{ in} & \quad \text{local } X \text{ in} \\
\langle \text{statement} \rangle & \quad \langle \text{statement} \rangle \\
\text{end} & \quad \text{end}
\end{align*}
\]

Transforming Declarations

Direct Assignment

\[
\begin{align*}
\text{local } X = \langle \text{expression} \rangle \text{ in} & \quad \text{local } X \text{ in} \\
\langle \text{statement} \rangle & \quad \langle \text{statement} \rangle \\
\text{end} & \quad \text{end}
\end{align*}
\]
Transforming Expressions

- Unfold function calls to procedure calls
- Use local declaration for intermediate values
- Order of unfolding:
  - left to right
  - innermost first
  - watch out: different for record construction (later)

Function Call to Procedure Call

\[ X = \{ F \ Y \} \quad \rightarrow \quad \{ F \ Y \ X \} \]
Unfolding Nested Calls

\[
\{ P \{ F \times Y \} Z \} \quad \Rightarrow \quad \begin{align*}
&\text{local } U_1 \text{ in } \\
&\quad \{ F \times Y U_1 \} \\
&\quad \{ P U_1 Z \} \\
&\text{end}
\end{align*}
\]

Unfolding Nested Calls

\[
\{ P \{ F \{ G \times X \} Y \} Z \} \quad \Rightarrow \quad \begin{align*}
&\text{local } U_2 \text{ in } \\
&\quad \text{local } U_1 \text{ in } \\
&\quad \{ G \times U_1 \} \\
&\quad \{ F U_1 Y U_2 \} \\
&\text{end} \\
&\quad \{ P U_2 Z \} \\
&\text{end}
\end{align*}
\]
Unfolding Conditionals

local B in
  if X>Y then
    ...
  else
    ...
end

if B then
  ...
else
  ...
end


Expressions to Statements

X = if B then
  ...
else
  ...
end

if B then
  X = ...
else
  X = ...
end
**Length (0)**

fun \{\text{Length Xs}\}

    case Xs
    of nil then 0
    [] X|Xr then 1+{\text{Length Xr}}
    end

end

**Length (1)**

proc \{\text{Length Xs N}\}

    N=case Xs
    of nil then 0
    [] X|Xr then 1+{\text{Length Xr}}
    end

end

- Make it a procedure
Length (2)

proc {Length Xs N}
  case Xs
    of nil then N=0
    [] X|Xr then N=1+{Length Xr}
  end
end

• Expressions to statements

Length (3)

proc {Length Xs N}
  case Xs
    of nil then N=0
    [] X|Xr then
      local U in
      {Length Xr U}
      N=1+U
    end
  end
end

• Unfold function call
Length (4)

\begin{verbatim}
proc {Length Xs N}
  case Xs
  of nil then N=0
  [] | Xr then
    local U in
    {Length Xr U}
    {Number.'+' I U N}
  end
end
end

- Replace operation (+, dot-access, <, >, ...): procedure!
\end{verbatim}

Summary

- Transform to kernel language
  - function definitions
  - function calls
  - expressions

- Kernel language
  - procedures
  - declarations
  - statements
Programming Model

- **Computation model**
  - describes a language and how sentences (expressions, statements) of the language are executed by an abstract machine

- **Set of programming techniques**
  - expresses solutions to problems you want to solve

- **Set of reasoning techniques**
  - reason about programs to increase confidence that they compute correctly and efficiently
Declarative Programming Model

- Guarantees that computations are evaluating functions on (partial) data structures
- Core of functional programming
  - LISP, Scheme, ML, Haskell
  - Functional part of Erlang
- Core of logic programming
  - Prolog, Mercury
  - Functional (non-relational) part
- Stateless programming
Description of a Language

- **Language = Syntax + Semantics**
- The *syntax* of a language is concerned with the *form* of a program: how expressions, commands, declarations etc. are put together to result in the final program.
- The *semantics* of a language is concerned with the *meaning* of a program: how the programs behave when executed on computers.

Programming Language Definition

- **Syntax**: grammatical structure
  - lexical: how words are formed
  - phrasal: how sentences are formed from words
- **Semantics**: meaning of programs
  - Informal: English documents (e.g. Reference manuals, language tutorials and FAQs etc.)
  - Formal:
    - Operational Semantics (execution on an abstract machine)
    - Denotational Semantics (each construct defines a function)
    - Axiomatic Semantics (each construct is defined by pre and post conditions)
Language Syntax

- Defines *legal* programs
  - programs that can be executed by machine
- Defined by *grammar rules*
  - define how to make ‘sentences’ out of ‘words’
- For programming languages
  - sentences are called statements (commands, expressions)
  - words are called tokens
  - grammar rules describe both tokens and statements

Language Syntax

- *Statement* is sequence of tokens
- *Token* is sequence of characters
- *Lexical analyzer* is a program
  - recognizes character sequence
  - produces token sequence
- *Parser* is a program
  - recognizes token sequence
  - produces statement representation
- Statements are represented as *parse trees*
Backus-Naur Form

- BNF (Backus-Naur Form) is a common notation to define grammars for programming languages
- A BNF grammar is set of grammar (rewriting) rules $\Omega$
- A set of terminal symbols $T$ (tokens)
- A set of Non-terminal symbols $N$
- One start symbol $\sigma$
- A grammar rule
  \[
  \langle \text{nonterminal} \rangle ::= \langle \text{sequence of terminal and nonterminal} \rangle
  \]

Examples of BNF

(A) BNF rules for robot commands
- A robot arm only accepts a command from
  \{up, down, left, right\}
- \langle move \rangle ::= \langle cmd \rangle
- \langle move \rangle ::= \langle cmd \rangle \langle move \rangle
- \langle cmd \rangle ::= up
- \langle cmd \rangle ::= down
- \langle cmd \rangle ::= left
- \langle cmd \rangle ::= right
Grammar Rules

- \(\text{digit}\) is defined to represent one of the ten tokens 0, 1, ..., 9
  
  \[
  \text{digit} ::= 0 | 1 | 2 | 3 | 5 | 6 | 7 | 8 | 9
  \]

- The symbol ‘\(|\)’ is read as ‘or’

- Another reading is that \(\text{digit}\) describes the set of tokens \{0,1,..., 9\}

Examples of BNF

(A) BNF rules for robot commands

- A robot arm only accepts a command from \{up, down, left, right\}
  
  \[
  \text{move} ::= \text{cmd} | \text{cmd} \text{move}
  \]

  \[
  \text{cmd} ::= \text{up} | \text{down} | \text{left} | \text{right}
  \]

- Examples of command sequences:
  
  - up
  - down left
  - up down down right left
Examples of BNF

- Integers
  \[ \langle \text{integer} \rangle ::= \langle \text{digit} \rangle | \langle \text{digit} \rangle \langle \text{integer} \rangle \]
  \[ \langle \text{digit} \rangle ::= 0 | 1 | 2 | 3 | 5 | 6 | 7 | 8 | 9 \]

- \langle \text{integer} \rangle is defined as the sequence of a \langle \text{digit} \rangle followed by zero or more \langle \text{digit} \rangle's

Extended Backus-Naur Form

- EBNF (Extended Backus-Naur Form) is a common notation to define grammars for programming languages
- Terminal symbols and non-terminal symbols
  - \textit{Terminal symbol} is a token
  - \textit{Nonterminal symbol} is a sequence of tokens, and is represented by a grammar rule
  \[ \langle \text{nonterminal} \rangle ::= \langle \text{rule body} \rangle \]
Grammar Rules

- Grammar rules may refer to other nonterminals
  
  \[ \langle \text{integer} \rangle ::= \langle \text{digit} \rangle \{ \langle \text{digit} \rangle \} \]

- \( \langle \text{integer} \rangle \) is defined as the sequence of a \( \langle \text{digit} \rangle \) followed by zero or more \( \langle \text{digit} \rangle \)'s

Grammar Rules Constructs

- \( \langle x \rangle \) nonterminal \( x \)
- \( \langle x \rangle ::= \text{Body} \) \( \langle x \rangle \) is defined by \text{Body}
- \( \langle x \rangle | \langle y \rangle \) either \( \langle x \rangle \) or \( \langle y \rangle \) (choice)
- \( \langle x \rangle \langle y \rangle \) the sequence \( \langle x \rangle \) followed by \( \langle y \rangle \)
- \( \{ \langle x \rangle \} \) sequence of zero or more occurrences of \( \langle x \rangle \)
- \( \{ \langle x \rangle \}^+ \) sequence of one or more occurrences of \( \langle x \rangle \)
- \( [ \langle x \rangle ] \) zero or one occurrence of \( \langle x \rangle \)
How to Read Grammar Rules

- From left to right

- Gives the following sequence
  - each terminal symbol is added to the sequence
  - each nonterminal is replaced by its definition
  - for each \( \langle x \rangle \mid \langle y \rangle \) pick any of the alternatives
  - for each \( \langle x \rangle \langle y \rangle \) is the sequence \( \langle x \rangle \) followed by the sequence \( \langle y \rangle \)

Examples

- \( \langle \text{statement} \rangle ::= \text{skip} \mid \langle \text{expression} \rangle \text{‘=}\langle \text{expression} \rangle \mid \ldots \)
- \( \langle \text{expression} \rangle ::= \langle \text{variable} \rangle \mid \langle \text{integer} \rangle \mid \ldots \)

- \( \langle \text{statement} \rangle ::= \text{if} \langle \text{expression} \rangle \text{then} \langle \text{statement} \rangle \)
  \{ \text{elseif} \langle \text{expression} \rangle \text{then} \langle \text{statement} \rangle \}
  \{ \text{else} \langle \text{statement} \rangle \} \text{end} \mid \ldots \)
Context-free Grammars

- Grammar rules can be used to
  - verify that a statement is legal
  - generate all possible statements
- The set of all possible statements generated from a grammar and one nonterminal symbol is called a (formal) language
- EBNF notation defines essentially a class of grammars called context-free grammars
- Expansion of a nonterminal is always the same regardless of where it is used

2. Context Free Grammar

Example 1:
- Let $N = \{\langle a \rangle\}$, $T = \{0,1\}$
  \[ \Omega = \{\langle a \rangle ::= 11a0, \langle a \rangle ::= 110\}, \quad \sigma = \langle a \rangle \]

<table>
<thead>
<tr>
<th>$110 \in L(G)$</th>
<th>$11100 \in L(G)$</th>
<th>But $011 \notin L(G)$</th>
</tr>
</thead>
</table>

These trees are called parse trees or syntax trees
4. More Examples of EBNF

(C) BNF rules for Real Numbers:

\[
\begin{align*}
\text{<real-#>} &::= \text{<int-part>} \cdot \text{<fraction>} \\
\text{<int-part>} &::= \text{<digit>} | \text{<int-part>} \text{<digit>} \\
\text{<fraction>} &::= \text{<digit>} | \text{<digit>} \text{<fraction>} \\
\text{<digit>} &::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\end{align*}
\]

```
137.91
```

Ambiguity

- A grammar is ambiguous if there exists a string which gives rise to more than one parse tree.
- Most common cause is due to infix binary operation

\[
\langle \text{expr} \rangle ::= \langle \text{num} \rangle | \langle \text{expr} \rangle - \langle \text{expr} \rangle
\]

Parse: 1-2-3
Ambiguity

\[ \langle \text{expr} \rangle ::= \langle \text{num} \rangle | \langle \text{expr} \rangle '-' \langle \text{expr} \rangle \]

Parse: \((1-2)-3\)

Which parse tree?
Ambiguity resolution for binary operators

- (A) Associative Rules
  Given a binary operator ‘op’ and a string
  \[ a_1 \ 'op' \ a_2 \ 'op' \ a_3 \]
  - If \[ a_1 \ 'op' \ a_2 \ 'op' \ a_3 \] is interpreted as \((a_1 \ 'op' \ a_2) \ 'op' \ a_3\),
    then ‘op’ is **left associative**.
  - If \[ a_1 \ 'op' \ a_2 \ 'op' \ a_3 \] is interpreted as \(a_1 \ 'op' \ (a_2 \ 'op' \ a_3)\),
    then ‘op’ is **right associative**.
  - It is possible that ‘op’ is neither left nor right associative. In which case \(a_1 \ 'op' \ a_2 \ 'op' \ a_3\) will be treated as a syntax error.

Ambiguity resolution for binary operators

- Example: We have seen that this BNF is ambiguous:
  \[
  \langle \text{expr} \rangle ::= \langle \text{num} \rangle | \langle \text{expr} \rangle - \langle \text{expr} \rangle
  \]
  To make it unambiguous, I want the ‘-’ to be...
  - Left associative:
    \[
    \langle \text{expr} \rangle ::= \langle \text{num} \rangle | \langle \text{expr} \rangle - \langle \text{num} \rangle
    \]
  - Right Associative:
    \[
    \langle \text{expr} \rangle ::= \langle \text{num} \rangle | \langle \text{num} \rangle - \langle \text{expr} \rangle
    \]
Ambiguity rules for binary operators

- (B) Precedence Rules
  Given two different binary operators ‘op₁’ and ‘op₂’
  \[ a₁ \text{ `op₁` } a₂ \text{ `op₂` } a₃ \]

  - If \( a₁ \text{ `op₁` } a₂ \text{ `op₂` } a₃ \) is interpreted as \( (a₁ \text{ `op₁` } a₂) \text{ `op₂` } a₃ \), then \( \text{op₁} \) has a higher precedence than \( \text{op₂} \).

  - If \( a₁ \text{ `op` } a₂ \text{ `op` } a₃ \) is interpreted as \( a₁ \text{ `op₁` } (a₂ \text{ `op₂` } a₃) \), then \( \text{op₂} \) has a higher precedence than \( \text{op₁} \).

Ambiguity (precedence rules)

- Example: This BNF is ambiguous:
  \[ <expr> ::= <num> | <expr> + <expr> | <expr> * <expr> \]

```
1 + 2 * 3
1 + (2 * 3)
```

Which One?
Ambiguity resolution (precedence)

Example: This BNF is ambiguous:

\[
<\text{expr}> ::= <\text{num}> \mid <\text{expr}> + <\text{expr}> \mid <\text{expr}> * <\text{expr}>
\]

To make it unambiguous, I want…

(Case 1) + to be of a higher precedence than *

\[
<\text{expr}> ::= <\text{expr2}> \mid <\text{expr2}> + <\text{expr}>
\]

\[
<\text{expr2}> ::= <\text{num}> \mid <\text{num}> + <\text{expr2}>
\]

1+2*3

(1+2)*3

Ambiguity resolution (precedence)

Example: This BNF is ambiguous:

\[
<\text{expr}> ::= <\text{num}> \mid <\text{expr}> + <\text{expr}> \mid <\text{expr}> * <\text{expr}>
\]

To make it unambiguous, I want…

(Case 2) * to be of a higher precedence than +

\[
<\text{expr}> ::= <\text{expr2}> \mid <\text{expr2}> + <\text{expr}>
\]

\[
<\text{expr2}> ::= <\text{num}> \mid <\text{num}> * <\text{expr2}>
\]

1+2*3

1+(2*3)
Ambiguity of operators

- For binary operators, we have to specify
  - the associativity of the operators, and
  - The precedence of the operators
- Alternatively, rewrite the grammar rules to get rid of ambiguity

Ambiguity of operators

- Version #1 of BNF:
  \[
  \begin{align*}
  \langle E \rangle & ::= \langle E \rangle + \langle E \rangle \\
  & \quad \mid \langle E \rangle - \langle E \rangle \\
  & \quad \mid \langle E \rangle \ast \langle E \rangle \\
  & \quad \mid \langle E \rangle / \langle E \rangle \\
  & \quad \mid \langle \text{num} \rangle \\
  & \quad \mid \langle \text{var} \rangle \\
  & \quad \mid (\langle E \rangle)
  \end{align*}
  \]

- Is the grammar ambiguous? Yes
- Version #2 of BNF:

  \[
  \begin{align*}
  \langle E \rangle & ::= \langle E \rangle + \langle T \rangle \\
  & \quad \mid \langle E \rangle - \langle T \rangle \\
  & \quad \mid \langle T \rangle \\
  \langle T \rangle & ::= \langle T \rangle \ast \langle F \rangle \\
  & \quad \mid \langle T \rangle / \langle F \rangle \\
  \langle F \rangle & ::= \langle \text{num} \rangle \\
  & \quad \mid \langle \text{var} \rangle \\
  & \quad \mid (\langle E \rangle)
  \end{align*}
  \]
Ambiguity (Dangling-else Ambiguity)

- 6.2.2 Ambiguity in general
  - Ambiguous grammar is **NOT** restricted to just binary operations:
  - Example:
    \[
    <S> ::= \begin{cases} 
    & \text{if } <E> \text{ then } <S> \\
    & \text{if } <E> \text{ then } <S> \text{ else } <S>
    \end{cases}
    \]
  - String: \text{if } <E_1> \text{ then } <E_2> \text{ then } <S_1> \text{ else } <S_2>
  - Parse Tree???

Context-sensitive Grammars

- For practical languages context-free grammar is not enough

- A condition on context is sometimes added
  - for example: identifier must be declared before use
Context-free and Context-sensitive Grammars

- Easy to read and understand
- Defines superset of language
- Expresses restrictions imposed by language
- Renders grammar rules context sensitive

Context-free grammar (e.g. with EBNF) + Set of extra conditions

Language Semantics
Language Semantics

- Defines what a program does when executed
- Goals
  - simple
  - allow programmer to reason about program (correctness, execution time, and memory use)
- How to achieve for a practical language used to build complex systems (millions lines of code)?
- The kernel language approach

Kernel Language Approach

- Define simple language (kernel language)
- Define its computation model
  - how language constructs (statements) manipulate (create and transform) data structures
- Define mapping scheme (translation) of full programming language into kernel language
- Two kinds of translations
  - linguistic abstractions
  - syntactic sugar
Kernel Language Approach

- Provides useful abstractions for programmer
- Can be extended with linguistic abstractions

```
fun \{Sqr X\} X*X end
B = \{Sqr \{Sqr A\}\}
```

```
proc \{Sqr X Y\}
    \{* X X Y\}
end
local T in
    \{Sqr A T\}
    \{Sqr T B\}
end
```

- Easy to understand and reason with
- Has a precise (formal) semantics

Linguistic Abstractions ↔ Syntactic Sugar

- Linguistic abstractions provide higher level concepts
  - programmer uses to model and reason about programs (systems)
  - examples: functions (fun), iterations (for), classes and objects (class)
- Functions (calls) are translated to procedures (calls)
- Translation answers questions about functions: \{F1 \{F2 X\} \{F3 X\}\}
Linguistic Abstractions ↔ Syntactic Sugar

- Linguistic abstractions: provide higher level concepts
- Syntactic sugar: short cuts and conveniences to improve readability

Approaches to Semantics

- Operational model
- Programming Language
  - Kernel Language
  - Formal calculus
  - Abstract machine

Aid programmer in reasoning and understanding
Mathematical study of programming (languages)
$\lambda$-calculus, predicate calculus, $\pi$-calculus
Aid implementer
Efficient execution on a real machine
Sequential Declarative Computation Model

- **Single assignment store**
  - declarative (dataflow) variables and values (together called entities)
  - values and their types
- **Kernel language syntax**
- **Environment**
  - maps textual variable names (variable identifiers) into entities in the store
- **Execution of kernel language statements**
  - execution stack of statements (defines control)
  - store
  - transforms store by sequence of steps

Our Roadmap

- Single assignment store
- Kernel language syntax
- Values and types
- Environments
- Execution
Single Assignment Store

- Single assignment store is a set of variables.
- Initially variables are unbound.
- Example: store with three variables, \( x_1 \), \( x_2 \), and \( x_3 \).
Single Assignment Store (2)

- Variables in store may be bound to values
- Example: assume we allow as values integers and lists of integers

```
  Store
  x₁ unbound
  x₂ unbound
  x₃ unbound
```

Single Assignment Store (3)

- Variables in store may be bound to values
- Assume we allow as values, integers and lists of integers
- Example:
  - x₁ is bound to integer 314
  - x₂ is bound to list [1 2 3]
  - x₃ is still unbound

```
  Store
  x₁ 314
  x₂ 1 | 2 | 3 | nil
  x₃ unbound
```
Declarative (Single-Assignment) Variables

- Created as being *unbound*
- Can be *bound* to exactly one value
- Once bound, stays bound
  - indistinguishable from its value

---

Value Store

- Store where all variables bound to values is called *value store*
- Example
  - $x_1$ bound to integer 314
  - $x_2$ to list [1 2 3]
  - $x_3$ to record
    - person(name: george
      - age: 25)
- Functional programming computes functions on values
Store Operations: Single Assignment

\[ \langle x \rangle = \langle v \rangle \]
- \( x_1 = 314 \)
- \( x_2 = [1 \ 2 \ 3] \)
- Assumes that \( \langle x \rangle \) is unbound

Single Assignment

\[ \langle x \rangle = \langle \text{value} \rangle \]
- \( x_1 = 314 \)
- \( x_2 = [1 \ 2 \ 3] \)
Single Assignment

\[ \langle x \rangle = \langle v \rangle \]
- \( x_1 = 314 \)
- \( x_2 = [1 \ 2 \ 3] \)
- **Single assignment operation** (\( '=' \))
  - constructs \( \langle v \rangle \) in store
  - binds variable \( \langle x \rangle \) to this value
- If variable already bound, operation tests compatibility of values
  - if test fails an error is raised

```
314
x_1

x_2 = [1 | 2 | 3 | nil]

x_3 = unbound
```

Variable Identifiers

- Refer to store entities
- Environment maps variable identifiers to variables
  - declare X
  - local X in ...
- "X" is variable identifier
- Corresponds to 'environment' \{"X" \( \rightarrow \) \( x_1 \)\}

```
"X"  X_1 = Unbound
```

2003-08-29  S. Haridi, CS2104, L03 (slides: C. Schulte, S. Haridi) 91

2003-08-29  S. Haridi, CS2104, L03 (slides: C. Schulte, S. Haridi) 92
### Variable-Value Binding Revisited

- \( X = [1 \ 2 \ 3] \)
- Once bound, variable indistinguishable from its value
- Traversing variable cell to get value: dereferencing
  - automatic
  - invisible

### Partial Values

- Data structure that may contain unbound variables
- The store contains the partial value:
  - person(name: george age: \( x_2 \))
- \( \text{declare} \ Y \ X \)
  - \( X = \text{person(name: george age: } Y) \)
- The identifier 'Y' refers to \( x_2 \).
Partial Values

- may be complete
  
  \[ \text{declare } Y \ X \]
  
  \[ X = \text{person(name: george age: } Y) \]
  
  \[ Y = 25 \]

Variable-variable Binding

\[ \langle x_1 \rangle = \langle x_2 \rangle \]

- Performs bind operation between variables

- Example:
  
  \[ X = Y \]
  
  \[ X = [1 \ 2 \ 3] \]

- Operation equates (merges) the two variables
Variable-variable Binding

\[ \langle x_1 \rangle = \langle x_2 \rangle \]

- Performs bind operation between variables
- Example:
  \[ X = Y \]
  \[ X = [1 \ 2 \ 3] \]
- Operation equates the two variables: forming an equivalence class

---

Variable-variable Binding

\[ \langle x_1 \rangle = \langle x_2 \rangle \]

- Performs bind operation between variables
- Example:
  \[ X = Y \]
  \[ X = [1 \ 2 \ 3] \]
- All variables (X and Y) are bound to [1 2 3]
Summary: Variables and Partial Values

- Declarative variable
  - resides in single-assignment store
  - is initially unbound
  - can be bound to exactly one (partial) value
  - can be bound to several (partial) values as long as they are compatible with each other

- Partial value
  - data-structure that may contain unbound variables
  - when one of the variables is bound, it is replaced by the (partial) value it is bound to
  - a complete value, or value for short is a data-structure that does not contain any unbound variable
Kernel Language Syntax

\( \langle s \rangle \) denotes a statement

\[
\langle s \rangle ::= \text{skip} \quad \text{empty statement} \\
| \langle x \rangle = \langle y \rangle \quad \text{variable-variable binding} \\
| \langle x \rangle = \langle v \rangle \quad \text{variable-value binding} \\
| \langle s_1 \rangle \langle s_2 \rangle \quad \text{sequential composition} \\
| \text{local} \langle x \rangle \text{in} \langle s_1 \rangle \text{end} \quad \text{declaration} \\
| \text{if} \langle x \rangle \text{then} \langle s_1 \rangle \text{else} \langle s_2 \rangle \text{end} \quad \text{conditional} \\
| \{ \langle x \rangle, \langle y_1 \rangle, \ldots, \langle y_n \rangle \} \quad \text{procedural application} \\
| \text{case} \langle x \rangle \text{of} \langle \text{pattern} \rangle \text{then} \langle s_1 \rangle \text{else} \langle s_2 \rangle \text{end} \quad \text{pattern matching}
\]

\( \langle v \rangle \) ::= ...

\( \langle \text{pattern} \rangle \) ::= ...

Variable Identifiers

- \( \langle x \rangle, \langle y \rangle, \langle z \rangle \) stand for variables
- Concrete kernel language variables
  - begin with upper-case letter
  - followed by (possibly empty) sequence of alphanumeric characters or underscore
- Any sequence of characters within backquote
- Examples:
  - X, Y1
  - Hello_World
  - “hello this is a $5 bill” (backquote)
Values and Types

- **Data type**
  - set of values
  - set of associated operations
- **Example:** Int is data type "Integer"
  - set of all integer values
  - 1 is *of type* Int
  - has set of operations including +, -, *, div, etc
- Model comes with a set of basic types
- Programs can define other types
  - for example: abstract data types ADT

Data Types

- Value
  - Number
    - Int
    - Float
    - Char
  - Record
  - Procedure
  - Tuple
    - Literal
      - Atom
      - Boolean
        - True
        - False
    - List
      - String
**Primitive Data Types**

- **Value**
  - **Number**
    - **Int**
    - **Float**
  - **Record**
    - **Tuple**
  - **Procedure**
  - **Literal**
    - **Atom**
    - **Boolean**
      - **True**
      - **False**
  - **List**
  - **String**

**Value Expressions**

\[
\langle v \rangle ::= \langle \text{procedure} \rangle \mid \langle \text{record} \rangle \mid \langle \text{number} \rangle
\]

\[
\langle \text{procedure} \rangle ::= \text{proc} \langle y_1 \rangle \ldots \langle y_n \rangle \langle s \rangle \text{ end}
\]

\[
\langle \text{record} \rangle, \langle \text{pattern} \rangle ::= \langle \text{literal} \rangle \mid \langle \text{feature}_1 \rangle : \langle x_1 \rangle \ldots \langle \text{feature}_n \rangle : \langle x_n \rangle
\]

\[
\langle \text{litera}l \rangle ::= \langle \text{atom} \rangle \mid \langle \text{bool} \rangle
\]

\[
\langle \text{feature} \rangle ::= \langle \text{int} \rangle \mid \langle \text{atom} \rangle \mid \langle \text{bool} \rangle
\]

\[
\langle \text{bool} \rangle ::= \text{true} \mid \text{false}
\]

\[
\langle \text{number} \rangle ::= \langle \text{int} \rangle \mid \langle \text{float} \rangle
\]
Numbers

- Integers
  - 314, 0
  - ~10 (minus 10)
- Floats
  - 1.0, 3.4, 2.0e2, 2.0E2 (2×10²)
- Number: either Integer or Float

Atoms and Booleans

- A sequence starting with a lower-case character followed by characters or digits, ...
  - person, peter
  - ‘Seif Haridi’
- Booleans
  - true
  - false
- Literal: atom or boolean
Records

- Compound representation (data-structures)
  - $\langle l \langle f_1 : x_1 \rangle \ldots \langle f_n : x_n \rangle \rangle$
  - $\langle l \rangle$ is a literal

- Examples
  - person(age:X1 name:X2)
  - person(1:X1 2:X2)
  - '1'(1:H 2:T)
  - nil
  - person

Syntactic Sugar

- Tuples
  - $\langle l \langle x_1 \rangle \ldots \langle x_n \rangle \rangle$ (tuple)
  - equivalent to record
    - $\langle l \langle 1 : \langle x_1 \rangle \ldots n : \langle x_n \rangle \rangle \rangle$

- Lists
Strings

- Is list of character codes enclosed with double quotes
  - example "E=mc^2"
  - same as [69 61 109 99 94 50]

Procedure Declarations

- Kernel language
  \[
  \langle x \rangle = \text{proc}\{\langle y_1 \rangle \ldots \langle y_n \rangle\} \langle s \rangle \text{ end}
  \]
  is a legal statement
  - binds \langle x \rangle to procedure value
  - declares (introduces a procedure)

- Familiar Syntactic variant
  \[
  \text{proc}\{\langle x \rangle \langle y_1 \rangle \ldots \langle y_n \rangle\} \langle s \rangle \text{ end}
  \]
  introduces (declares) the procedure \langle x \rangle
Operations on Basic Types

- **Numbers**
  - floats: +,-,*, /
  - integers: +,-,*, div, mod

- **Records**
  - Arity, Label, Width, and "."
  - \( X = \text{person(name:"George" age:25)} \)
  - \( \{\text{Arity } X\} = \{\text{age name}\} \)
  - \( \{\text{Label } X\} = \text{person, X.age = 25} \)

- **Comparisons**
  - equality: \( =, \neq \)
  - order: \( \leq, <, >, \geq \)

Value expressions

\[
\langle V \rangle ::= \langle \text{procedure} \rangle \mid \langle \text{record} \rangle \mid \langle \text{number} \rangle \mid \langle \text{basicExpr} \rangle
\]

\[
\langle \text{basicExpr} \rangle ::= \ldots \mid \langle \text{numberExpr} \rangle \mid \ldots
\]

\[
\langle \text{numberExpr} \rangle ::= \langle X \rangle_1 + \langle X \rangle_2 \mid \ldots
\]

.....
Summary: Values and Types

- For kernel language
  - numbers
  - literals
  - records
  - procedures

Outlook

- How do statements compute
  - describe for each statement
  - how environment is affected
  - how store is affected
  - how statements change
Have a Nice Weekend!