Programming Language Concepts, cs2104
Lecture 03 (2003-08-29)

Seif Haridi
Department of Computer Science, NUS
haridi@comp.nus.edu.sg

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

```
fun {L Xs N}
   case Xs
   of nil then N
   [] X|Xr then {L Xr N+1}
   end
end
fun {Length Xs}
   {L Xs 0}
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
	no list syntax
• local declarations	no nested calls
	no nested construction

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

```
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

```
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

`local X in … end`

- Introduces the variable identifier X
  - visible between `in` and `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

```
local X Y in
    ⟨statement⟩
end
```

⇨

```
local X in
    local Y in
        ⟨statement⟩
    end
end
```


Transforming Declarations Direct Assignment

```
local
    X=⟨expression⟩
in
    ⟨statement⟩
end
```

⇨

```
local X in
    X=⟨expression⟩
    ⟨statement⟩
end
```

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} ⇨ {F Y X}

Unfolding Nested Calls

```
{P {F X Y} Z}        ⇨    local U1 in
                              {F X Y U1}
                              {P U1 Z}
                          end
```


Unfolding Nested Calls

```
                          local U2 in
                             local U1 in
                                {G X U1}
{P {F {G X} Y} Z}    ⇨          {F U1 Y U2}
                             end
                             {P U2 Z}
                          end
```

Unfolding Conditionals

```
if X>Y then
   …
else
   …
end
```

⇨

```
local B in
   B = (X>Y)
   if B then
      …
   else
      …
   end
end
```


Expressions to Statements

```
X = if B then
        …
    else
        …
    end
```

⇨

```
if B then
   X = …
else
   X = …
end
```

Length (0)

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


Length (1)

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

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

- Replace operation (+, dot-access, <, >, …): procedure!

Summary

- Transform to kernel language
  - function definitions
  - function calls
  - expressions
- Kernel language
  - procedures
  - declarations
  - statements

Programming Model

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

Language Syntax

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

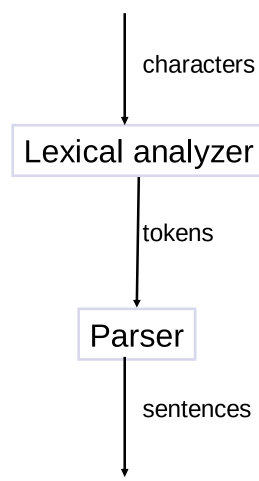
- Syntax: grammatical structure
  - lexical — how words are formed
  - phrasal — how sentences are formed from words
- Semantics: meaning of programs
  - Informal: English documents (e.g. Reference manuals, language tutorials and FAQs etc.)
  - Formal:
    - Operational Semantics (execution on an abstract 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 Ω
- A set of terminal symbols T (tokens)
- A set of Non-terminal symbols N
- One start symbol σ
- A grammar rule

  ⟨nonterminal⟩ ::= ⟨sequence of terminal and nonterminal⟩

Examples of BNF

(A) BNF rules for robot commands

- A robot arm only accepts a command from {up, down, left, right}

  ⟨move⟩ ::= ⟨cmd⟩
  ⟨move⟩ ::= ⟨cmd⟩ ⟨move⟩

  ⟨cmd⟩ ::= up
  ⟨cmd⟩ ::= down
  ⟨cmd⟩ ::= left
  ⟨cmd⟩ ::= right

Grammar Rules

- ⟨digit⟩ is defined to represent one of the ten tokens 0, 1, …, 9

  ⟨digit⟩ ::= 0 | 1 | 2 | 3 | 5 | 6 | 7 | 8 | 9

- The symbol '|' is read as 'or'
- Another reading is that ⟨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}

  ⟨move⟩ ::= ⟨cmd **|** ⟨cmd⟩ ⟨move⟩

  ⟨cmd⟩ ::= up **|** down **|** left **|** right

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

Examples of BNF

- Integers

  ⟨integer⟩ ::= ⟨digit⟩ | ⟨digit⟩ ⟨integer⟩

  ⟨digit⟩ ::= 0 | 1 | 2 | 3 | 5 | 6 | 7 | 8 | 9

- ⟨integer⟩ is defined as the sequence of a ⟨digit⟩ followed by zero or more ⟨digit⟩'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
- *Terminal symbol* is a token
- *Nonterminal symbol* is a sequence of tokens, and is represented by a grammar rule

  ⟨nonterminal⟩ ::= ⟨rule body⟩

Grammar Rules

- Grammar rules may refer to other nonterminals

  ⟨integer⟩ ::= ⟨digit⟩ { ⟨digit⟩ }

- ⟨integer⟩ is defined as the sequence of a ⟨digit⟩ followed by zero or more ⟨digit⟩'s

Grammar Rules Constructs

- ⟨*x*⟩ nonterminal *x*
- ⟨*x*⟩ ::= *Body* ⟨*x*⟩ is defined by *Body*
- ⟨*x*⟩ | ⟨*y*⟩ either ⟨*x*⟩ or ⟨*y*⟩ (choice)
- ⟨*x*⟩ ⟨*y*⟩ the sequence ⟨*x*⟩ followed by ⟨*y*⟩
- { ⟨*x*⟩ } sequence of zero or more occurrences of ⟨*x*⟩
- { ⟨*x*⟩ }$^{+}$ sequence of one or more occurrences of ⟨*x*⟩
- [⟨*x*⟩] zero or one occurrence of ⟨*x*⟩

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 ⟨*x*⟩ | ⟨*y*⟩ pick any of the alternatives
  - for each ⟨*x*⟩ ⟨*y*⟩ is the sequence ⟨*x*⟩ followed by the sequence ⟨*y*⟩

Examples

- ⟨statement⟩ **::=** skip **|** ⟨expression⟩ '=' ⟨expression⟩ **|** …
- ⟨expression⟩ **::=** ⟨variable⟩ **|** ⟨integer⟩ **| …**
- ⟨statement⟩ **::=** if ⟨expression⟩ then ⟨statement⟩
  { elseif ⟨expression⟩ then ⟨statement⟩ **}**
  [else ⟨statement⟩ **]** end **| …**

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

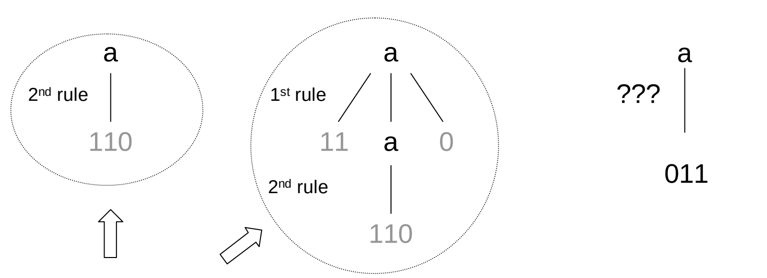
2. Context Free Grammar

Example 1:

- Let $N = \{\langle a \rangle\}$, $T = \{0,1\}$

  $\Omega = \{\langle a \rangle ::= 11a0,\ \langle a \rangle ::= 110\}$, $\sigma = \langle a \rangle$

110 $\in$ L(G) 111100 $\in$ L(G) But 011 $\notin$ L(G)

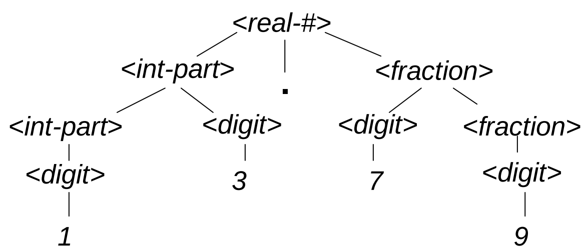


These trees are called parse trees or syntax trees

4. More Examples of EBNF

(C) BNF rules for Real Numbers;

<real-#>	::=	*<int-part> . <fraction>*
<int-part>	::=	*<digit> | <int-part> <digit>*
<fraction>	::=	*<digit> | <digit> <fraction>*
<digit>	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



Ambiguity

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

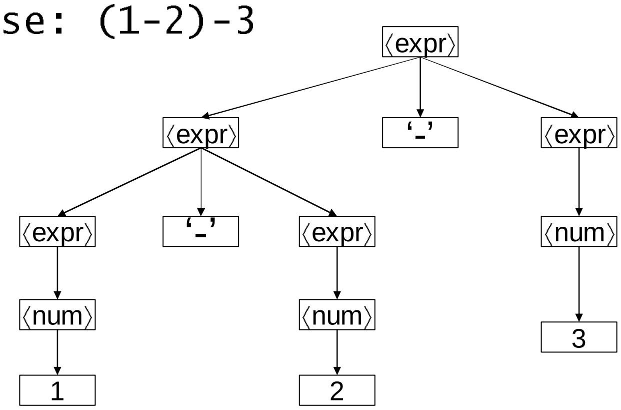
```
⟨expr⟩ ::= ⟨num⟩|⟨expr⟩'-'⟨expr⟩

Parse: 1-2-3
```

Ambiguity

⟨expr⟩ ::= ⟨num⟩|⟨expr⟩'-'⟨expr⟩

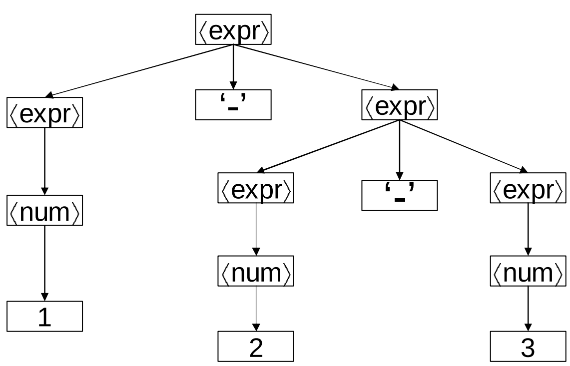
Parse: (1-2)-3



Ambiguity

⟨expr⟩ ::= ⟨num⟩|⟨expr⟩'-'⟨expr⟩

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:
  $\langle\texttt{expr}\rangle$ ::= $\langle\texttt{num}\rangle$ | $\langle\texttt{expr}\rangle$ – $\langle\texttt{expr}\rangle$

  To make it unambiguous, I want the '–' to be…
  - Left associative:
    $\langle\texttt{expr}\rangle$ ::= $\langle\texttt{num}\rangle$ | $\langle\texttt{expr}\rangle$ – $\langle\texttt{num}\rangle$
  - Right Associative:
    $\langle\texttt{expr}\rangle$ ::= $\langle\texttt{num}\rangle$ | $\langle\texttt{num}\rangle$– $\langle\texttt{expr}\rangle$

Ambiguity rules for binary operators

- (B) Precedence Rules

  Given two **different** binary operators ‘op_1’ and ‘op_2’

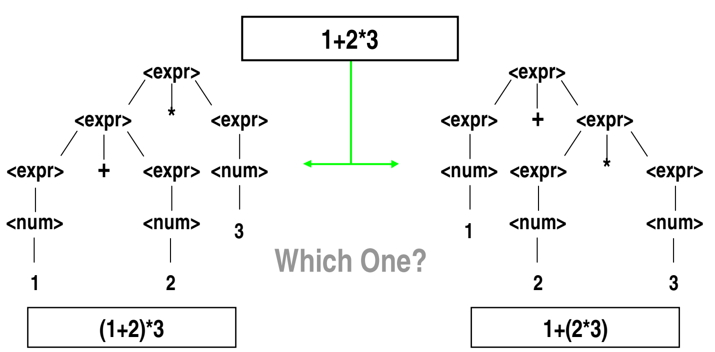
  a_1 ‘op_1’ a_2 ‘op_2’ a_3

  - If a_1 ‘op_1’ a_2 ‘op_2’ a_3 is interpreted as (a_1 ‘op_1’ a_2) ‘op_2’ a_3, then op_1 has a *higher precedence* than op_2.
  - Ifa$_1$ ‘op’ a_2 ‘op’ a_3 is interpreted as a_1 ‘op_1’ (a_2 ‘op_2’ a_3), then op_2 has a *higher precedence* than op_1.

Ambiguity (precedence rules)

- Example: This BNF is ambiguous:

  <expr> ::= <num> | <expr> + <expr> | <expr> * <expr>



Ambiguity resolution (precedence)

Example: This BNF is ambiguous:

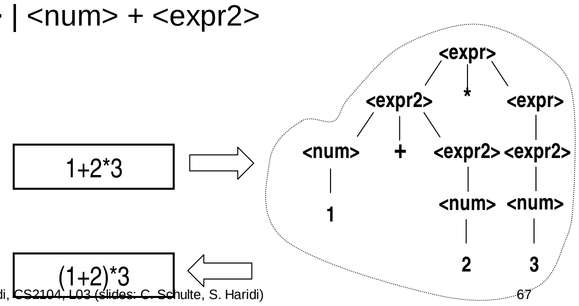
<expr> ::= <num> | <expr> + <expr> | <expr> * <expr>

To make it unambiguous, I want…

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

<expr> ::= <expr2> | <expr2> * <expr>

<expr2>::= <num> | <num> + <expr2>



Ambiguity resolution (precedence)

Example: This BNF is ambiguous:

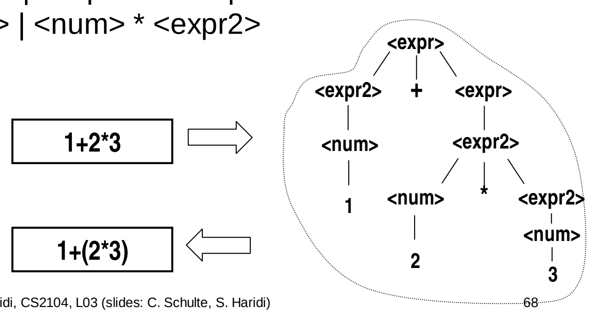
<expr> ::= <num> | <expr> + <expr> | <expr> * <expr>

To make it unambiguous, I want…

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

<expr> ::= <expr2> | <expr2> + <expr>

<expr2>::= <num> | <num> * <expr2>



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:

  <E> ::= <E> + <E> |
  <E> - <E> | <E> * <E> | <E> /<E>|
  <num> | <var> | (<E>)
- Is the grammar ambiguous? Yes
- Version #2 of BNF:

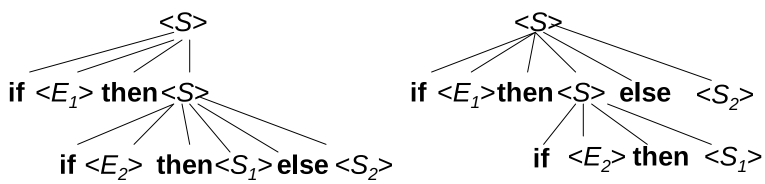
  <E> ::= <E> + <T> | <E> - <T> | <T>
  <T> ::= <T> * <F> | <T> / <F> | <F>
  <F> ::= <num> | <var> | (<E>)

Ambiguity (Dangling-else Ambiguity)

- 6.2.2 Ambiguity in general
  - Ambiguous grammar is **NOT** restricted to just binary operation:
  - Example:

    <S> ::= **if** *<E>* **then** *<S>*
    | **if** *<E>* **then** *<S>* **else** *<S>*

  - String: **if** <E_1> **then if** <E_2> **then** <S_1> **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

practical language

translation

kernel language

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

```
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
```


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

```
if N==1 then [1]
else
   local L in
      ...
   end
end
```

```
if N==1 then [1]
else L in
      ...
end
```


Approaches to Semantics

Programming Language

Operational model

Kernel Language | Formal calculus | Abstract machine

Aid programmer in reasoning and understanding

Mathematical study of programming (languages) λ-calculus, predicate calculus, π-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

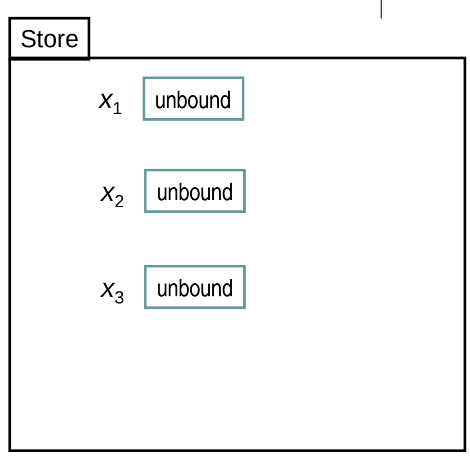
Our Roadmap

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

Single Assignment Store

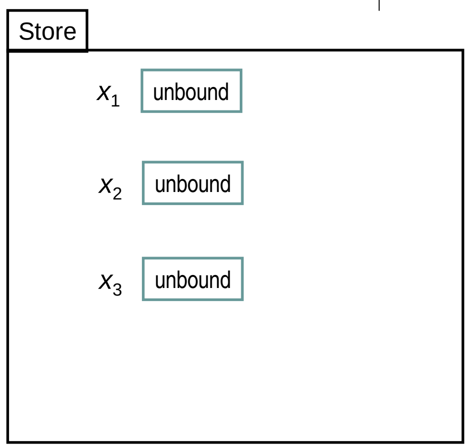
Single Assignment Store

- Single assignment store is store (set) of variables
- Initially variables are unbound
- Example: store with three variables, x_1, x_2, and x_3



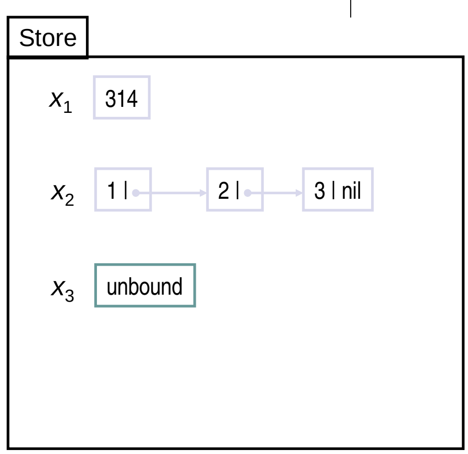
Single Assignment Store (2)

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



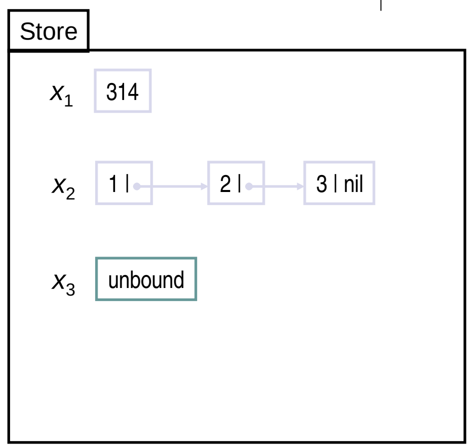
Single Assignment Store (3)

- Variables in store may be bound to values
- Assume we allow as values, integers and lists of integers
- Example:
  - x_1 is bound to integer 314
  - x_2 is bound to list [1 2 3]
  - x_3 is still unbound



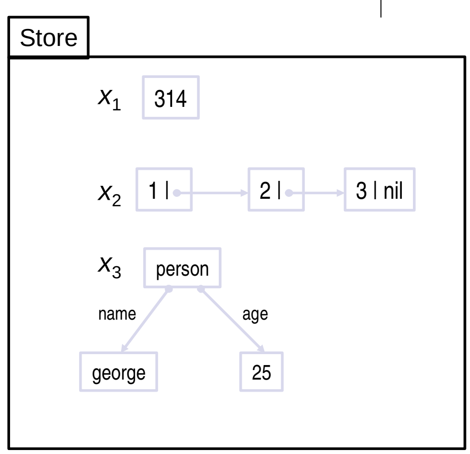
Declarative (Single-Assignment) Variables

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



Value Store

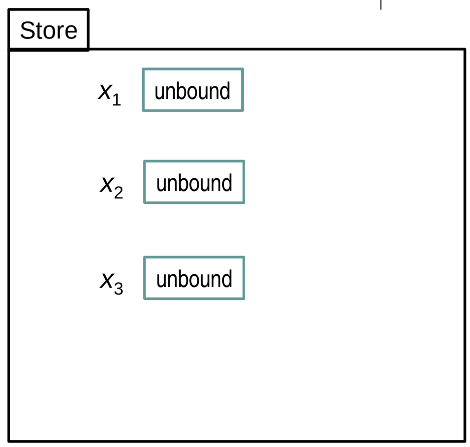
- Store where all variables bound to values is called *value store*
- Example
  - x_1 bound to integer 314
  - x_2 to list [1 2 3]
  - x_3 to record person(name:george age: 25)
- Functional programming computes functions on values



Store Operations: Single Assignment

⟨*x*⟩ = ⟨*v*⟩

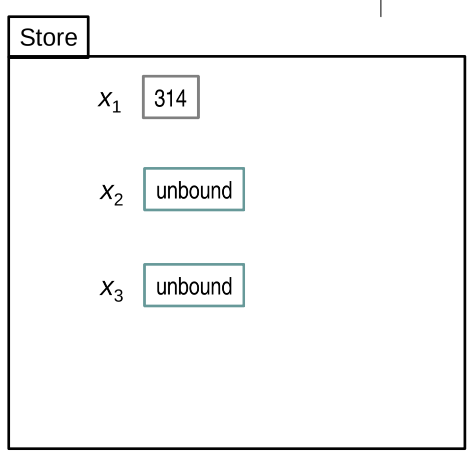
- x_1 = 314
- x_2 = [1 2 3]
- Assumes that ⟨*x*⟩ is unbound



Single Assignment

⟨*x*⟩ = ⟨value⟩

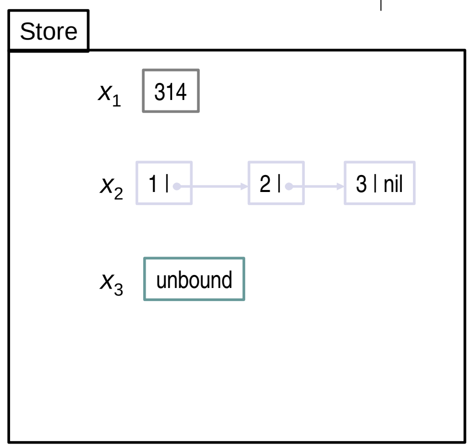
- $\boldsymbol{x_1}$ **= 314**
- x2 = [1 2 3]



Single Assignment

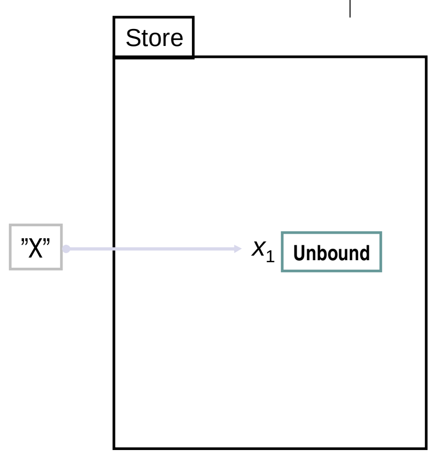
$\langle x \rangle = \langle v \rangle$

- 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



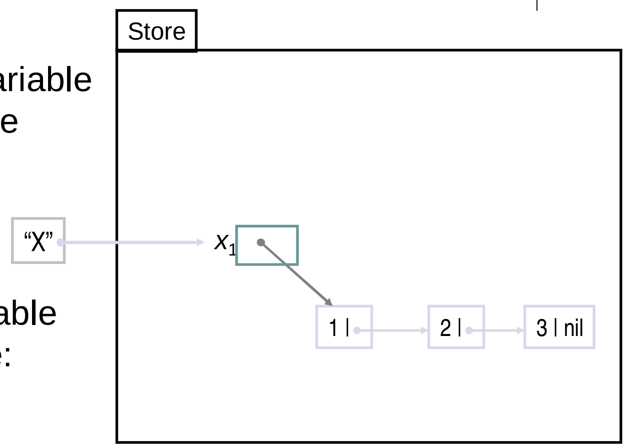
Variable Identifiers

- Refer to store entities
- Environment maps variable identifiers to variables
  - declare X
  - local X in ...
- ”X” is variable identifier
- Corresponds to ’environment’ {”X” $\rightarrow x_1$}



Variable-Value Binding Revisited

- X = [1 2 3]
- Once bound, variable indistinguishable from its value



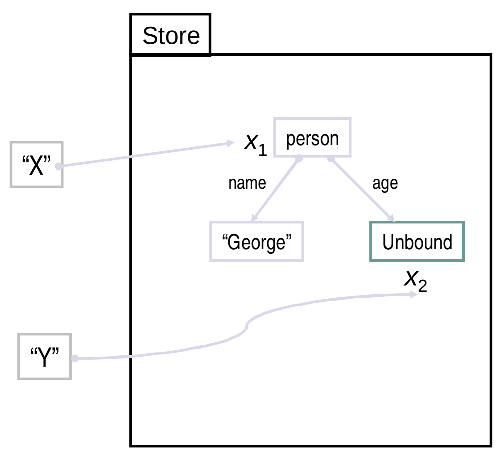
- Traversing variable cell to get value: *dereferencing*
  - automatic
  - invisible

Partial Values

- Data structure that may contain unbound variables
- The store contains the partial value:

  person(name: george age: x_2)
- declare Y X

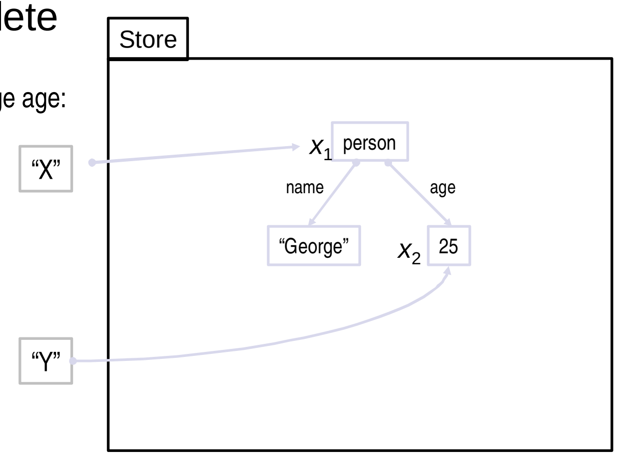
  X = person(name: george age: Y)
- The identifier ’Y’ refers to x_2



Partial Values

- may be complete

  declare Y X
  X = person(name: george age: Y)
- Y = 25



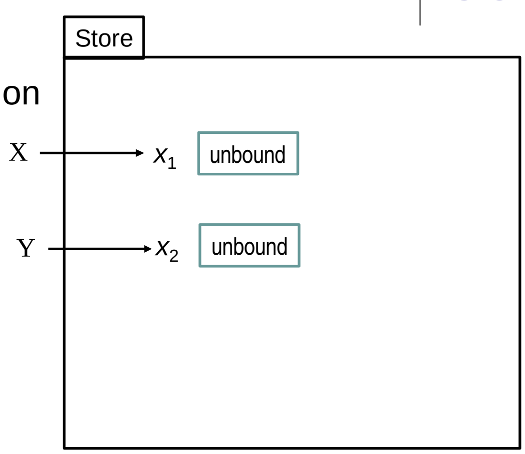
Variable-variable Binding

$\langle x_1 \rangle = \langle x_2 \rangle$

- Performs bind operation between variables
- Example:

  X = Y

  X = [1 2 3]
- Operation equates (merges) the two variables



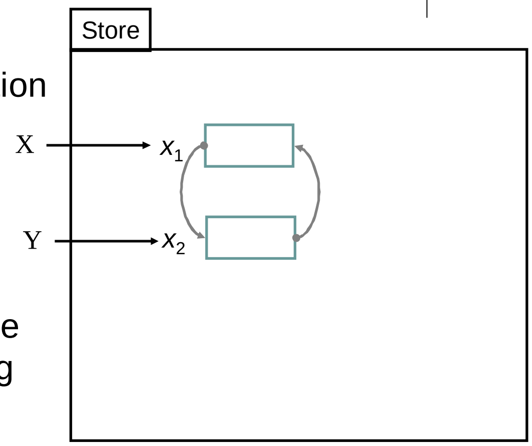
Variable-variable Binding

$\langle x_1 \rangle = \langle x_2 \rangle$

- Performs bind operation between variables
- Example:

  X = Y

  X = [1 2 3]
- Operation equates the two variables: forming an equivalence class



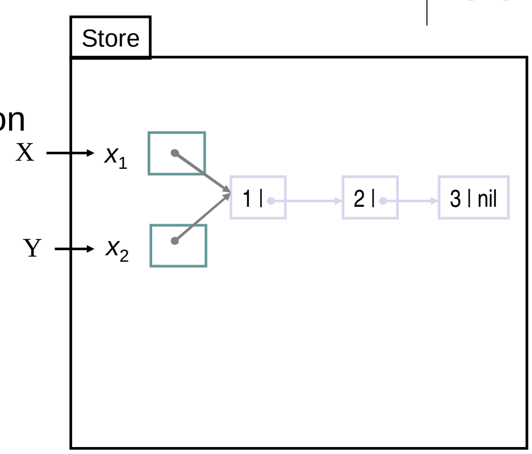
Variable-variable Binding

$\langle x_1 \rangle = \langle x_2 \rangle$

- 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

Kernel Language Syntax

⟨s⟩ denotes a statement

⟨s⟩ ::=	skip	*empty statement*
|	⟨x⟩ = ⟨y⟩	*variable-variable binding*
|	⟨x⟩ = ⟨v⟩	*variable-value binding*
|	$\langle s_1\rangle \langle s_2\rangle$	*sequential composition*
|	local ⟨x⟩ in $\langle s_1\rangle$ end	*declaration*
|	if ⟨x⟩ then $\langle s_1\rangle$ else $\langle s_2\rangle$ end	*conditional*
|	{ ⟨x⟩ $\langle y_1\rangle$ … $\langle y_n\rangle$ }	*procedural application*
|	case ⟨x⟩ of ⟨pattern⟩ then $\langle s_1\rangle$ else $\langle s_2\rangle$ end	*pattern matching*
⟨v⟩ ::=	...	*value expression*
⟨pattern⟩ ::=	...	

Variable Identifiers

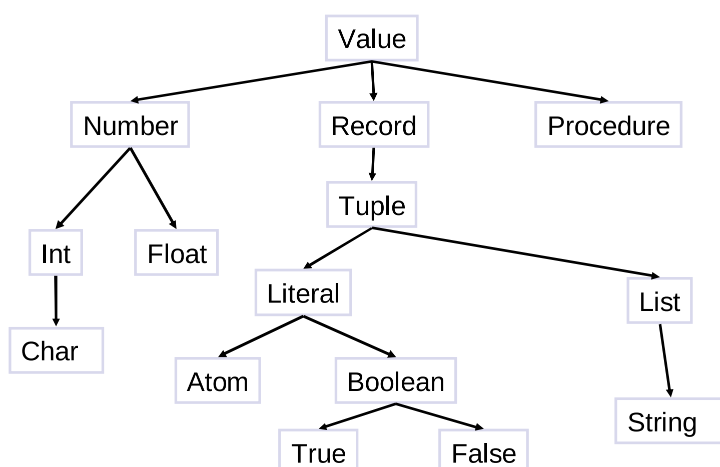
- ⟨*x*⟩ , ⟨*y*⟩, ⟨*z*⟩ stand for variables
- Concrete kernel language variables
  - begin with upper-case letter
  - followed by (possibly empty) sequence of alphanumeric characters or underscore
- Any sequence of characters within backquote
- Examples:
  - X, Y1
  - Hello_World
  - \`hello this is a $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

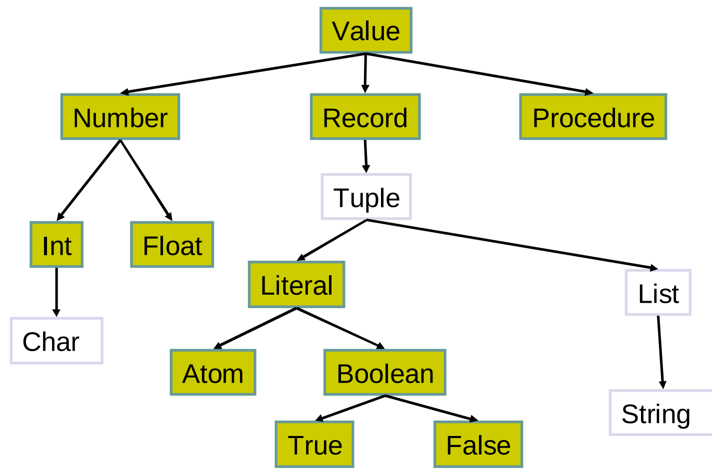
2003-08-29 S. Haridi, CS2104, L03 (slides: C. Schulte, S. Haridi) 103

Data Types



2003-08-29 S. Haridi, CS2104, L03 (slides: C. Schulte, S. Haridi) 104

Primitive Data Types



Value Expressions

⟨v⟩ ::= ⟨procedure⟩ | ⟨record⟩ | ⟨number⟩

⟨procedure⟩ ::= proc {$ $\langle y_1 \rangle \ldots \langle y_n \rangle$} $\langle s \rangle$ end

⟨record⟩, ⟨pattern⟩ ::= ⟨literal⟩
| ⟨literal⟩ $(\langle \text{feature}_1 \rangle : \langle x_1 \rangle \ldots \langle \text{feature}_n \rangle : \langle x_n \rangle)$

⟨literal⟩ ::= ⟨atom⟩ | ⟨bool⟩
⟨feature⟩ ::= ⟨int⟩ | ⟨atom⟩ | ⟨bool⟩

⟨bool⟩ ::= true | false

⟨number⟩ ::= ⟨int⟩ | ⟨float⟩

Numbers

- Integers
  - 314, 0
  - ~10 (minus 10)
- Floats
  - 1.0, 3.4, 2.0e2, 2.0E2 (2×10^2)
- 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)
  - $\langle l \rangle(\langle f_1 \rangle : \langle x_1 \rangle \ldots \langle f_n \rangle : \langle x_n \rangle)$
  - $\langle l \rangle$ is a literal
- Examples
  - person(age:X1 name:X2)
  - person(1:X1 2:X2)
  - '|'(1:H 2:T)
  - nil
  - person

Syntactic Sugar

- Tuples

  $\langle l \rangle(\langle x_1 \rangle \ldots \langle x_n \rangle)$ (tuple)

  equivalent to record

  $\langle l \rangle(1{:}\, \langle x_1 \rangle \ldots n{:}\, \langle x_n \rangle)$
- Lists

Strings

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

2003-08-29 S. Haridi, CS2104, L03 (slides: C. Schulte, S. Haridi) 111

Procedure Declarations

- Kernel language

  $\langle x \rangle$ = proc {\$ $\langle y_1 \rangle \ldots \langle y_n \rangle$} $\langle s \rangle$ end

  is a legal statement
  - binds $\langle x \rangle$ to procedure value
  - declares (introduces a procedure)
- Familiar Syntactic variant

  proc {$\langle x \rangle$ $\langle y_1 \rangle \ldots \langle y_n \rangle$} $\langle s \rangle$ end

  introduces (declares) the procedure $\langle x \rangle$

2003-08-29 S. Haridi, CS2104, L03 (slides: C. Schulte, S. Haridi) 112

Operations on Basic Types

- Numbers
  - floats: +,-,*, /
  - integers: +,-,*,div, mod
- Records
  - Arity, Label, Width, and "."
  - X = person(name:"George" age:25)
  - {Arity X} = [age name]
  - {Label X} = person, X.age = 25
- Comparisons
  - equality: ==, \=
  - order: =<, <, >=
    integers, floats, and atoms

Value expressions

⟨v⟩ ::= ⟨procedure⟩ | ⟨record⟩ | ⟨number⟩ | ⟨basicExpr⟩

⟨basicExpr⟩ ::= ... | ⟨numberExpr⟩ | ...

⟨numberExpr⟩ ::= $\langle x \rangle_1 + \langle x \rangle_2$ | ...

.....

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!