Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Contact Information - Elsa L Gunter

- Office: 2112 SC
- Office hours:
  - Mondays 11:00am – 11:50am
  - Tuesdays 3:30pm – 4:20pm
  - Thursdays 12:30pm – 1:40pm
  - Also by appointment
- Email: egunter@illinois.edu

Contact Information - TAs

- Teaching Assistants Office: 0207 SC
- Kyle Blocher
  - Email: blocher1@illinois.edu
  - Hours: Tues 1:00pm – 1:50pm &
    Thurs 11:00am – 11:50am
- Hassan Samee
  - Email: samee1@illinois.edu
  - Hours: Wed 1:30pm – 2:20pm &
    Fri 4:00pm - 4:50pm

Course Website

- http://courses.engr.illinois.edu/cs421
- Main page - summary of news items
- Policy - rules governing course
- Lectures - syllabus and slides
- MPs - information about homework
- Exams
- Unit Projects - for 4 credit students
- Resources - tools and helpful info
- FAQ

Some Course References

- No required textbook.
- Essentials of Programming Languages (2nd Edition) by Daniel P. Friedman, Mitchell Wand and Christopher T. Haynes, MIT Press 2001.
- Compilers: Principles, Techniques, and Tools, (also known as "The Dragon Book"); by Aho, Sethi, and Ullman. Published by Addison-Wesley. ISBN: 0-201-10088-6.
- Modern Compiler Implementation in ML by Andrew W. Appel, Cambridge University Press 1998
- Additional ones for Ocaml given separately

Course Grading

- Homework 20%
  - About 12 MPs (in Ocaml) and 12 written assignments
  - Submitted by **handin** on EWS linux machines
    - MPs – plain text code that compiles; HWs – pdf
  - Late submission penalty: 20% of assignments total value
- 2 Midterms - 20% each
  - In class – **Oct 9, Nov 13**
  - **DO NOT MISS EXAM DATES!**
- Final 40% - Dec 14, 7:00pm – 10:00pm
- Percentages are approximate
  - Exams may weigh more if homework is much better

Course Homework

- You may discuss homeworks and their solutions with others
- You may work in groups, but you must list members with whom you worked if you share solutions or solution outlines
- Each student must turn in their own solution separately
- You may look at examples from class and other similar examples from any source
  - Note: University policy on plagiarism still holds - cite your sources if you are not the sole author of your solution
- Problems from homework may appear verbatim, or with some modification on exams

Course Objectives

- New programming paradigm
  - Functional programming
  - Tail Recursion
  - Continuation Passing Style
- Phases of an interpreter / compiler
  - Lexing and parsing
  - Type checking
  - Evaluation
- Programming Language Semantics
  - Lambda Calculus
  - Operational Semantics

OCAML

- Compiler is on the EWS-linux systems at
- /usr/local/bin/ocaml
- A (possibly better, non-PowerPoint) text version of this lecture can be found at
- http://course.engr.illinois.edu/class/cs421/lectures/ocaml-intro-shell.txt
- For the OCAML code for today's lecture see
- http://course.engr.illinois.edu/class/cs421/lectures/ocaml-intro.ml

WWW Addresses for OCAML

- Main CAML home: http://caml.inria.fr/index.en.html
- To install OCAML on your computer see:
- http://caml.inria.fr/ocaml/release.en.html

References for CAML

- Supplemental texts (not required):
- The Objective Caml system release 3.09, by Xavier Leroy, online manual
- Introduction to the Objective Caml Programming Language, by Jason Hickey
- Developing Applications With Objective Caml, by Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano, on O'Reilly
  - Available online from course resources

OCAML

- CAML is European descendant of original ML
  - American/British version is SML
  - O is for object-oriented extension
- ML stands for Meta-Language
- ML family designed for implementing theorem provers
  - It was the meta-language for programming the "object" language of the theorem prover
  - Despite obscure original application area, OCAML is a full general-purpose programming language

Features of OCAML

- Higher order applicative language
- Call-by-value parameter passing
- Modern syntax
- Parametric polymorphism
  - Aka structural polymorphism
- Automatic garbage collection
- User-defined algebraic data types

- It’s fast - winners of the 1999 and 2000 ICFP Programming Contests used OCAML

Why learn OCAML?

- Many features not clearly in languages you have already learned
- Assumed basis for much research in programming language research
- OCAML is particularly efficient for programming tasks involving languages (eg parsing, compilers, user interfaces)
- Used at Microsoft for writing SLAM, a formal methods tool for C programs

Session in OCAML

```
% ocaml
Objective Caml version 3.12.0
# (* Read-eval-print loop; expressions and
   declarations *)
   2 + 3;;     (* Expression *)
-  : int = 5
# 3 < 2;;
- : bool = false
```

No Overloading for Basic Arithmetic Operations

```
# 15 * 2;;
- : int = 30
# 1.35 + 0.23;;  (* Wrong type of addition *)
Characters 0-4:
  1.35 + 0.23;;  (* Wrong type of addition *)
  ^^^^
Error: This expression has type float but an
    expression was expected of type
        int
# 1.35 +. 0.23;;
- : float = 1.58
```

No Implicit Coercion

```
# 1.0 * 2;; (* No Implicit Coercion *)
Characters 0-3:
  1.0 * 2;; (* No Implicit Coercion *)
  ^^^
Error: This expression has type float but an
    expression was expected of type
       int
```

Sequencing Expressions

```
# "Hi there";;  (* has type string *)
- : string = "Hi there"
# print_string "Hello world\n";;  (* has type unit *)
Hello world
- : unit = ()
# (print_string "Bye\n"; 25);;  (* Sequence of exp *)
Bye
- : int = 25
```

Terminology

- *Output* refers both to the result returned from a function application
  - As in + outputs integers, whereas +. outputs floats
- And to text printed as a side-effect of a computation
  - As in print_string "\n" outputs a carriage return
  - In terms of values, it outputs ( ) ("unit")
- We will standardly use "output" to refer to the value returned

Declarations; Sequencing of Declarations

```
# let x = 2 + 3;;   (* declaration *)
val x : int = 5
# let test = 3 < 2;;
val test : bool = false
# let a = 3 let b = a + 2;; (* Sequence of dec *)
val a : int = 3
val b : int = 5
```

Environments

- *Environments* record what value is associated with a given identifier
- Central to the semantics and implementation of a language
- Notation

$$\rho = \{name_1 \to value_1, name_2 \to value_2, \ldots\}$$

  Using set notation, but describes a partial function
- Often stored as list, or stack
  - To find value start from left and take first match

Global Variable Creation

```
# 2 + 3;;    (* Expression *)
// doesn't affect the environment
# let test = 3 < 2;;    (* Declaration *)
val test : bool = false
```

// ρ_1 = {test → false}

```
# let a = 1 let b = a + 4;; (* Seq of dec *)
```

// ρ_2 = {b → 5, a → 1, test → false}

New Bindings Hide Old

// $\rho_2 = \{b \rightarrow 5, a \rightarrow 1, test \rightarrow false\}$

let a = 3;;

- What is the environment after this declaration?

New Bindings Hide Old

// $\rho_2 = \{b \to 5, a \to 1, test \to false\}$

let a = 3;;

- What is the environment after this declaration?

// $\rho_3 = \{a \to 3, b \to 5, test \to false\}$

Local let binding

```
// ρ3 = {a → 3, b → 5, test → false}
# let c =
    let b = a + a
// ρ4 = {b → 6} + ρ2
//      ={b → 6, a → 3, test → false}
    in b * b;;
val c : int = 36
// ρ5 = {c → 36, a → 3, b → 5, test → false}
# b;;
- : int = 5
```

Local Variable Creation

// $\rho_5 = \{c \rightarrow 36, b \rightarrow 5, a \rightarrow 3, test \rightarrow false\}$

```
# let b = 5 * 4
```

// $\rho_6 = \{b \rightarrow 20, c \rightarrow 36, a \rightarrow 3, test \rightarrow false\}$

```
    in 2 * b;;
- : int = 40
```

// $\rho_7 = \rho_5$

```
# b;;
- : int = 5
```

Booleans (aka Truth Values)

```
# true;;
- : bool = true
# false;;
- : bool = false

# if y > x then 25 else 0;;
- : int = 25
```

Booleans

```
# 3 > 1 && 4 > 6;;
- : bool = false
# 3 > 1 || 4 > 6;;
- : bool = true
# (print_string "Hi\n"; 3 > 1) || 4 > 6;;
Hi
- : bool = true
# 3 > 1 || (print_string "Bye\n"; 4 > 6);;
- : bool = true
# not (4 > 6);;
- : bool = true
```

Tuples

```
# let s = (5,"hi",3.2);;
val s : int * string * float = (5, "hi", 3.2)
# let (a,b,c) = s;;  (* (a,b,c) is a pattern *)
val a : int = 5
val b : string = "hi"
val c : float = 3.2
# let x = 2, 9.3;; (* tuples don't require parens in
    Ocaml *)
val x : int * float = (2, 9.3)
```

Tuples

```
# (*Tuples can be nested *)
let d = ((1,4,62),("bye",15),73.95);;
val d : (int * int * int) * (string * int) * float =
  ((1, 4, 62), ("bye", 15), 73.95)
# (*Patterns can be nested *)
let (p,(st,_),_) = d;; (* _ matches all, binds nothing
   *)
val p : int * int * int = (1, 4, 62)
val st : string = "bye"
```

Functions

```
# let plus_two n = n + 2;;
val plus_two : int -> int = <fun>
# plus_two 17;;
- : int = 19
# let plus_two = fun n -> n + 2;;
val plus_two : int -> int = <fun>
# plus_two 14;;
- : int = 16
```

First definition syntactic sugar for second

Using a nameless function

```
# (fun x -> x * 3) 5;;   (* An application *)
- : int = 15
# ((fun y -> y +. 2.0), (fun z -> z * 3));;
  (* As data *)
- : (float -> float) * (int -> int) = (<fun>,
  <fun>)
```

Note: in fun v -> exp(v), scope of variable is only the body exp(v)

Values fixed at declaration time

```
# let x = 12;;
val x : int = 12
# let plus_x y = y + x;;
val plus_x : int -> int = <fun>
# plus_x 3;;
```

What is the result?

Values fixed at declaration time

```
# let x = 12;;
val x : int = 12
# let plus_x y = y + x;;
val plus_x : int -> int = <fun>
# plus_x 3;;
- : int = 15
```

Values fixed at declaration time

```
# let x = 7;;   (* New declaration, not an
  update *)
val x : int = 7
# plus_x 3;;
```

What is the result this time?

Values fixed at declaration time

```
# let x = 7;;   (* New declaration, not an update *)
val x : int = 7
# plus_x 3;;
- : int = 15
```

Functions with more than one argument

```
# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>
# let t = add_three 6 3 2;;
val t : int = 11
# let add_three =
   fun x -> (fun y -> (fun z -> x + y + z));;
val add_three : int -> int -> int -> int = <fun>
```

Again, first syntactic sugar for second

Partial application of functions

```
let add_three x y z = x + y + z;;
```

```
# let h = add_three 5 4;;
val h : int -> int = <fun>
# h 3;;
- : int = 12
# h 7;;
- : int = 16
```

Functions as arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = thrice plus_two;;
val g : int -> int = <fun>
# g 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

Question

- Observation: Functions are first-class values in this language

- Question: What value does the environment record for a function variable?

- Answer: a closure

Save the Environment!

- A *closure* is a pair of an environment and an association of a sequence of variables (the input variables) with an expression (the function body), written:

$$f \rightarrow < (v1,\ldots,vn) \rightarrow exp, \rho_f >$$

- Where ρ_f is the environment in effect when f is defined (if f is a simple function)

Closure for plus_x

- When plus_x was defined, had environment:

$$\rho_{plus\_x} = \{x \rightarrow 12, \ldots, y \rightarrow 24, \ldots\}$$

- Closure for plus_x:

$$<y \rightarrow y + x, \rho_{plus\_x} >$$

- Environment just after plus_x defined:

$$\{plus\_x \rightarrow <y \rightarrow y + x, \rho_{plus\_x} >\} + \rho_{plus\_x}$$

Evaluation of Application of plus_x;;

- Have environment:

  $\rho = \{plus\_x \to <y \to y + x, \rho_{plus\_x}>, \ldots ,$
  $y \to 3, \ldots\}$

  where $\rho_{plus\_x} = \{x \to 12, \ldots , y \to 24, \ldots\}$
- Eval (plus_x y, ρ) rewrites to
- Eval (app $<y \to y + x, \rho_{plus\_x}>$ 3, ρ) rewrites to
- Eval ($y + x$, $\{y \to 3\} + \rho_{plus\_x}$) rewrites to
- Eval ($3 + 12$, ρ_{plus_x}) = 15

Functions on tuples

```
# let plus_pair (n,m) = n + m;;
val plus_pair : int * int -> int = <fun>
# plus_pair (3,4);;
- : int = 7
# let double x = (x,x);;
val double : 'a -> 'a * 'a = <fun>
# double 3;;
- : int * int = (3, 3)
# double "hi";;
- : string * string = ("hi", "hi")
```

Match Expressions

```
# let triple_to_pair triple =
  match triple
  with (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);;
val triple_to_pair : int * int * int -> int * int =
  <fun>
```

- Each clause: pattern on left, expression on right
- Each x, y has scope of only its clause
- Use first matching clause

Closure for plus_pair

- Assume ρ_{plus_pair} was the environment just before plus_pair defined
- Closure for plus_pair:

$$<(n,m) \rightarrow n + m, \rho_{plus\_pair}>$$

- Environment just after plus_pair defined:

$$\{plus\_pair \rightarrow <(n,m) \rightarrow n + m, \rho_{plus\_pair}>\} + \rho_{plus\_pair}$$

Evaluation of Application with Closures

- In environment ρ, evaluate left term to closure, $c = <(x_1,\dots,x_n) \rightarrow b, \rho>$
- $(x_1,\dots,x_n)$ variables in (first) argument
- Evaluate the right term to values, $(v_1,\dots,v_n)$
- Update the environment ρ to

  $\rho' = \{x_1 \rightarrow v_1,\dots, x_n \rightarrow v_n\} + \rho$
- Evaluate body b in environment ρ'

Evaluation of Application of plus_pair

- Assume environment

$\rho = \{x \rightarrow 3\ldots,$

$\quad plus\_pair \rightarrow <(n,m) \rightarrow n + m, \rho_{plus\_pair}>\} +$

$\quad \rho_{plus\_pair}$

- Eval (plus_pair (4,x), ρ)=
- Eval (app <(n,m) $\rightarrow$n + m, ρ_{plus_pair}> (4,x), ρ)) =
- Eval (app <(n,m) $\rightarrow$n + m, ρ_{plus_pair}> (4,3), ρ)) =
- Eval (n + m, {n -> 4, m -> 3} + ρ_{plus_pair}) =
- Eval (4 + 3, {n -> 4, m -> 3} + ρ_{plus_pair}) = 7

Curried vs Uncurried

- Recall

val add_three : int -> int -> int -> int = <fun>

- How does it differ from

\# let add_triple (u,v,w) = u + v + w;;

val add_triple : int * int * int -> int = <fun>

- add_three is *curried*;
- add_triple is *uncurried*

Curried vs Uncurried

```
# add_triple (6,3,2);;
- : int = 11
# add_triple 5 4;;
Characters 0-10:
  add_triple 5 4;;
  ^^^^^^^^^^
This function is applied to too many arguments,
maybe you forgot a `;'
# fun x -> add_triple (5,4,x);;
: int -> int = <fun>
```

Scoping Question

Consider this code:

```
let x = 27;;
let f x =
    let x = 5 in
        (fun x -> print_int x) 10;;
f 12;;
```

What value is printed?

- 5
- 10
- 12
- 27

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://www.cs.uiuc.edu/class/cs421/

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Personal History

- First began programming more than 35 years ago
- First languages: Basic, DG Nova assembler
- Since have programmed in at least 10 different languages
  - Not including AWK, sed, shell scripts, latex, HTML, etc

Personal History - Moral

One language may not last you all day, let alone your whole programming life

Programming Language Goals

- Original Model:
  - Computers expensive, people cheap; hand code to keep computer busy
- Today:
  - People expensive, computers cheap; write programs efficiently and correctly

Programming Language Goals

- Mythical Man-Month Author Fred Brookes

  "The most important two tools for system programming … are (1) high-level programming languages and (2) interactive languages"

Languages as Abstractions

- Abstraction from the Machine
- Abstraction from the Operational Model
- Abstraction of Errors
- Abstraction of Data
- Abstraction of Components
- Abstraction for Reuse

Why Study Programming Languages?

Helps you to:

- understand efficiency costs of given constructs
- reduce bugs by understanding semantics of constructs
- think about programming in new ways
- choose best language for task
- design better program interfaces (and languages)
- learn new languages

Study of Programming Languages

- Design and Organization
  - Syntax: How a program is written
  - Semantics: What a program means
  - Implementation: How a program runs
- Major Language Features
  - Imperative / Applicative / Rule-based
  - Sequential / Concurrent

Historical Environment

- Mainframe Era
  - Batch environments (through early 60’ s and 70’ s)
    - Programs submitted to operator as a pile of punch cards; programs were typically run over night and output put in programmer’ s bin

Historical Environment

- Mainframe Era
  - Interactive environments
    - Multiple teletypes and CRT’s hooked up to single mainframe
    - Time-sharing OS (Multics) gave users time slices
    - Lead to compilers with read-eval-print loops

Historical Environment

- Personal Computing Era
  - Small, cheap, powerful
  - Single user, single-threaded OS (at first any way)
  - Windows interfaces replaced line input
  - Wide availability lead to inter-computer communications and distributed systems

Historical Environment

- Networking Era
  - Local area networks for printing, file sharing, application sharing
  - Global network
    - First called ARPANET, now called Internet
    - Composed of a collection of protocols: FTP, Email (SMTP), HTTP (HMTL), URL

Features of a Good Language

- Simplicity – few clear constructs, each with unique meaning
- Orthogonality - every combination of features is meaningful, with meaning given by each feature
- Flexible control constructs

Features of a Good Language

- Rich data structures – allows programmer to naturally model problem
- Clear syntax design – constructs should suggest functionality
- Support for abstraction - program data reflects problem being solved; allows programmers to safely work locally

Features of a Good Language

- Expressiveness – concise programs
- Good programming environment
- Architecture independence and portability

Features of a Good Language

- Readability
  - Simplicity
  - Orthogonality
  - Flexible control constructs
  - Rich data structures
  - Clear syntax design

Features of a Good Language

- Writability
  - Simplicity
  - Orthogonality
  - Support for abstraction
  - Expressivity
  - Programming environment
  - Portability

Features of a Good Language

- Usually readability and writability call for the same language characteristics
- Sometimes they conflict:
  - Comments: Nested comments (e.g /*… /* … */ … */) enhance writability, but decrease readability

Features of a Good Language

- Reliability
  - Readability
  - Writability
  - Type Checking
  - Exception Handling
  - Restricted aliasing

Language Paradigms – Imperative Languages

- Main focus: machine state – the set of values stored in memory locations
- Command-driven: Each statement uses current state to compute a new

- Syntax: S1; S2; S3; ...
- Example languages: C, Pascal, FORTRAN, COBOL

Language Paradigms – Object-oriented Languages

- Classes are complex data types grouped with operations (methods) for creating, examining, and modifying elements (objects); subclasses include (inherit) the objects and methods from superclasses

Language Paradigms – Object-oriented Languages

- Computation is based on objects sending messages (methods applied to arguments) to other objects
- Syntax: Varies, object <- method(args)
- Example languages: Java, C++, Smalltalk

Language Paradigms – Applicative Languages

- Applicative (functional) languages
  - Programs as functions that take arguments and return values; arguments and returned values may be functions

Language Paradigms – Applicative Languages

- Applicative (functional) languages
  - Programming consists of building the function that computes the answer; function application and composition main method of computation
  - Syntax: P1(P2(P3 X))
  - Example languages: ML, LISP, Scheme, Haskell, Miranda

Language Paradigms – Logic Programming

- Rule-based languages
  - Programs as sets of basic rules for decomposing problem
  - Computation by deduction: search, unification and backtracking main components
  - Syntax: Answer :- specification rule
  - Example languages: (Prolog, Datalog,BNF Parsing)

Programming Language Implementation

- Develop layers of machines, each more primitive than the previous
- Translate between successive layers
- End at basic layer
- Ultimately hardware machine at bottom

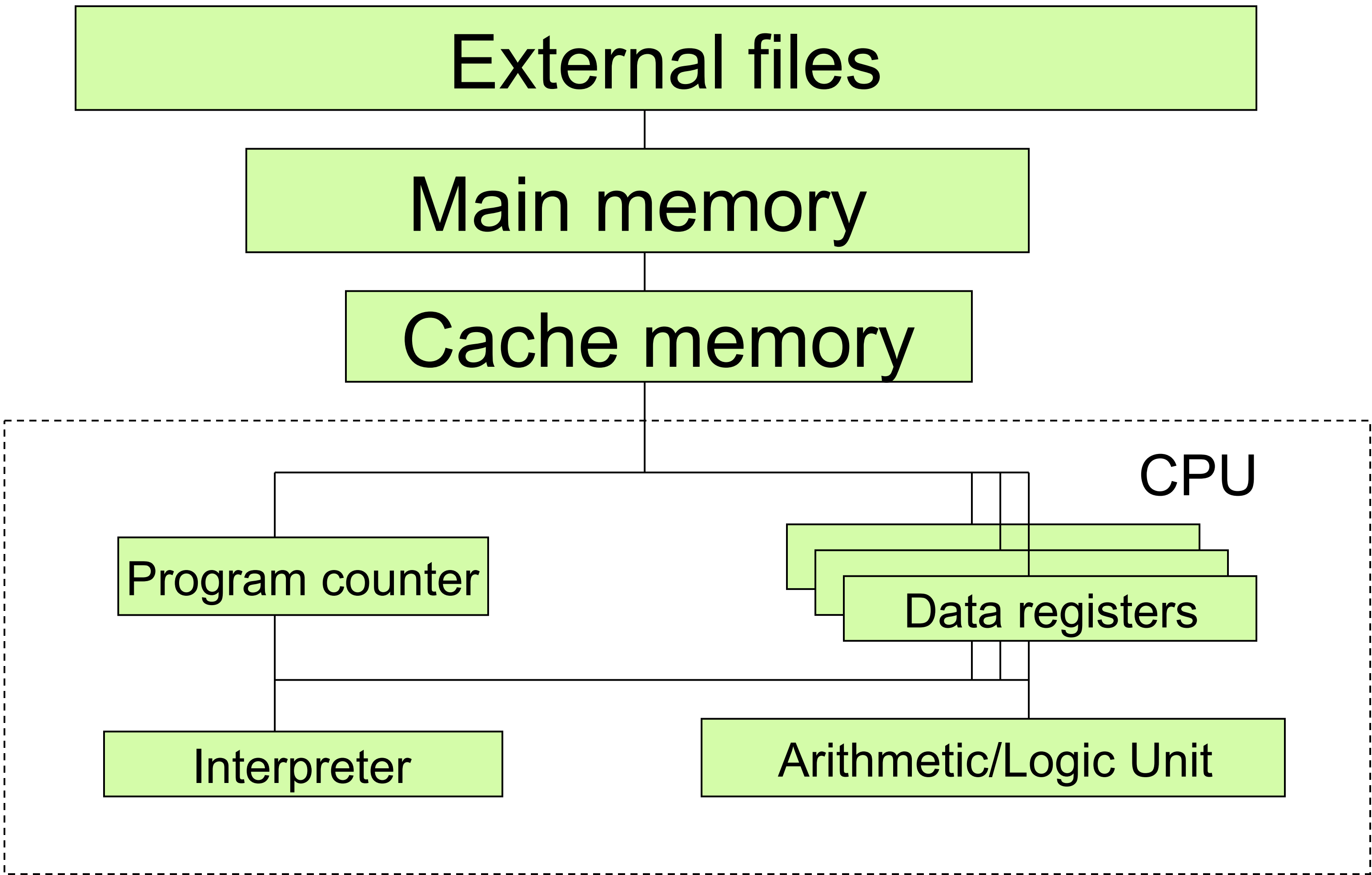
Basic Machine Components

- **Data**: basic data types and elements of those types
- **Primitive operations**: for examining, altering, and combining data
- **Sequence control**: order of execution of primitive operations

Basic Machine Components

- **Data access**: control of supply of data to operations
- **Storage management**: storage and update of program and data
- **External I/O**: access to data and programs from external sources, and output results

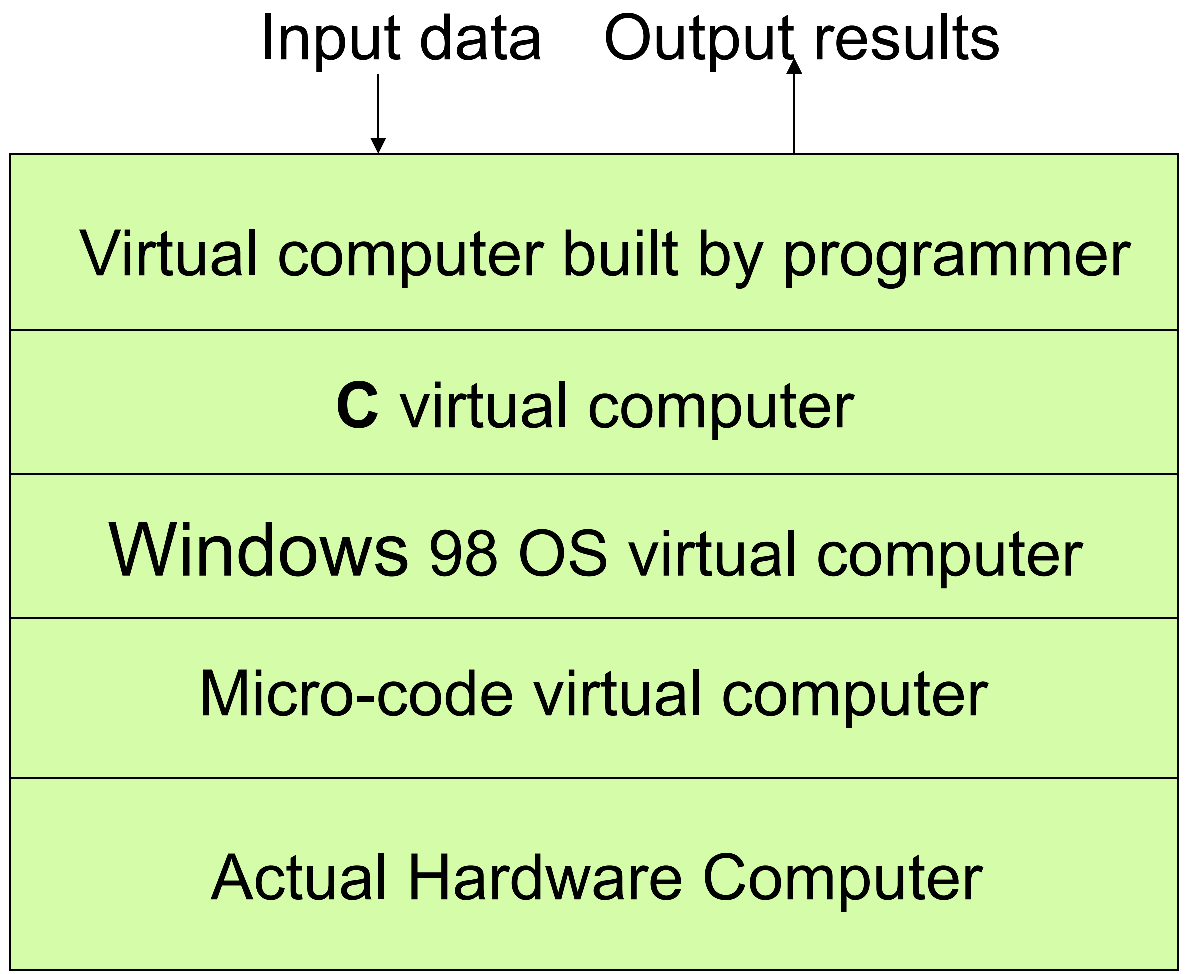
Basic Computer Architecture



Virtual (Software) Machines

- At first, programs written in assembly language (or at very first, machine language)
- Hand-coded to be very efficient
- Now, no longer write in native assembly language
- Use layers of software (e.g. operating system)
- Each layer makes a *virtual machine* in which the next layer is defined

Example Layers of Virtual Computers for a **C** Program



Virtual Machines Within Compilers

- Compilers often define layers of virtual machines
  - Functional languages: Untyped lambda calculus -> continuations -> generic pseudo-assembly -> machine specific code
  - May compile to intermediate language that is interpreted or compiled separately
    - Java virtual machine, CAML byte code

To Class

- Name some examples of virtual machines
- Name some examples of things that aren’t virtual machines

Interpretation Versus Compilation

- A **compiler** from language L1 to language L2 is a program that takes an L1 program and for each piece of code in L1 generates a piece of code in L2 of same meaning

Interpretation Versus Compilation

- An **interpreter** of L1 in L2 is an L2 program that executes the meaning of a given L1 program
- Compiler would examine the body of a loop once; an interpreter would examine it every time the loop was executed

Program Aspects

- Syntax: what valid programs look like
- Semantics: what valid programs mean; what they should compute
- Compiler must contain both information

Major Phases of a Compiler

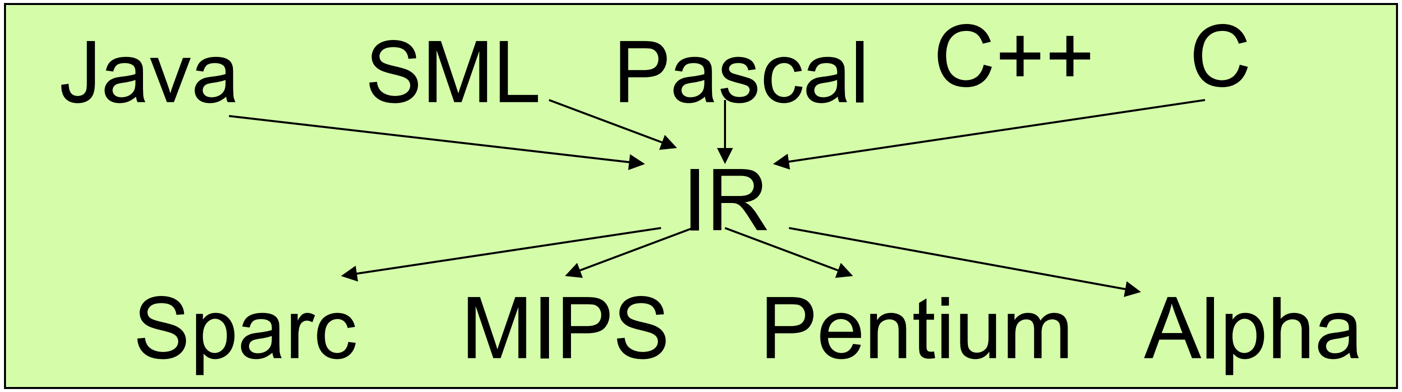
- Lex
  - Break the source into separate tokens
- Parse
  - Analyze phrase structure and apply semantic actions, usually to build an abstract syntax tree

Major Phases of a Compiler

- Semantic analysis
  - Determine what each phrase means, connect variable name to definition (typically with symbol tables), check types

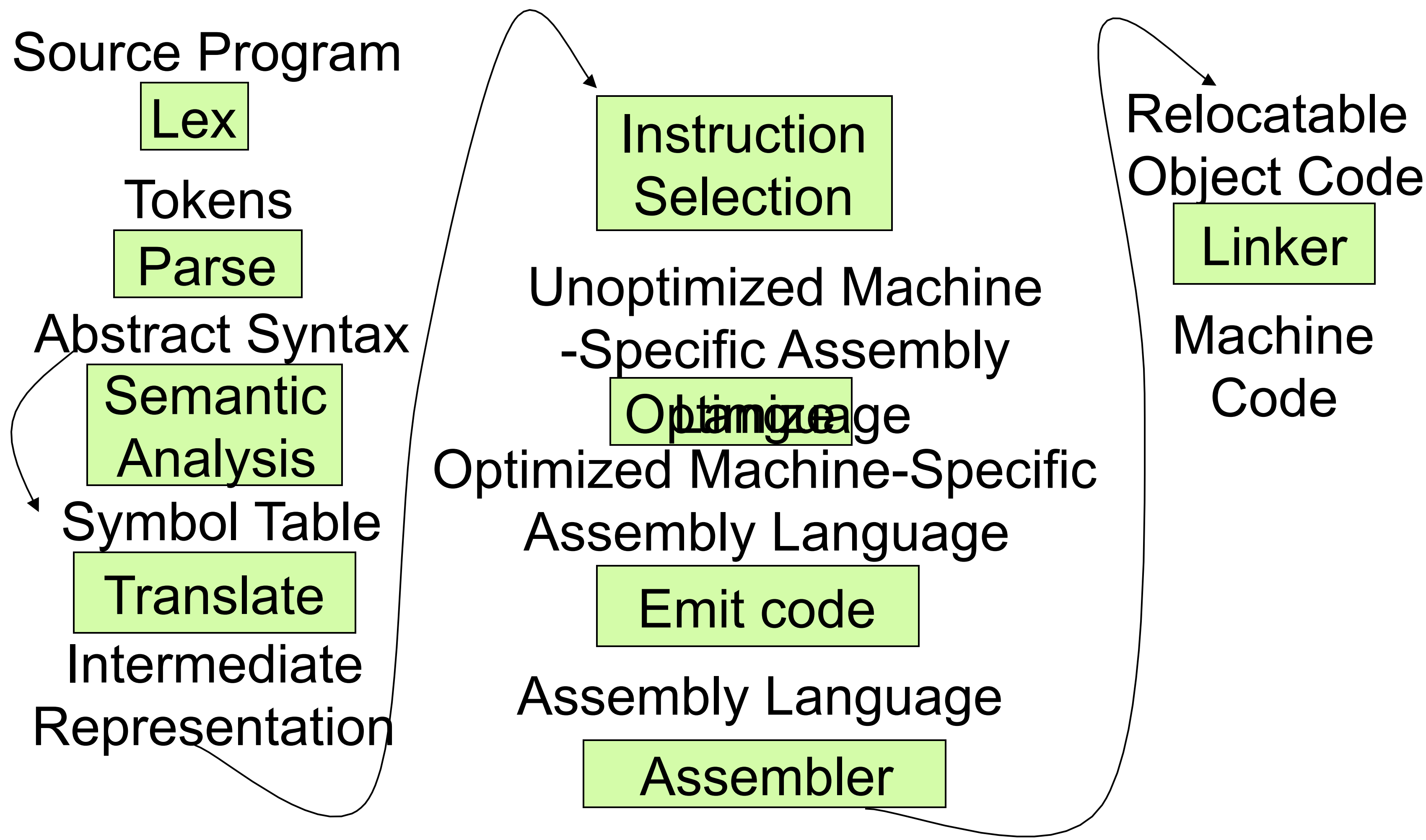
Major Phases of a Compiler

- Translate to intermediate representation



- Instruction selection
- Optimize
- Emit final machine code

Major Phases of a Compiler



Modified from "Modern Compiler Implementation in ML", by Andrew Appel

Example of Intermediate Representation

- Program code: X = Y + Z + W
  - tmp = Y + Z
  - X = tmp + W
  - Simpler language with no compound arithmetic expressions

Example of Optimization

Program code: X = Y + Z + W

▪ Load reg1 with Y ▪ Load reg2 with Z ▪ Add reg1 and reg2, saving to reg1 ▪ Store reg1 to tmp **	▪ Load reg1 with tmp ** ▪ Load reg2 with W ▪ Add reg1 and reg2, saving to reg1 ▪ Store reg1 to X

Eliminate two steps marked **

Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Question

- Observation: Functions are first-class values in this language
- Question: What value does the environment record for a function variable?
- Better question: What is the value of a fun expression?
- Answer: a closure

Save the Environment!

- A *closure* is a pair of an environment and an association of a sequence of variables (the input variables) with an expression (the function body), written:

$$f \rightarrow < (v1,\ldots,vn) \rightarrow exp, \rho_f >$$

- Where ρ_f is the environment in effect when f is defined (if f is a simple function)

Closure for plus_x

- When plus_x was defined, had environment:

$$\rho_{plus\_x} = \{x \to 12, \ldots, y \to 24, \ldots\}$$

- Recall: let plus_x y = y + x

  is really let plus_x = fun y -> y + x

- Closure for plus_x:

$$<y \to y + x, \rho_{plus\_x}>$$

- Environment just after plus_x defined:

$$\{plus\_x \to <y \to y + x, \rho_{plus\_x}>\} + \rho_{plus\_x}$$

Evaluation of Application of plus_x;;

- Have environment:

  $\rho = \{plus\_x \rightarrow <y \rightarrow y + x, \rho_{plus\_x}>, \ldots,$
  $y \rightarrow 3, \ldots\}$

  where $\rho_{plus\_x} = \{x \rightarrow 12, \ldots, y \rightarrow 24, \ldots\}$
- Eval (plus_x y, ρ) rewrites to
- Eval (app $<y \rightarrow y + x, \rho_{plus\_x}>$ 3, ρ) rewrites to
- Eval ($y + x$, $\{y \rightarrow 3\} + \rho_{plus\_x}$) rewrites to
- Eval ($3 + 12$, ρ_{plus_x}) = 15

Functions on tuples

```
# let plus_pair (n,m) = n + m;;
val plus_pair : int * int -> int = <fun>
# plus_pair (3,4);;
- : int = 7
# let double x = (x,x);;
val double : 'a -> 'a * 'a = <fun>
# double 3;;
- : int * int = (3, 3)
# double "hi";;
- : string * string = ("hi", "hi")
```

Match Expressions

```
# let triple_to_pair triple =
  match triple
  with (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);;
val triple_to_pair : int * int * int -> int * int =
  <fun>
```

- Each clause: pattern on left, expression on right
- Each x, y has scope of only its clause
- Use first matching clause

Closure for plus_pair

- Assume ρ_{plus_pair} was the environment just before plus_pair defined
- Closure for plus_pair:

$$<(n,m) \rightarrow n + m, \rho_{plus\_pair}>$$

- Environment just after plus_pair defined:

$$\{plus\_pair \rightarrow <(n,m) \rightarrow n + m, \rho_{plus\_pair} >\}$$
$$+ \rho_{plus\_pair}$$

Evaluation of Application with Closures

- In environment ρ, evaluate left term to closure, $c = <(x_1,\ldots,x_n) \rightarrow b, \rho>$
- $(x_1,\ldots,x_n)$ variables in (first) argument
- Evaluate the right term to values, $(v_1,\ldots,v_n)$
- Update the environment ρ to

  $\rho' = \{x_1 \rightarrow v_1,\ldots, x_n \rightarrow v_n\}+ \rho$
- Evaluate body b in environment ρ'

Evaluation of Application of plus_pair

- Assume environment

$\rho = \{x \rightarrow 3\ldots,$

$\quad plus\_pair \rightarrow <(n,m) \rightarrow n + m, \rho_{plus\_pair}>\} +$

$\quad \rho_{plus\_pair}$

- Eval (plus_pair (4,x), ρ)=
- Eval (app <(n,m) $\rightarrow$n + m, ρ_{plus_pair}> (4,x), ρ)) =
- Eval (app <(n,m) $\rightarrow$n + m, ρ_{plus_pair}> (4,3), ρ)) =
- Eval (n + m, {n -> 4, m -> 3} + ρ_{plus_pair}) =
- Eval (4 + 3, {n -> 4, m -> 3} + ρ_{plus_pair}) = 7

Curried vs Uncurried

- Recall

val add_three : int -> int -> int -> int = <fun>

- How does it differ from

```
# let add_triple (u,v,w) = u + v + w;;
val add_triple : int * int * int -> int = <fun>
```

- add_three is *curried*;
- add_triple is *uncurried*

Curried vs Uncurried

```
# add_triple (6,3,2);;
- : int = 11
# add_triple 5 4;;
Characters 0-10:
  add_triple 5 4;;
  ^^^^^^^^^^
This function is applied to too many arguments,
maybe you forgot a `;'
# fun x -> add_triple (5,4,x);;
: int -> int = <fun>
```

Scoping Question

Consider this code:

```
let x = 27;;
let f x =
    let x = 5 in
        (fun x -> print_int x) 10;;
f 12;;
```

What value is printed?

5

10

12

27

Higher Order Functions

- A function is *higher-order* if it takes a function as an argument or returns one as a result
- Example:

```
# let compose f g = fun x -> f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c ->
  'b = <fun>
```

- The type ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b is a higher order type because of ('a -> 'b) and ('c -> 'a) and -> 'c -> 'b

Thrice

- Recall:

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

- How do you write thrice with compose?

Thrice

- Recall:

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

- How do you write thrice with compose?

```
# let thrice f = compose f (compose f f);;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

- Is this the only way?

Partial Application

```
# (+);;
- : int -> int -> int = <fun>
# (+) 2 3;;
- : int = 5
# let plus_two = (+) 2;;
val plus_two : int -> int = <fun>
# plus_two 7;;
- : int = 9
```

- Patial application also called *sectioning*

Lambda Lifting

- You must remember the rules for evaluation when you use partial application

```
# let add_two = (+) (print_string "test\n"; 2);;
test
val add_two : int -> int = <fun>
# let add2 =      (* lambda lifted *)
   fun x -> (+) (print_string "test\n"; 2) x;;
val add2 : int -> int = <fun>
```

Lambda Lifting

```
# thrice add_two 5;;
- : int = 11
# thrice add2 5;;
test
test
test
- : int = 11
```

- Lambda lifting delayed the evaluation of the argument to (+) until the second argument was supplied

Partial Application and “Unknown Types”

- Recall compose plus_two:

```
# let f1 = compose plus_two;;
val f1 : ('_a -> int) -> '_a -> int = <fun>
```

- Compare to lambda lifted version:

```
# let f2 = fun g ->  compose plus_two g;;
val f2 : ('a -> int) -> 'a -> int = <fun>
```

- What is the difference?

Partial Application and “Unknown Types”

- ‘_a can only be instantiated once for an expression

```
# f1 plus_two;;
- : int -> int = <fun>
# f1 List.length;;
Characters 3-14:
  f1 List.length;;
     ^^^^^^^^^^^
This expression has type 'a list -> int but is here used
   with type int -> int
```

Partial Application and "Unknown Types"

- 'a can be repeatedly instantiated

```
# f2 plus_two;;
- : int -> int = <fun>
# f2 List.length;;
- : '_a list -> int = <fun>
```

Recursive Functions

```
# let rec factorial n =
    if n = 0 then 1 else n * factorial (n - 1);;
  val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
# (* rec  is needed for recursive function
   declarations *)
```

Recursion Example

Compute n^2 recursively using:

$$n^2 = (2 * n - 1) + (n - 1)^2$$

```
# let rec nthsq n =          (* rec for recursion *)
    match n                  (* pattern matching for cases *)
    with 0 -> 0              (* base case *)
    | n -> (2 * n -1)        (* recursive case *)
          + nthsq (n -1);;   (* recursive call *)
val nthsq : int -> int = <fun>
# nthsq 3;;
-   : int = 9
```

Structure of recursion similar to inductive proof

Recursion and Induction

```
# let rec nthsq n = match n with 0 -> 0
        | n -> (2 * n - 1) + nthsq (n - 1) ;;
```

- Base case is the last case; it stops the computation
- Recursive call must be to arguments that are somehow smaller - must progress to base case
- **if** or **match** must contain base case
- Failure of these may cause failure of termination

Lists

- First example of a recursive datatype (aka algebraic datatype)

- Unlike tuples, lists are homogeneous in type (all elements same type)

Lists

- List can take one of two forms:
  - Empty list, written [ ]
  - Non-empty list, written x :: xs
    - x is head element, xs is tail list, :: called "cons"
  - Syntactic sugar: [x] == x :: [ ]
  - [x1; x2; …; xn] == x1 :: x2 :: … :: xn :: [ ]

Lists

```
# let fib5 = [8;5;3;2;1;1];;
val fib5 : int list = [8; 5; 3; 2; 1; 1]
# let fib6 = 13 :: fib5;;
val fib6 : int list = [13; 8; 5; 3; 2; 1; 1]
# (8::5::3::2::1::1::[ ]) = fib5;;
- : bool = true
# fib5 @ fib6;;
- : int list = [8; 5; 3; 2; 1; 1; 13; 8; 5; 3; 2; 1;
  1]
```

Lists are Homogeneous

```
# let bad_list = [1; 3.2; 7];;
Characters 19-22:
  let bad_list = [1; 3.2; 7];;
                     ^^^
This expression has type float but is here
  used with type int
```

Question

- Which one of these lists is invalid?

1. [2; 3; 4; 6]
2. [2,3; 4,5; 6,7]
3. [(2.3,4); (3.2,5); (6,7.2)]
4. [[“hi”; “there”]; [“wahcha”]; [ ]; [“doin”]]

Answer

- Which one of these lists is invalid?

1. [2; 3; 4; 6]
2. [2,3; 4,5; 6,7]
3. [(2.3,4); (3.2,5); (6,7.2)]
4. [[“hi”; “there”]; [“wahcha”]; [ ]; [“doin”]]

- 3 is invalid because of last pair

Functions Over Lists

```
# let rec double_up list =
    match list
    with [ ] -> [ ]  (* pattern before ->,
                        expression after *)
      | (x :: xs) -> (x :: x :: double_up xs);;
val double_up : 'a list -> 'a list = <fun>
# let fib5_2 = double_up fib5;;
val fib5_2 : int list = [8; 8; 5; 5; 3; 3; 2; 2; 1;
  1; 1; 1]
```

Functions Over Lists

```
# let silly = double_up ["hi"; "there"];;
val silly : string list = ["hi"; "hi"; "there"; "there"]
# let rec poor_rev list =
  match list
  with [] -> []
     | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
# poor_rev silly;;
- : string list = ["there"; "there"; "hi"; "hi"]
```

Functions Over Lists

```
# let rec map f list =
  match list
  with [] -> []
  | (h::t) -> (f h) :: (map f t);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]
# map (fun x -> x - 1) fib6;;
: int list = [12; 7; 4; 2; 1; 0; 0]
```

Iterating over lists

```
# let rec fold_left f a list =
  match list
  with [] -> a
  | (x :: xs) -> fold_left f (f a x) xs;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a =
   <fun>
# fold_left
   (fun () -> print_string)
   ()
   ["hi"; "there"];;
hithere- : unit = ()
```

Iterating over lists

```
# let rec fold_right f list b =
  match list
  with [] -> b
  | (x :: xs) -> f x (fold_right f xs b);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b =
  <fun>
# fold_right
   (fun s -> fun () -> print_string s)
   ["hi"; "there"]
   ();;
therehi- : unit = ()
```

Structural Recursion

- Functions on recursive datatypes (eg lists) tend to be recursive
- Recursion over recursive datatypes generally by structural recursion
  - Recursive calls made to components of structure of the same recursive type
  - Base cases of recursive types stop the recursion of the function

Structural Recursion : List Example

```
# let rec length list = match list
  with [ ] -> 0   (* Nil case *)
  | x :: xs -> 1 + length xs;;  (* Cons case *)
val length : 'a list -> int = <fun>
# length [5; 4; 3; 2];;
- : int = 4
```

- Nil case [ ] is base case
- Cons case recurses on component list xs

Forward Recursion

- In Structural Recursion, split input into components and (eventually) recurse
- Forward Recursion form of Structural Recursion
- In forward recursion, first call the function recursively on all recursive components, and then build final result from partial results
- Wait until whole structure has been traversed to start building answer

Forward Recursion: Examples

```
# let rec double_up list =
    match list
    with [ ] -> [ ]
       | (x :: xs) -> (x :: x :: double_up xs);;
val double_up : 'a list -> 'a list = <fun>

# let rec poor_rev list =
  match list
  with [] -> []
     | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
```

Encoding Recursion with Fold

```
# let rec append list1 list2 = match list1 with
  [ ] -> list2 | x::xs -> x :: append xs list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

Base Case Operation Recursive Call

```
# let append list1 list2 =
   fold_right (fun x y -> x :: y) list1 list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [1;2;3] [4;5;6];;
 - : int list = [1; 2; 3; 4; 5; 6]
```

Mapping Recursion

- One common form of structural recursion applies a function to each element in the structure

```
# let rec doubleList list = match list
  with [ ] -> [ ]
  | x::xs -> 2 * x :: doubleList xs;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```

Mapping Recursion

- Can use the higher-order recursive map function instead of direct recursion

```
# let doubleList list =
    List.map (fun x -> 2 * x) list;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```

- Same function, but no rec

Folding Recursion

- Another common form “folds” an operation over the elements of the structure

```
# let rec multList list = match list
  with [ ] -> 1
  | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;4;6];;
- : int = 48
```

- Computes (2 * (4 * (6 * 1)))

Folding Recursion

- multList folds to the right
- Same as:

```
# let multList list =
    List.fold_right
    (fun x -> fun p -> x * p)
    list 1;;
val multList : int list -> int = <fun>
# multList [2;4;6];;
- : int = 48
```

How long will it take?

- Remember the big-O notation from CS 225 and CS 273
- Question: given input of size *n*, how long to generate output?
- Express output time in terms of input size, omit constants and take biggest power

How long will it take?

Common big-O times:

- Constant time $O(1)$
  - input size doesn't matter
- Linear time $O(n)$
  - double input $\Rightarrow$ double time
- Quadratic time $O(n^2)$
  - double input $\Rightarrow$ quadruple time
- Exponential time $O(2^n)$
  - increment input $\Rightarrow$ double time

Linear Time

- Expect most list operations to take linear time $O(n)$
- Each step of the recursion can be done in constant time
- Each step makes only one recursive call
- List example: multList, append
- Integer example: factorial

Quadratic Time

- Each step of the recursion takes time proportional to input
- Each step of the recursion makes only one recursive call.
- List example:

```
# let rec poor_rev list = match list
  with [] -> []
    | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
```

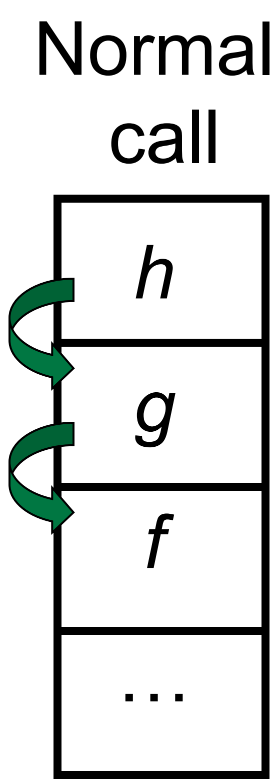
Exponential running time

- Hideous running times on input of any size
- Each step of recursion takes constant time
- Each recursion makes two recursive calls
- Easy to write naïve code that is exponential for functions that can be linear

Exponential running time

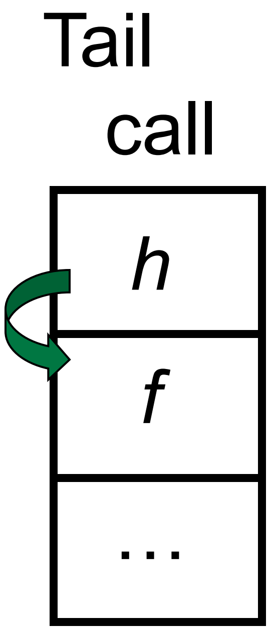
```
# let rec naiveFib n = match n
  with 0 -> 0
  | 1 -> 1
  | _ -> naiveFib (n-1) + naiveFib (n-2);;
val naiveFib : int -> int = <fun>
```

An Important Optimization



- When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished
- What if *f* calls *g* and *g* calls *h*, but calling *h* is the last thing *g* does (a *tail call*)?

An Important Optimization



- When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished
- What if *f* calls *g* and *g* calls *h*, but calling *h* is the last thing *g* does (a *tail call*)?
- Then *h* can return directly to *f* instead of *g*

Tail Recursion

- A recursive program is tail recursive if all recursive calls are tail calls
- Tail recursive programs may be optimized to be implemented as loops, thus removing the function call overhead for the recursive calls
- Tail recursion generally requires extra "accumulator" arguments to pass partial results
  - May require an auxiliary function

Tail Recursion - Example

```
# let rec rev_aux list revlist =
  match list with [ ] -> revlist
  | x :: xs -> rev_aux xs (x::revlist);;
val rev_aux : 'a list -> 'a list -> 'a list = <fun>

# let rev list = rev_aux list [ ];;
val rev : 'a list -> 'a list = <fun>
```

- What is its running time?

Comparison

- poor_rev [1,2,3] =
- (poor_rev [2,3]) @ [1] =
- ((poor_rev [3]) @ [2]) @ [1] =
- (((poor_rev [ ]) @ [3]) @ [2]) @ [1] =
- (([ ] @ [3]) @ [2]) @ [1]) =
- ([3] @ [2]) @ [1] =
- (3:: ([ ] @ [2])) @ [1] =
- [3,2] @ [1] =
- 3 :: ([2] @ [1]) =
- 3 :: (2:: ([ ] @ [1])) = [3, 2, 1]

Comparison

- rev [1,2,3] =
- rev_aux [1,2,3] [ ] =
- rev_aux [2,3] [1] =
- rev_aux [3] [2,1] =
- rev_aux [ ] [3,2,1] = [3,2,1]

Folding Functions over Lists

How are the following functions similar?

```
# let rec sumlist list = match list with
  [ ] -> 0 | x::xs -> x + sumlist xs;;
val sumlist : int list -> int = <fun>
# sumlist [2;3;4];;
- : int = 9
# let rec prodlist list = match list with
  [ ] -> 1 | x::xs -> x * prodlist xs;;
val prodlist : int list -> int = <fun>
# prodlist [2;3;4];;
- : int = 24
```

Folding

```
# let rec fold_left f a list = match list
  with [] -> a | (x :: xs) -> fold_left f (f a x) xs;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a =
    <fun>
```

$\boxed{\text{fold\_left f a } [x_1; x_2; \ldots; x_n] = \text{f}(\ldots(\text{f } (\text{f a } x_1)\ x_2)\ldots)x_n}$

```
# let rec fold_right f list b = match list
  with [ ] -> b | (x :: xs) -> f x (fold_right f xs b);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b =
    <fun>
```

$\boxed{\text{fold\_right f } [x_1; x_2; \ldots; x_n] \text{ b} = \text{f } x_1 (\text{f } x_2 (\ldots(\text{f } x_n \text{ b})\ldots))}$

Folding - Forward Recursion

\# let sumlist list = fold_right (+) list 0;;

val sumlist : int list -> int = <fun>

\# sumlist [2;3;4];;

\- : int = 9

\# let prodlist list = fold_right (*) list 1;;

val prodlist : int list -> int = <fun>

\# prodlist [2;3;4];;

\- : int = 24

Folding - Tail Recursion

```
- # let rev list =
-        fold_left
-          (fun l -> fun x -> x :: l)      //comb op
           []                //accumulator cell
           list
```

Folding

- Can replace recursion by fold_right in any forward primitive recursive definition
  - Primitive recursive means it only recurses on immediate subcomponents of recursive data structure
- Can replace recursion by fold_left in any tail primitive recursive definition

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Functions Over Lists

```
# let rec map f list =
  match list
  with [] -> []
  | (h::t) -> (f h) :: (map f t);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]
# map (fun x -> x - 1) fib6;;
: int list = [12; 7; 4; 2; 1; 0; 0]
```

Iterating over lists

```
# let rec fold_left f a list =
  match list
  with [] -> a
  | (x :: xs) -> fold_left f (f a x) xs;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a =
   <fun>
# fold_left
   (fun () -> print_string)
   ()
   ["hi"; "there"];;
hithere- : unit = ()
```

Iterating over lists

```
# let rec fold_right f list b =
  match list
  with [] -> b
  | (x :: xs) -> f x (fold_right f xs b);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b =
   <fun>
# fold_right
    (fun s -> fun () -> print_string s)
    ["hi"; "there"]
    ();;
therehi- : unit = ()
```

Structural Recursion

- Functions on recursive datatypes (eg lists) tend to be recursive
- Recursion over recursive datatypes generally by structural recursion
  - Recursive calls made to components of structure of the same recursive type
  - Base cases of recursive types stop the recursion of the function

Structural Recursion : List Example

```
# let rec length list = match list
  with [ ] -> 0   (* Nil case *)
  | x :: xs -> 1 + length xs;;  (* Cons case *)
val length : 'a list -> int = <fun>
# length [5; 4; 3; 2];;
- : int = 4
```

- Nil case [ ] is base case
- Cons case recurses on component list xs

Forward Recursion

- In Structural Recursion, split input into components and (eventually) recurse
- Forward Recursion form of Structural Recursion
- In forward recursion, first call the function recursively on all recursive components, and then build final result from partial results
- Wait until whole structure has been traversed to start building answer

Forward Recursion: Examples

```
# let rec double_up list =
    match list
    with [ ] -> [ ]
       | (x :: xs) -> (x :: x :: double_up xs);;
val double_up : 'a list -> 'a list = <fun>

# let rec poor_rev list =
  match list
  with [] -> []
     | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
```

Encoding Recursion with Fold

let rec append list1 list2 = match list1 with
[ ] -> list2 | x::xs -> x :: append xs list2;;
val append : 'a list -> 'a list -> 'a list = <fun>

Base Case | Operation | Recursive Call

let append list1 list2 =
fold_right (fun x y -> x :: y) list1 list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
append [1;2;3] [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]

Mapping Recursion

- One common form of structural recursion applies a function to each element in the structure

```
# let rec doubleList list = match list
   with [ ] -> [ ]
   | x::xs -> 2 * x :: doubleList xs;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```

Mapping Recursion

- Can use the higher-order recursive map function instead of direct recursion

```
# let doubleList list =
    List.map (fun x -> 2 * x) list;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```

- Same function, but no rec

Folding Recursion

- Another common form “folds” an operation over the elements of the structure

```
# let rec multList list = match list
  with [ ] -> 1
  | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;4;6];;
- : int = 48
```

- Computes (2 * (4 * (6 * 1)))

Folding Recursion

- multList folds to the right
- Same as:

```
# let multList list =
    List.fold_right
    (fun x -> fun p -> x * p)
    list 1;;
val multList : int list -> int = <fun>
# multList [2;4;6];;
- : int = 48
```

Folding Functions over Lists

How are the following functions similar?

```
# let rec sumlist list = match list with
  [ ] -> 0 | x::xs -> x + sumlist xs;;
val sumlist : int list -> int = <fun>
# sumlist [2;3;4];;
- : int = 9
# let rec prodlist list = match list with
  [ ] -> 1 | x::xs -> x * prodlist xs;;
val prodlist : int list -> int = <fun>
# prodlist [2;3;4];;
- : int = 24
```

Folding

```
# let rec fold_left f a list = match list
  with [] -> a | (x :: xs) -> fold_left f (f a x) xs;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a =
   <fun>
```

$\boxed{\text{fold\_left f a } [x_1;\ x_2;\ldots;x_n] = \text{f}(\ldots(\text{f } (\text{f a } x_1)\ x_2)\ldots)x_n}$

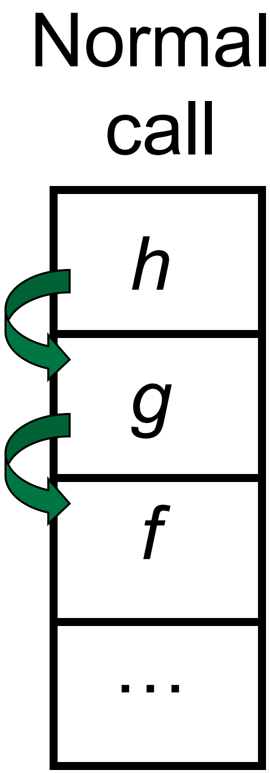
```
# let rec fold_right f list b = match list
  with [ ] -> b | (x :: xs) -> f x (fold_right f xs b);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b =
   <fun>
```

$\boxed{\text{fold\_right f } [x_1;\ x_2;\ldots;x_n]\text{ b} = \text{f } x_1(\text{f } x_2\,(\ldots(\text{f } x_n\text{ b})\ldots))}$

Folding - Forward Recursion

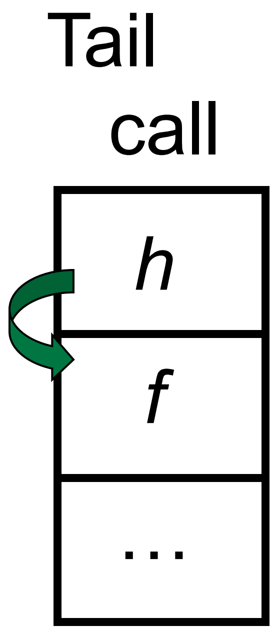
```
# let sumlist list = fold_right (+) list 0;;
val sumlist : int list -> int = <fun>
# sumlist [2;3;4];;
- : int = 9
# let prodlist list = fold_right ( * ) list 1;;
val prodlist : int list -> int = <fun>
# prodlist [2;3;4];;
- : int = 24
```

An Important Optimization



- When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished
- What if *f* calls *g* and *g* calls *h*, but calling *h* is the last thing *g* does (a *tail call*)?

An Important Optimization



- When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished
- What if *f* calls *g* and *g* calls *h*, but calling *h* is the last thing *g* does (a *tail call*)?
- Then *h* can return directly to *f* instead of *g*

Tail Recursion

- A recursive program is tail recursive if all recursive calls are tail calls
- Tail recursive programs may be optimized to be implemented as loops, thus removing the function call overhead for the recursive calls
- Tail recursion generally requires extra “accumulator” arguments to pass partial results
  - May require an auxiliary function

Example of Tail Recursion

```
# let rec prod l =
    match l with [] -> 1
    | (x :: rem) -> x * prod rem;;
val prod : int list -> int = <fun>
# let prod list =
    let rec prod_aux l acc =
        match l with [] -> acc
        | (y :: rest) -> prod_aux rest (acc * y)
(* Uses associativity of multiplication *)
    in prod_aux list 1;;
 val prod : int list -> int = <fun>
```

Recall

```
# let rec poor_rev list = match list
  with [] -> []
    | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
```

- What is its running time?

Quadratic Time

- Each step of the recursion takes time proportional to input
- Each step of the recursion makes only one recursive call.
- List example:

```
# let rec poor_rev list = match list
  with [] -> []
    | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
```

Tail Recursion - Example

```
# let rec rev_aux list revlist =
  match list with [ ] -> revlist
  | x :: xs -> rev_aux xs (x::revlist);;
val rev_aux : 'a list -> 'a list -> 'a list = <fun>

# let rev list = rev_aux list [ ];;
val rev : 'a list -> 'a list = <fun>
```

- What is its running time?

Comparison

- poor_rev [1,2,3] =
- (poor_rev [2,3]) @ [1] =
- ((poor_rev [3]) @ [2]) @ [1] =
- (((poor_rev [ ]) @ [3]) @ [2]) @ [1] =
- (([ ] @ [3]) @ [2]) @ [1]) =
- ([3] @ [2]) @ [1] =
- (3:: ([ ] @ [2])) @ [1] =
- [3,2] @ [1] =
- 3 :: ([2] @ [1]) =
- 3 :: (2:: ([ ] @ [1])) = [3, 2, 1]

Comparison

- rev [1,2,3] =
- rev_aux [1,2,3] [ ] =
- rev_aux [2,3] [1] =
- rev_aux [3] [2,1] =
- rev_aux [ ] [3,2,1] = [3,2,1]

Folding - Tail Recursion

```
# let rev list =
        fold_left
          (fun l -> fun x -> x :: l)     //comb op
          []                //accumulator cell
          list
```

Encoding Tail Recursion with fold_left

```
# let prod list = let rec prod_aux l acc =
      match l with [] -> acc
      | (y :: rest) -> prod_aux rest (acc * y)
    in prod_aux list 1;;
val prod : int list -> int = <fun>
```

Init Acc Value   Recursive Call   Operation

```
# let prod list =
  List.fold_left (fun acc y -> acc * y) 1 list;;
val prod: int list -> int = <fun>
# prod [4;5;6];;
 - : int =120
```

Folding

- Can replace recursion by fold_right in any forward primitive recursive definition
  - Primitive recursive means it only recurses on immediate subcomponents of recursive data structure
- Can replace recursion by fold_left in any tail primitive recursive definition

Map from Fold

```
# let map f list =
 fold_right (fun x y -> f x :: y) list [ ];;
val map : ('a -> 'b) -> 'a list -> 'b list =
  <fun>
# map ((+)1) [1;2;3];;
- : int list = [2; 3; 4]
```

- Can you write fold_right (or fold_left) with just map? How, or why not?

Higher Order Functions

- A function is *higher-order* if it takes a function as an argument or returns one as a result
- Example:

let compose f g = fun x -> f (g x);;

val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

- The type ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b is a higher order type because of ('a -> 'b) and ('c -> 'a) and -> 'c -> 'b

Partial Application

```
# (+);;
- : int -> int -> int = <fun>
# (+) 2 3;;
- : int = 5
# let plus_two = (+) 2;;
val plus_two : int -> int = <fun>
# plus_two 7;;
- : int = 9
```

- Patial application also called *sectioning*

Lambda Lifting

- You must remember the rules for evaluation when you use partial application

```
# let add_two = (+) (print_string "test\n"; 2);;
test
val add_two : int -> int = <fun>
# let add2 =      (* lambda lifted *)
    fun x -> (+) (print_string "test\n"; 2) x;;
val add2 : int -> int = <fun>
```

Lambda Lifting

```
# thrice add_two 5;;
- : int = 11
# thrice add2 5;;
test
test
test
- : int = 11
```

- Lambda lifting delayed the evaluation of the argument to (+) until the second argument was supplied

Partial Application and “Unknown Types”

- Recall compose plus_two:

```
# let f1 = compose plus_two;;
val f1 : ('_a -> int) -> '_a -> int = <fun>
```

- Compare to lambda lifted version:

```
# let f2 = fun g -> compose plus_two g;;
val f2 : ('a -> int) -> 'a -> int = <fun>
```

- What is the difference?

Partial Application and “Unknown Types”

- ‘_a can only be instantiated once for an expression

```
# f1 plus_two;;
- : int -> int = <fun>
# f1 List.length;;
Characters 3-14:
  f1 List.length;;
     ^^^^^^^^^^^
This expression has type 'a list -> int but is here used
   with type int -> int
```

Partial Application and “Unknown Types”

- ‘a can be repeatedly instantiated

```
# f2 plus_two;;
- : int -> int = <fun>
# f2 List.length;;
- : '_a list -> int = <fun>
```

Continuation Passing Style

- A programming technique for all forms of “non-local” control flow:
  - non-local jumps
  - exceptions
  - general conversion of non-tail calls to tail calls
- Essentially it’s a higher-order function version of GOTO

Continuations

- Idea: Use functions to represent the control flow of a program
- Method: Each procedure takes a function as an argument to which to pass its result; outer procedure “returns” no result
- Function receiving the result called a continuation
- Continuation acts as “accumulator” for work still to be done

Example of Tail Recursion

```
# let rec app fl x =
    match fl with [] -> x
     | (f :: rem_fs) -> f (app rem_fs x);;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
# let app fs x =
    let rec app_aux fl acc=
        match fl with [] -> acc
        | (f :: rem_fs) -> app_aux rem_fs
                              (fun z -> acc (f z))
    in app_aux fs (fun y -> y) x;;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
```

Continuation Passing Style

- Writing procedures so that they take a continuation to which to give (pass) the result, and return no result, is called continuation passing style (CPS)

Example of Tail Recursion & CSP

```
# let app fs x =
    let rec app_aux fl acc=
        match fl with [] -> acc
        | (f :: rem_fs) -> app_aux rem_fs
                              (fun z -> acc (f z))
    in app_aux fs (fun y -> y) x;;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
# let rec appk fl x k =
    match fl with [] -> k x
    | (f :: rem_fs) -> appk rem_fs x (fun z -> k (f z));;
val appk : ('a -> 'a) list -> 'a -> ('a -> 'b) -> 'b
```

Continuation Passing Style

- A compilation technique to implement non-local control flow, especially useful in interpreters.
- A formalization of non-local control flow in denotational semantics

Terms

- A function is in Direct Style when it returns its result back to the caller.
- A Tail Call occurs when a function returns the result of another function call without any more computations (eg tail recursion)
- A function is in Continuation Passing Style when it passes its result to another function.
- Instead of returning the result to the caller, we pass it forward to another function.

Example

- Simple reporting continuation:

```
# let report x = (print_int x; print_newline( ) );;
val report : int -> unit = <fun>
```

- Simple function using a continuation:

```
# let plusk a b k = k (a + b)
val plusk : int -> int -> (int -> 'a) -> 'a = <fun>
# plusk 20 22 report;;
42
- : unit = ()
```

Simple Functions Taking Continuations

- Given a primitive operation, can convert it to pass its result forward to a continuation
- Examples:

```
# let subk x y k = k(x + y);;
val subk : int -> int -> (int -> 'a) -> 'a = <fun>
# let eqk x y k = k(x = y);;
val eqk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>
# let timesk x y k = k(x * y);;
val timesk : int -> int -> (int -> 'a) -> 'a = <fun>
```

Nesting Continuations

```
# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>
# let add_three x y z= let p = x + y in  p + z;;
val add_three : int -> int -> int -> int = <fun>
# let add_three_k x y z k =
   addk x y (fun p -> addk p z k );;
val add_three_k : int -> int -> int -> (int -> 'a)
   -> 'a = <fun>
```

Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Continuations

- A programming technique for all forms of “non-local” control flow:
  - non-local jumps
  - exceptions
  - general conversion of non-tail calls to tail calls
- Essentially it’s a higher-order function version of GOTO

Continuations

- Idea: Use functions to represent the control flow of a program
- Method: Each procedure takes a function as an extra argument to which to pass its result; outer procedure "returns" no result
- Function receiving the result called a continuation
- Continuation acts as "accumulator" for work still to be done

Example of Tail Recursion

```
# let rec app fl x =
    match fl with [] -> x
     | (f :: rem_fs) -> f (app rem_fs x);;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
# let app fs x =
    let rec app_aux fl acc=
        match fl with [] -> acc
        | (f :: rem_fs) -> app_aux rem_fs
                                  (fun z -> acc (f z))
    in app_aux fs (fun y -> y) x;;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
```

Example of Tail Recursion

```
# let rec app fl x =
    match fl with [] -> x
     | (f :: rem_fs) -> f (app rem_fs x);;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
# let app fs x =
    let rec app_aux fl acc=
        match fl with [] -> acc
        | (f :: rem_fs) -> app_aux rem_fs
                              (fun z -> acc (f z))
    in app_aux fs (fun y -> y) x;;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
```

Continuation Passing Style

- Writing procedures such that all procedure calls take a continuation to which to give (pass) the result, and return no result, is called continuation passing style (CPS)

Example of Tail Recursion & CSP

```
# let app fs x =
    let rec app_aux fl acc=
        match fl with [] -> acc
        | (f :: rem_fs) -> app_aux rem_fs
                                 (fun z -> acc (f z))
    in app_aux fs (fun y -> y) x;;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
# let rec appk fl x k =
    match fl with [] -> k x
    | (f :: rem_fs) -> appk rem_fs x (fun z -> k (f z));;
val appk : ('a -> 'a) list -> 'a -> ('a -> 'b) -> 'b
```

Continuation Passing Style

- A compilation technique to implement non-local control flow, especially useful in interpreters.
- A formalization of non-local control flow in denotational semantics
- Possible intermediate state in compiling functional code

Why CPS?

- Makes order of evaluation explicitly clear
- Allocates variables (to become registers) for each step of computation
- Essentially converts functional programs into imperative ones
  - Major step for compiling to assembly or byte code
- Tail recursion easily identified
- Strict forward recursion converted to tail recursion

Terms

- A function is in Direct Style when it returns its result back to the caller.
- A Tail Call occurs when a function returns the result of another function call without any more computations (eg tail recursion)
- A function is in Continuation Passing Style when it, and every function call in it, passes its result to another function.
- Instead of returning the result to the caller, we pass it forward to another function.

Example

- Simple reporting continuation:

```
# let report x = (print_int x; print_newline( ) );;
val report : int -> unit = <fun>
```

- Simple function using a continuation:

```
# let addk a b k = k (a + b);;
val addk : int -> int -> (int -> ’a) -> ’a = <fun>
# addk 22 20 report;;
2
- : unit = ()
```

Simple Functions Taking Continuations

- Given a primitive operation, can convert it to pass its result forward to a continuation
- Examples:

```
# let subk x y k = k(x + y);;
val timesk : int -> int -> (int -> 'a) -> 'a = <fun>
# let eqk x y k = k(x = y);;
val eqk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>
# let timesk x y k = k(x * y);;
val timesk : int -> int -> (int -> 'a) -> 'a = <fun>
```

Nesting Continuations

```
# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>
# let add_three x y z= let p = x + y in  p + z;;
val add_three : int -> int -> int -> int = <fun>
# let add_three_k x y z k =
   addk x y (fun p -> addk p z k );;
val add_three_k : int -> int -> int -> (int -> 'a)
   -> 'a = <fun>
```

Recursive Functions

- Recall:

```
# let rec factorial n =
    if n = 0 then 1 else n * factorial (n - 1);;
  val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
```

Recursive Functions

```
# let rec factorial n =
    let b = (n = 0) in (* First computation *)
    if b then 1 (* Returned value *)
    else let s = n – 1 in (* Second computation *)
         let r = factorial s in  (* Third computation *)
         n * r in (* Returned value *) ;;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
```

Recursive Functions

```
# let rec factorialk n k =
   eqk n 0
   (fun b ->  (* First computation *)
    if b then k 1 (* Passed value *)
    else subk n 1  (* Second computation *)
    (fun s -> factorialk s  (* Third computation *)
    (fun r -> timesk n r k))) (* Passed value *)
val factorialk : int -> int = <fun>
# factorialk 5 report;;
120
- : unit = ()
```

Recursive Functions

- To make recursive call, must build intermediate continuation to
  - take recursive value: r
  - build it to final result: n * r
  - And pass it to final continuation:
  - times n r k = k (n * r)

CPS for length

```
# let rec lengthk list k = match list with [ ] -> k 0
    | x :: xs -> lengthk xs (fun r -> k (r + 1));;
val lengthk : 'a list -> (int -> 'b) -> 'b = <fun>
# let rec lengthk list k = match list with [ ] -> k 0
    | x :: xs -> lengthk xs (fun r -> addk r 1 k);;
val lengthk : 'a list -> (int -> 'b) -> 'b = <fun>
# lengthk [2;4;6;8] report;;
4
- : unit = ()
```

Terminology

- Tail Position: A subexpression s of expressions e, such that if evaluated, will be taken as the value of e
  - if (x>3) then $\boxed{\text{x + 2}}$ else $\boxed{\text{x - 4}}$
  - let x = 5 in $\boxed{\text{x + 4}}$
- Tail Call: A function call that occurs in tail position
  - if (h x) then $\boxed{\text{f x}}$ else $\boxed{\text{(x + g x)}}$

Terminology

- Available: A function call that can be executed by the current expression
- The fastest way to be unavailable is to be guarded by an abstraction (anonymous function, lambda lifted).
  - if (h x) then f x else (x + g x)
  - if (h x) then (fun x -> f x) else (g (x + x))

Not available

CPS Transformation

- Step 1: Add continuation argument to any function definition:
  - let f arg = e $\Rightarrow$ let f arg k = e
  - Idea: Every function takes an extra parameter saying where the result goes
- Step 2: A simple expression in tail position should be passed to a continuation instead of returned:
  - return a $\Rightarrow$ k a
  - Assuming a is a constant or variable.
  - "Simple" = "No available function calls."

CPS Transformation

- Step 3: Pass the current continuation to every function call in tail position
  - return f arg $\Rightarrow$ f arg k
  - The function “isn’t going to return,” so we need to tell it where to put the result.

CPS Transformation

- Step 4: Each function call not in tail position needs to be converted to take a new continuation (containing the old continuation as appropriate)
  - return op (f arg) $\Rightarrow$ f arg (fun r -> k(op r))
  - op represents a primitive operation
  - return f(g arg) $\Rightarrow$ g arg (fun r-> f r k)

Example

Before:

```
let rec add_list lst =
match lst with
  [ ] -> 0
| 0 :: xs -> add_list xs
| x :: xs -> (+) x
   (add_list xs);;
```

After:

```
let rec add_listk lst k =
                        (* rule 1 *)
match lst with
| [ ] -> k 0 (* rule 2 *)
| 0 :: xs -> add_listk xs k
                           (* rule 3 *)
| x :: xs -> add_listk xs
           (fun r -> k ((+) x r));;
                     (* rule 4 *)
```

Other Uses for Continuations

- CPS designed to preserve order of evaluation
- Continuations used to express order of evaluation
- Can be used to change order of evaluation
- Implements:
  - Exceptions and exception handling
  - Co-routines
  - (pseudo, aka green) threads

Exceptions - Example

```
# exception Zero;;
exception Zero
# let rec list_mult_aux list =
    match list with [ ] -> 1
    | x :: xs ->
     if x = 0 then raise Zero
                 else x * list_mult_aux xs;;
val list_mult_aux : int list -> int = <fun>
```

Exceptions - Example

```
# let list_mult list =
    try list_mult_aux list with Zero -> 0;;
val list_mult : int list -> int = <fun>
# list_mult [3;4;2];;
- : int = 24
# list_mult [7;4;0];;
- : int = 0
# list_mult_aux [7;4;0];;
Exception: Zero.
```

Exceptions

- When an exception is raised
  - The current computation is aborted
  - Control is “thrown” back up the call stack until a matching handler is found
  - All the intermediate calls waiting for a return values are thrown away

Implementing Exceptions

```
# let multkp m n k =
   let r = m * n in
    (print_string "product result: ";
     print_int r; print_string "\n";
     k r);;
val multkp : int -> int -> (int -> 'a) -> 'a
  = <fun>
```

Implementing Exceptions

```
# let rec list_multk_aux list k kexcp =
    match list with [ ] -> k 1
    | x :: xs -> if x = 0 then  kexcp  0
      else list_multk_aux xs
             (fun r -> multkp x r k) kexcp;;
val list_multk_aux : int list -> (int -> 'a) -> (int -> 'a)
  -> 'a = <fun>
# let rec list_multk list k = list_multk_aux list  k  k;;
val list_multk : int list -> (int -> 'a) -> 'a = <fun>
```

Implementing Exceptions

```
# list_multk [3;4;2] report;;
product result: 2
product result: 8
product result: 24
24
- : unit = ()
# list_multk [7;4;0] report;;
0
- : unit = ()
```

Programming Languages and Compilers (CS 421)

Elsa L Gunter

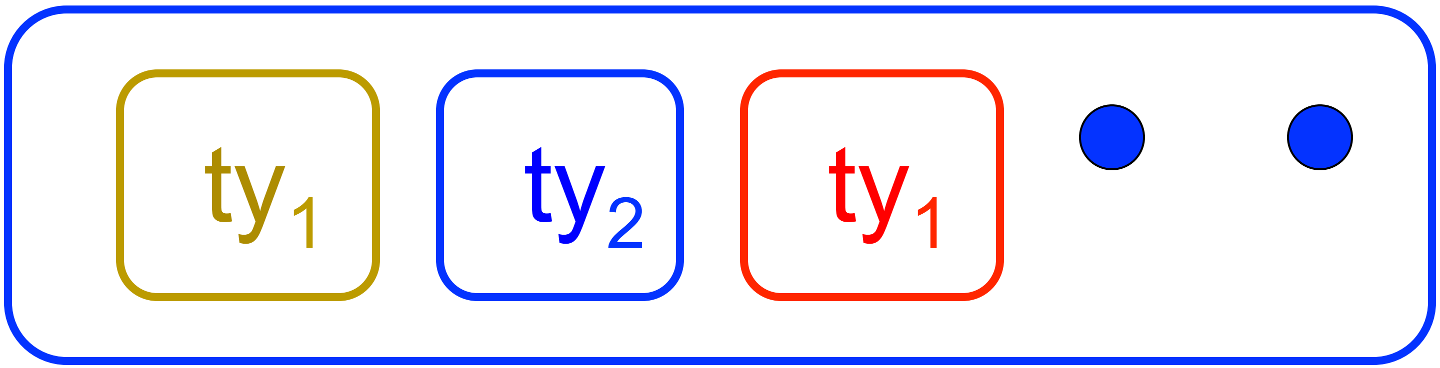
2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Disjoint Union Types

- Disjoint union of types, with some possibly occurring more than once



- We can also add in some new singleton elements

Disjoint Union Types

```
# type id = DriversLicense of int
   | SocialSecurity of int | Name of string;;
type id = DriversLicense of int | SocialSecurity
   of int | Name of string
# let check_id id = match id with
      DriversLicense num ->
       not (List.mem num [13570; 99999])
    | SocialSecurity num -> num < 900000000
    | Name str -> not (str = "John Doe");;
 val check_id : id -> bool = <fun>
```

Polymorphism in Variants

- The type 'a option is gives us something to represent non-existence or failure

```
# type 'a option = Some of 'a | None;;
type 'a option = Some of 'a | None
```

- Used to encode partial functions
- Often can replace the raising of an exception

Functions producing option

```
# let rec first p list =
    match list with [ ] -> None
    | (x::xs) -> if p x then Some x else first p xs;;
val first : ('a -> bool) -> 'a list -> 'a option = <fun>
# first (fun x -> x > 3) [1;3;4;2;5];;
- : int option = Some 4
# first (fun x -> x > 5) [1;3;4;2;5];;
- : int option = None
```

Functions over option

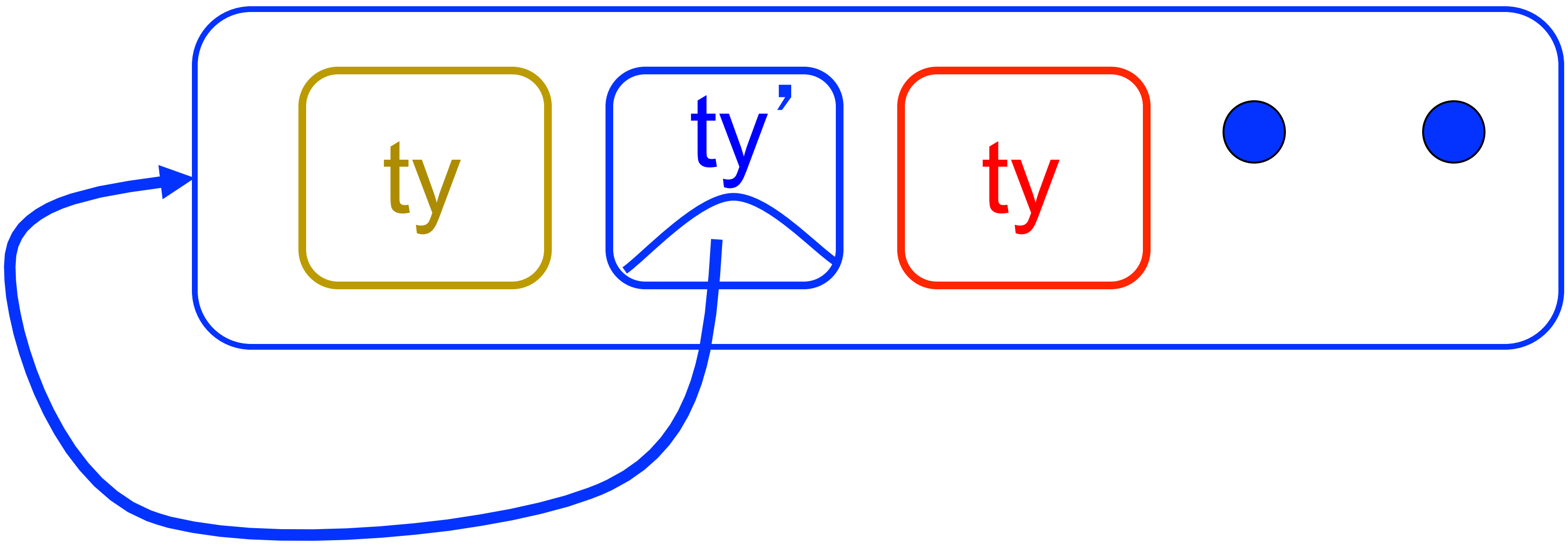
```
# let result_ok r =
   match r with None -> false
   | Some _ -> true;;
val result_ok : 'a option -> bool = <fun>
# result_ok (first (fun x -> x > 3) [1;3;4;2;5]);;
- : bool = true
# result_ok (first (fun x -> x > 5) [1;3;4;2;5]);;
- : bool = false
```

Folding over Variants

```
# let optionFold someFun noneVal opt =
    match opt with None -> noneVal
    | Some x -> someFun x;;
val optionFold : ('a -> 'b) -> 'b -> 'a option ->
    'b = <fun>
# let optionMap f opt =
    optionFold (fun x -> Some (f x)) None opt;;
val optionMap : ('a -> 'b) -> 'a option -> 'b
    option = <fun>
```

Recursive Types

- The type being defined may be a component of itself



Mapping over Variants

```
# let optionMap f opt =
    match opt with None -> None
    | Some x -> Some (f x);;
val optionMap : ('a -> 'b) -> 'a option -> 'b
    option = <fun>
# optionMap
  (fun x -> x - 2)
  (first (fun x -> x > 3) [1;3;4;2;5]);;
- : int option = Some 2
```

Recursive Data Types

```
# type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree *
   int_Bin_Tree);;
```

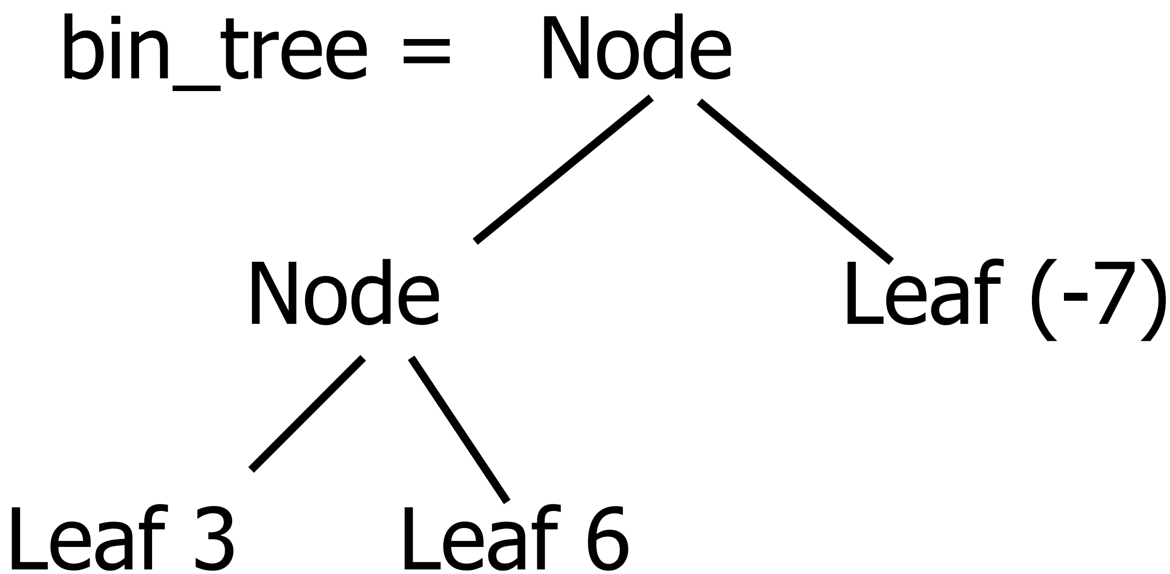
```
type int_Bin_Tree = Leaf of int | Node of
   (int_Bin_Tree * int_Bin_Tree)
```

Recursive Data Type Values

```
# let bin_tree =
 Node(Node(Leaf 3, Leaf 6),Leaf (-7));;

val bin_tree : int_Bin_Tree = Node (Node
   (Leaf 3, Leaf 6), Leaf (-7))
```

Recursive Data Type Values



Recursive Functions

```
# let rec first_leaf_value tree =
    match tree with (Leaf n) -> n
    | Node (left_tree, right_tree) ->
     first_leaf_value left_tree;;
val first_leaf_value : int_Bin_Tree -> int =
    <fun>
# let left = first_leaf_value bin_tree;;
val left : int = 3
```

Mapping over Recursive Types

```
# let rec ibtreeMap f tree =
    match tree with (Leaf n) -> Leaf (f n)
    | Node (left_tree, right_tree) ->
     Node (ibtreeMap f left_tree,
           ibtreeMap f right_tree);;
val ibtreeMap : (int -> int) -> int_Bin_Tree ->
    int_Bin_Tree = <fun>
```

Mapping over Recursive Types

```
# ibtreeMap ((+) 2) bin_tree;;

- : int_Bin_Tree = Node (Node (Leaf 5, Leaf
  8), Leaf (-5))
```

Folding over Recursive Types

```
# let rec ibtreeFoldRight leafFun nodeFun tree =
    match tree with Leaf n -> leafFun n
    | Node (left_tree, right_tree) ->
      nodeFun
      (ibtreeFoldRight leafFun nodeFun left_tree)
      (ibtreeFoldRight leafFun nodeFun right_tree);;
val ibtreeFoldRight : (int -> 'a) -> ('a -> 'a -> 'a) ->
    int_Bin_Tree -> 'a = <fun>
```

Folding over Recursive Types

```
# let tree_sum =
    ibtreeFoldRight (fun x -> x) (+);;
val tree_sum : int_Bin_Tree -> int = <fun>
# tree_sum bin_tree;;
- : int = 2
```

Mutually Recursive Types

```
# type 'a tree = TreeLeaf of 'a
    | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree
    | More of ('a tree * 'a treeList);;
```

type 'a tree = TreeLeaf of 'a | TreeNode of 'a treeList

and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList)

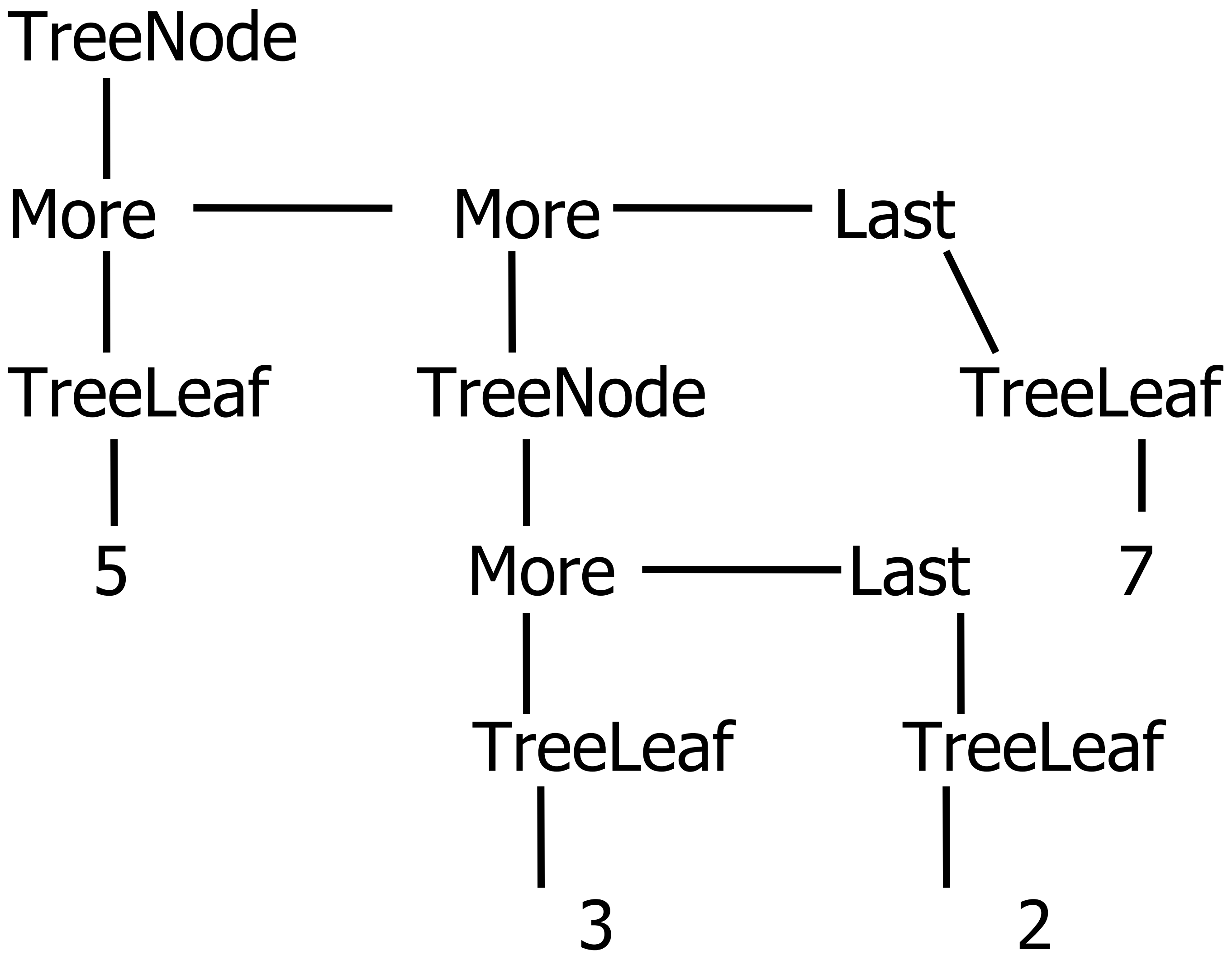
Mutually Recursive Types - Values

```
# let tree =
   TreeNode
    (More (TreeLeaf 5,
          (More (TreeNode
                (More (TreeLeaf 3,
                      Last (TreeLeaf 2))),
                Last (TreeLeaf 7)))));;
```

Mutually Recursive Types - Values

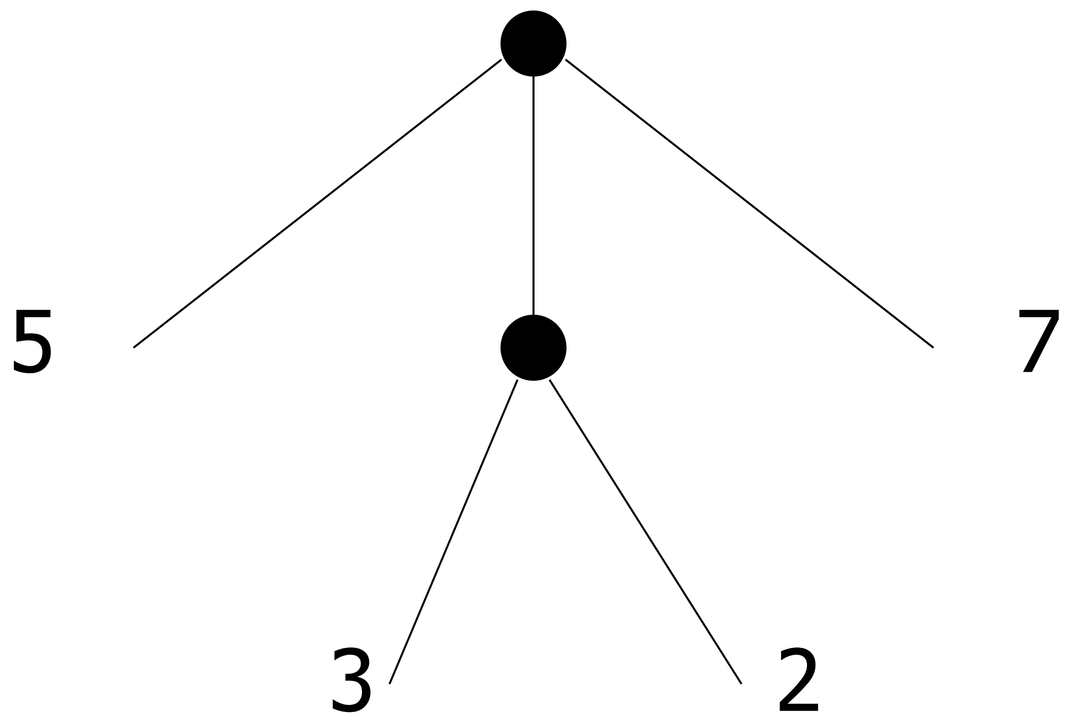
```
val tree : int tree =
 TreeNode
  (More
    (TreeLeaf 5,
     More
      (TreeNode (More (TreeLeaf 3, Last
   (TreeLeaf 2))), Last (TreeLeaf 7))))
```

Mutually Recursive Types - Values



Mutually Recursive Types - Values

A more conventional picture



Mutually Recursive Functions

```
# let rec fringe tree =
    match tree with (TreeLeaf x) -> [x]
  | (TreeNode list) -> list_fringe list
and list_fringe tree_list =
    match tree_list with (Last tree) -> fringe tree
  | (More (tree,list)) ->
    (fringe tree) @ (list_fringe list);;

val fringe : 'a tree -> 'a list = <fun>
val list_fringe : 'a treeList -> 'a list = <fun>
```

Mutually Recursive Functions

```
# fringe tree;;
- : int list = [5; 3; 2; 7]
```

Nested Recursive Types

```
# type 'a labeled_tree =
 TreeNode of ('a * 'a labeled_tree
   list);;
type 'a labeled_tree = TreeNode of ('a
   * 'a labeled_tree list)
```

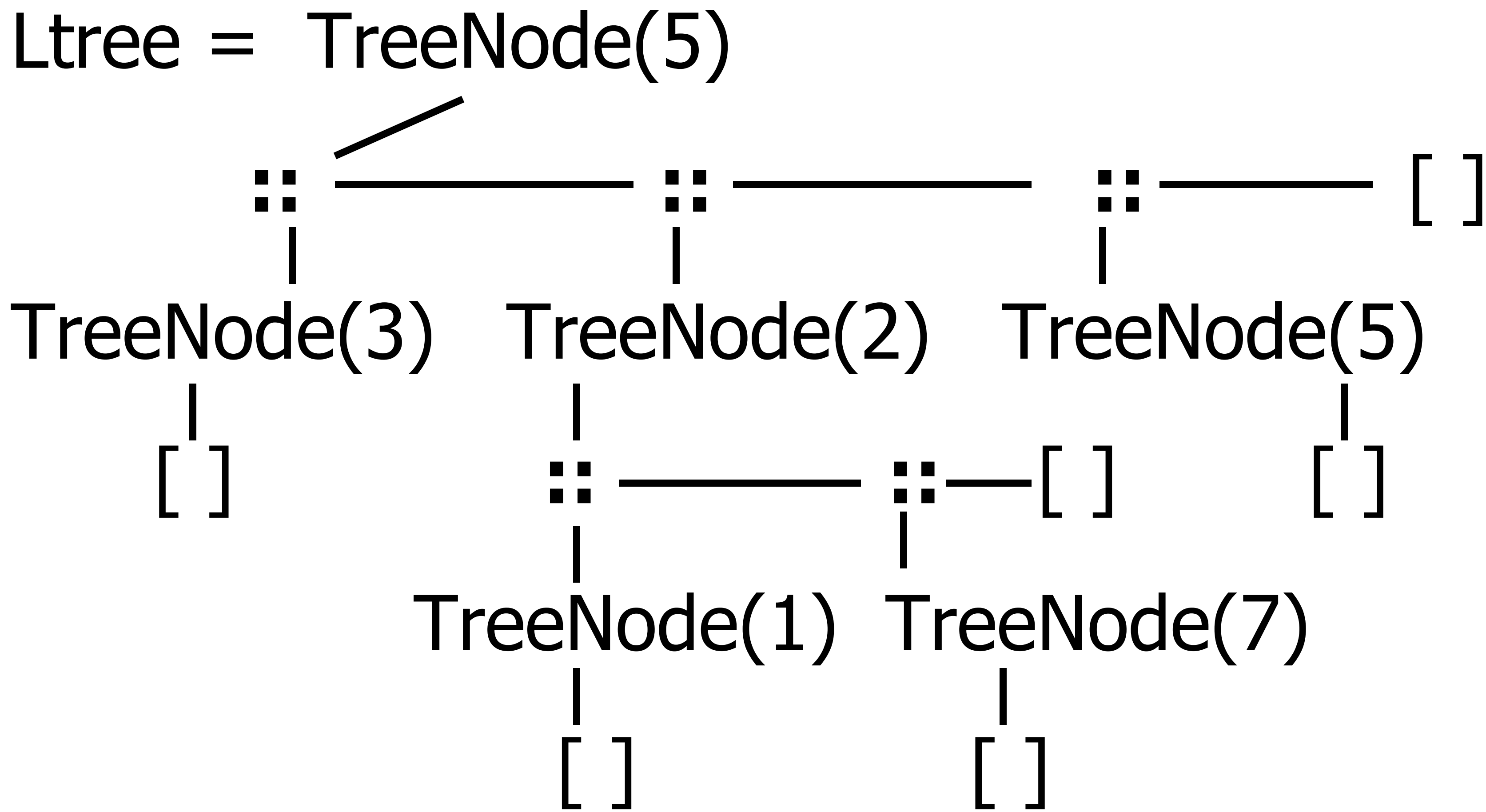
Nested Recursive Type Values

```
# let ltree =
  TreeNode(5,
    [TreeNode (3, []);
     TreeNode (2, [TreeNode (1, []);
                   TreeNode (7, [])]);
     TreeNode (5, [])]);;
```

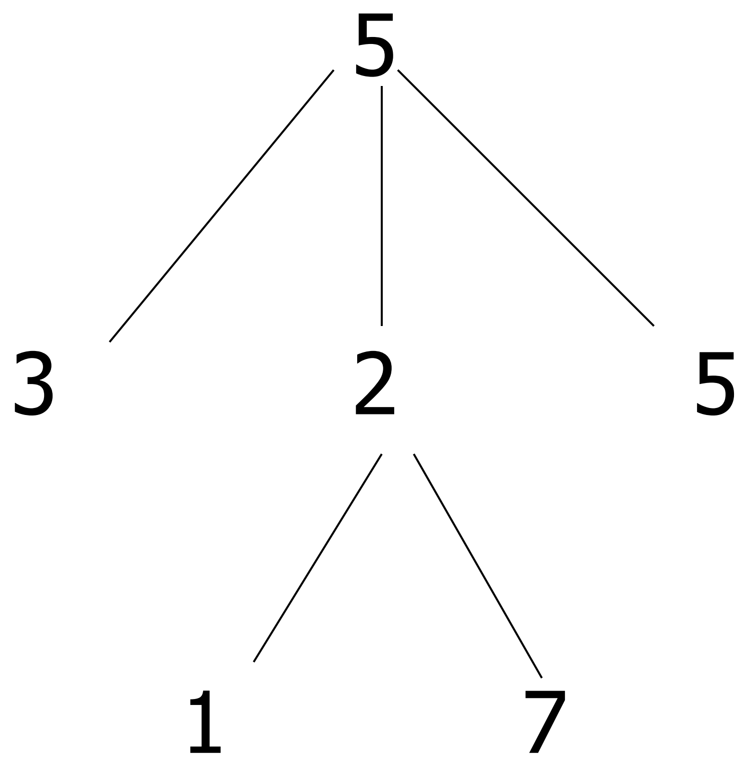
Nested Recursive Type Values

```
val ltree : int labeled_tree =
  TreeNode
   (5,
    [TreeNode (3, []); TreeNode (2,
   [TreeNode (1, []); TreeNode (7, [])]);
     TreeNode (5, [])])
```

Nested Recursive Type Values



Nested Recursive Type Values



Mutually Recursive Functions

```
# let rec flatten_tree labtree =
    match labtree with TreeNode (x,treelist)
      -> x::flatten_tree_list treelist
    and flatten_tree_list treelist =
    match treelist with [] -> []
    | labtree::labtrees
      -> flatten_tree labtree
        @ flatten_tree_list labtrees;;
```

Mutually Recursive Functions

val flatten_tree : 'a labeled_tree -> 'a list = <fun>

val flatten_tree_list : 'a labeled_tree list -> 'a list = <fun>

flatten_tree ltree;;

- : int list = [5; 3; 2; 1; 7; 5]

- Nested recursive types lead to mutually recursive functions

Infinite Recursive Values

```
# let rec ones = 1::ones;;
val ones : int list =
  [1; 1; 1; 1; ...]
# match ones with x::_ -> x;;
Characters 0-25:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
  match ones with x::_ -> x;;
  ^^^^^^^^^^^^^^^^^^^^^^^^^
- : int = 1
```

Infinite Recursive Values

```
# let rec lab_tree = TreeNode(2, tree_list)
    and tree_list = [lab_tree; lab_tree];;
val lab_tree : int labeled_tree =
  TreeNode (2, [TreeNode(...); TreeNode(...)])
val tree_list : int labeled_tree list =
  [TreeNode (2, [TreeNode(...);
    TreeNode(...)]);
   TreeNode (2, [TreeNode(...);
    TreeNode(...)])]
```

Infinite Recursive Values

```
# match lab_tree
   with TreeNode (x, _) -> x;;
- : int = 2
```

Records

- Records serve the same programming purpose as tuples
- Provide better documentation, more readable code
- Allow components to be accessed by label instead of position
  - Labels (aka *field names* must be unique)
  - Fields accessed by suffix dot notation

Record Types

- Record types must be declared before they can be used in OCaml

```
# type person = {name : string; ss : (int * int * int); age : int};;
type person = { name : string; ss : int * int * int; age : int; }
```

- person is the type being introduced
- name, ss and age are the labels, or fields

Record Values

- Records built with labels; order does not matter

```
# let teacher = {name = "Elsa L. Gunter";
   age = 102; ss = (119,73,6244)};;
val teacher : person =
  {name = "Elsa L. Gunter"; ss = (119, 73,
   6244); age = 102}
```

Record Pattern Matching

```
# let {name = elsa; age = age; ss =
  (_,_,s3)} = teacher;;
val elsa : string = "Elsa L. Gunter"
val age : int = 102
val s3 : int = 6244
```

Record Field Access

```
# let soc_sec = teacher.ss;;
val soc_sec : int * int * int = (119,
  73, 6244)
```

Record Values

```
# let student = {ss=(325,40,1276);
   name="Joseph Martins"; age=22};;
val student : person =
  {name = "Joseph Martins"; ss = (325, 40,
   1276); age = 22}
# student = teacher;;
- : bool = false
```

New Records from Old

```
# let birthday person = {person with age =
    person.age + 1};;
val birthday : person -> person = <fun>
# birthday teacher;;
- : person = {name = "Elsa L. Gunter"; ss =
    (119, 73, 6244); age = 103}
```

New Records from Old

```
# let new_id name soc_sec person =
 {person with name = name; ss = soc_sec};;
val new_id : string -> int * int * int -> person
   -> person = <fun>
# new_id "Guieseppe Martin" (523,04,6712)
   student;;
- : person = {name = "Guieseppe Martin"; ss
   = (523, 4, 6712); age = 22}
```

A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention*

John Wilander and Mariam Kamkar
Dept. of Computer and Information Science, Linköpings universitet
{johwi, marka}@ida.liu.se

Abstract

The size and complexity of software systems is growing, increasing the number of bugs. Many of these bugs constitute security vulnerabilities. Most common of these bugs is the buffer overflow vulnerability. In this paper we implement a testbed of 20 different buffer overflow attacks, and use it to compare four publicly available tools for dynamic intrusion prevention aiming to stop buffer overflows. The tools are compared empirically and theoretically. The best tool is effective against only 50% of the attacks and there are six attack forms which none of the tools can handle.

Keywords: security intrusion; buffer overflow; intrusion prevention; dynamic analysis

1 Introduction

The size and complexity of software systems is growing, increasing the number of bugs. According to statistics from Coordination Center at Carnegie Mellon University, CERT, the number of reported vulnerabilities in software has increased with nearly 500% in two years [5] as shown in figure 1.

Now there is good news and bad news. The good news is that there is lots of information available on how these security vulnerabilities occur, how the attacks against them work, and most importantly how they can be avoided. The bad news is that this information apparently does not lead to fewer vulnerabilities. The same mistakes are made over and over again which, for instance, is shown in the statistics for the infamous *buffer overflow* vulnerability. David Wagner et al from University of California at Berkeley show that buffer overflows alone stand for about 50% of the vulnerabilities reported by CERT [35].

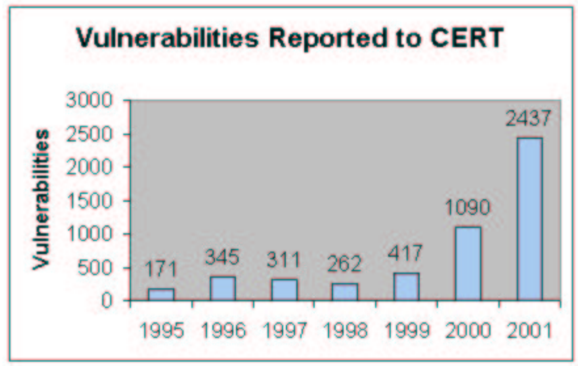


Figure 1. Software vulnerabilities reported to CERT 1995–2001.

In the middle of January 2002 the discussion about responsibility for security intrusions took a new turn. The US National Academies released a prepublication recommending that policy-makers create laws that would hold companies accountable for security breaches resulting from vulnerable products [30], which received global media attention [3, 28]. So far, only the intruder can be charged in court. In the future, software companies may be charged for not preventing intrusions. This stresses the importance of helping software engineers to produce more secure software. Automated development and testing tools aimed for security could be one of the solutions for this growing problem.

One starting point would, or could be tools that can be applied directly to the source code and solve or warn about security vulnerabilities. This means trying to solve the problems in the implementation and testing phase. Applying security related methodologies throughout the whole development cycle would most likely be more effective, but given the amount of existing software ("legacy code"), the desire for modular design using software components programmed earlier, and the time it would take to educate software engineers in secure analysis and design, we argue that security tools that aim to clean up vulnerable

*This work has been supported by the national computer graduate school in computer science (CUGS), commissioned by the Swedish government and the board of education.

source code are necessary. A further discussion of this issue can be found in the January/February 2002 issue of IEEE Software [18].

In this paper we investigate the effectiveness of four publicly available tools for dynamic prevention of buffer overflow attacks–namely the GCC compiler patches StackGuard, Stack Shield, and ProPolice, and the security library Libsafe/Libverify. Our approach has been to first develop an in-depth understanding of how buffer overflow attacks work and from this knowledge build a testbed with all the identified attack forms. Then the four tools are compared theoretically and empirically with the testbed. This work is a follow-up of John Wilander's Master's Thesis "Security Intrusions and Intrusion Prevention" [36].

1.1 Scope

We have tested publicly available tools for run-time prevention of buffer overflow attacks. The tools all apply to C source code, but using them requires no modifications of the source code. We do not consider approaches that use system specific features, modified kernels, or require the user to install separate run-time security components. The twenty buffer overflows represent a sample of the potential instances of buffer overflow attacks and not on the likelihood of a specific attack using the sample instance.

1.2 Paper Overview

The rest of the paper is organized as follows. Section 2 describes process memory management in UNIX and how buffer overflow attacks work. Section 3 presents the concept of intrusion prevention and describes the techniques used in the four analyzed tools. Section 4 defines our testbed of twenty attack forms and presents our theoretical and empirical comparison of the tools' effectiveness against the previously described attack forms. Section 5 describes the common shortcomings of current dynamic intrusion prevention. Finally sections 6 and 7 present related work and our conclusions.

2 Attack Methods

The analysis of intrusions in this paper concerns a subset of all violations of security policies that would constitute a security intrusion according to definitions in, for example, the Internet Security Glossary [31]. In our context an intrusion or a successful attack aims to *change the flow of control*, letting the attacker execute arbitrary code. We consider this class of vulnerabilities the worst possible since "arbitrary code" often means starting a new *shell*. This shell will have the same access rights to the system as the process attacked. If the process had *root access*, so will the attacker in the new shell, leaving the whole system open for any kind of manipulation.

2.1 Changing the Flow of Control

Changing the flow of control and executing arbitrary code involves two steps for an attacker:

1. Injecting *attack code* or *attack parameters* into some memory structure (e.g. a buffer) of the vulnerable process.

2. Abusing some vulnerable function that writes to memory of the process to alter data that controls execution flow.

Attack code could mean assembly code for starting a shell (less than 100 bytes of space will do) whereas attack parameters are used as input to code already existing in the vulnerable process, for example using the parameter `"/bin/sh"` as input to the `system()` library function would start a shell.

Our biggest concern is step two—redirecting control flow by writing to memory. That is the hard part and the possibility of changing the flow of control in this way is the most unlikely condition of the two to hold. The possibility of injecting attack code or attack parameters is higher since it does not necessarily have to violate any rules or restrictions of the program.

Changing the flow of control occurs by altering a *code pointer*. A code pointer is basically a value which gives the *program counter* a new memory address to start executing code at. If a code pointer can be made to point to attack code the program is vulnerable. The most popular target is the return address on the stack. But programmer defined *function pointers* and so called *longjmp buffers* are equally effective targets of attack.

2.2 Memory Layout in UNIX

To get a picture of the memory layout of processes in UNIX we can look at two simplified models (for a complete description see "Memory Layout in Program Execution" by Frederick Giasson [19]). Each process has a (partial) memory layout as in the figure below:

High address	Stack ↓ ↑ Heap
	BSS segment
	Data segment
Low address	Text segment

Figure 2. Memory layout of a UNIX process.

The machine code is stored in the text segment and constants, arguments, and variables defined by the programmer are stored in the other memory areas. A small C-program shows this (the comments show where each piece of data is stored in process memory):

```
static int GLOBAL_CONST = 1;      // Data segment
static int global_var;            // BSS segment

// argc & argv on stack, local
int main(argc **argv[]) {
  int local_dynamic_var;          // Stack
  static int local_static_var;    // BSS segment
  int *buf_ptr=(int *)malloc(32); // Heap
  ... }
```

For each function call a new *stack frame* is set up on top of the stack. It contains the return address, the calling function's base pointer, locally declared variables, and more. When the function ends, the return address instructs the processor where to continue executing code and the stored base pointer gives the offset for the stack frame to use.

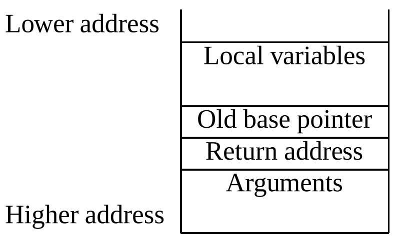


Figure 3. The UNIX stack frame.

2.3 Attack Targets

As stated above the target for a successful change of control flow is a code pointer. There are three types of code pointers to attack [11]. But Hiroaki Etoh and Kunikazu Yoda propose using the old base pointer as an attack target [15]. We have implemented their proposed attack form and proven that the old base pointer is just as dangerous a target as the return address (see section 2.4 and 4). So we have four attack targets:

1. The return address, allocated on the stack.
2. The old base pointer, allocated on the stack.
3. Function pointers, allocated on the heap, in the BSS or data segment, or on the stack either as a local variable or as a parameter.
4. Longjmp buffers, allocated on the heap, in the BSS or data segment, or on the stack either as a local variable or as a parameter.

A function pointer in C is declared as `int (*func_ptr) (char)`, in this example a pointer to a function taking a `char` as input and returns an `int`. It points to executable code.

Longjmp in C allows the programmer to explicitly jump back to functions, not going through the chain of return addresses. Let's say function A first calls `setjmp()`, then calls function B which in turn calls function C. If C now calls `longjmp()` the control is directly transferred back to function A, popping both C's and B's stack frames of the stack.

2.4 Buffer Overflow Attacks

Buffer overflow attacks are the most common security intrusion attack [35, 16] and have been extensively analyzed and described in several papers and on-line documents [29, 24, 7, 14]. Buffers, wherever they are allocated in memory, may be overflown with too much data if there is no check to ensure that the data being written into the buffer actually fits there. When too much data is written into a buffer the extra data will "spill over" into the adjacent memory structure, effectively overwriting anything that was stored there before. This can be abused to overwrite a code pointer and change the flow of control either by directly overflowing the code pointer or by first overflowing another pointer and redirect that pointer to the code pointer.

The most common buffer overflow attack is shown in the simplified example below. A local buffer allocated on the stack is overflown with 'A's and eventually the return address is overwritten, in this case with the address `0xbffff740`.

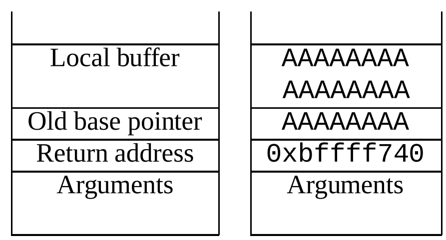


Figure 4. A buffer overflow overwriting the return address.

If an attacker can supply the input to the buffer he or she can design the data to redirect the return address to his or her attack code.

The second attack target, the old base pointer, can be abused by building a fake stack frame with a return address pointing to attack code and then overflow the buffer to overwrite the old base pointer with the address of this fake stack frame. Upon return, control will be passed to the fake stack frame which immediately returns again redirecting flow of control to the attack code.

The third attack target is function pointers. If the function pointer is redirected to the attack code the attack will be executed when the function pointer is used.

The fourth and last attack target is longjmp buffers. They contain the environment data required to resume execution from the point `setjmp()` was called. This environment data includes a base pointer and a program counter. If the program counter is redirected to attack code the attack will be executed when `longjmp()` is called.

Combining all these buffer overflow techniques, locations in memory and attack targets leaves us with no less than twenty attack forms. They are all listed in section 4 and constitute our testbed for testing of the intrusion prevention tools.

3 Intrusion Prevention

There are several ways of trying to prohibit intrusions. Halme and Bauer present a taxonomy of *anti-intrusion techniques* called *AINT* [20] where they define:

Intrusion prevention. Precludes or severely handicaps the likelihood of a particular intrusion's success.

We divide intrusion prevention into *static intrusion prevention* and *dynamic intrusion prevention*. In this section we will first describe the differences between these two categories. Secondly, we describe four publicly available tools for dynamic intrusion prevention, describe shortly how they work, and in the end compare their effectiveness against the intrusions and vulnerabilities described in section 2.4. This is not a complete survey of intrusion prevention techniques, rather a subset with the following constraints:

- Techniques used in the implementation phase of the software.
- Techniques that require no altering of source code to disarm security vulnerabilities.
- Techniques that are generic, implemented and publicly available, not prototypes or system specific tools.

Our motivation for this is to evaluate and compare tools that could easily and quickly be introduced to software developers and increase software quality from a security point of view.

3.1 Static Intrusion Prevention

Static intrusion prevention tries to prevent attacks by finding the security bugs in the source code so that the programmer can remove them. Removing all security bugs from a program is considered infeasible [23] which makes the static solution incomplete. Nevertheless, removing bugs known to be exploitable brings down the likelihood of successful attacks against all possible targets. Static intrusion prevention removes the attacker's method of entry, the security bugs. The two main drawbacks of this approach is that someone has to keep an updated database of programming flaws to test for, and since the tools only *detect* vulnerabilities the user has to know how to fix the problem once a warning has been issued.

3.2 Dynamic Intrusion Prevention

The dynamic or *run-time* intrusion prevention approach is to change the run-time environment or system functionality making vulnerable programs harmless, or at least less vulnerable. This means that in an ordinary environment the program would still be vulnerable (the security bugs are still there) but in the new, more secure environment those same vulnerabilities cannot be exploited in the same way—it protects *known* targets from attacks.

Dynamic intrusion prevention, as we will see, often ends up becoming an intrusion detection system building on program and/or environment specific solutions, terminating execution in case of an attack. The techniques are often complete in the way that they can provably secure the targets they are designed to protect (one proof can be found in a paper by Chiueh and Hsu [6]) and will produce no false positives. Their general weakness lies in the fact that they all try to solve *known* security problems, i.e. how bugs are known to be exploited today, while not getting rid of the actual bugs in the programs. Whenever an attacker has figured out a new way of exploiting a bug, these dynamic solutions often stand defenseless. On the other hand they will be effective against exploitation of any new bugs using the same attack method.

3.3 StackGuard

The *StackGuard* compiler invented and implemented by Crispin Cowan et al [10] is perhaps the most well referenced of the current dynamic intrusion prevention techniques. It is designed for detecting and stopping stack-based buffer overflows targeting the return address.

3.3.1 The StackGuard Concept

The key idea behind StackGuard is that buffer overflow attacks overwrite everything on their way towards their target. In the case of a buffer overflow on the stack targeting the return address, the attacker has to fill the buffer, then overwrite any other local variables below (i.e. on higher stack addresses), then overwrite the old base pointer until it finally reaches the return address. If we place a dummy value in between the return address and the stack data above, and then check whether this value has been overwritten or not before we allow the return address to be used, we could detect this kind of attack and possibly prevent it. The inventors have chosen to call this dummy value the *canary*.

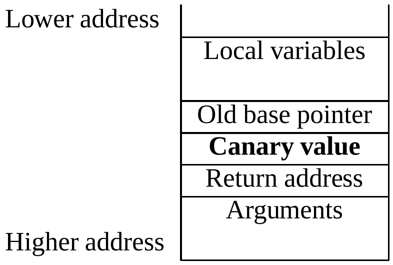


Figure 5. The StackGuard stack frame.

A potentially successful attack against such a system would be to somehow leave the canary intact while changing the return address, either by overwriting the canary with its correct value and thus not changing it, or by overwriting the return address through a pointer, not touching the canary. To solve the first problem, two canary versions have been suggested—firstly the *random canary* which consists of a random 32-bit value calculated at run-time, and secondly the *terminator canary* which consists of all four kinds of string termination sequences, namely `Null`, `Carriage Return`, `-1` and `Line Feed`. In the random canary case the attacker has to guess, or somehow retrieve, the random value at run-time. In the terminator canary case the attacker has to input all the termination sequences to keep the canary intact during the overflow. This is not possible since the string function receiving the input will terminate on one of the sequences.

Note that these techniques only stop overflow attacks that overwrite everything along the stack, not general attacks against the return address. The attacker can still abuse a pointer, making it point at the return address and writing a new address to that memory position. This shortcoming of StackGuard was discovered by Mariusz Woloszyn, alias "Emsi" and presented by Bulba and Kil3er [4]. The StackGuard team has addressed this problem by not only saving the canary value but the `XOR` of the canary and the correct return address. In this way an abused return address with an intact canary preceding it would still be detected since the `XOR` of the canary and the return address has changed. If the `XOR` scheme is used the canary has to be random since the terminator canary `XOR`ed with an address would not terminate strings anymore.

3.3.2 Random Canaries Unsupported

While testing StackGuard we noticed that the compiler did not respond to the flag set for random canary. We e-mailed Crispin Cowan and according to him: "There is only one threat that the XOR canary defeats, and the terminator canary does not: Emsi's attack. However, if you have a vulnerability that enables you to deploy Emsi's attack, then you have many other targets to attack besides function return address values. Therefore, we dropped support for random canaries [8]". We agree that the return address is not the only attack target but it is the most popular and unlike function pointers and longjmp buffers, the return address is always present. According to Cowan's e-mail and a WireX paper a better solution is on its way called *PointGuard* which will protect the integrity of pointers in general with the same kind of canary solution [11]. This implies that PointGuard will protect against all attack forms overflowing pointers (See attack forms 3a–f and 4a–f in section 4).

StackGuard is available for download at `http://www.immunix.org/`

3.4 Stack Shield

Stack Shield is a compiler patch for GCC made by Vendicator [33]. In the current version 0.7 it implements three types of protection, two against overwriting of the return address (both can be used at the same time) and one against overwriting of function pointers.

3.4.1 Global Ret Stack

The *Global Ret Stack* protection of the return address is the default choice for Stack Shield. It is a separate stack for storing the return addresses of functions called during execution. The stack is a global array of 32-bit entries. Whenever a function call is made, the return address being pushed onto the normal stack is at the same time copied into the Global Ret Stack array. When the function returns, the return address on the normal stack is replaced by the copy on the Global Ret Stack. If an attacker had overwritten the return address in one way or another the attack would be stopped without terminating the process execution. Note that no comparison is made between the return address on the stack and the copy on the Global Ret Stack. This means only prevention and no detection of an attack. The Global Ret Stack has by default 256 entries which limits the nesting depth to 256 protected function calls. Further function calls will be unprotected but execute normally.

3.4.2 Ret Range Check

A somewhat simpler but faster version of Stack Shield's protection of return addresses is the *Ret Range Check*. It uses a global variable to store the return address of the current function. Before returning, the return address on the stack is compared with the stored copy in the global variable. If there is a difference the execution is halted. Note that the Ret Range Check can detect an attack as opposed to the Global Ret Stack described above.

3.4.3 Protection of Function Pointers

Stack Shield also aims to protect function pointers from being overwritten. The idea is that function pointers normally should point into the text segment of the process' memory. That's where the programmer is likely to have implemented the functions to point at. If the process can ensure that no function pointer is allowed to point into other parts of memory than the text segment, it will be impossible for an attacker to make it point at code injected into the process, since injection of data only can be done into the data segment, the BSS segment, the heap, or the stack.

Stack Shield adds checking code before all function calls that make use of function pointers. A global variable is then declared in the data segment and its address is used as a boundary value. The checking function ensures that any function pointer about to be dereferenced points to memory below the address of the global boundary variable. If it points above the boundary the process is terminated. This protection will give false positives if the programmer has intended to use dynamically allocated function pointers.

Stack Shield is available for download at `http://www.angelfire.com/sk/stackshield/`

3.5 ProPolice

Hiroaki Etoh and Kunikazu Yoda from IBM Research in Tokyo have implemented the perhaps most sophisticated compiler protection called *ProPolice* [15].

3.5.1 The ProPolice Concept

Etoh's and Yoda's GCC patch ProPolice borrows the main idea from StackGuard (see section 3.3)—they use canary values to detect attacks on the stack. The novelty is the protection of stack allocated variables by rearranging the local variables so that `char` buffers always are allocated at the bottom, next to the old base pointer, where they cannot be overflown to harm any other local variables.

3.5.2 Building a Safe Stack Frame

After a program has been compiled with ProPolice the stack frame of functions look like that shown in figure 6.

No matter in what order local variables, pointers, and buffers are declared by the programmer, they are rearranged in stack memory to reflect the structure shown above. In this way we know that local `char` buffers can only be overflown to harm each other, the old base pointer and below. No variables can be attacked unless they are part of a `char` buffer. And by placing the canary which they call the *guard* between these buffers and the old base pointer all attacks outside the `char` buffer segment will

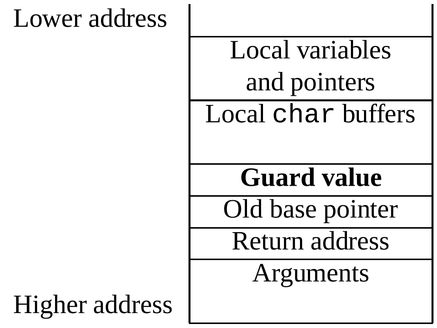


Figure 6. The ProPolice stack frame.

be detected. When an attack is detected the process is terminated.

When testing ProPolice we noticed some irregularities in when and was not the buffer overflow protection was included. It seems like small char buffers (e.g. 5 bytes) confuse ProPolice, causing it to skip the protection even if the user has set the protector flag. This gives the overall impression maybe that ProPolice is somewhat unstable.

ProPolice is available for download at `http://www.trl.ibm.com/projects/security/ssp/`

3.6 Libsafe and Libverify

Another defense against buffer overflows presented by Arash Baratloo et al [1] is *Libsafe*. This tool actually provides a combination of static and dynamic intrusion prevention. Statically it patches library functions in C that constitute potential buffer overflow vulnerabilities. A range check is made before the actual function call which ensures that the return address and the base pointer cannot be overwritten. Further protection has been provided [2] with *Libverify* using a similar dynamic approach to StackGuard (see Section 3.3).

3.6.1 Libsafe

The key idea behind Libsafe is to estimate a safe boundary for buffers on the stack at run-time and then check this boundary before any vulnerable function is allowed to write to the buffer. Vulnerable functions they consider to be the ones in table 1 below.

As a boundary value Libsafe uses the old base pointer pushed onto the stack after the return address. No local variable should be allowed to expand further down the stack than the beginning of the old base pointer. In this way a stack-based buffer overflow cannot overwrite the return address.

Function	Vulnerability
`strcpy(char *dest, const char *src)`	May overflow `dest`
`strcat(char *dest, const char *src)`	May overflow `dest`
`getwd(char *buf)`	May overflow `buf`
`gets(char *s)`	May overflow `s`
`[vf]scanf(const char *format, ...)`	May overflow arguments
`realpath(char *path, char resolved_path[])`	May overflow `path`
`[v]sprintf(char *str, const char *format, ...)`	May overflow `str`

Table 1. Vulnerable C functions that Libsafe adds protection to.

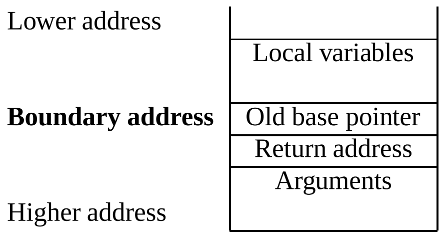


Figure 7. The Libsafe stack frame.

This boundary is enforced by overloading the functions in table 1 with wrapping functions. These wrappers first compute the length of the input as well as the allowed buffer size (i.e. from the buffer's starting point to the old base pointer) and then performs a boundary check. If the input is within the boundary the original functionality is carried out. If not the wrapper writes an alert to the system's log file and then halts the program. Observe that overflows within the local variables on the stack, such as function pointers, are not stopped.

3.6.2 Libverify

Libverify is an enhancement of Libsafe, implementing return address verification similar to StackGuard. But since this is a library it does not require recompilation of the software. As with Libsafe the library is pre-loaded and linked to any program running on the system.

The key idea behind Libverify is to alter all functions in a process so that the first thing done in every function is to copy the return address onto a *canary stack* located on the heap, and the last thing done before returning is to verify the return address by comparing it with the address saved on the canary stack. If the return address is still correct the process is allowed to continue executing. But if the return address does not match the saved copy, execution is halted and a security alert is raised. Libverify does not protect the integrity of the canary stack. They propose protecting it with `mprotect()` as in RAD (see section 3.7) but as in the RAD case this will most probably impose a very serious performance penalty [6].

To be able to do this, Libverify has to rearrange the code quite a bit. First each function is copied whole to the heap (requires executable heap) where it can be altered. Then the saving and verifying of the return address is injected into each function by overwriting the first instruction with a call to `wrapper_entry` and all return instructions with a call to `wrapper_exit`. The need for copying the code to the heap is due to the Intel CPU architecture. On other platforms this could be solved without copying the code [2].

Libverify is needed to give a more complete protection of the return address since Libsafe only addresses standard C library functions (as pointed out by Istvan Simon [32]). With Libsafe vulnerabilities could still occur where the programmer has implemented his/her own memory handling.

Libsafe and Libverify are available for download at `http://www.research.avayalabs.com/project/libsafe/`

3.7 Other Dynamic Solutions

The dynamic intrusion prevention techniques presented above are not the only ones. Other researchers have had similar ideas and implemented alternatives.

Tzi-cker Chiueh and Fu-Hau Hsu from State University of New York at Stony Brook have presented a compiler patch for protection of the return address [6]. They call their GCC patch *Return Address Defender*, or *RAD* for short. The key idea behind RAD is quite similar to the return address protection of Stack Shield described in Section 3.4. Every time a function call is made and a new stack frame is created, RAD stores a copy of the new return address. When a function returns, the return address about to be dereferenced is first checked against its copy. RAD is not publicly available.

The GCC patch *StackGhost* [25] by Mike Frantzen and Mike Shuey makes use of system specific features of the Sun Sparc Station to implement a sophisticated protection of the return address. They propose both XORing a random value with the return address (as StackGuard) as well as keeping a separate return address stack (as Stack Shield, RAD and Libverify). They also suggest using cryptographic methods instead of XOR to enhance security.

CCured and Cyclone are two recent research projects aiming to significantly enhance type and bounds checking in C. They both use a combination of static analysis and run-time checks.

CCured [27, 26] is an extension of the C programming language that distinguishes between various kinds of pointers depending on their usage. The purpose of this distinction is to be able to prevent improper usage of pointers and thus to guarantee that programs do not access memory areas they shouldn't access. CCured will change C programs slightly so that they are type safe. CCured does not change code that does not use pointers or arrays.

Cyclone [21] is a C dialect that prevents safety violations such as buffer overflows, dangling pointers, and format string attacks by ruling out certain parts of ANSI C and replacing them with safer versions. For instance `setjmp()` and `longjmp()` are unsupported (in some cases exceptions are used instead). Also pointer arithmetic is restricted. An average of 10% of the lines of code have to be changed when porting programs from C to Cyclone.

Richard Jones and Paul Kelly 1997 presented a GCC compiler patch in which they implemented run-time bounds checking of variables [22]. For each declared storage pointer they keep an entry in a table where the base and limit of the storage is kept. Before any pointer arithmetic or pointer dereferencing is made, the base and limit is checked in the table. While not explicitly aimed for security, this technique would effectively stop all kinds of buffer overflow attacks. Sadly their solution suffered both from performance penalties of more than 400 %, as well as incompatibilities with real-world programs (according to Crispin Cowan et al [9]). Because of the bad performance and compatibility we considered Jones' and Kelly's solution less interesting for software development and excluded it from our test.

It is also possible to have support for dynamic intrusion prevention in the operating system. A popular idea is the non-executable stack. This would make injection of attack code into the stack useless. But there are many ways around this protection. A few examples include using code already linked into the program from libraries (for instance calling `system()` with the parameter `"/bin/sh"`), injecting the attack code into other memory structures such as environment variables, or by exploiting buffer overflows on the heap or in the BSS/data segment. The Linux kernel patch from the Openwall Project is publicly available and implements a non-executable stack as well as protection against attacks using library functions [13]. Since it is a kernel patch it is up to the user and not the producer of software to install it. Therefore we did not include it in our test.

David Wagner and Drew Dean have presented an interesting approach for intrusion detection that relates to the functionality of the tools described in this paper [34]. They model the program's correct execution behavior via static analysis of the source code, building up callgraphs or even equivalent context-free languages defining the set of possible system call traces. Then these models are used for run-time monitoring of execution. Any deviation from the defined 'good' behavior will make the model enter an unaccepting state and trigger the intrusion alarm. As the metric for precision in intrusion detection they propose the branching factor of the model. A low branching factor means that the attacker has few choices of what to do next if he or she wants to evade detection.

4 Comparison of the Tools

Here we define our testbed of twenty buffer overflow attack forms and then present the outcome of our empirical and theoretical comparison of the tools from section 3.2.

We define an attack form as a combination of a technique, a location, and an attack target. As described in section 2.3 we have identified two techniques, two types of location and four attack targets:

Techniques. Either we overflow the buffer all the way to the attack target or we overflow the buffer to redirect a pointer to the target.

Locations. The types of location for the buffer overflow are the stack or the heap/BSS/data segment.

Attack Targets. We have four targets—the return address, the old base pointer, function pointers, and longjmp buffers. The last two can be either variables or function parameters.

Considering all practically possible combinations gives us the twenty attack forms listed below.

1. Buffer overflow on the stack all the way to the target:
   (a) Return address
   (b) Old base pointer
   (c) Function pointer as local variable
   (d) Function pointer as parameter
   (e) Longjmp buffer as local variable
   (f) Longjmp buffer as function parameter
2. Buffer overflow on the heap/BSS/data all the way to the target:
   (a) Function pointer
   (b) Longjmp buffer

Development Tool	Attacks prevented	Attacks halted	Attacks missed	Abnormal behavior
StackGuard Terminator Canary	0 (0%)	3 (15%)	16 (80%)	1 (5%)
Stack Shield Global Ret Stack	5 (25%)	0 (0%)	14 (70%)	1 (5%)
Stack Shield Range Ret Check	0 (0%)	0 (0%)	17 (85%)	3 (15%)
Stack Shield Global & Range	6 (30%)	0 (0%)	14 (70%)	0 (0%)
ProPolice	8 (40%)	2 (10%)	9 (45%)	1 (5%)
Libsafe and Libverify	0 (0%)	4 (20%)	15 (75%)	1 (5%)

Table 2. Empirical test of dynamic intrusion prevention tools. 20 attack forms tested. “Prevented” means that the process execution is unharmed. “Halted” means that the attack is detected but the process is terminated.

3. Buffer overflow of a pointer on the stack and then pointing at target:
   (a) Return address
   (b) Base pointer
   (c) Function pointer as variable
   (d) Function pointer as function parameter
   (e) Longjmp buffer as variable
   (f) Longjmp buffer as function parameter
4. Buffer overflow of a pointer on the heap/BSS/data and then pointing at target:
   (a) Return address
   (b) Base pointer
   (c) Function pointer as variable
   (d) Function pointer as function parameter
   (e) Longjmp buffer as variable
   (f) Longjmp buffer as function parameter

Note that we do not consider differences in the likelihood of certain attack forms being possible, nor current statistics on which attack forms are most popular. However, we have observed that most of the dynamic intrusion prevention tools focus on the protection of the return address. Bulba and Kil3r did not present any real-life examples of their attack forms that defeated StackGuard and Stack Shield. Also the Immunix operating system (Linux hardened with StackGuard and more) came in second place at the Defcon “Capture the Flag” competition where nearly 100 crackers and security experts tried to compromise the competing systems [12]. This implies that the tools presented here are effective against many of the currently used attack forms. The question is: will this will change as soon as this kind of protection is wide spread?

Also worth noting is that just because a attack form is prevented or halted does not mean that the very same buffer overflow can not be abused in another attack form. All of these attack forms have been implemented on the Linux platform and the source code is available from our homepage: `http://www.ida.liu.se/˜johwi`.

To set up the test, the source code was compiled with StackGuard, Stack Shield, or ProPolice, or linked with Libsafe/Libverify. The overall results are shown in table 2. We also made a theoretical comparison to investigate the potential of the ideas and concepts used in the tools. The overall results of the theoretical analysis are shown in table 3. For details of the tests see appendix A and B.

Most interesting in the overall test results is that the most effective tool, namely ProPolice, is able to prevent only 50% of the attack forms. Buffer overflows on the heap/BSS/data targeting function pointers or longjmp buffers are not prevented or halted by any of the tools, which means that a combination of all techniques built into one tool would still miss 30% of the attack forms.

This however does not comply with the result from the theoretical comparison. Stack Shield was not able to protect function pointers as stated by Vendicator. Another difference is the abnormal behavior of StackGuard and Stack Shield when confronted with a fake stack frame in the BSS segment.

These poor results are all evidence of the weakness in dynamic intrusion prevention discussed in section 3.2, the tested tools all aim to protect *known* attack targets. The return address has been a popular target and therefore all tools are fairly effective in protecting it.

Worth noting is that StackGuard halts attacks against the old base pointer although that was not mentioned as an explicit design goal.

Only ProPolice and Stack Shield offer real intrusion prevention—the other tools are more or less intrusion detection systems. But still the general behavior of all these tools is termination of process execution during attack.

Development Tool	Attacks prevented	Attacks halted	Attacks missed
StackGuard Terminator Canary	0 (0%)	4 (20%)	16 (80%)
StackGuard Random XOR Canary	0 (0%)	6 (30%)	14 (70%)
Stack Shield Global Ret Stack	6 (30%)	7 (35%)	7 (35%)
Stack Shield Range Ret Check	0 (0%)	10 (50%)	10 (50%)
Stack Shield Global & Range	6 (30%)	7 (35%)	7 (35%)
ProPolice	8 (40%)	3 (15%)	9 (45%)
Libsafe and Libverify	0 (0%)	6 (30%)	14 (70%)

Table 3. Theoretical comparison of dynamic intrusion prevention tools. 20 attack forms used. “Prevented” means that the process execution is unharmed. “Halted” means that the attack is detected but the process is terminated.

5 Common Shortcomings

There are several shortcomings worth discussing. We have identified four generic problems worth highlighting, especially when considering future research in this area.

5.1 Denial of Service Attacks

Since three out of four tools terminate execution upon detecting an attack they actually offer more of intrusion detection than intrusion prevention. More important is that the vulnerabilities still allow for Denial of Service attacks. Terminating a web service process is a common goal in security attacks. Process termination results in a much less serious attack but will still be a security issue.

5.2 Storage Protection

Canaries or separate return address stacks have to be protected from attacks. If the canary template or the stored copy of the return address can be tampered with, the protection is fooled. Only StackGuard with the terminator canary offers protection in this sense. The other tools have no protection implemented and the performance penalty of such protection can be very serious—up to 200 times [6].

5.3 Recompilation of Code

The three compiler patches have the common shortcoming of demanding recompilation of all code to provide protection. For software vendors shipping new products this is a natural thing but for running operating systems and legacy systems this is a serious drawback. Libsafe/Libverify offers a much more convenient solution in this sense. The StackGuard and ProPolice teams have addressed this issue by offering protected versions of Linux and FreeBSD.

5.4 Limited Nesting Depth

When keeping a separate stack with copies of return addresses, the nesting depth of the process is limited. Only Vendicator, author of Stack Shield, discusses this issue but offers no real solution to the problem.

6 Related Work

Three other studies of defenses against buffer overflow attacks have been made.

In late 2000 Crispin Cowan et al published their paper “Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade” [11]. They implicitly discuss several of our attack forms but leave out the old base pointer as an attack target. Comparison of defenses is broader considering also operating system patches, choice of programming language and code auditing but there is only a theoretical analysis, no comparative testing is done. Also the only dynamic tools discussed are their own StackGuard and their forthcoming PointGuard.

Only a month later Istvan Simon published his paper “A Comparative Analysis of Methods of Defense against Buffer Overflow Attacks” [32]. It discusses pros and cons with operating system patches, StackGuard, Libsafe, and similar solutions. The major drawback in his analysis is the lack of categorization of buffer overflow attack forms (only three of our attack forms are explicitly mentioned) and any structured comparison of the tool’s effectiveness. No testing is done.

In March 2002 Pierre-Alain Fayolle and Vincent Glaume published their lengthy report “A Buffer Overflow Study, Attacks & Defenses” [17]. They describe and compare Libsafe with a non-executable stack and an intrusion detection system. Tests are performed for two of our twenty attack forms. No proper categorization of buffer overflow attack forms is made or used for testing.

7 Conclusions

There are several run-time techniques for stopping the most common of security intrusion attack—the buffer overflow. But we have shown that none of these can handle the diverse forms of attacks known today. In practice at best 40% of the attack forms were prevented and another 10% detected and halted, leaving 50% of the attacks still at large. Combining all the techniques in theory would still leave us with nearly a third of the attack forms missed. In our opinion this is due to the general weakness of the dynamic intrusion prevention solution—the tools all aim at protecting *known* attack targets, not all targets. Nevertheless these tools and the ideas they are built on are effective against many security attacks that harm software users today.

8 Acknowledgments

We are grateful to the readers who have previewed and improved our paper, especially Crispin Cowan.

References

[1] A. Baratloo, N. Singh, and T. Tsai. Libsafe: Protecting critical elements of stacks. White Paper `http://www.research.avayalabs.com/project/libsafe/`, December 1999.

[2] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Technical Conference*, San Diego, California, USA, June 2000.

[3] L. M. Bowman. Companies on the hook for security. `http://news.com.com/2100-1023-821266.html`, January 2002.

[4] Bulba and Kil3r. Bypassig StackGuard and StackShield. Phrack Magazine 56 `http://www.phrack.org/phrack/56/p56-0x05`, May 2000.

[5] C. C. Center. Cert/cc statistics 1988-2001. `http://www.cert.org/stats/`, February 2002.

[6] T. cker Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21th International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, Arizona, USA, April 2001.

[7] M. Conover and w00w00 Security Team. w00w00 on heap overflows. `http://www.w00w00.org/files/articles/heaptut.txt`, January 1999.

[8] C. Cowan. Personal communication, February 2002.

[9] C. Cowan, S. Beattie, R. Day, C. Pu, P. Wagle, and E. Walthinsen. Protecting systems from stack smashing attacks with StackGuard. Linux Expo `http://www.cse.ogi.edu/~crispin/`, May 1999.

[10] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.

[11] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of the DARPA Information Survivability Conference and Expo (DISCEX)*, pages 119–129, Hilton Head, South Carolina, January 2000.

[12] W. Crispin Cowan. Nearly 100 hackers fail to crack wirex immunix server, August 2002.

[13] S. Designer. Linux kernel patch from the openwall project. `http://www.openwall.com/linux/README`.

[14] DilDog. The tao of Windows buffer overflow. `http://www.cultdeadcow.com/cDc_files/cDc-351/`, April 1998.

[15] H. Etoh. GCC extension for protecting applications from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp/`, June 2000.

[16] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, February 2002.

[17] P.-A. Fayolle and V. Glaume. A buffer overflow study, attacks & defenses. `http://www.enseirb.fr/~glaume/indexen.html`, March 2002.

[18] A. K. Ghosh, C. Howell, and J. A. Whittaker. Building software securely from the ground up. *IEEE Software*, 19(1):14–16, February 2002.

[19] F. Giasson. Memory layout in program execution. `http://www.decatomb.com/articles/memorylayout.txt`, October 2001.

[20] L. R. Halme and R. K. Bauer. AINT misbehaving: A taxonomy of anti-intrusion techniques. `http://www.sans.org/newlook/resources/IDFAQ/aint.htm`, April 2000.

[21] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[22] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the Third International Workshop on Automatic Debugging AADEBUG'97*, Linkoping, Sweden, May 1997.

[23] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, Washington DC, USA, August 2001.

[24] G. McGraw and J. Viega. An analysis of how buffer overflow attacks work. IBM developerWorks: Security: Security articles `http://www-106.ibm.com/developerworks/security/library/smash.html?dwzon%e=security`, March 2000.

[25] M. S. Mike Frantzen. StackGhost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[26] G. Necula, S. McPeak, and W. Weimer. CCured: Typesafe retrofitting of legacy code. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, Portland, OR, January 2002.

[27] G. Necula, S. McPeak, and W. Weimer. Taming C pointers. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, June 2002.

[28] B. News. Software security law call. http://news.bbc.co.uk/hi/english/sci/tech/newsid_1762000/1762261.stm, January 2002.

[29] A. One. Smashing the stack for fun and profit. http://immunix.org/StackGuard/profit.html, November 1996.

[30] C. Science and N. R. C. Telecommunications Board. Cybersecurity today and tomorrow: Pay now or pay later (prepublication). Technical report, National Academies, USA, http://www.nap.edu/books/0309083125/html/, January 2002.

[31] R. W. Shirey. Request for comments: 2828, Internet security glossary. http://www.faqs.org/rfcs/rfc2828.html, May 2000.

[32] I. Simon. A comparative analysis of methods of defense against buffer overflow attacks. http://www.mcs.csuhayward.edu/˜simon/security/boflo.html, January 2001.

[33] Vendicator. Stack Shield technical info file v0.7. http://www.angelfire.com/sk/stackshield/, January 2001.

[34] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–169, May 2001.

[35] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed System Security Symposium*, pages 3–17, Catamaran Resort Hotel, San Diego, California, February 2000.

[36] J. Wilander. Security intrusions and intrusion prevention. Master's thesis, Linkopings universitet, http://www.ida.liu.se/˜johwi, April 2002.

A Details of Empirical Test

Attack Target / Development Tool	Return address	Old Base Pointer	Func Ptr Variable	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Halted	Halted	Missed	Missed	Missed	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Missed	Missed	Missed	Missed
Stack Shield Range Ret Check	Abnormal	Missed	Missed	Missed	Missed	Missed
Stack Shield Global & Range	Prevented	Prevented	Missed	Missed	Missed	Missed
ProPolice	Halted	Halted	Prevented	Abnormal	Prevented	Missed
Libsafe and Libverify	Halted	Halted	Missed	Halted	Missed	Halted

Table 4. **Prevention of buffer overflow on the stack all the way to the target.**

Attack Target / Development Tool	Func Ptr Variable	Longjmp Buf Variable
StackGuard Terminator Canary	Missed	Missed
Stack Shield Global Ret Stack	Missed	Missed
Stack Shield Range Ret Check	Missed	Missed
Stack Shield Global & Range	Missed	Missed
ProPolice	Missed	Missed
Libsafe and Libverify	Missed	Missed

Table 5. **Prevention of buffer overflow on the heap/BSS/data all the way to the target.**

Attack Target / Development Tool	Return address	Old Base Pointer	Func Ptr Variable	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Missed	Halted	Missed	Missed	Missed	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Missed	Missed	Missed	Missed
Stack Shield Range Ret Check	Abnormal	Missed	Missed	Missed	Missed	Missed
Stack Shield Global & Range	Prevented	Prevented	Missed	Missed	Missed	Missed
ProPolice	Prevented	Prevented	Prevented	Prevented	Prevented	Prevented
Libsafe and Libverify	Missed	Abnormal	Missed	Missed	Missed	Missed

Table 6. **Prevention of buffer overflow of pointer on the stack and then pointing at target.**

Attack Target / Development Tool	Return address	Old Base Pointer	Func Ptr Variable	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Missed	Abnormal	Missed	Missed	Missed	Missed
Stack Shield Global Ret Stack	Prevented	Abnormal	Missed	Missed	Missed	Missed
Stack Shield Range Ret Check	Abnormal	Missed	Missed	Missed	Missed	Missed
Stack Shield Global & Range	Prevented	Prevented	Missed	Missed	Missed	Missed
ProPolice	Missed	Missed	Missed	Missed	Missed	Missed
Libsafe and Libverify	Missed	Missed	Missed	Missed	Missed	Missed

Table 7. **Prevention of buffer overflow of a pointer on the heap/BSS/data and then pointing at target.**

B Details of Theoretical Test

Development Tool \ Attack Target	Return address	Old Base Pointer	Func Ptr Variable	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Halted	Halted	Missed	Missed	Missed	Missed
StackGuard Random XOR Canary	Halted	Halted	Missed	Missed	Missed	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Halted	Halted	Missed	Missed
Stack Shield Range Ret Check	Halted	Missed	Halted	Halted	Missed	Missed
Stack Shield Global & Range	Prevented	Prevented	Halted	Halted	Missed	Missed
ProPolice	Halted	Halted	Prevented	Missed	Halted	Missed
Libsafe and Libverify	Halted	Halted	Missed	Halted	Missed	Halted

Table 8. **Prevention of buffer overflow on the stack all the way to the target.**

Development Tool \ Attack Target	Func Ptr Variable	Longjmp Buf Variable
StackGuard Terminator Canary	Missed	Missed
StackGuard Random XOR Canary	Missed	Missed
Stack Shield Global Ret Stack	Missed	Missed
Stack Shield Range Ret Check	Missed	Missed
Stack Shield Global & Range	Missed	Missed
ProPolice	Missed	Missed
Libsafe and Libverify	Missed	Missed

Table 9. **Prevention of buffer overflow on the heap/BSS/data all the way to the target.**

Development Tool \ Attack Target	Return address	Old Base Pointer	Func Ptr Variable	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Missed	Halted	Missed	Missed	Missed	Missed
StackGuard Random XOR Canary	Halted	Halted	Missed	Missed	Missed	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Halted	Halted	Missed	Missed
Stack Shield Range Ret Check	Halted	Missed	Halted	Halted	Missed	Missed
Stack Shield Global & Range	Prevented	Prevented	Halted	Halted	Missed	Missed
ProPolice	Prevented	Prevented	Prevented	Prevented	Prevented	Prevented
Libsafe and Libverify	Halted	Halted	Missed	Missed	Missed	Missed

Table 10. **Prevention of buffer overflow of pointer on the stack and then pointing at target.**

Development Tool \ Attack Target	Return address	Old Base Pointer	Func Ptr Variable	Func Ptr Parameter	Longjmp Buf Variable	Longjmp Buf Parameter
StackGuard Terminator Canary	Missed	Halted	Missed	Missed	Missed	Missed
StackGuard Random XOR Canary	Halted	Halted	Missed	Missed	Missed	Missed
Stack Shield Global Ret Stack	Prevented	Prevented	Halted	Halted	Missed	Missed
Stack Shield Range Ret Check	Halted	Halted	Halted	Halted	Missed	Missed
Stack Shield Global & Range	Prevented	Prevented	Halted	Halted	Missed	Missed
ProPolice	Missed	Halted	Missed	Missed	Missed	Missed
Libsafe and Libverify	Halted	Halted	Missed	Missed	Missed	Missed

Table 11. **Prevention of buffer overflow of a pointer on the heap/BSS/data and then pointing at target.**

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Format of Type Judgments

- A *type judgement* has the form

$$\Gamma \text{ |- exp} : \tau$$

- Γ is a typing environment
  - Supplies the types of variables and functions
  - Γ is a list of the form [$x : \sigma$, . . .]
- exp is a program expression
- τ is a type to be assigned to exp
- |- pronounced "turnstyle", or "entails" (or "satisfies")

Axioms - Constants

$$\frac{}{\text{|- } n \text{ : int}}$$ (assuming n is an integer constant)

$$\frac{}{\text{|- true : bool}} \qquad \frac{}{\text{|- false : bool}}$$

- These rules are true with any typing environment
- *n* is a meta-variable

Axioms – Variables (Monomorphic Rule)

Notation: Let $\Gamma(x) = \sigma$ if $x : \sigma \in \Gamma$ and there is no $x : \tau$ to the left of $x : \sigma$ in Γ

Variable axiom:

$$\frac{}{\Gamma \mid\!\!- \ x : \sigma} \quad \text{if } \Gamma(x) = \sigma$$

Simple Rules - Arithmetic

Primitive operators ($\oplus \in \{ +, -, *, \ldots\}$):

$$\frac{\Gamma \mid\!- e_1 : \tau \qquad \Gamma \mid\!- e_2 : \tau \qquad (\oplus) : \tau \rightarrow \tau \rightarrow \tau}{\Gamma \mid\!- e_1 \oplus e_2 : \tau}$$

Relations ($\sim \; \in \{ <, >, =, <=, >= \}$):

$$\frac{\Gamma \mid\!- e_1 : \tau \qquad \Gamma \mid\!- e_2 : \tau}{\Gamma \mid\!- e_1 \sim e_2 : \text{bool}}$$

For the moment, think τ is int

Simple Rules - Booleans

Connectives

$$\frac{\Gamma \mid\!- \; e_1 : \text{bool} \qquad \Gamma \mid\!- \; e_2 : \text{bool}}{\Gamma \mid\!- \; e_1 \;\&\&\; e_2 : \text{bool}}$$

$$\frac{\Gamma \mid\!- \; e_1 : \text{bool} \qquad \Gamma \mid\!- \; e_2 : \text{bool}}{\Gamma \mid\!- \; e_1 \;||\; e_2 : \text{bool}}$$

Type Variables in Rules

- If_then_else rule:

$$\frac{\Gamma \mid\!- \; e_1 : \text{bool} \quad \Gamma \mid\!- \; e_2 : \tau \quad \Gamma \mid\!- \; e_3 : \tau}{\Gamma \mid\!- \; (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau}$$

- τ is a type variable (meta-variable)
- Can take any type at all
- All instances in a rule application must get same type
- Then branch, else branch and if_then_else must all have same type

Function Application

- Application rule:

$$\frac{\Gamma \mid\!- \; e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \mid\!- \; e_2 : \tau_1}{\Gamma \mid\!- \; (e_1 \; e_2) : \tau_2}$$

- If you have a function expression e_1 of type $\tau_1 \rightarrow \tau_2$ applied to an argument of type τ_1, the resulting expression has type τ_2

Fun Rule

- Rules describe types, but also how the environment Γ may change
- Can only do what rule allows!
- fun rule:

$$\frac{[x : \tau_1] + \Gamma \mid\!\!- \; e : \tau_2}{\Gamma \mid\!\!- \text{fun } x \text{ -> } e : \tau_1 \rightarrow \tau_2}$$

Fun Examples

$$\frac{[y : int] + \Gamma \mid\!- \ y + 3 \ : int}{\Gamma \mid\!- \ fun\ y \rightarrow y + 3 \ : int \rightarrow int}$$

$$\frac{[f : int \rightarrow bool] + \Gamma \mid\!- \ f\ 2 :: [true] \ : bool\ list}{\Gamma \mid\!- \ (fun\ f \rightarrow f\ 2 :: [true]) \ : (int \rightarrow bool) \rightarrow bool\ list}$$

(Monomorphic) Let and Let Rec

- let rule:

$$\frac{\Gamma \text{ |- } e_1 : \tau_1 \qquad [x : \tau_1] + \Gamma \text{ |- } e_2 : \tau_2}{\Gamma \text{ |- } (\text{let } x = e_1 \text{ in } e_2) : \tau_2}$$

- let rec rule:

$$\frac{[x: \tau_1] + \Gamma \text{ |- } e_1 : \tau_1 \quad [x: \tau_1] + \Gamma \text{ |- } e_2 : \tau_2}{\Gamma \text{ |- } (\text{let rec } x = e_1 \text{ in } e_2) : \tau_2}$$

Example

- Which rule do we apply?

$$\frac{?}{\text{|- (let rec one = 1 :: one in let x = 2 in fun y -> (x :: y :: one)) : int} \to \text{int list}}$$

Example

- Let rec rule: ② [one : int list] |- (let x = 2 in fun y -> (x :: y :: one)) : int → int list

① [one : int list] |- (1 :: one) : int list

|- (let rec one = 1 :: one in
let x = 2 in
fun y -> (x :: y :: one)) : int → int list

Proof of 1

- Which rule?

[one : int list] |- (1 :: one) : int list

Proof of 1

- Application

③ [one : int list] |- ((::) 1): int list→ int list

④ [one : int list] |- one : int list

$$\frac{\text{③ [one : int list] |- ((::) 1): int list} \to \text{int list} \qquad \text{④ [one : int list] |- one : int list}}{\text{[one : int list] |- (1 :: one) : int list}}$$

Proof of 3

$$\dfrac{\dfrac{\text{Constants Rule}}{[\text{one} : \text{int list}] \mid\!- \ (::) : \text{int} \to \text{int list} \to \text{int list}} \qquad \dfrac{\text{Constants Rule}}{[\text{one} : \text{int list}] \mid\!- \ 1 : \text{int}}}{[\text{one} : \text{int list}] \mid\!- \ ((::)\ 1) : \text{int list} \to \text{int list}}$$

Proof of 4

- Rule for variables

$$\overline{[\text{one} : \text{int list}] \mid\!\!- \text{one:int list}}$$

Proof of 2

5 [x:int; one : int list] |- fun y -> (x :: y :: one))

- Constant

$$\dfrac{\dfrac{}{\text{[one : int list] |- 2:int}} \qquad \text{⑤ [x:int; one : int list] |- fun y -> (x :: y :: one)) : int} \rightarrow \text{int list}}{\text{[one : int list] |- (let x = 2 in fun y -> (x :: y :: one)) : int} \rightarrow \text{int list}}$$

Proof of 5

$$\frac{?}{[x{:}int;\ one : int\ list]\ |\text{-}\ fun\ y\ \text{->}\ (x :: y :: one) : int \rightarrow int\ list}$$

Proof of 5

$$\frac{\dfrac{?}{\text{[y:int; x:int; one : int list] |- (x :: y :: one) : int list}}}{\text{[x:int; one : int list] |- fun y -> (x :: y :: one)) : int} \rightarrow \text{int list}}$$

Proof of 5

6

[y:int; x:int; one : int list] |- ((::) x):int list→ int list

7

[y:int; x:int; one : int list] |- (y :: one) : int list

[y:int; x:int; one : int list] |- (x :: y :: one) : int list

[x:int; one : int list] |- fun y -> (x :: y :: one)) : int → int list

Proof of 6

Constant   Variable

$$\dfrac{\dfrac{}{[\ldots] \mid\!- (::) : \text{int} \rightarrow \text{int list} \rightarrow \text{int list}} \quad \dfrac{}{[\ldots;\ x:\text{int};\ldots] \mid\!- x:\text{int}}}{[y:\text{int};\ x:\text{int};\ \text{one} : \text{int list}] \mid\!- ((::)\ x) : \text{int list} \rightarrow \text{int list}}$$

Proof of 7

Pf of 6 [y/x] Variable

⋮

[y:int; …] |- ((::) y) :int list→ int list   [...; one: int list] |- one: int list

[y:int; x:int; one : int list] |- (y :: one) : int list

Curry - Howard Isomorphism

- Type Systems are logics; logics are type systems
- Types are propositions; propositions are types
- Terms are proofs; proofs are terms

- Functions space arrow corresponds to implication; application corresponds to modus ponens

Curry - Howard Isomorphism

- Modus Ponens

$$\frac{A \Rightarrow B \quad A}{B}$$

- Application

$$\frac{\Gamma \mid\!- \ e_1 : \alpha \rightarrow \beta \quad \Gamma \mid\!- \ e_2 : \alpha}{\Gamma \mid\!- \ (e_1 \ e_2) : \beta}$$

Mia Copa

- The above system can't handle polymorphism as in OCAML
- No type variables in type language (only meta-variable in the logic)
- Would need:
  - Object level type variables and some kind of type quantification
  - **let** and **let rec** rules to introduce polymorphism
  - Explicit rule to eliminate (instantiate) polymorphism

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Curry - Howard Isomorphism

- Type Systems are logics; logics are type systems
- Types are propositions; propositions are types
- Terms are proofs; proofs are terms
- Functions space arrow corresponds to implication; application corresponds to modus ponens

Curry - Howard Isomorphism

- Modus Ponens

$$\frac{A \Rightarrow B \quad A}{B}$$

- Application

$$\frac{\Gamma \mid\!- \; e_1 : \alpha \rightarrow \beta \quad \Gamma \mid\!- \; e_2 : \alpha}{\Gamma \mid\!- \; (e_1 \; e_2) : \beta}$$

Mia Copa

- The above system can't handle polymorphism as in OCAML
- No type variables in type language (only meta-variable in the logic)
- Would need:
  - Object level type variables and some kind of type quantification
  - **let** and **let rec** rules to introduce polymorphism
  - Explicit rule to eliminate (instantiate) polymorphism

Support for Polymorphic Types

- Monomorpic Types (τ):
  - Basic Types: int, bool, float, string, unit, …
  - Type Variables: α, β, γ δ ε
  - Compound Types: $\alpha \rightarrow \beta$, int * string, bool list, …
- Polymorphic Types:
  - Monomorphic types τ
  - Universally quantified monomorphic types
  - $\forall \alpha_1, \ldots, \alpha_n . \tau$
  - Can think of τ as same as $\forall . \tau$

Support for Polymorphic Types

- Typing Environment Γ supplies polymorphic types (which will often just be monomorphic) for variables
- Free variables of monomorphic type just type variables that occur in it
  - Write FreeVars(τ)
- Free variables of polymorphic type removes variables that are universally quantified
  - FreeVars($\forall\alpha_1, \ldots , \alpha_n . \tau$) = FreeVars($\tau$) – $\{\alpha_1, \ldots , \alpha_n \}$
- FreeVars(Γ) = all FreeVars of types in range of Γ

Monomorphic to Polymorphic

- Given:
  - type environment Γ
  - monomorphic type τ
  - τ shares type variables with Γ
- Want most polymorphic type for τ that doesn't break sharing type variables with Γ
- $Gen(\tau, \Gamma) = \forall \alpha_1, \ldots, \alpha_n . \tau$ where $\{\alpha_1, \ldots, \alpha_n\} = freeVars(\tau) - freeVars(\Gamma)$

Polymorphic Typing Rules

- A *type judgement* has the form

$$\Gamma \text{ |- exp} : \tau$$

  - Γ uses polymorphic types
  - τ still monomorphic
- Most rules stay same (except use more general typing environments)
- Rules that change:
  - Variables
  - Let and Let Rec
  - Allow polymorphic constants
- Worth noting functions again

Polymorphic Let and Let Rec

- let rule:

$$\frac{\Gamma \mid\!- \; e_1 : \tau_1 \quad [x : \mathrm{Gen}(\tau_1,\Gamma)] + \Gamma \mid\!- \; e_2 : \tau_2}{\Gamma \mid\!- \; (\text{let } x = e_1 \text{ in } e_2) : \tau_2}$$

- let rec rule:

$$\frac{[x: \tau_1] + \Gamma \mid\!- \; e_1 : \tau_1 \quad [x : \mathrm{Gen}(\tau_1,\Gamma)] + \Gamma \mid\!- \; e_2 : \tau_2}{\Gamma \mid\!- \; (\text{let rec } x = e_1 \text{ in } e_2) : \tau_2}$$

Polymorphic Variables (Identifiers)

Variable axiom:

$$\frac{}{\Gamma \mid\!- \; x : \varphi(\tau)} \quad \text{if } \Gamma(x) = \forall \alpha_1, \ldots, \alpha_n \,.\, \tau$$

- Where φ replaces all occurrences of $\alpha_1, \ldots, \alpha_n$ by monotypes $\tau_1, \ldots, \tau_n$
- Note: Monomorphic rule special case:

$$\frac{}{\Gamma \mid\!- \; x : \tau} \quad \text{if } \Gamma(x) = \tau$$

- Constants treated same way

Fun Rule Stays the Same

- fun rule:

$$\frac{[x : \tau_1] + \Gamma \mid\!\text{-}\ e : \tau_2}{\Gamma \mid\!\text{-}\ \text{fun } x \text{ -> } e : \tau_1 \rightarrow \tau_2}$$

- Types τ_1, τ_2 monomorphic
- Function argument must always be used at same type in function body

Polymorphic Example

- Assume additional constants:
- hd :$\forall\alpha.\ \alpha$ list -> α
- tl: $\forall\alpha.\ \alpha$ list -> α list
- is_empty : $\forall\alpha.\ \alpha$ list -> bool
- :: : $\forall\alpha.\ \alpha$ -> α list -> α list
- [] : $\forall\alpha.\ \alpha$ list

Polymorphic Example

- Show:

$$\frac{?}{\{\}\ |\text{-}\ \texttt{let rec length = fun l -> if is\_empty l then 0 else 1 + length (tl l) in length ((::) 2 []) + length((::) true []) : int}}$$

```
                                   ?
----------------------------------------------------------------
{} |- let rec length =
        fun l -> if is_empty l then 0
                 else 1 + length (tl l)
  in length ((::) 2 []) + length((::) true []) : int
```

Polymorphic Example: Let Rec Rule

- Show: (1) (2)

$$\frac{\{length:\alpha\ list \to int\} \vdash fun\ l \to \ldots : \alpha\ list \to int \qquad \{length:\forall\alpha.\ \alpha\ list \to int\} \vdash length\ ((::)\ 2\ [\,]) + length((::)\ true\ [\,]) : int}{\{\} \vdash let\ rec\ length = fun\ l \to if\ is\_empty\ l\ then\ 0\ else\ 1 + length\ (tl\ l)\ in\ length\ ((::)\ 2\ [\,]) + length((::)\ true\ [\,]) : int}$$

Polymorphic Example (1)

- Show:

$$\frac{?}{\{\text{length}:\alpha \text{ list} \to \text{int}\} \vdash \text{fun } l \to \text{if is\_empty } l \text{ then } 0 \text{ else } 1 + \text{length } (\text{tl } l) : \alpha \text{ list} \to \text{int}}$$

Polymorphic Example (1): Fun Rule

- Show: (3)

$$\frac{\{length:\alpha\ list \rightarrow int,\ l: \alpha\ list\}\ |\!-\ \text{if is\_empty l then 0 else length (hd l) + length (tl l)} : int}{\{length:\alpha\ list \rightarrow int\}\ |\!-\ \text{fun l -> if is\_empty l then 0 else 1 + length (tl l)} : \alpha\ list \rightarrow int}$$

Polymorphic Example (3)

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\frac{?}{\Gamma \mid\!- \text{ if is\_empty l then 0 else 1 + length (tl l) : int}}$$

Polymorphic Example (3):IfThenElse

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\frac{\begin{array}{ccc} (4) & (5) & (6) \\ \Gamma\text{|- is\_empty l} & \Gamma\text{|- 0:int} & \Gamma\text{|- 1 +} \\ \text{: bool} & & \text{length (tl l) : int} \end{array}}{\begin{array}{l}\Gamma\text{|- if is\_empty l then 0} \\ \quad\text{else 1 + length (tl l) : int}\end{array}}$$

Polymorphic Example (4)

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\frac{?}{\Gamma \mid\!- \text{is\_empty l : bool}}$$

Polymorphic Example (4):Application

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\dfrac{\dfrac{?}{\Gamma \mid\text{- is\_empty : } \alpha \text{ list -> bool}} \qquad \dfrac{?}{\Gamma \mid\text{- l : } \alpha \text{ list}}}{\Gamma \mid\text{- is\_empty l : bool}}$$

Polymorphic Example (4)

- Let Γ ={length:α list -> int, l: α list }
- Show

By Const since α list -> bool is instance of $\forall\alpha.\ \alpha$ list -> bool ?

$$\dfrac{\dfrac{}{\Gamma \mid\!- \text{ is\_empty} : \alpha \text{ list} \to \text{bool}} \qquad \dfrac{}{\Gamma \mid\!- \text{ l} : \alpha \text{ list}}}{\Gamma \mid\!- \text{ is\_empty l} : \text{bool}}$$

Polymorphic Example (4)

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\dfrac{\dfrac{\text{By Const since } \alpha \text{ list -> bool is instance of } \forall\alpha.\ \alpha \text{ list -> bool}}{\Gamma \mid\!- \text{ is\_empty} : \alpha \text{ list -> bool}} \qquad \dfrac{\text{By Variable } \Gamma(\text{l}) = \alpha \text{ list}}{\Gamma \mid\!- \text{ l} : \alpha \text{ list}}}{\Gamma \mid\!- \text{ is\_empty l} : \text{bool}}$$

- This finishes (4)

Polymorphic Example (5):Const

- Let Γ ={length:α list -> int, l: α list }
- Show

By Const Rule

$$\frac{}{\Gamma \vdash 0:\text{int}}$$

Polymorphic Example (6):Arith Op

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\dfrac{\dfrac{\text{By Const}}{\Gamma \mid\!-\ 1:\text{int}} \qquad \dfrac{\dfrac{\text{By Variable}}{\Gamma \mid\!-\ \text{length} : \alpha\ \text{list} \to \text{int}} \qquad \overset{(7)}{\Gamma \mid\!-\ (\text{tl l}) : \alpha\ \text{list}}}{\Gamma \mid\!-\ \text{length (tl l)} : \text{int}}}{\Gamma \mid\!-\ 1 + \text{length (tl l)} : \text{int}}$$

Polymorphic Example (7):App Rule

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\dfrac{\dfrac{\text{By Const}}{\Gamma \mid\!- \text{ (tl l) : } \alpha \text{ list -> } \alpha \text{ list}} \qquad \dfrac{\text{By Variable}}{\Gamma \mid\!- \text{ l : } \alpha \text{ list}}}{\Gamma \mid\!- \text{ (tl l) : } \alpha \text{ list}}$$

By Const since α list -> α list is instance of $\forall\alpha$. α list -> α list

Polymorphic Example: (2) by ArithOp

- Let Γ' = {length:$\forall\,\alpha.\ \alpha$ list -> int}
- Show:

$$\dfrac{\begin{array}{cc} (8) & (9) \\ \Gamma' \mid\!- & \Gamma' \mid\!- \\ \text{length ((::) 2 []) :int} & \text{length((::) true []) : int} \end{array}}{\begin{array}{l} \text{\{length: } \alpha.\ \alpha \text{ list -> int\}} \\ \mid\!- \text{ length ((::) 2 []) + length((::) true []) : int} \end{array}}$$

Polymorphic Example: (8)AppRule

- Let Γ' = {length:$\forall\ \alpha.\ \alpha$ list -> int}
- Show:

$$\frac{\Gamma' \text{ |- length : int list ->int list} \quad \Gamma' \text{ |- ((::)2 []):int list}}{\Gamma' \text{ |- length ((::) 2 []) :int}}$$

Polymorphic Example: (8)AppRule

- Let Γ' = {length:$\forall\ \alpha.\ \alpha$ list -> int}
- Show:

By Var since int list -> int is instance of

$\forall\alpha.\ \alpha$ list -> int

$$\dfrac{\dfrac{}{\Gamma' \text{ |- length : int list ->int list}} \qquad \dfrac{(10)}{\Gamma' \text{ |- ((::)2 []):int list}}}{\Gamma' \text{ |- length ((::) 2 []) :int}}$$

Polymorphic Example: (10)AppRule

- Let Γ' = {length:$\forall\,\alpha.\ \alpha$ list -> int}
- Show:
- By Const since α list is instance of $\forall\alpha.\ \alpha$ list

$$\dfrac{\Gamma' \mid\!- ((::)\ 2) : \text{int list -> int list} \qquad \dfrac{\quad}{\Gamma' \mid\!- [\,] : \text{int list}}}{\Gamma' \mid\!- ((::)\ 2\ [\,]) : \text{int list}}\ (11)$$

Polymorphic Example: (11)AppRule

- Let Γ' = {length:$\forall \alpha. \alpha$ list -> int}
- Show:
- By Const since α list is instance of $\forall\alpha. \alpha$ list

$$\dfrac{\dfrac{\text{By Const}}{\Gamma' \text{ |- (::) : int -> int list -> int list}} \qquad \dfrac{\text{By Const}}{\Gamma' \text{ |- 2 : int}}}{\Gamma' \text{ |- ((::) 2) : int list -> int list}}$$

Polymorphic Example: (9)AppRule

- Let Γ' = {length:$\forall$ α. α list -> int}
- Show:

$$\frac{\Gamma' \text{ |- length:bool list ->int} \qquad \Gamma' \text{ |- ((::) true []):bool list}}{\Gamma' \text{ |- length ((::) true []) :int}}$$

Polymorphic Example: (9)AppRule

- Let Γ' = {length:$\forall\ \alpha.\ \alpha$ list -> int}
- Show:

By Var since bool list -> int is instance of $\forall\alpha.\ \alpha$ list -> int

$$\dfrac{\dfrac{}{\Gamma' \text{ |- length:bool list ->int}} \qquad \dfrac{(12)}{\Gamma' \text{ |- ((::) true []):bool list}}}{\Gamma' \text{ |- length ((::) true []) :int}}$$

Polymorphic Example: (12)AppRule

- Let Γ' = {length:$\forall \alpha. \alpha$ list -> int}
- Show:
- By Const since α list is instance of $\forall\alpha. \alpha$ list

$$\frac{\Gamma' \text{ |-((::)true):bool list ->bool list} \quad \dfrac{(13)}{\Gamma' \text{ |- []:bool list}}}{\Gamma' \text{ |- ((::) true []) :bool list}}$$

Polymorphic Example: (13)AppRule

- Let Γ' = {length:$\forall \alpha.\ \alpha$ list -> int}
- Show:

By Const since bool list is instance of $\forall\alpha.\ \alpha$ list

By Const

$$\dfrac{\dfrac{}{\Gamma' \text{ |- } (::)\text{:bool -> bool list -> bool list}} \qquad \dfrac{}{\Gamma' \text{ |- } \text{true : bool}}}{\Gamma' \text{ |- } ((::)\text{ true}) : \text{bool list -> bool list}}$$

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Regular Expressions

- Start with a given character set – **a, b, c**…
- Each character is a regular expression
  - It represents the set of one string containing just that character

Regular Expressions

- If **x** and **y** are regular expressions, then **xy** is a regular expression
  - It represents the set of all strings made from first a string described by **x** then a string described by **y**

If x={a,ab} and y={c,d} then xy ={ac,ad,abc,abd}.

- If **x** and **y** are regular expressions, then **x**∨**y** is a regular expression
  - It represents the set of strings described by either **x** or **y**

  If x={a,ab} and y={c,d} then x ∨ y={a,ab,c,d}

Regular Expressions

- If **x** is a regular expression, then so is **(x)**
  - It represents the same thing as **x**

- If **x** is a regular expression, then so is **x***
  - It represents strings made from concatenating zero or more strings from **x**

  If x = {a,ab}

  then x* ={“”,a,ab,aa,aab,abab,aaa,aaab,…}

- **ε**
  - It represents {“”}, set containing the empty string

Example Regular Expressions

- **(0**∨**1)*1**
  - The set of all strings of **0**'s and **1**'s ending in 1, **{1, 01, 11,…}**
- **a*b(a*)**
  - The set of all strings of a's and b's with exactly one b
- **((01)** ∨**(10))***
  - You tell me
- Regular expressions (equivalently, regular grammars) important for lexing, breaking strings into recognized words

Example: Lexing

- Regular expressions good for describing lexemes (words) in a programming language
  - Identifier = (a ∨ b ∨ … ∨ z ∨ A ∨ B ∨ … ∨ Z) (a ∨ b ∨ … ∨ z ∨ A ∨ B ∨ … ∨ Z ∨ 0 ∨ 1 ∨ … ∨ 9)*
  - Digit = (0 ∨ 1 ∨ … ∨ 9)
  - Number = 0 ∨ (1 ∨ … ∨ 9)(0 ∨ … ∨ 9)* ∨ ~ (1 ∨ … ∨ 9)(0 ∨ … ∨ 9)*
  - Keywords: if = if, while = while,…

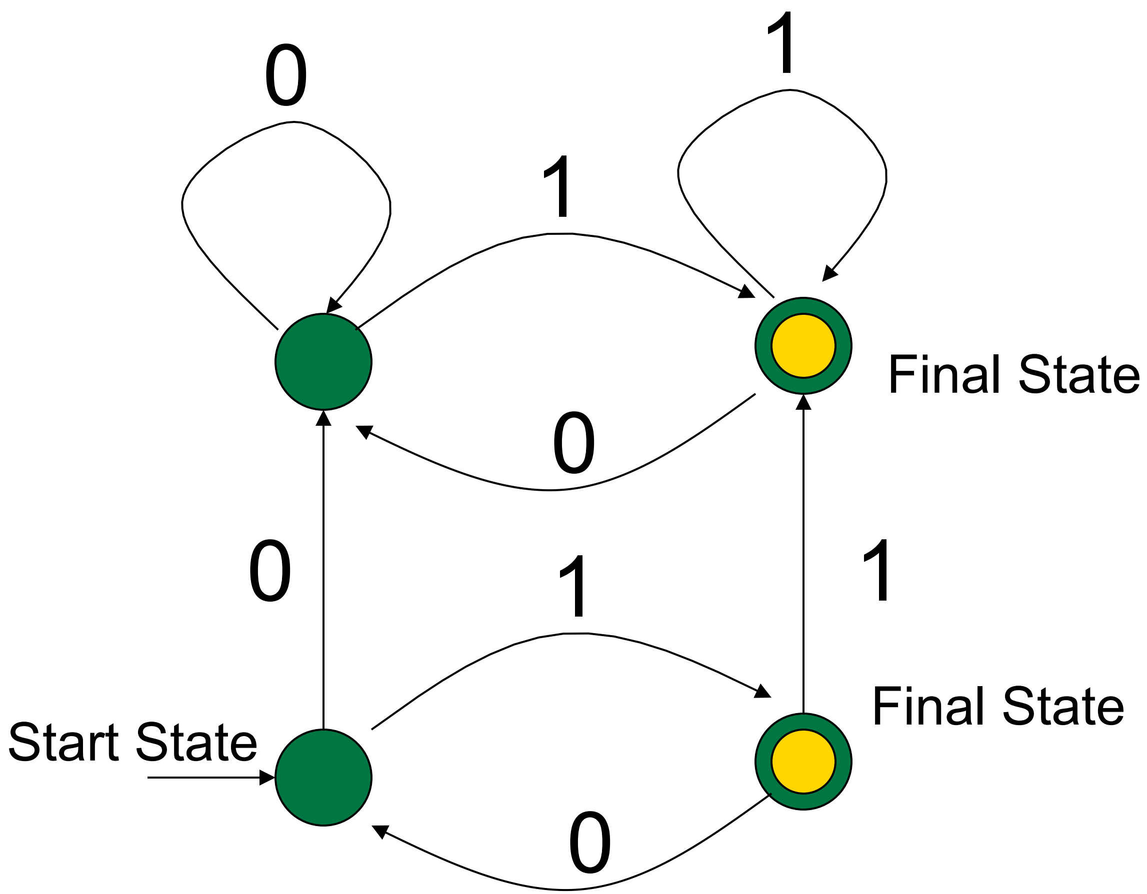
Implementing Regular Expressions

- Regular expressions reasonable way to generate strings in language
- Not so good for recognizing when a string is in language
- Problems with Regular Expressions
  - which option to choose,
  - how many repetitions to make
- Answer: finite state automata

Finite State Automata

- A finite state automata over an alphabet is:
  - a directed graph
  - a finite set of states defined by the nodes
  - edges are labeled with elements of alphabet, or empty string; they define state transition
  - some nodes (or *states*), marked as final
  - one node marked as start state
- Syntax of FSA

Example FSA



Deterministic FSA’s

- If FSA has for every state *exactly one* edge for each letter in alphabet then FSA is *deterministic*
  - No edge labeled with ε
- In general FSA in *non-deterministic*.
  - NFSA also allows edges labeled by ε
- Deterministic FSA special kind of non-deterministic FSA

DFSA Language Recognition

- Think of a DFSA as a board game; DFSA is board
- You have string as a deck of cards; one letter on each card
- Start by placing a disc on the start state

DFSA Language Recognition

- Move the disc from one state to next along the edge labeled the same as top card in deck; discard top card

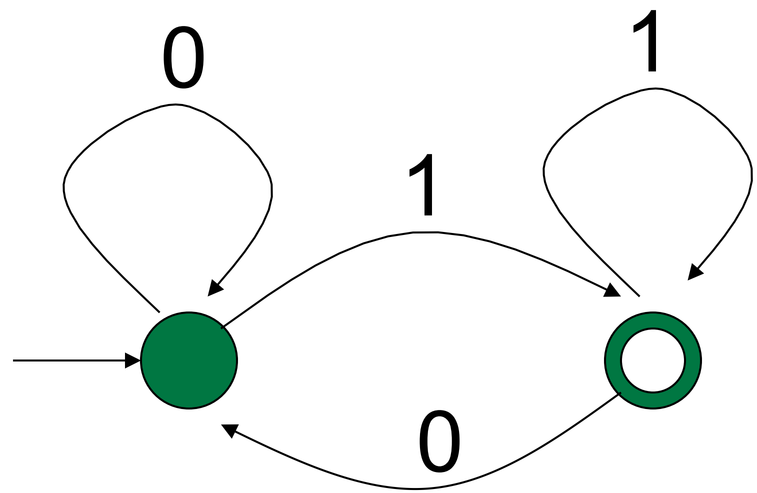
- When you run out of cards,
  - if you are in final state, you win; string is in language
  - if you are not in a final state, you lose; string is not in language

- Given a string over alphabet
- Start at start state
- Move over edge labeled with first letter to new state
- Remove first letter from string
- Repeat until string gone
- If end in final state then string in language

- Semantics of FSA

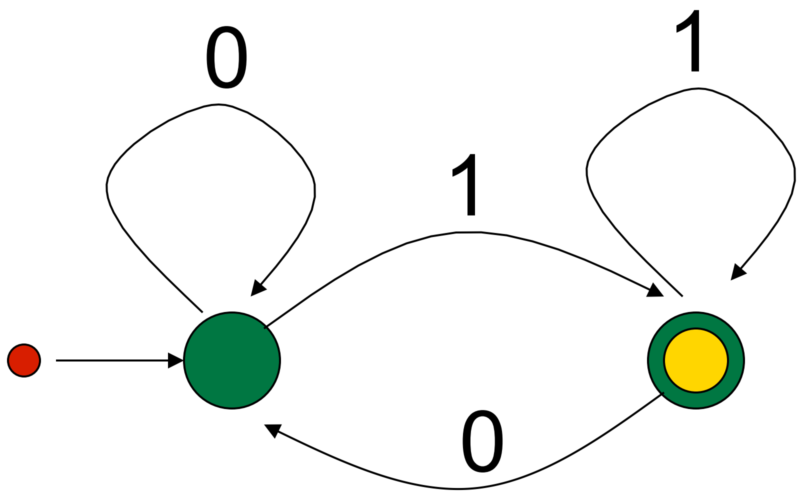
Example DFSA

- Regular expression: (0 ∨ 1)* 1
- Deterministic FSA



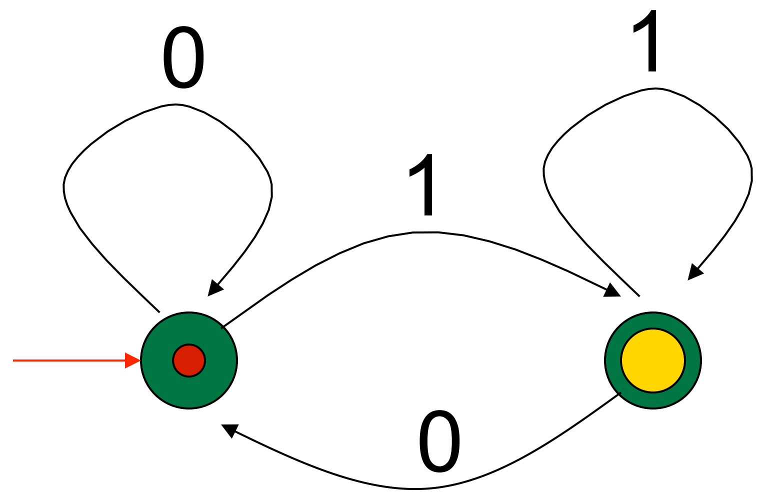
Example DFSA

- Regular expression: (0 ∨ 1)* 1
- Accepts string 0 1 1 0 1



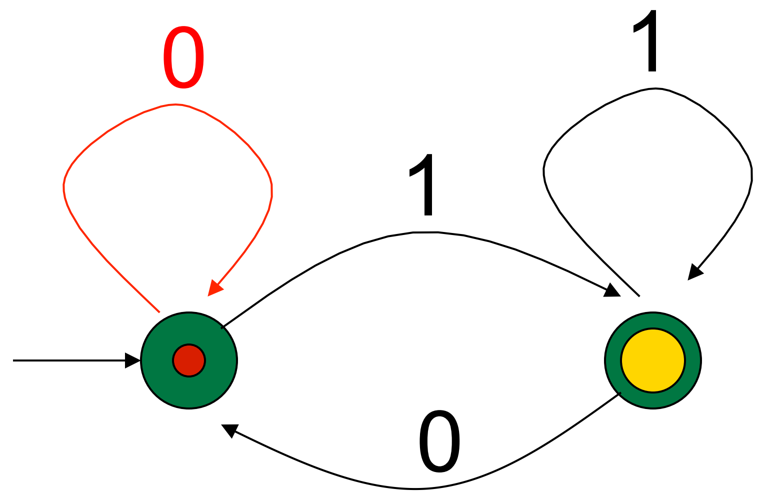
Example DFSA

- Regular expression: (0 ∨ 1)* 1
- Accepts string 0 1 1 0 1



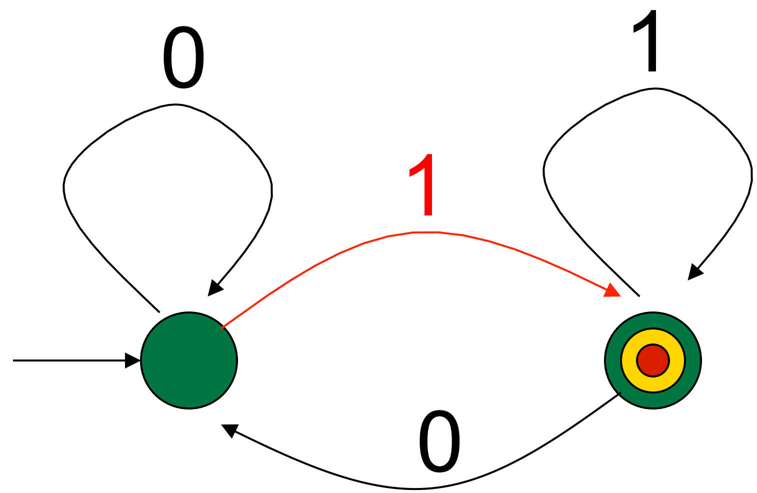
Example DFSA

- Regular expression: (0 ∨ 1)* 1
- Accepts string ~~0~~ 1 1 0 1



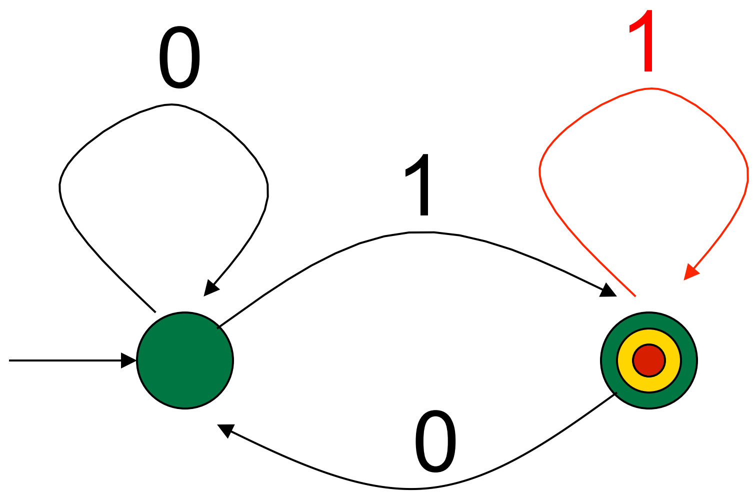
Example DFSA

- Regular expression: (0 v 1)* 1
- Accepts string ~~0~~ ~~1~~ 1 0 1



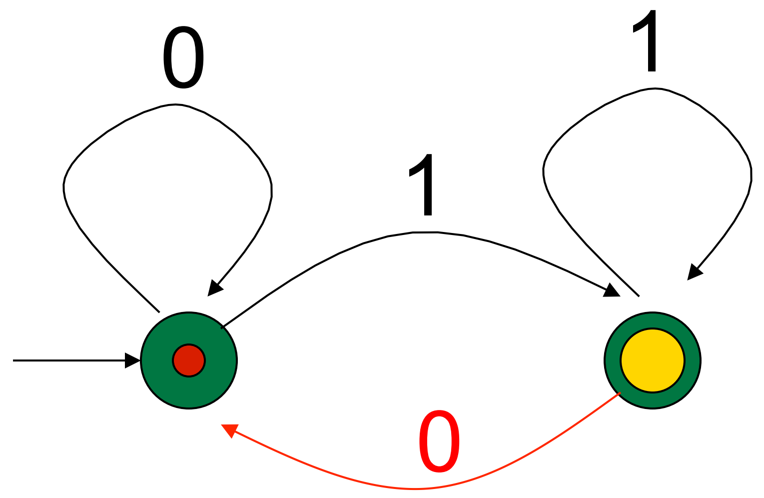
Example DFSA

- Regular expression: (0 ∨ 1)* 1
- Accepts string ~~0~~ ~~1~~ ~~1~~ 0 1



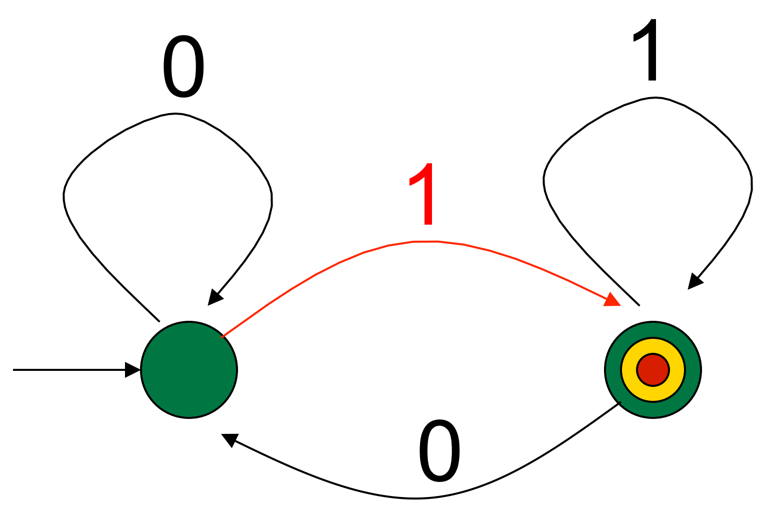
Example DFSA

- Regular expression: (0 v 1)* 1
- Accepts string ~~0~~ ~~1~~ ~~1~~ ~~0~~ 1



Example DFSA

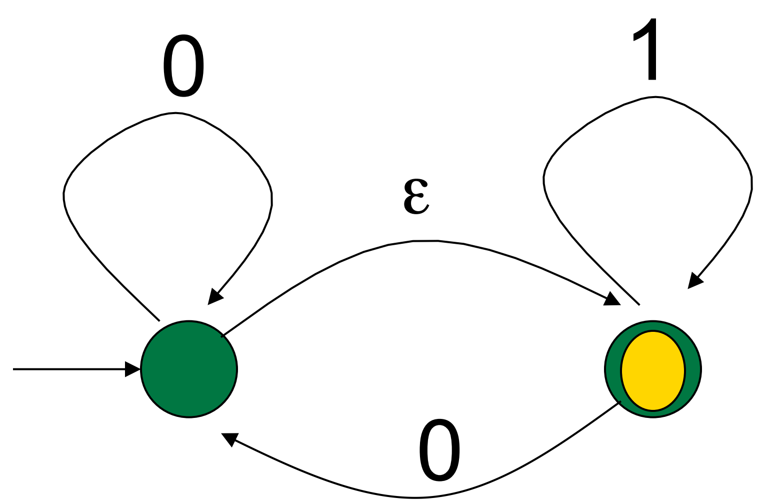
- Regular expression: (0 v 1)* 1
- Accepts string ~~0~~ ~~1~~ ~~1~~ ~~0~~ ~~1~~



Non-deterministic FSA’s

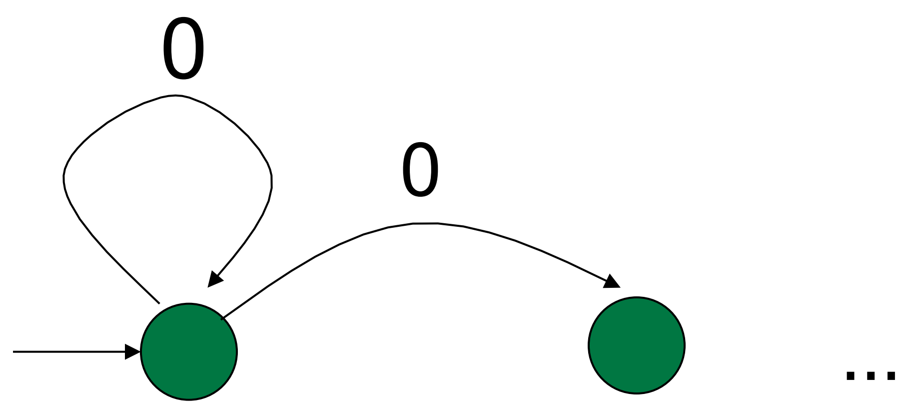
- NFSA generalize DFSA in two ways:
- Include edges labeled by ε
  - Allows process to non-deterministically change state

change state



Non-deterministic FSA's

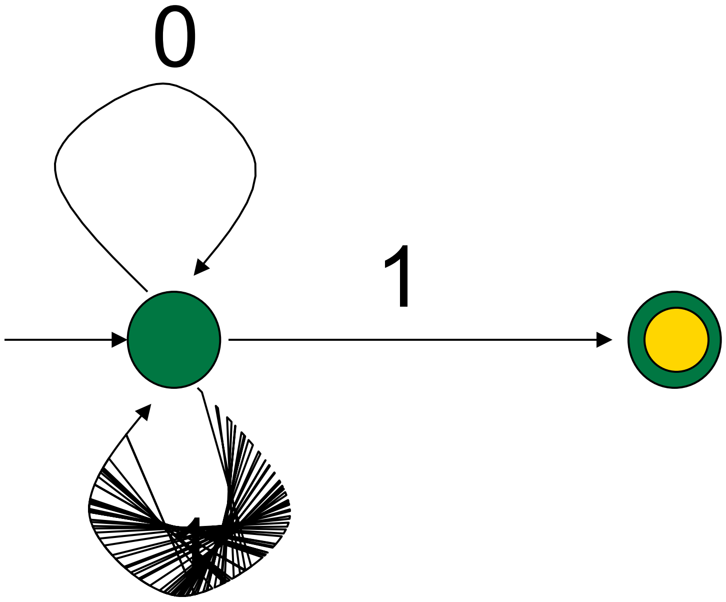
- Given a letter, non-deterministically choose an edge to use



NFSA Language Recognition

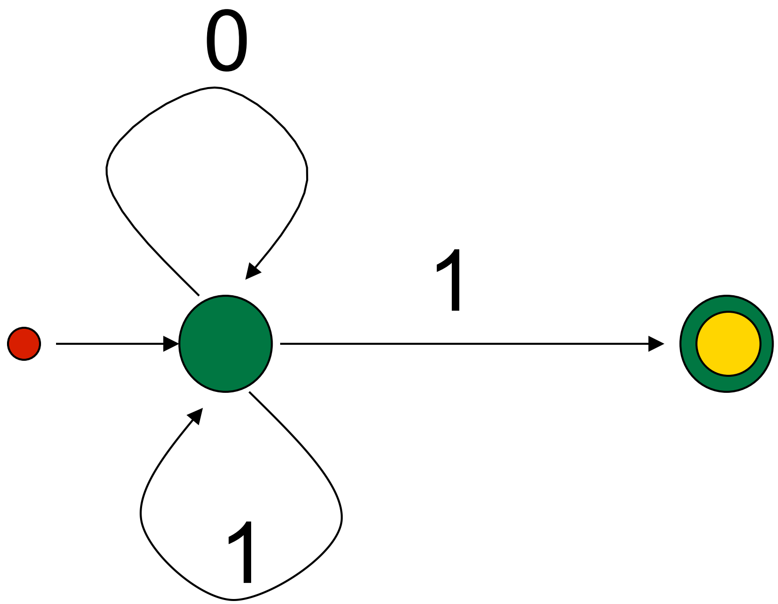
- When you run out of letters, if you are in final state, you win; string is in language
- You can take one or more moves back and try again
- If have tried all possible paths without success, then you lose; string not in language

If have tried all possible paths without success, then you lose; string not in language


0
1

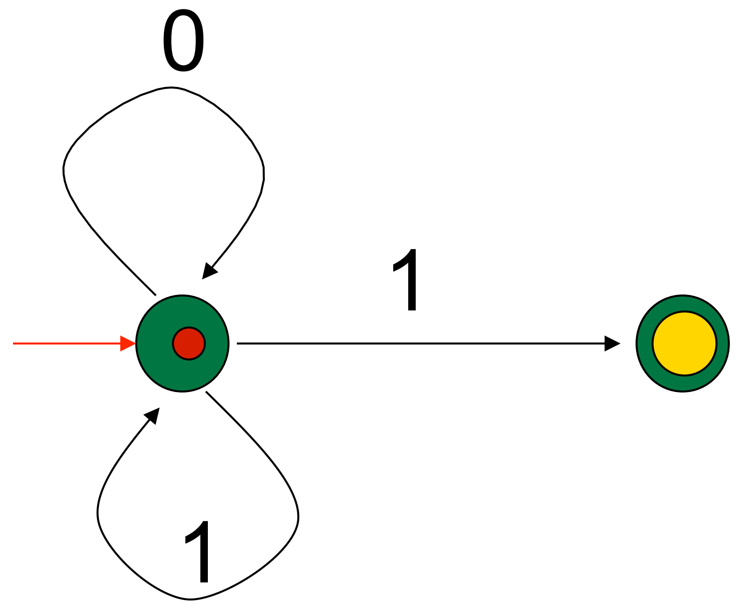
Example NFSA

- Regular expression: (0 v 1)* 1
- Accepts string   0 1 1 0 1



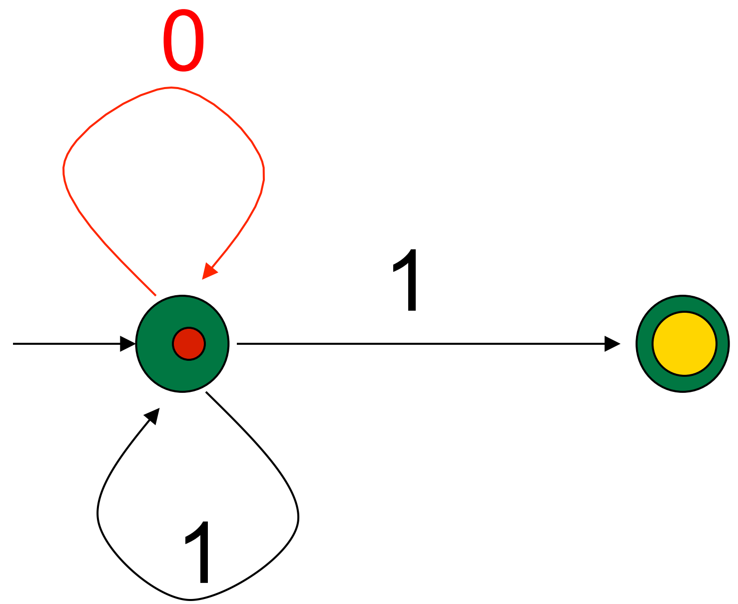
Example NFSA

- Regular expression: (0 ∨ 1)* 1
- Accepts string   0 1 1 0 1



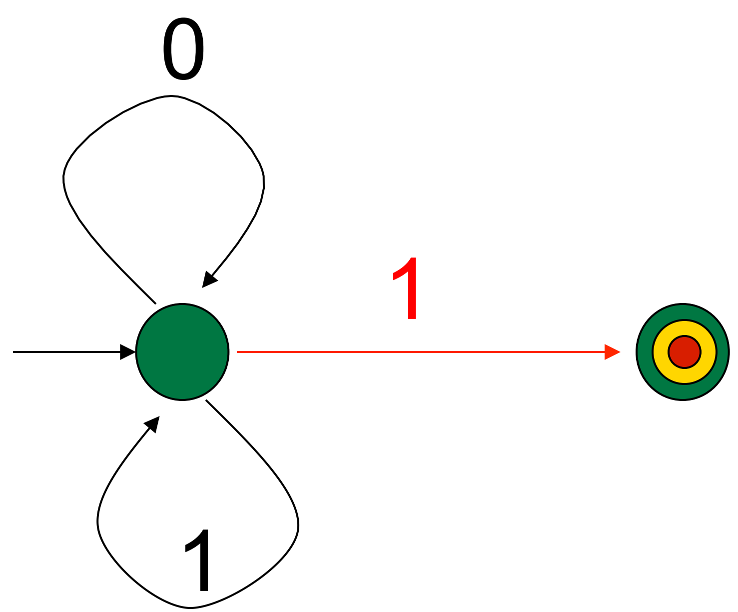
Example NFSA

- Regular expression: (0 ∨ 1)* 1
- Accepts string ~~0~~ 1 1 0 1



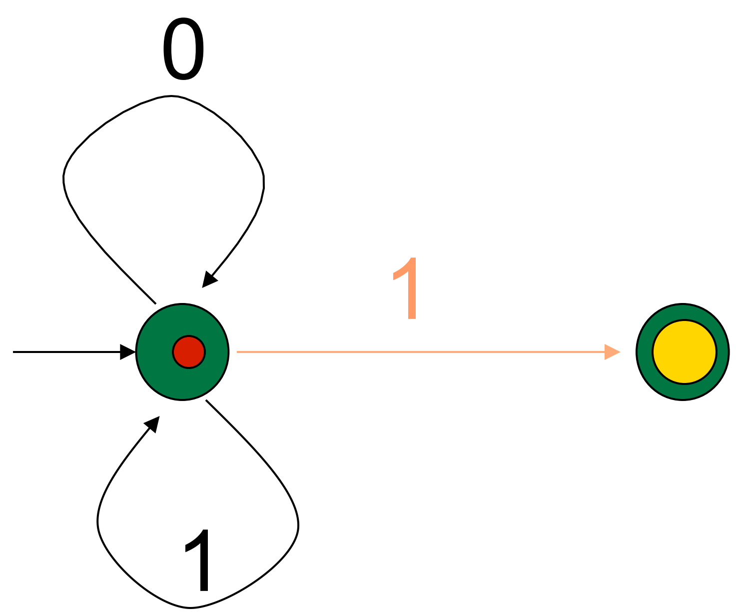
Example NFSA

- Regular expression: (0 ∨ 1)* 1
- Accepts string ~~0~~ ~~1~~ 1 0 1
- Guess



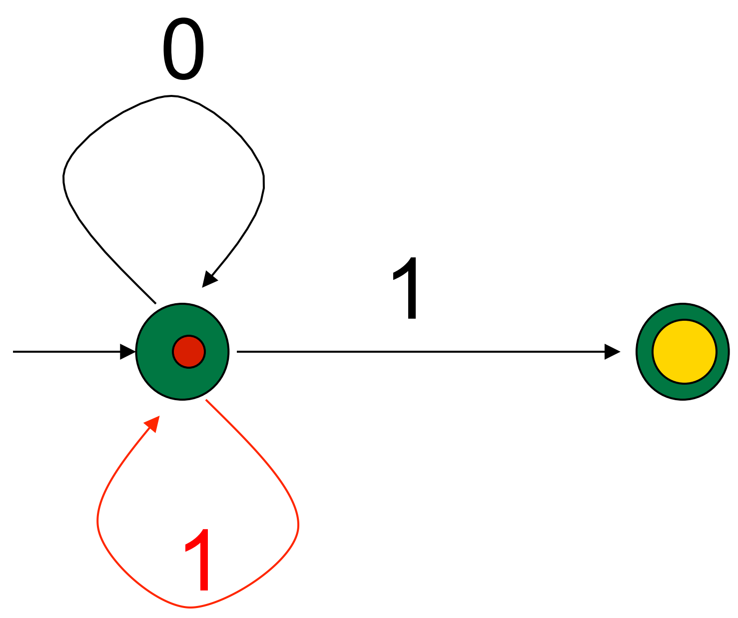
Example NFSA

- Regular expression: (0 ∨ 1)* 1
- Accepts string   0 1 1 0 1
- Backtrack



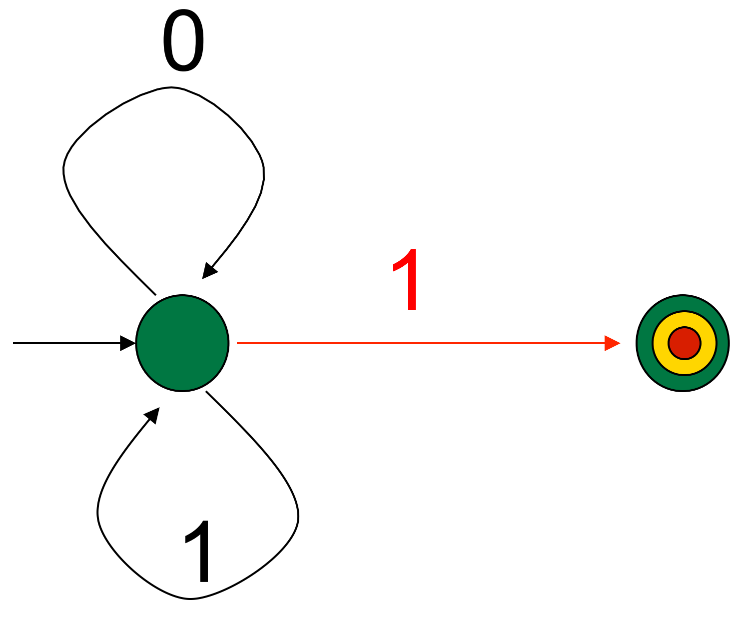
Example NFSA

- Regular expression: (0 ∨ 1)* 1
- Accepts string ~~0~~ ~~1~~ 1 1 0 1
- Guess again



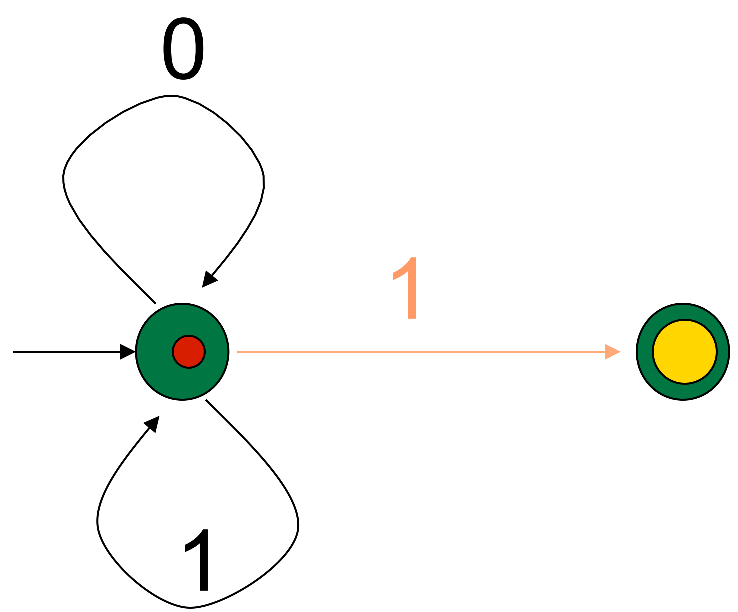
Example NFSA

- Regular expression: (0 ∨ 1)* 1
- Accepts string ~~0~~ ~~1~~ ~~1~~ 0 1
- Guess



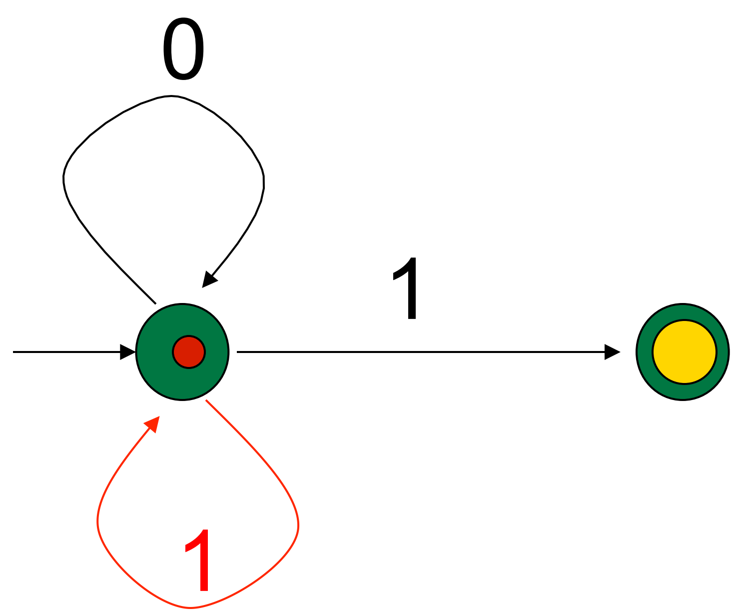
Example NFSA

- Regular expression: (0 v 1)* 1
- Accepts string ~~0~~ ~~1~~ 1 0 1
- Backtrack



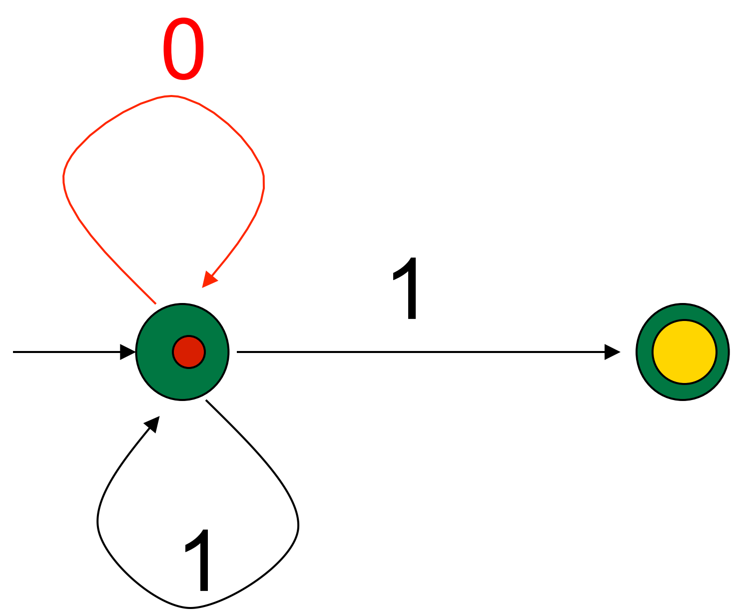
Example NFSA

- Regular expression: (0 ∨ 1)* 1
- Accepts string ~~0~~ ~~1~~ ~~1~~ 0 1
- Guess again



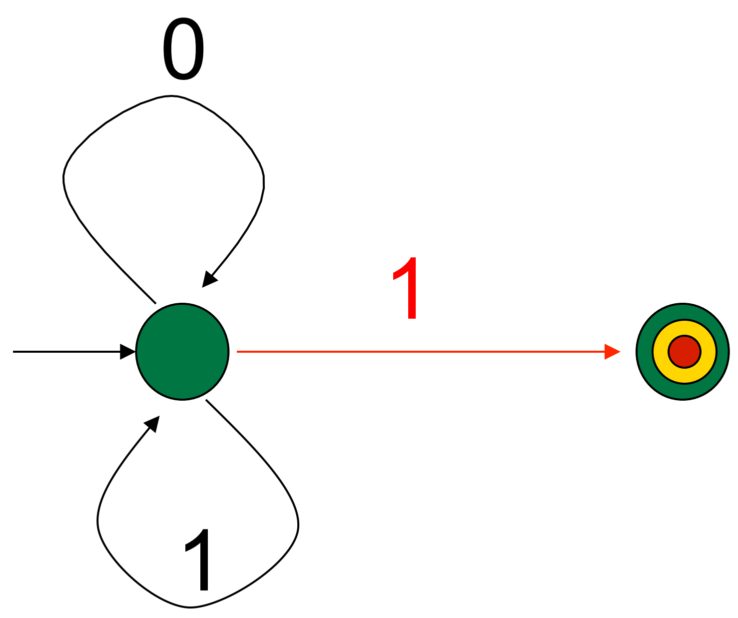
Example NFSA

- Regular expression: (0 ∨ 1)* 1
- Accepts string ~~0~~ ~~1~~ ~~1~~ ~~0~~ 1



Example NFSA

- Regular expression: (0 v 1)* 1
- Accepts string ~~0~~ ~~1~~ ~~1~~ ~~0~~ ~~1~~
- Guess (Hurray!!)



Rule Based Execution

- Search
- When stuck backtrack to last point with choices remaining
- Executing the NFSA in last example was example of rule based execution
- FSA's are rule-based programs; transitions between states (labeled edges) are rules; set of all FSA's is programming language

Rule Based Execution

- Search
- When stuck backtrack to last point with choices remaining

- FSA's are rule-based programs; transitions between states (labeled edges) are rules; set of all FSA's is programming language

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Unification Algorithm

- Let $S = \{(s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)\}$ be a unification problem.
- Case $S = \{\ \}$: Unif(S) = Identity function (i.e., no substitution)
- Case $S = \{(s, t)\} \cup S'$: Four main steps

Unification Algorithm

- Delete: if s = t (they are the same term) then Unif(S) = Unif(S')
- Decompose: if $s = f(q_1, \ldots, q_m)$ and $t = f(r_1, \ldots, r_m)$ (same f, same m!), then $\mathrm{Unif}(S) = \mathrm{Unif}(\{(q_1, r_1), \ldots, (q_m, r_m)\} \cup S')$
- Orient: if t = x is a variable, and s is not a variable, $\mathrm{Unif}(S) = \mathrm{Unif}(\{(x,s)\} \cup S')$

Unification Algorithm

- **Eliminate:** if s = x is a variable, and x does not occur in t (the occurs check), then
  - Let $\varphi = x \mid\!\rightarrow t$
  - Let $\psi = \text{Unif}(\varphi(S'))$
  - $\text{Unif}(S) = \{x \mid\!\rightarrow \psi(t)\} \circ \psi$
    - Note: $\{x \mid\!\rightarrow a\} \circ \{y \mid\!\rightarrow b\} = \{y \mid\!\rightarrow (\{x \mid\!\rightarrow a\}(b))\} \circ \{x \mid\!\rightarrow a\}$ if y not in a

Tricks for Efficient Unification

- Don't return substitution, rather do it incrementally
- Make substitution be constant time
  - Requires implementation of terms to use mutable structures (or possibly lazy structures)
  - We won't discuss these

Example

- x,y,z variables, f,g constructors
- $S = \{(f(x), f(g(y,z))), (g(y,f(y)), x)\}$

Example

- x,y,z variables, f,g constructors
- S is nonempty

- S = {(f(x), f(g(y,z))), (g(y,f(y)), x)}

Example

- x,y,z variables, f,g constructors
- Pick a pair: (g(y,f(y)), x)

- S = {(f(x), f(g(y,z))), (g(y,f(y)), x)}

Example

- x,y,z variables, f,g constructors
- Pick a pair: (g(y,f(y))), x)
- Orient: (x, g(y,f(y)))
- S = {(f(x), f(g(y,z))), (g(y,f(y)), x)}
- -> {(f(x), f(g(y,z))), (x, g(y,f(y)))}

Example

- x,y,z variables, f,g constructors
- S -> {(f(x), f(g(y,z))), (x, g(y,f(y)))}

Example

- x,y,z variables, f,g constructors
- Pick a pair: (f(x), f(g(y,z)))

- S -> {(f(x), f(g(y,z))), (x, g(y,f(y)))}

Example

- x,y,z variables, f,g constructors
- Pick a pair: (f(x), f(g(y,z)))
- Decompose: (x, g(y,z))
- S -> {(f(x), f(g(y,z))), (x, g(y,f(y)))}
- -> {(x, g(y,z)), (x, g(y,f(y)))}

Example

- x,y,z variables, f,g constructors
- Pick a pair: (x, g(y,f(y)))
- Substitute: {x |-> g(y,f(y))}
- S -> {(x, g(y,z)), (x, g(y,f(y)))}
- -> {(g(y,f(y)), g(y,z))}

- With {x |-> g(y,f(y))}

Example

- x,y,z variables, f,g constructors
- Pick a pair: (g(y,f(y)), g(y,z))

- S -> {(g(y,f(y)), g(y,z))}

With {x |→ g(y,f(y))}

Example

- x,y,z variables, f,g constructors
- Pick a pair: (g(y,f(y)), g(y,z))
- Decompose: (y, y) and (f(y), z)
- S -> {(g(y,f(y)), g(y,z))}
- -> {(y, y), (f(y), z)}

With {x $\mapsto$ g(y,f(y))}

Example

- x,y,z variables, f,g constructors
- Pick a pair: (y, y)

- S -> {(y, y), (f(y), z)}

With $\{x \mapsto g(y,f(y))\}$

Example

- x,y,z variables, f,g constructors
- Pick a pair: (y, y)
- Delete
- S -> {(y, y), (f(y), z)}
- -> {(f(y), z)}

With {x $\mapsto$ g(y,f(y))}

Example

- x,y,z variables, f,g constructors
- Pick a pair: (f(y), z)
- S -> {(f(y), z)}

With {x $\mapsto$ g(y,f(y))}

Example

- x,y,z variables, f,g constructors
- Pick a pair: (f(y), z)
- Orient: (z, f(y))
- S -> {(f(y), z)}
- -> {(z, f(y))}

With $\{x \mapsto g(y,f(y))\}$

Example

- x,y,z variables, f,g constructors
- Pick a pair: (z, f(y))

- S -> {(z, f(y))}

With {x |→ g(y,f(y))}

Example

- x,y,z variables, f,g constructors
- Pick a pair: (z, f(y))
- Eliminate: {z|-> f(y)}
- S -> {(z, f(y))}
- -> { }

With $\{x \mapsto \{z \mapsto f(y)\}\ (g(y,f(y)))\ \}$

$\circ\ \{z \mapsto f(y)\}$

Example

- x,y,z variables, f,g constructors
- Pick a pair: (z, f(y))
- Eliminate: {z|-> f(y)}
- S -> {(z, f(y))}
- -> { }

With {x |→ g(y,f(y))} o {(z |→ f(y))}

Example

$S = \{(f(x), f(g(y,z))), (g(y,f(y)),x)\}$

Solved by $\{x \mid\!\rightarrow g(y,f(y))\}$ o $\{(z \mid\!\rightarrow f(y))\}$

$$f(\underbrace{g(y,f(y))}_{x}) = f(g(y,\underbrace{f(y)}_{z}))$$

and

$$g(y,f(y)) = \underbrace{g(y,f(y))}_{x}$$

Example of Failure: Decompose

- S = {(f(x,g(y)), f(h(y),x))}
- Decompose: (f(x,g(y)), f(h(y),x))
- S -> {(x,h(y)), (g(y),x)}
- Orient: (g(y),x)
- S -> {(x,h(y)), (x,g(y))}
- Eliminate: (x,h(y))
- S -> {(h(y), g(y))} with {x |→ h(y)}
- No rule to apply! Decompose fails!

Example of Failure: Occurs Check

- S = {(f(x,g(x)), f(h(x),x))}
- Decompose: (f(x,g(x)), f(h(x),x))
- S -> {(x,h(x)), (g(x),x)}
- Orient: (g(y),x)
- S -> {(x,h(x)), (x,g(x))}
- No rules apply.

Where We Are Going

- We want to turn strings (code) into computer instructions
- Done in phases
- Turn strings into abstract syntax trees (parse)
- Translate abstract syntax trees into executable instructions (interpret or compile)

Major Phases of a Compiler

Source Program

Lex

Tokens

Parse

Abstract Syntax

Semantic Analysis

Symbol Table

Translate

Intermediate Representation

Optimize

Optimized IR

Instruction Selection

Unoptimized Machine -Specific Assembly Language

Optimize

Optimized Machine-Specific Assembly Language

Emit code

Assembly Language

Assembler

Relocatable Object Code

Linker

Machine Code

Modified from "Modern Compiler Implementation in ML", by Andrew Appel

Meta-discourse

- Language Syntax and Semantics
- Syntax
  - Regular Expressions, DFSAs and NDFSAs
  - Grammars
- Semantics
  - Natural Semantics
  - Transition Semantics

Language Syntax

- Syntax is the description of which strings of symbols are meaningful expressions in a language
- It takes more than syntax to understand a language; need meaning (semantics) too
- Syntax is the entry point

Syntax of English Language

- Pattern 1

Subject	Verb
David	*sings*
The dog	*barked*
Susan	*yawned*

- Pattern 2

Subject	Verb	Direct Object
David	*sings*	*ballads*
The professor	*wants*	*to retire*
The jury	*found*	*the defendant guilty*

Elements of Syntax

- Character set – previously always ASCII, now often 64 character sets
- Keywords – usually reserved
- Special constants – cannot be assigned to
- Identifiers – can be assigned to
- Operator symbols
- Delimiters (parenthesis, braces, brackets)
- Blanks (aka white space)

Elements of Syntax

- Expressions

  if ... then begin ... ; ... end else begin ... ; ... end

- Type expressions

  $typexpr_1$ -> $typexpr_2$

- Declarations (in functional languages)

  let $pattern_1$ = $expr_1$ in $expr$

- Statements (in imperative languages)

  a = b + c

- Subprograms

  let $pattern_1$ = let rec inner = … in $expr$

Elements of Syntax

- Modules
- Interfaces
- Classes (for object-oriented languages)

Lexing and Parsing

- Converting strings to abstract syntax trees done in two phases
  - **Lexing:** Converting string (or streams of characters) into lists (or streams) of tokens (the “words” of the language)
    - Specification Technique: Regular Expressions
  - **Parsing:** Convert a list of tokens into an abstract syntax tree
    - Specification Technique: BNF Grammars

Formal Language Descriptions

- Regular expressions, regular grammars, finite state automata
- Context-free grammars, BNF grammars, syntax diagrams
- Whole family more of grammars and automata – covered in automata theory

Grammars

- Grammars are formal descriptions of which strings over a given character set are in a particular language
- Language designers write grammar
- Language implementers use grammar to know what programs to accept
- Language users use grammar to know how to write legitimate programs

Regular Expressions - Review

- Start with a given character set – **a, b, c**…

- Each character is a regular expression
  - It represents the set of one string containing just that character

Regular Expressions

- If **x** and **y** are regular expressions, then **xy** is a regular expression
  - It represents the set of all strings made from first a string described by **x** then a string described by **y**

If x={a,ab} and y={c,d} then xy ={ac,ad,abc,abd}.

- If **x** and **y** are regular expressions, then **x**∨**y** is a regular expression
  - It represents the set of strings described by either **x** or **y**

If x={a,ab} and y={c,d} then x ∨ y={a,ab,c,d}

Regular Expressions

- If **x** is a regular expression, then so is **(x)**
  - It represents the same thing as **x**

- If **x** is a regular expression, then so is **x***
  - It represents strings made from concatenating zero or more strings from **x**

  If x = {a,ab}

  then x* ={“”,a,ab,aa,aab,abab,aaa,aaab,...}

- **ε**
  - It represents {“”}, set containing the empty string

Example Regular Expressions

- **(0**v**1)*1**
  - The set of all strings of **0**'s and **1**'s ending in 1, **{1, 01, 11,…}**
- **a*b(a*)**
  - The set of all strings of a's and b's with exactly one b
- **((01)** v**(10))***
  - You tell me
- Regular expressions (equivalently, regular grammars) important for lexing, breaking strings into recognized words

Example: Lexing

- Regular expressions good for describing lexemes (words) in a programming language
  - Identifier = (a ∨ b ∨ … ∨ z ∨ A ∨ B ∨ … ∨ Z) (a ∨ b ∨ … ∨ z ∨ A ∨ B ∨ … ∨ Z ∨ 0 ∨ 1 ∨ … ∨ 9)*
  - Digit = (0 ∨ 1 ∨ … ∨ 9)
  - Number = 0 ∨ (1 ∨ … ∨ 9)(0 ∨ … ∨ 9)* ∨ ~ (1 ∨ … ∨ 9)(0 ∨ … ∨ 9)*
  - Keywords: if = if, while = while,…

Implementing Regular Expressions

- Regular expressions reasonable way to generate strings in language
- Not so good for recognizing when a string is in language
- Problems with Regular Expressions
  - which option to choose,
  - how many repetitions to make
- Answer: finite state automata
- Should have covered this in CS373

Lexing

- Different syntactic categories of “words”: tokens

Example:

- Convert sequence of characters into sequence of strings, integers, and floating point numbers.
- "asd 123 jkl 3.14" will become:

  [String "asd"; Int 123; String "jkl"; Float 3.14]

Lex, ocamllex

- Could write the reg exp, then translate to DFA by hand
  - A lot of work
- Better: Write program to take reg exp as input and automatically generates automata
- Lex is such a program
- ocamllex version for ocaml

How to do it

- To use regular expressions to parse our input we need:
  - Some way to identify the input string — call it a lexing buffer
  - Set of regular expressions,
  - Corresponding set of actions to take when they are matched.

How to do it

- The lexer will take the regular expressions and generate a state machine.
- The state machine will take our lexing buffer and apply the transitions...
- If we reach an accepting state from which we can go no further, the machine will perform the appropriate action.

Mechanics

- Put table of reg exp and corresponding actions (written in ocaml) into a file <*filename*>.mll
- Call

  ocamllex <*filename*>.mll

- Produces Ocaml code for a lexical analyzer in file <*filename*>.ml

Sample Input

```
rule main = parse
 ['0'-'9']+ { print_string "Int\n"}
 | ['0'-'9']+'.'['0'-'9']+ { print_string "Float\n"}
 | ['a'-'z']+ { print_string "String\n"}
 | _ { main lexbuf }
 {
 let newlexbuf = (Lexing.from_channel stdin) in
 print_string "Ready to lex.\n";
 main newlexbuf
}
```

General Input

{ *header* }

let *ident* = *regexp* …

rule *entrypoint* [*arg1*… *argn*] = parse

regexp { *action* }

| …

| *regexp* { *action* }

and *entrypoint* [*arg1*… *argn*] = parse …and …

{ *trailer* }

Ocamllex Input

- *header* and *trailer* contain arbitrary ocaml code put at top an bottom of *<filename>*.ml

- let *ident = regexp* ... Introduces *ident* for use in later regular expressions

Ocamllex Input

- *<filename>*.ml contains one lexing function per *entrypoint*
  - Name of function is name given for *entrypoint*
  - Each entry point becomes an Ocaml function that takes *n*+1 arguments, the extra implicit last argument being of type Lexing.lexbuf
- *arg1*... a*rgn* are for use in *action*

Ocamllex Regular Expression

- Single quoted characters for letters: ‘a’
- _: (underscore) matches any letter
- Eof: special “end_of_file” marker
- Concatenation same as usual
- “*string*”: concatenation of sequence of characters
- $e_1 \mid e_2$: choice - what was $e_1 \vee e_2$

Ocamllex Regular Expression

- $[c_1 - c_2]$: choice of any character between first and second inclusive, as determined by character codes
- $[\verb|^|c_1 - c_2]$: choice of any character NOT in set
- $e*$: same as before
- $e+$: same as $e\ e*$
- $e?$: option - was $e_1 \vee \varepsilon$

Ocamllex Regular Expression

- e_1 # e_2: the characters in e_1 but not in e_2; e_1 and e_2 must describe just sets of characters
- *ident*: abbreviation for earlier reg exp in let *ident* = *regexp*
- e_1 as *id*: binds the result of e_1 to *id* to be used in the associated *action*

Ocamllex Manual

- More details can be found at

http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html

Example : test.mll

{ type result = Int of int | Float of float | String of string }

let digit = ['0'-'9']

let digits = digit +

let lower_case = ['a'-'z']

let upper_case = ['A'-'Z']

let letter = upper_case | lower_case

let letters = letter +

Example : test.mll

```
rule main = parse
   (digits)'.'digits as f  { Float (float_of_string f) }
 | digits as n               { Int (int_of_string n) }
 | letters as s              { String s}
 | _ { main lexbuf }
 { let newlexbuf = (Lexing.from_channel stdin) in
 print_string "Ready to lex.";
 print_newline ();
 main newlexbuf  }
```

Example

#use "test.ml";;

…

val main : Lexing.lexbuf -> result = <fun>

val __ocaml_lex_main_rec : Lexing.lexbuf -> int -> result = <fun>

Ready to lex.

hi there 234 5.2

- : result = String "hi"

What happened to the rest?!?

Example

```
# let b = Lexing.from_channel stdin;;
# main b;;
hi 673 there
- : result = String "hi"
# main b;;
- : result = Int 673
# main b;;
- : result = String "there"
```

Problem

- How to get lexer to look at more than the first token at one time?
- Answer: *action* has to tell it to -- recursive calls
- Side Benefit: can add "state" into lexing
- Note: already used this with the _ case

Example

```
rule main = parse
  (digits) '.' digits as f { Float
  (float_of_string f) :: main lexbuf}
 | digits as n        { Int (int_of_string n) ::
  main lexbuf }
 | letters as s       { String s :: main
  lexbuf}
 | eof                 { [] }
 | _                   { main lexbuf }
```

Example Results

Ready to lex.

hi there 234 5.2

- : result list = [String "hi"; String "there"; Int 234; Float 5.2]

#

Used Ctrl-d to send the end-of-file signal

Dealing with comments

First Attempt

```
let open_comment = "(*"
let close_comment = "*)"
rule main = parse
  (digits) '.' digits as f { Float (float_of_string
  f) :: main lexbuf}
 | digits as n          { Int (int_of_string n) ::
  main lexbuf }
 | letters as s         { String s :: main lexbuf}
```

Dealing with comments

```
 | open_comment          { comment  lexbuf}
 | eof                   { [] }
 | _ { main lexbuf }
and comment = parse
   close_comment         { main lexbuf }
 | _                     { comment lexbuf }
```

Dealing with nested comments

```
rule main = parse …
 | open_comment        { comment 1 lexbuf}
 | eof              { [] }
 | _ { main lexbuf }
and comment depth = parse
   open_comment        { comment (depth+1)
   lexbuf }
 | close_comment       { if depth = 1
                  then main lexbuf
                  else comment (depth - 1) lexbuf }
 | _               { comment depth lexbuf }
```

Dealing with nested comments

```
rule main = parse
  (digits) '.' digits as f { Float (float_of_string f) :: main lexbuf}
 | digits as n              { Int (int_of_string n) :: main lexbuf }
 | letters as s             { String s :: main lexbuf}
 | open_comment             { (comment 1 lexbuf}
 | eof                      { [] }
 | _ { main lexbuf }
```

Dealing with nested comments

```
and comment depth = parse
   open_comment        { comment (depth+1) lexbuf }
 | close_comment       { if depth = 1
                         then main lexbuf
                         else comment (depth - 1) lexbuf }
 | _                   { comment depth lexbuf }
```

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Lexing and Parsing

- Converting strings to abstract syntax trees done in two phases
  - **Lexing:** Converting string (or streams of characters) into lists (or streams) of tokens (the “words” of the language)
    - Specification Technique: Regular Expressions
  - **Parsing:** Convert a list of tokens into an abstract syntax tree
    - Specification Technique: BNF Grammars

Formal Language Descriptions

- Regular expressions, regular grammars, finite state automata
- Context-free grammars, BNF grammars, syntax diagrams
- Whole family more of grammars and automata – covered in automata theory

Grammars

- Grammars are formal descriptions of which strings over a given character set are in a particular language
- Language designers write grammar
- Language implementers use grammar to know what programs to accept
- Language users use grammar to know how to write legitimate programs

Regular Expressions - Review

- Start with a given character set – **a, b, c**…
- Each character is a regular expression
  - It represents the set of one string containing just that character

Regular Expressions

- If **x** and **y** are regular expressions, then **xy** is a regular expression
  - It represents the set of all strings made from first a string described by **x** then a string described by **y**

  If x={a,ab} and y={c,d} then xy ={ac,ad,abc,abd}.
- If **x** and **y** are regular expressions, then **x**∨**y** is a regular expression
  - It represents the set of strings described by either **x** or **y**

    If x={a,ab} and y={c,d} then x ∨ y={a,ab,c,d}

Regular Expressions

- If **x** is a regular expression, then so is **(x)**
  - It represents the same thing as **x**

- If **x** is a regular expression, then so is **x***
  - It represents strings made from concatenating zero or more strings from **x**

  If x = {a,ab}

  then x* ={“”,a,ab,aa,aab,abab,aaa,aaab,…}

- **ε**
  - It represents {“”}, set containing the empty string

Example Regular Expressions

- **(0**∨**1)*1**
  - The set of all strings of **0**'s and **1**'s ending in 1, **{1, 01, 11,...}**
- **a*b(a*)**
  - The set of all strings of a's and b's with exactly one b
- **((01)** ∨**(10))***
  - You tell me
- Regular expressions (equivalently, regular grammars) important for lexing, breaking strings into recognized words

Example: Lexing

- Regular expressions good for describing lexemes (words) in a programming language
  - Identifier = (a ∨ b ∨ … ∨ z ∨ A ∨ B ∨ … ∨ Z) (a ∨ b ∨ … ∨ z ∨ A ∨ B ∨ … ∨ Z ∨ 0 ∨ 1 ∨ … ∨ 9)*
  - Digit = (0 ∨ 1 ∨ … ∨ 9)
  - Number = 0 ∨ (1 ∨ … ∨ 9)(0 ∨ … ∨ 9)* ∨ ~ (1 ∨ … ∨ 9)(0 ∨ … ∨ 9)*
  - Keywords: if = if, while = while,…

Implementing Regular Expressions

- Regular expressions reasonable way to generate strings in language
- Not so good for recognizing when a string is in language
- Problems with Regular Expressions
  - which option to choose,
  - how many repetitions to make
- Answer: finite state automata
- Should have covered this in CS373

Lexing

- Different syntactic categories of “words”: tokens

Example:

- Convert sequence of characters into sequence of strings, integers, and floating point numbers.
- "asd 123 jkl 3.14" will become:

  [String "asd"; Int 123; String "jkl"; Float 3.14]

Lex, ocamllex

- Could write the reg exp, then translate to DFA by hand
  - A lot of work
- Better: Write program to take reg exp as input and automatically generates automata
- Lex is such a program
- ocamllex version for ocaml

How to do it

- To use regular expressions to parse our input we need:
  - Some way to identify the input string — call it a lexing buffer
  - Set of regular expressions,
  - Corresponding set of actions to take when they are matched.

How to do it

- The lexer will take the regular expressions and generate a state machine.
- The state machine will take our lexing buffer and apply the transitions...
- If we reach an accepting state from which we can go no further, the machine will perform the appropriate action.

Mechanics

- Put table of reg exp and corresponding actions (written in ocaml) into a file <*filename*>.mll
- Call

  ocamllex <*filename*>.mll
- Produces Ocaml code for a lexical analyzer in file <*filename*>.ml

Sample Input

```
rule main = parse
 ['0'-'9']+ { print_string "Int\n"}
 | ['0'-'9']+'.'['0'-'9']+ { print_string "Float\n"}
 | ['a'-'z']+ { print_string "String\n"}
 | _ { main lexbuf }
 {
 let newlexbuf = (Lexing.from_channel stdin) in
 print_string "Ready to lex.\n";
 main newlexbuf
}
```

General Input

{ *header* }

let *ident* = *regexp* ...

rule *entrypoint* [*arg1*... *argn*] = parse

regexp { *action* }

| ...

| *regexp* { *action* }

and *entrypoint* [*arg1*... *argn*] = parse ...and ...

{ *trailer* }

Ocamllex Input

- *header* and *trailer* contain arbitrary ocaml code put at top an bottom of *<filename>*.ml
- let *ident* = *regexp* ... Introduces *ident* for use in later regular expressions

Ocamllex Input

- *<filename>*.ml contains one lexing function per *entrypoint*
  - Name of function is name given for *entrypoint*
  - Each entry point becomes an Ocaml function that takes *n*+1 arguments, the extra implicit last argument being of type Lexing.lexbuf
- *arg1*... a*rgn* are for use in *action*

Ocamllex Regular Expression

- Single quoted characters for letters: ‘a’
- _: (underscore) matches any letter
- Eof: special “end_of_file” marker
- Concatenation same as usual
- “*string*”: concatenation of sequence of characters
- $e_1 \mid e_2$: choice - what was $e_1 \vee e_2$

Ocamllex Regular Expression

- $[c_1 - c_2]$: choice of any character between first and second inclusive, as determined by character codes
- $[\wedge c_1 - c_2]$: choice of any character NOT in set
- $e*$: same as before
- $e+$: same as $e\ e*$
- $e?$: option - was $e_1 \vee \varepsilon$

Ocamllex Regular Expression

- e_1 # e_2: the characters in e_1 but not in e_2; e_1 and e_2 must describe just sets of characters
- *ident*: abbreviation for earlier reg exp in let *ident* = *regexp*
- e_1 as *id*: binds the result of e_1 to *id* to be used in the associated *action*

Ocamllex Manual

- More details can be found at

http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html

Example : test.mll

```
{ type result = Int of int | Float of float |
  String of string }
let digit = ['0'-'9']
let digits = digit +
let lower_case = ['a'-'z']
let upper_case = ['A'-'Z']
let letter = upper_case | lower_case
let letters = letter +
```

Example : test.mll

```
rule main = parse
   (digits)'.'digits as f  { Float (float_of_string f) }
 | digits as n              { Int (int_of_string n) }
 | letters as s             { String s}
 | _ { main lexbuf }
 { let newlexbuf = (Lexing.from_channel stdin) in
 print_string "Ready to lex.";
 print_newline ();
 main newlexbuf  }
```

Example

#use "test.ml";;

...

val main : Lexing.lexbuf -> result = <fun>

val __ocaml_lex_main_rec : Lexing.lexbuf -> int -> result = <fun>

Ready to lex.

hi there 234 5.2

- : result = String "hi"

What happened to the rest?!?

Example

```
# let b = Lexing.from_channel stdin;;
# main b;;
hi 673 there
- : result = String "hi"
# main b;;
- : result = Int 673
# main b;;
- : result = String "there"
```

Problem

- How to get lexer to look at more than the first token at one time?
- Answer: *action* has to tell it to -- recursive calls
- Side Benefit: can add “state” into lexing
- Note: already used this with the _ case

Example

```
rule main = parse
  (digits) '.' digits as f { Float
  (float_of_string f) :: main lexbuf}
 | digits as n            { Int (int_of_string n) ::
  main lexbuf }
 | letters as s           { String s :: main
  lexbuf}
 | eof                     { [] }
 | _                       { main lexbuf }
```

Example Results

Ready to lex.

hi there 234 5.2

- : result list = [String "hi"; String "there"; Int 234; Float 5.2]

#

Used Ctrl-d to send the end-of-file signal

Dealing with comments

First Attempt

```
let open_comment = "(*"
let close_comment = "*)"
rule main = parse
  (digits) '.' digits as f { Float (float_of_string
  f) :: main lexbuf}
 | digits as n          { Int (int_of_string n) ::
  main lexbuf }
 | letters as s         { String s :: main lexbuf}
```

Dealing with comments

```
 | open_comment        { comment  lexbuf}
 | eof                 { [] }
 | _ { main lexbuf }
and comment = parse
   close_comment       { main lexbuf }
 | _                   { comment lexbuf }
```

Dealing with nested comments

```
rule main = parse …
 | open_comment          { comment 1 lexbuf}
 | eof                   { [] }
 | _ { main lexbuf }
and comment depth = parse
   open_comment          { comment (depth+1)
   lexbuf }
 | close_comment         { if depth = 1
                           then main lexbuf
                           else comment (depth - 1) lexbuf }
 | _                     { comment depth lexbuf }
```

Dealing with nested comments

```
rule main = parse
  (digits) '.' digits as f { Float (float_of_string f) ::
  main lexbuf}
 | digits as n          { Int (int_of_string n) :: main
  lexbuf }
 | letters as s         { String s :: main lexbuf}
 | open_comment           { (comment 1 lexbuf}
 | eof                 { [] }
 | _ { main lexbuf }
```

Dealing with nested comments

```
and comment depth = parse
   open_comment        { comment (depth+1) lexbuf }
 | close_comment       { if depth = 1
                          then main lexbuf
                          else comment (depth - 1) lexbuf }
 | _                   { comment depth lexbuf }
```

Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

General Input

{ *header* }

let *ident* = *regexp* …

rule *entrypoint* [*arg1*… *argn*] = parse

regexp { *action* }

| …

| *regexp* { *action* }

and *entrypoint* [*arg1*… *argn*] = parse …and …

{ *trailer* }

Ocamllex Input

- *header* and *trailer* contain arbitrary ocaml code put at top an bottom of *<filename>*.ml

- let *ident* = *regexp* ... Introduces *ident* for use in later regular expressions

Ocamllex Input

- *<filename>*.ml contains one lexing function per *entrypoint*
  - Name of function is name given for *entrypoint*
  - Each entry point becomes an Ocaml function that takes *n*+1 arguments, the extra implicit last argument being of type Lexing.lexbuf
- *arg1*... a*rgn* are for use in *action*

Ocamllex Regular Expression

- Single quoted characters for letters: 'a'
- _: (underscore) matches any letter
- Eof: special "end_of_file" marker
- Concatenation same as usual
- "*string*": concatenation of sequence of characters
- e_1 / e_2 : choice - what was $e_1 \vee e_2$

Ocamllex Regular Expression

- $[c_1 - c_2]$: choice of any character between first and second inclusive, as determined by character codes
- $[\verb|^| c_1 - c_2]$: choice of any character NOT in set
- $e*$: same as before
- $e+$: same as $e\ e*$
- $e?$: option - was $e_1 \vee \varepsilon$

Ocamllex Regular Expression

- *e_1 # e_2*: the characters in e_1 but not in e_2; e_1 and e_2 must describe just sets of characters
- *ident*: abbreviation for earlier reg exp in let *ident = regexp*
- e_1 as *id*: binds the result of e_1 to *id* to be used in the associated *action*

Ocamllex Manual

- More details can be found at

http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html

Example : test.mll

{ type result = Int of int | Float of float | String of string }

let digit = ['0'-'9']

let digits = digit +

let lower_case = ['a'-'z']

let upper_case = ['A'-'Z']

let letter = upper_case | lower_case

let letters = letter +

Example : test.mll

```
rule main = parse
   (digits)'.'digits as f  { Float (float_of_string f) }
 | digits as n              { Int (int_of_string n) }
 | letters as s             { String s}
 | _ { main lexbuf }
 { let newlexbuf = (Lexing.from_channel stdin) in
 print_string "Ready to lex.";
 print_newline ();
 main newlexbuf  }
```

Example

```
# #use "test.ml";;
…
val main : Lexing.lexbuf -> result = <fun>
val __ocaml_lex_main_rec : Lexing.lexbuf -> int ->
  result = <fun>
Ready to lex.
hi there 234 5.2
- : result = String "hi"
```

What happened to the rest?!?

Example

```
# let b = Lexing.from_channel stdin;;
# main b;;
hi 673 there
- : result = String "hi"
# main b;;
- : result = Int 673
# main b;;
- : result = String "there"
```

Problem

- How to get lexer to look at more than the first token at one time?
- Answer: *action* has to tell it to -- recursive calls
- Side Benefit: can add “state” into lexing
- Note: already used this with the _ case

Example

```
rule main = parse
  (digits) '.' digits as f { Float
  (float_of_string f) :: main lexbuf}
 | digits as n          { Int (int_of_string n) ::
  main lexbuf }
 | letters as s         { String s :: main
  lexbuf}
 | eof                   { [] }
 | _                     { main lexbuf }
```

Example Results

Ready to lex.

hi there 234 5.2

- : result list = [String "hi"; String "there"; Int 234; Float 5.2]

#

Used Ctrl-d to send the end-of-file signal

Dealing with comments

First Attempt

```
let open_comment = "(*"
let close_comment = "*)"
rule main = parse
   (digits) '.' digits as f { Float (float_of_string
   f) :: main lexbuf}
 | digits as n          { Int (int_of_string n) ::
   main lexbuf }
 | letters as s         { String s :: main lexbuf}
```

Dealing with comments

```
 | open_comment          { comment  lexbuf}
 | eof                   { [] }
 | _ { main lexbuf }
and comment = parse
   close_comment         { main lexbuf }
 | _                     { comment lexbuf }
```

Dealing with nested comments

```
rule main = parse ...
 | open_comment          { comment 1 lexbuf}
 | eof                   { [] }
 | _ { main lexbuf }
and comment depth = parse
   open_comment          { comment (depth+1)
   lexbuf }
 | close_comment         { if depth = 1
                           then main lexbuf
                           else comment (depth - 1) lexbuf }
 | _                     { comment depth lexbuf }
```

Dealing with nested comments

```
rule main = parse
  (digits) '.' digits as f { Float (float_of_string f) :: main lexbuf}
 | digits as n            { Int (int_of_string n) :: main lexbuf }
 | letters as s           { String s :: main lexbuf}
 | open_comment              { (comment 1 lexbuf}
 | eof                  { [] }
 | _ { main lexbuf }
```

Dealing with nested comments

```
and comment depth = parse
   open_comment         { comment (depth+1) lexbuf }
 | close_comment        { if depth = 1
                          then main lexbuf
                          else comment (depth - 1) lexbuf }
 | _                    { comment depth lexbuf }
```

Types of Formal Language Descriptions

- Regular expressions, regular grammars
- Context-free grammars, BNF grammars, syntax diagrams
- Finite state automata

- Whole family more of grammars and automata – covered in automata theory

Sample Grammar

- Language: Parenthesized sums of 0’s and 1’s

- <Sum> ::= 0
- <Sum >::= 1
- <Sum> ::= <Sum> + <Sum>
- <Sum> ::= (<Sum>)

BNF Grammars

- Start with a set of characters, **a,b,c,...**
  - We call these *terminals*
- Add a set of different characters, **X,Y, Z,...**
  - We call these *nonterminals*
- One special nonterminal **S** called *start symbol*

BNF Grammars

- BNF rules (aka *productions*) have form

  X ::= *y*

  where **X** is any nonterminal and *y* is a string of terminals and nonterminals
- BNF *grammar* is a set of BNF rules such that every nonterminal appears on the left of some rule

Sample Grammar

- Terminals: 0 1 + ( )
- Nonterminals: <Sum>
- Start symbol = <Sum>

- <Sum> ::= 0
- <Sum >::= 1
- <Sum> ::= <Sum> + <Sum>
- <Sum> ::= (<Sum>)
- Can be abbreviated as

<Sum> ::= 0 | 1
          | <Sum> + <Sum> | (<Sum>)

BNF Deriviations

- Given rules

$$\mathbf{X}::= y\mathbf{Z}w \text{ and } \mathbf{Z}::= v$$

we may replace **Z** by *v* to say

$$\mathbf{X} \Rightarrow y\mathbf{Z}w \Rightarrow yvw$$

- Sequence of such replacements called *derivation*
- Derivation called *right-most* if always replace the right-most non-terminal

BNF Derivations

- Start with the start symbol:

<Sum> =>

BNF Derivations

- Pick a non-terminal

<Sum> =>

BNF Derivations

- Pick a rule and substitute:
  - <Sum> ::= <Sum> + <Sum>

<Sum> => <Sum> + <Sum >

BNF Derivations

- Pick a non-terminal:

<Sum> => <Sum> + <Sum >

BNF Derivations

- Pick a rule and substitute:
  - <Sum> ::= (<Sum>)

<Sum> => <Sum> + <Sum >

=> (<Sum>) + <Sum>

BNF Derivations

- Pick a non-terminal:

<Sum> => <Sum> + <Sum >

=> (<Sum>) + <Sum>

BNF Derivations

- Pick a rule and substitute:
  - <Sum> ::= <Sum> + <Sum>

<Sum> => <Sum> + <Sum >

=> (<Sum>) + <Sum>

=> (<Sum> + <Sum>) + <Sum>

BNF Derivations

- Pick a non-terminal:

<Sum> => <Sum> + <Sum >
=> (<Sum>) + <Sum>
=> (<Sum> + <Sum>) + <Sum>

BNF Derivations

- Pick a rule and substitute:
  - <Sum >::= 1

<Sum> => <Sum> + <Sum >

=> (<Sum>) + <Sum>

=> (<Sum> + <Sum>) + <Sum>

=> (<Sum> + 1) + <Sum>

BNF Derivations

- Pick a non-terminal:

<Sum> => <Sum> + <Sum >
=> (<Sum>) + <Sum>
=> (<Sum> + <Sum>) + <Sum>
=> (<Sum> + 1) + <Sum>

BNF Derivations

- Pick a rule and substitute:
  - <Sum >::= 0

<Sum> => <Sum> + <Sum >

=> (<Sum>) + <Sum>

=> (<Sum> + <Sum>) + <Sum>

=> (<Sum> + 1) + <Sum>

=> (<Sum> + 1) + 0

BNF Derivations

- Pick a non-terminal:

<Sum> => <Sum> + <Sum >
=> (<Sum>) + <Sum>
=> (<Sum> + <Sum>) + <Sum>
=> (<Sum> + 1) + <Sum>
=> (<Sum> + 1) + 0

BNF Derivations

- Pick a rule and substitute
  - <Sum> ::= 0

<Sum> => <Sum> + <Sum >
=> (<Sum>) + <Sum>
=> (<Sum> + <Sum>) + <Sum>
=> (<Sum> + 1) + <Sum>
=> (<Sum> + 1) 0
=> (0 + 1) + 0

BNF Derivations

- (0 + 1) + 0 is generated by grammar

<Sum> => <Sum> + <Sum >
=> (<Sum>) + <Sum>
=> (<Sum> + <Sum>) + <Sum>
=> (<Sum> + 1) + <Sum>
=> (<Sum> + 1) + 0
=> (0 + 1) + 0

<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

<Sum> =>

BNF Semantics

- The meaning of a BNF grammar is the set of all strings consisting only of terminals that can be derived from the Start symbol

Extended BNF Grammars

- Alternatives: allow rules of from X::=*y*|*z*
  - Abbreviates X::= *y*, X*::= z*
- Options: X::=*y*[*v*]*z*
  - Abbreviates X::=*yvz*, X::=*yz*
- Repetition: X::=*y*{*v*}**z*
  - Can be eliminated by adding new nonterminal V and rules X::=*yz*, X::=yVz, V::=*v*, V::=*v*V

Regular Grammars

- Subclass of BNF
- Only rules of form
  <nonterminal>::=<terminal><nonterminal> or
  <nonterminal>::=<terminal> or
  <nonterminal>::=ε
- Defines same class of languages as regular expressions
- Important for writing lexers (programs that convert strings of characters into strings of tokens)

Example

- Regular grammar:

  <Balanced> ::= ε

  <Balanced> ::= 0<OneAndMore>

  <Balanced> ::= 1<ZeroAndMore>

  <OneAndMore> ::= 1<Balanced>

  <ZeroAndMore> ::= 0<Balanced>

- Generates even length strings where every initial substring of even length has same number of 0's as 1's

Parse Trees

- Graphical representation of derivation
- Each node labeled with either non-terminal or terminal
- If node is labeled with a terminal, then it is a leaf (no sub-trees)
- If node is labeled with a non-terminal, then it has one branch for each character in the right-hand side of rule used to substitute for it

Example

- Consider grammar:

  <exp> ::= <factor>
  | <factor> + <factor>
  <factor> ::= <bin>
  | <bin> * <exp>
  <bin> ::= 0 | 1

- Problem: Build parse tree for 1 * 1 + 0 as an <exp>

Example cont.

- 1 * 1 + 0:   <exp>

<exp> is the start symbol for this parse tree

Example cont.

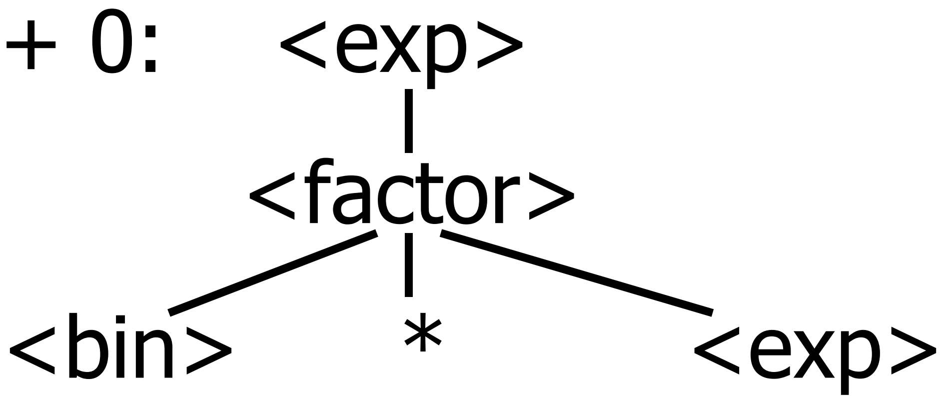
- 1 * 1 + 0: <exp>

$$\begin{array}{c} \langle exp \rangle \\ | \\ \langle factor \rangle \end{array}$$

Use rule: <exp> ::= <factor>

Example cont.

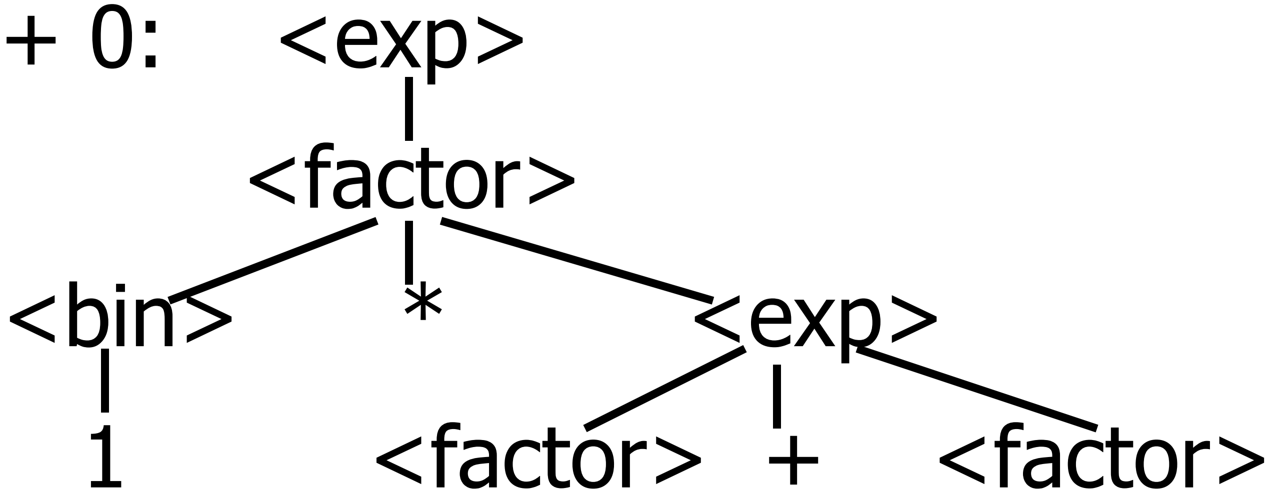
- 1 * 1 + 0:



Use rule: <factor> ::= <bin> * <exp>

Example cont.

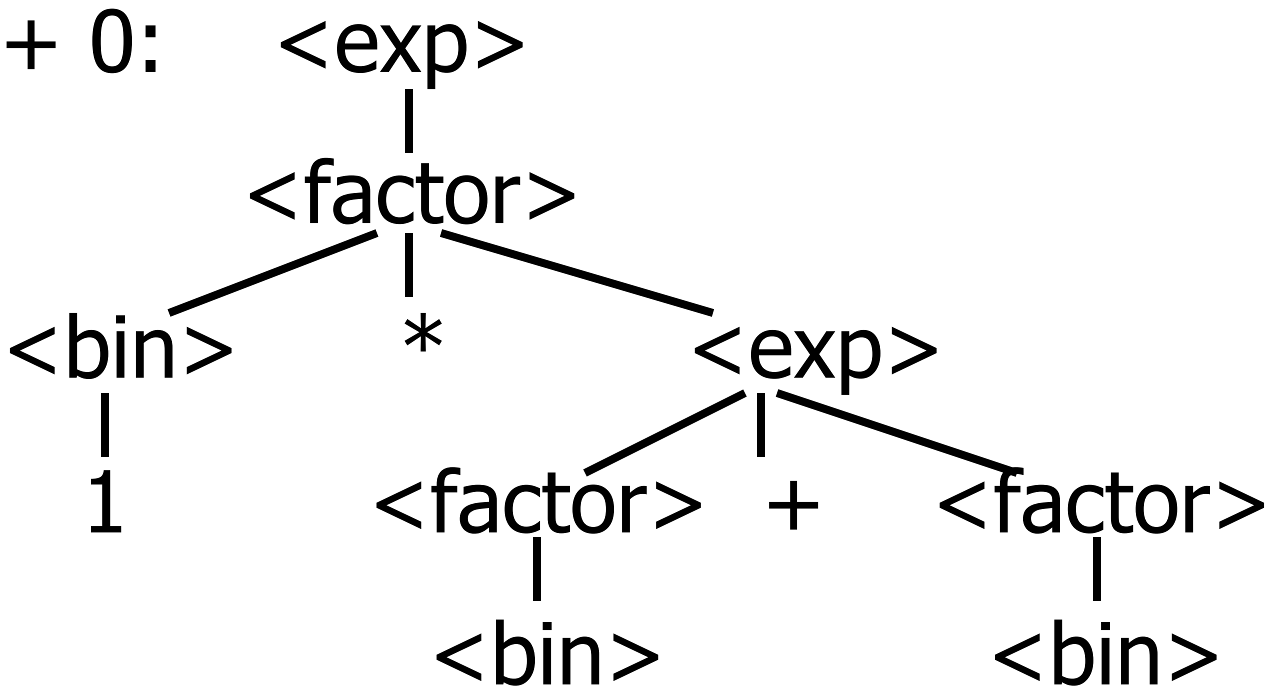
- 1 * 1 + 0:



Use rules: <bin> ::= 1 and
<exp> ::= <factor> + <factor>

Example cont.

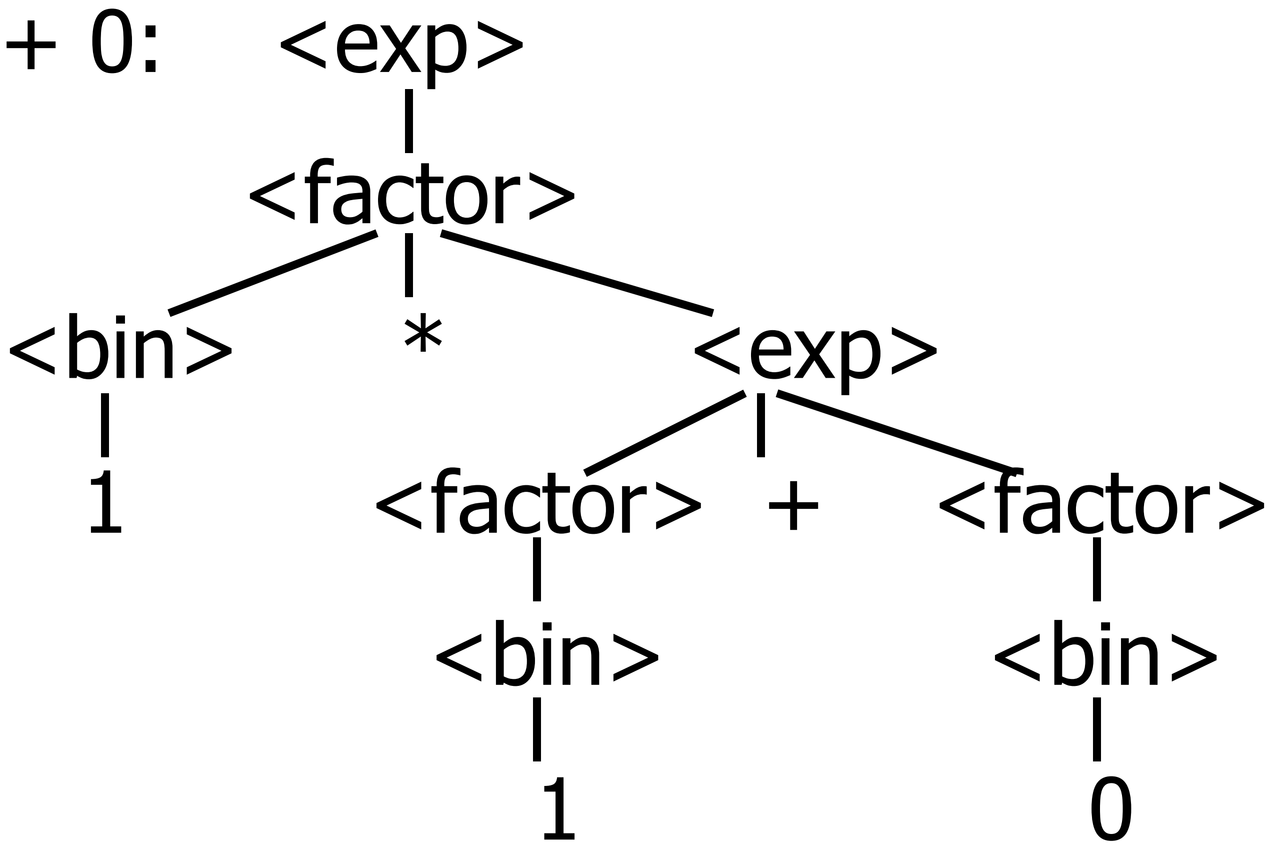
- 1 * 1 + 0:



Use rule: <factor> ::= <bin>

Example cont.

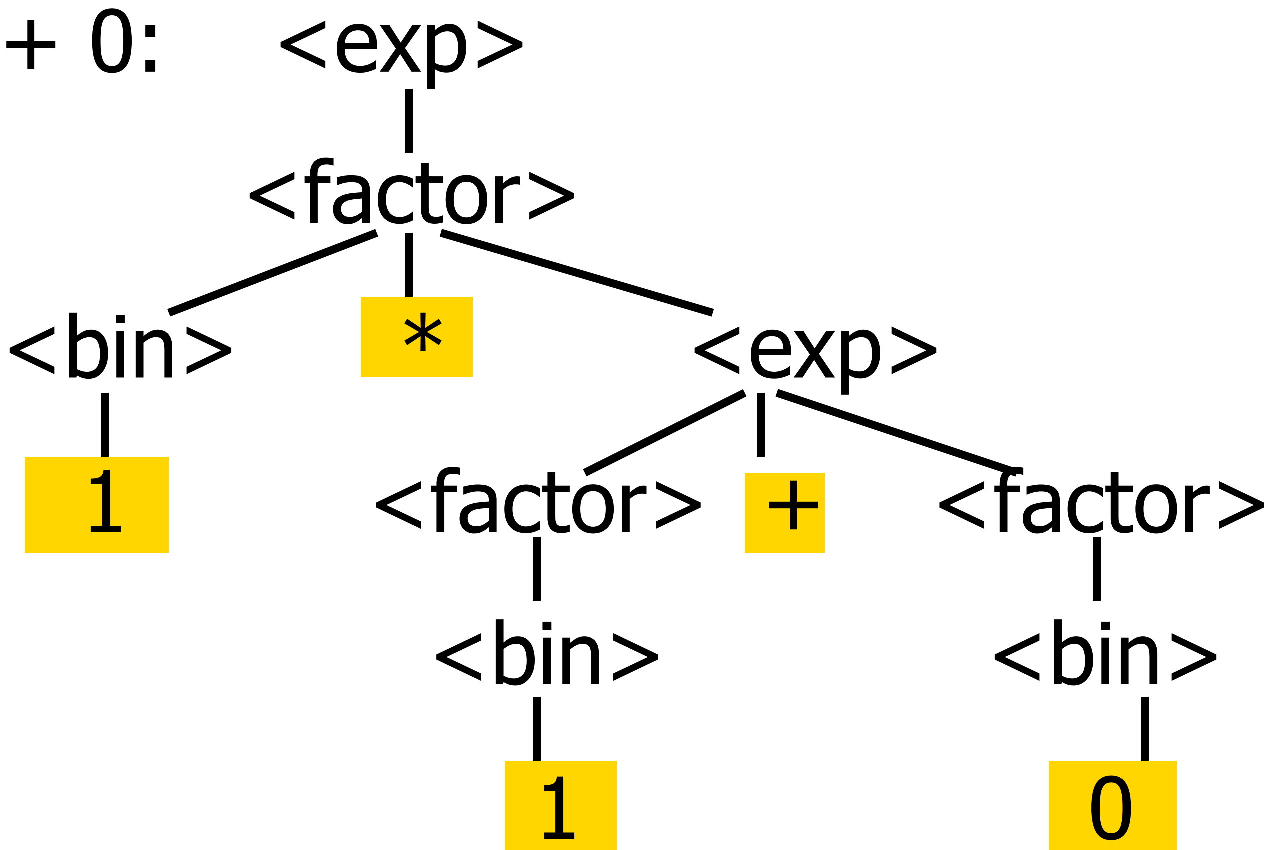
- 1 * 1 + 0:



Use rules: <bin> ::= 1 | 0

Example cont.

- 1 * 1 + 0:



Fringe of tree is string generated by grammar

Your Turn: 1 * 0 + 0 * 1

Parse Tree Data Structures

- Parse trees may be represented by OCaml datatypes
- One datatype for each nonterminal
- One constructor for each rule
- Defined as mutually recursive collection of datatype declarations

Example

- Recall grammar:

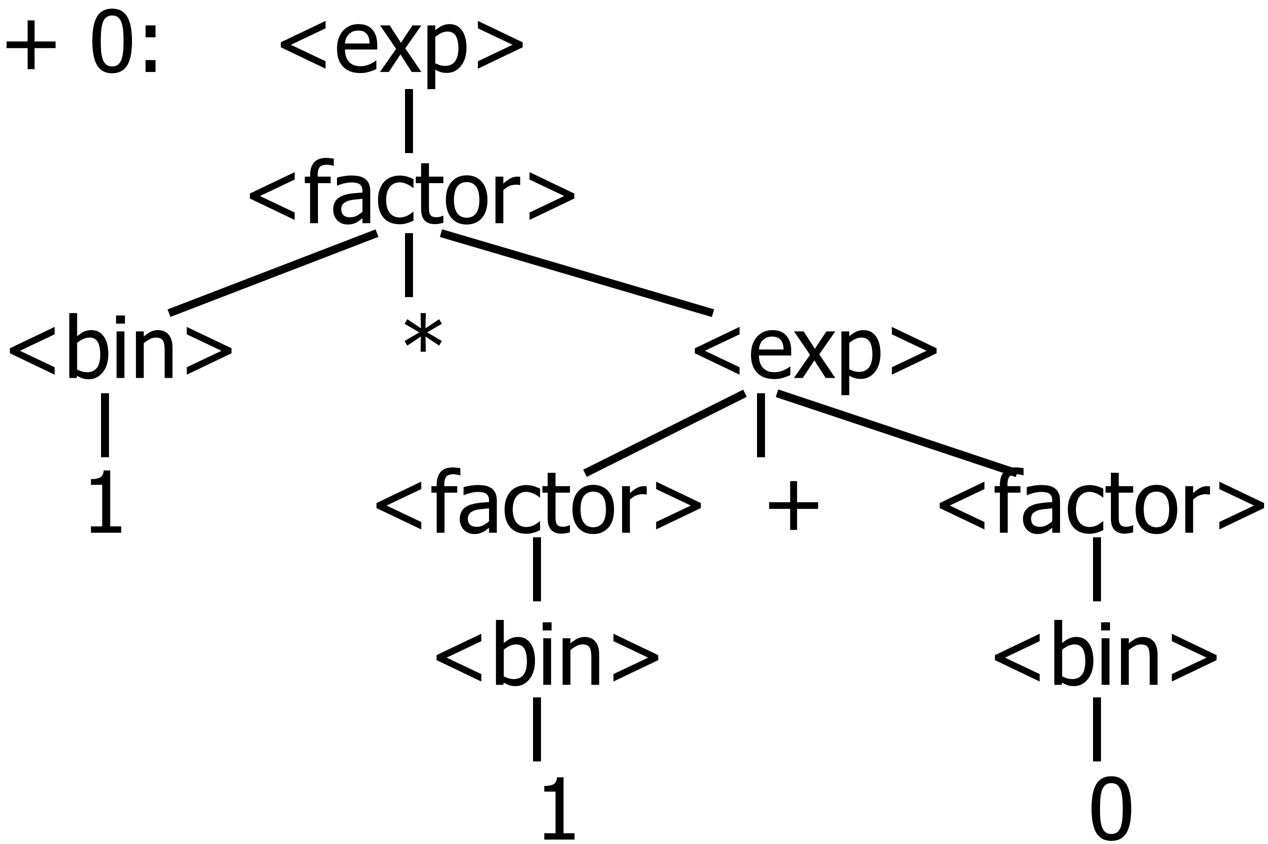
```
<exp>  ::= <factor>  |  <factor> +  <factor>
<factor>  ::=  <bin> |  <bin>  *  <exp>
<bin> ::=  0  | 1
```

- type exp = Factor2Exp of factor

```
type exp = Factor2Exp of factor
                 | Plus of factor * factor
 and factor = Bin2Factor of bin
                   | Mult of bin * exp
 and bin = Zero | One
```

Example cont.

- 1 * 1 + 0:



Example cont.

- Can be represented as

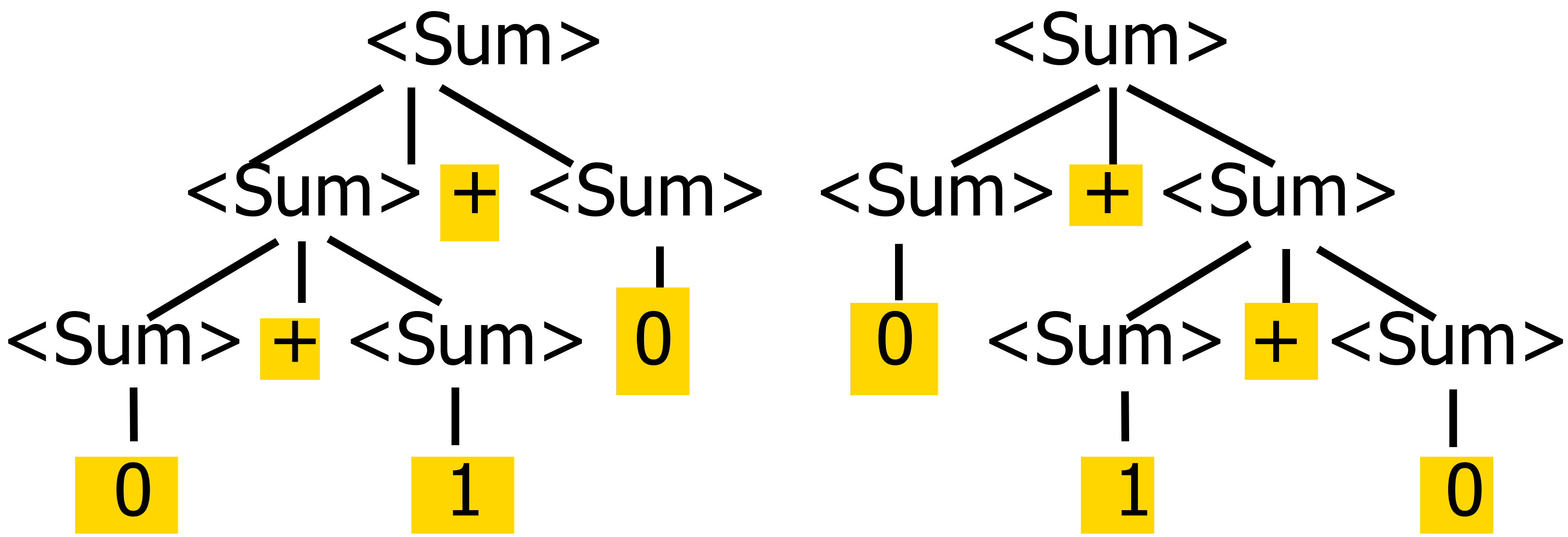
Factor2Exp
(Mult(One,
        Plus(Bin2Factor One,
                Bin2Factor Zero)))

Ambiguous Grammars and Languages

- A BNF grammar is *ambiguous* if its language contains strings for which there is more than one parse tree
- If all BNF's for a language are ambiguous then the language is *inherently ambiguous*

Example: Ambiguous Grammar

- 0 + 1 + 0



Example

- What is the result for:

$$3 + 4 * 5 + 6$$

Example

- What is the result for:

$$3 + 4 * 5 + 6$$

- Possible answers:
  - $41 = ((3 + 4) * 5) + 6$
  - $47 = 3 + (4 * (5 + 6))$
  - $29 = (3 + (4 * 5)) + 6 = 3 + ((4 * 5) + 6)$
  - $77 = (3 + 4) * (5 + 6)$

Example

- What is the value of:

$$7 - 5 - 2$$

Example

- What is the value of:

$$7 - 5 - 2$$

- Possible answers:
  - In Pascal, C++, SML assoc. left

  $7 - 5 - 2 = (7 - 5) - 2 = 0$
  - In APL, associate to right

  $7 - 5 - 2 = 7 - (5 - 2) = 4$

Two Major Sources of Ambiguity

- Lack of determination of operator precedence
- Lack of determination of operator assoicativity

- Not the only sources of ambiguity

CS241 Systems Programming

Synchronization Problems

Lawrence Angrave

CS241 Administrative

This week

- HW due Friday

Next week

- Midterm Monday in class

This lecture

Goals:

- Introduce classic synchronization problems

Topics

- Producer-Consumer
- Dining Philosophers

Producer Consumer

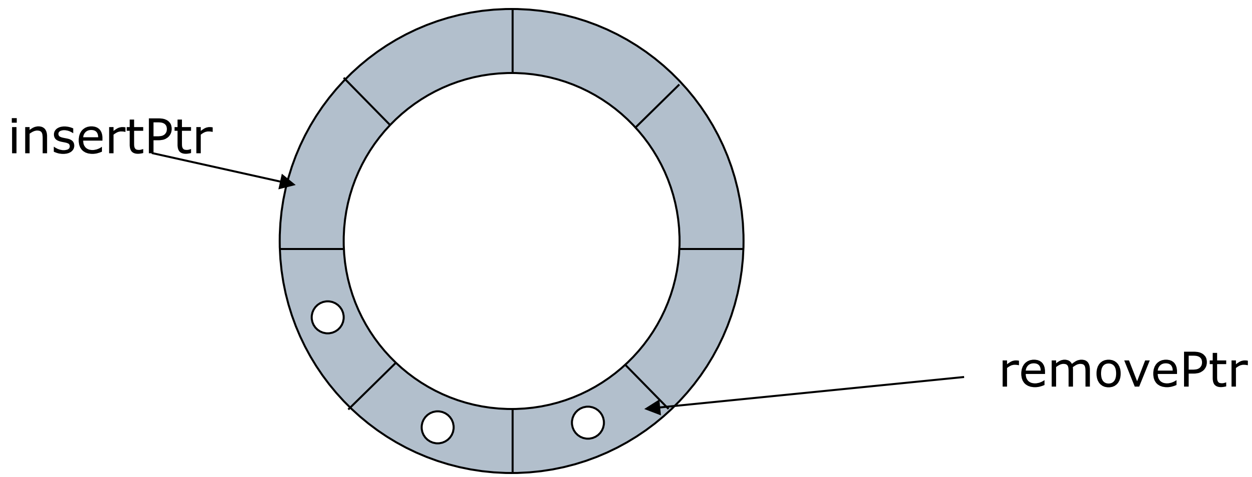
Problem occurs in system & application programming

e.g. Web Server dispatches incoming web requests to a waiting process(es)

e.g. GUI events from keyboard,& mouse are queued by O/S and consumed by applications.

Pipelines (Hardware & software examples)

Producer-Consumer



Producer-Consumer Problem

Producers insert items

Consumers remove items

Shared bounded buffer *

* Efficient implementation is a circular buffer with an insert and a removal pointer.

Challenge?

What the abstract requirements that our solution must satisfy?

Challenge

- Prevent buffer overflow
- Prevent buffer underflow
- Proper synchronization
  - Mutual Exclusion
  - Progress
  - Bounded wait
  - i.e. Prevent deadlock

Buffer underflow

Imagine:

Producer inserts an item

Consumer removes an item

Consumer removes another item

Conclusions:

Make sure consumer only retrieves valid items from the buffer

Consumer should block (or return an error code) if there are no items available

Buffer overflow

Producer inserts too many items and the buffer overflows

Conclusion:

Block Producer if the buffer is full

Mutual Exclusion

Producer inserts items. Updates insertion pointer.

Consumer executes destructive reads on the buffer. Update removal pointer

Both update information about how full/empty the buffer is.

Solution should allow multiple readers/writers

What could possibly go wrong?

Check the tricky "Edge cases"

e.g. Producer is waiting(blocked) but the buffer is already full. Now Consumer reads an item.

What could go wrong at this point?

What could possibly go wrong?

Check the tricky "Edge cases"

e.g. Producer is waiting(blocked) but the buffer is already full. Now Consumer calls remove()

What could go wrong at this point?

- Consumer cannot continue because Producer is in critical section => deadlock
- Producer is never unblocked => deadlock
- Consumer unblocks Producer too early. Failed mutual exclusion => corrupt data.
- Suppose *another* consumer/producer arrives… will our implementation still work?

Other edge cases

- Two producers call insert() at the same time
- Consumer(s) is/are waiting on an empty buffer. Producer tries to insert one item.

Implementation?

What do we need to prevent buffer underflow?

What do we need to prevent buffer overflow?

What do we need to protect updates to buffer?

2 Counting Semaphores and a mutex

Counting semaphore to count # items in buffer

Counting semaphore to count # free slots

Mutex to protect accesses to shared buffer & pointers.

Assembling the solution

sem_wait(slots), sem_post(slots)

sem_wait(items), sem_post(items)

mutex_lock(m) mutex_unlock(m)

insertptr=(insertptr+1) % N

removalptr=(removalptr+1) % N

Initialize semaphore *slots* to size of buffer

Initialize semaphore *items* to zero.

Pseudocode getItem()

Error checking/EINTR handling not shown

```
sem_wait(items)

mutex_lock(mutex)
    result=buffer[ removePtr ];
    removePtr=(removePtr +1 ) % N
mutex_unlock(mutex)

sem_post(slots)
```

so what about insertItem()?

Pseudocode putItem(*data*)

Error checking/EINTR handling not shown

```
sem_wait(slots)

mutex_lock(mutex)
    buffer[ insertPtr]= data;
    insertPtr=(insertPtr + 1 ) % N
mutex_unlock(mutex)

sem_post(items)
```

Analysis#1 What's the precise problem?

```
putItem(data) {

mutex_lock(mutex)

sem_wait(slots)
  buffer[ insertPtr]= …
  insertPtr=…
sem_post(items)

mutex_unlock(mutex)

}
```

Underflow?Overflow?

```
getItem() {

mutex_lock(mutex)

sem_wait(items)
  result=buffer[ removePtr ];
  removePtr=…
sem_post(slots)

mutex_unlock(mutex)

}
```

Deadlock? Failed Mut Excl?

Deadlock e.g Consumer waits for producer to insert a new item but Producer is waiting for Consumer to release mutex

```
putItem(data) {

mutex_lock(mutex) #2

sem_wait(slots)
  buffer[ insertPtr]= …
  insertPtr=…
sem_post(items)
mutex_unlock(mutex)
}
```

```
getItem() {

mutex_lock(mutex)

sem_wait(items) BLOCKS #1
  result=buffer[ removePtr ];
  removePtr=…
sem_post(slots)
mutex_unlock(mutex)
}
```

Analysis#2

```
putItem(data) {

sem_wait(slots)

mutex_lock(mutex)
  buffer[ insertPtr]= …
  insertPtr=…
sem_post(items)

mutex_unlock(mutex)
}
```

```
getItem() {

sem_wait(items)
sem_post(slots)

mutex_lock(mutex)
  result=buffer[ removePtr ];
  removePtr=…
mutex_unlock(mutex)

}
```

Buffer overflow when reader removes item from a full buffer: Producer inserts item too early

```
putItem(data) {

sem_wait(slots)

mutex_lock(mutex)
  buffer[ insertPtr]= …
  insertPtr=…
sem_post(items)

mutex_unlock(mutex)
}
```

```
getItem() {

sem_wait(items)
sem_post(slots)

mutex_lock(mutex)
  result=buffer[ removePtr ];
  removePtr=…
mutex_unlock(mutex)
}
```

Other considerations

Early cancelation using signals?

Limited Production?

(Possibly no items)

… Consumers wait forever

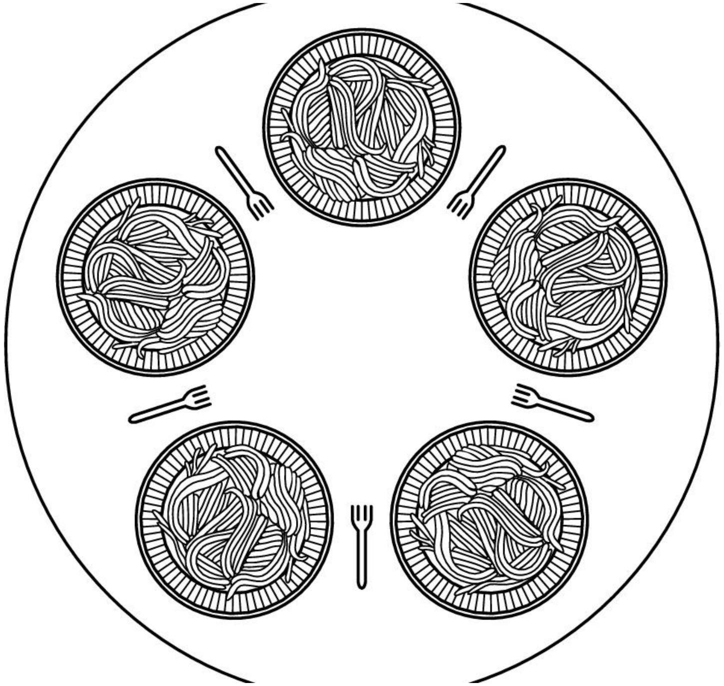
… Consumers quit too early?

One implementation: Insert special end-value into queue which consumers read (but may not consume)

Other reasonable implementations (condition variables;not covered in CS241)

Priority of Consumers & Producers

Dining Philosophers



History

Dijkstra 1971

Originally set as a exam question

Illustrates Starvation, Deadlock

Test shared resource allocation algorithms

See Stallings Ch.6.6 p.276

Dining Philosopher Challenge

{ Think | Eat }

N Philosophers circular table with N chopsticks

To eat the Philosopher must first pickup two chopsticks

i^{th} Philosopher needs i^{th} & $i+1^{th}$ chopstick

Only put down chopstick when Philosopher has finished eating

Devise a solution which satisfies mutual exclusion but avoids starvation and deadlock

Seems simple enough …?

```
while(true) {
 think()
 sem_wait(chopstick[i])
 sem_wait(chopstick[(i+1) % N])
 eat()
 sem_post(chopstick[(i+1) % N])
 sem_post(chopstick[i])
}
```

… Deadlock (Each P. holds left fork)

```
while(true) {
 think()
 sem_wait(chopstick[i])
 sem_wait(chopstick[(i+1) % N])
 eat()
 sem_post(chopstick[(i+1) % N])
 sem_post(chopstick[i])
}
```

What if?

Made picking up left and right chopsticks an atomic operation?

or,

N-1 Philosophers but N chopsticks?

What if?

Made picking up left and right chopsticks an atomic operation?

or,

N-1 Philosophers but N chopsticks?

… both changes prevent deadlock.

Formal requirements for deadlock

Mutual exclusion

Hold and wait condition

No preemption condition

Circular wait condition

Original scenario & our proposed ritual had all four of these properties.

Formal requirements for deadlock

Mutual exclusion

- Exclusive use of chopsticks

Hold and wait condition

- Hold 1 chopstick

No preemption condition

- Cannot force another P. to undo their hold

Circular wait condition

- N Philosophers N chopsticks

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

LR Parsing

- Read tokens left to right (L)
- Create a rightmost derivation (R)
- How is this possible?
- Start at the bottom (left) and work your way up
- Last step has only one non-terminal to be replaced so is right-most
- Working backwards, replace mixed strings by non-terminals
- Always proceed so that there are no non-terminals to the right of the string to be replaced

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

= ● (0 + 1) + 0 shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=	(0 + 1) + 0	shift
=	(0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=> (0 ● + 1) + 0	reduce
= (● 0 + 1) + 0	shift
= ● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=	(<Sum> ● + 1) + 0	shift
=>	(0 ● + 1) + 0	reduce
=	(● 0 + 1) + 0	shift
=	● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=	(<Sum> + ● 1) + 0	shift
=	(<Sum> ● + 1) + 0	shift
=>	(0 ● + 1) + 0	reduce
=	(● 0 + 1) + 0	shift
=	● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=>	(<Sum> + 1 ●) + 0	reduce
=	(<Sum> + ● 1) + 0	shift
=	(<Sum> ● + 1) + 0	shift
=>	(0 ● + 1) + 0	reduce
=	(● 0 + 1) + 0	shift
=	● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=>	(<Sum> + <Sum> ●) + 0	reduce
=>	(<Sum> + 1 ●) + 0	reduce
=	(<Sum> + ● 1) + 0	shift
=	(<Sum> ● + 1) + 0	shift
=>	(0 ● + 1) + 0	reduce
=	(● 0 + 1) + 0	shift
=	● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=	(<Sum> ●) + 0	shift
=>	(<Sum> + <Sum> ●) + 0	reduce
=>	(<Sum> + 1 ●) + 0	reduce
=	(<Sum> + ● 1) + 0	shift
=	(<Sum> ● + 1) + 0	shift
=>	(0 ● + 1) + 0	reduce
=	(● 0 + 1) + 0	shift
=	● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=> (<Sum>) ● + 0	reduce
= (<Sum> ●) + 0	shift
=> (<Sum> + <Sum> ●) + 0	reduce
=> (<Sum> + 1 ●) + 0	reduce
= (<Sum> + ● 1) + 0	shift
= (<Sum> ● + 1) + 0	shift
=> (0 ● + 1) + 0	reduce
= (● 0 + 1) + 0	shift
= ● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

= <Sum> ● + 0 shift
=> (<Sum>) ● + 0 reduce
= (<Sum> ●) + 0 shift
=> (<Sum> + <Sum> ●) + 0 reduce
=> (<Sum> + 1 ●) + 0 reduce
= (<Sum> + ● 1) + 0 shift
= (<Sum> ● + 1) + 0 shift
=> (0 ● + 1) + 0 reduce
= (● 0 + 1) + 0 shift
= ● (0 + 1) + 0 shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=	<Sum> + ● 0	shift
=	<Sum> ● + 0	shift
=>	(<Sum>) ● + 0	reduce
=	(<Sum> ●) + 0	shift
=>	(<Sum> + <Sum> ●) + 0	reduce
=>	(<Sum> + 1 ●) + 0	reduce
=	(<Sum> + ● 1) + 0	shift
=	(<Sum> ● + 1) + 0	shift
=>	(0 ● + 1) + 0	reduce
=	(● 0 + 1) + 0	shift
=	● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum>	=>		
	=>	<Sum> + 0 ●	reduce
	=	<Sum> + ● 0	shift
	=	<Sum> ● + 0	shift
	=>	(<Sum>) ● + 0	reduce
	=	(<Sum> ●) + 0	shift
	=>	(<Sum> + <Sum> ●) + 0	reduce
	=>	(<Sum> + 1 ●) + 0	reduce
	=	(<Sum> + ● 1) + 0	shift
	=	(<Sum> ● + 1) + 0	shift
	=>	(0 ● + 1) + 0	reduce
	=	(● 0 + 1) + 0	shift
	=	● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum>	=> <Sum> + <Sum > ●	reduce
	=> <Sum> + 0 ●	reduce
	= <Sum> + ● 0	shift
	= <Sum> ● + 0	shift
	=> (<Sum>) ● + 0	reduce
	= (<Sum> ●) + 0	shift
	=> (<Sum> + <Sum> ●) + 0	reduce
	=> (<Sum> + 1 ●) + 0	reduce
	= (<Sum> + ● 1) + 0	shift
	= (<Sum> ● + 1) + 0	shift
	=> (0 ● + 1) + 0	reduce
	= (● 0 + 1) + 0	shift
	= ● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> ●	=> <Sum> + <Sum > ●	reduce
	=> <Sum> + 0 ●	reduce
	= <Sum> + ● 0	shift
	= <Sum> ● + 0	shift
	=> (<Sum>) ● + 0	reduce
	= (<Sum> ●) + 0	shift
	=> (<Sum> + <Sum> ●) + 0	reduce
	=> (<Sum> + 1 ●) + 0	reduce
	= (<Sum> + ● 1) + 0	shift
	= (<Sum> ● + 1) + 0	shift
	=> (0 ● + 1) + 0	reduce
	= (● 0 + 1) + 0	shift
	= ● (0 + 1) + 0	shift

Example

(0 + 1) + 0

Example

(0 + 1) + 0

Example

(0 + 1) + 0

Example

<Sum>

(0 + 1) + 0

Example

<Sum>

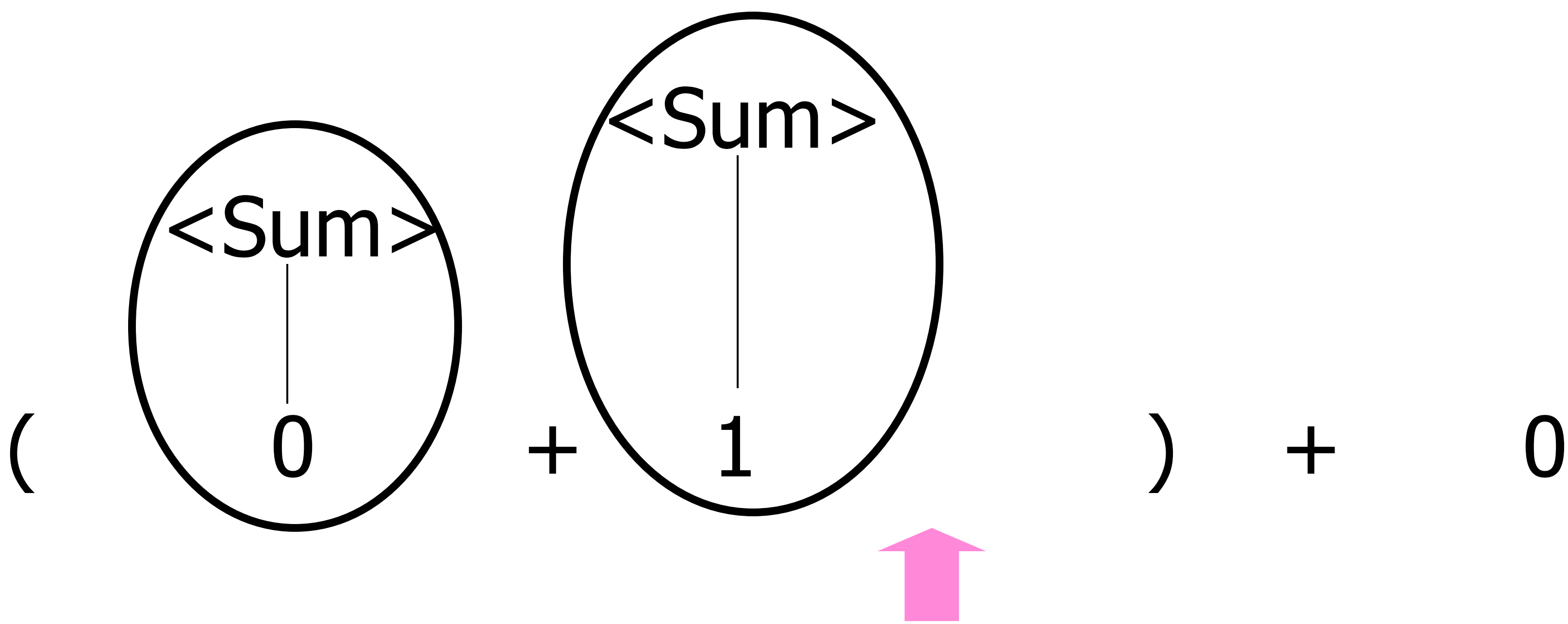
(0 + 1) + 0

Example

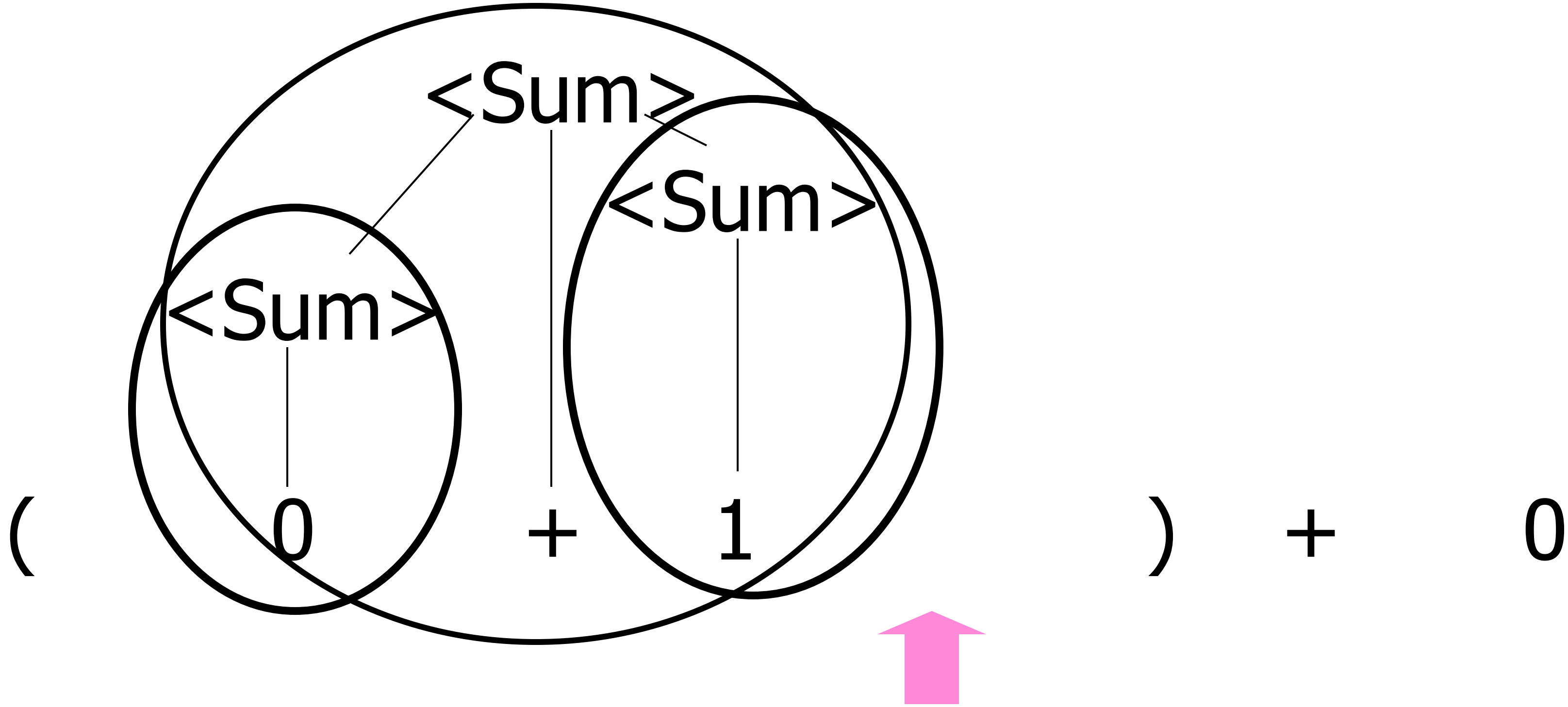
<Sum>

(0 + 1) + 0

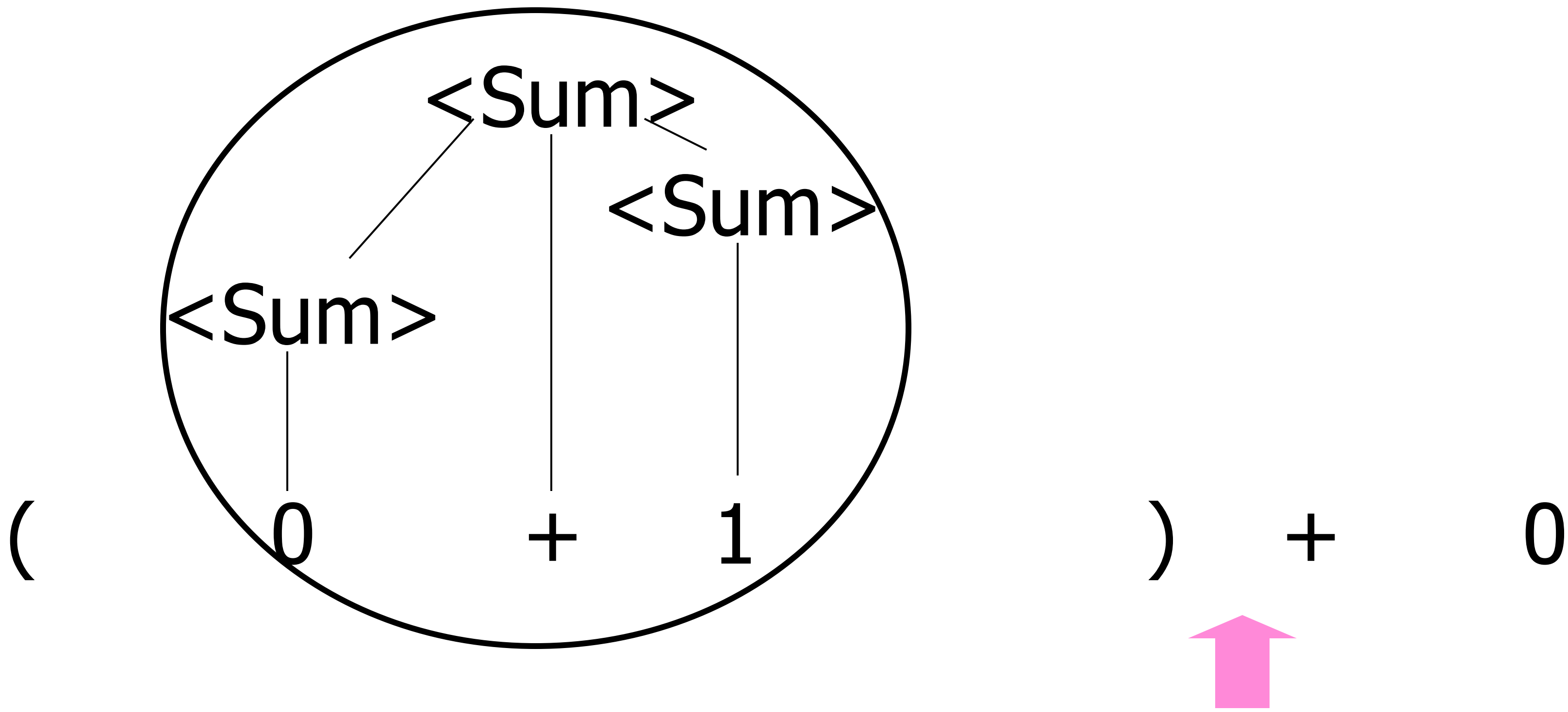
Example



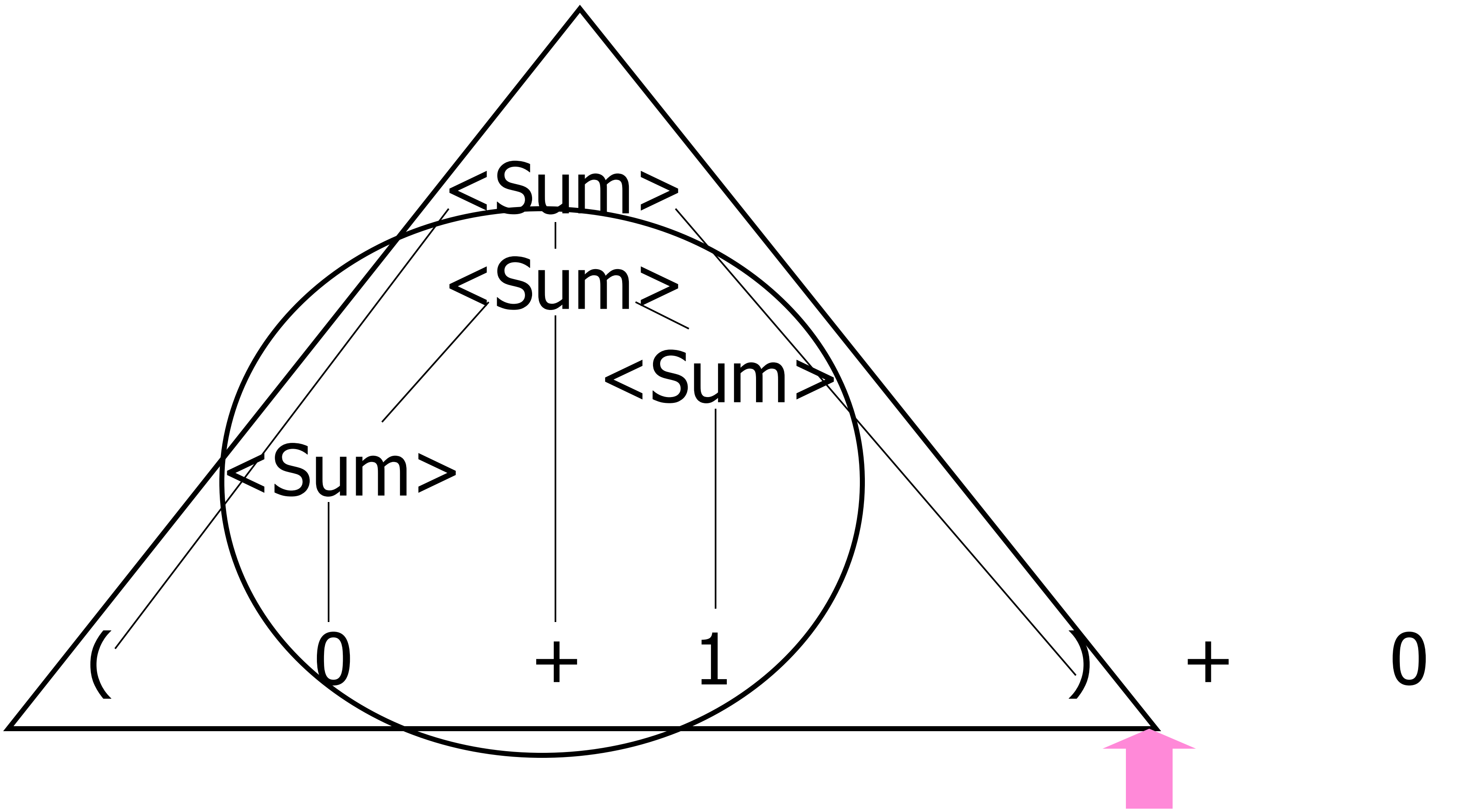
Example



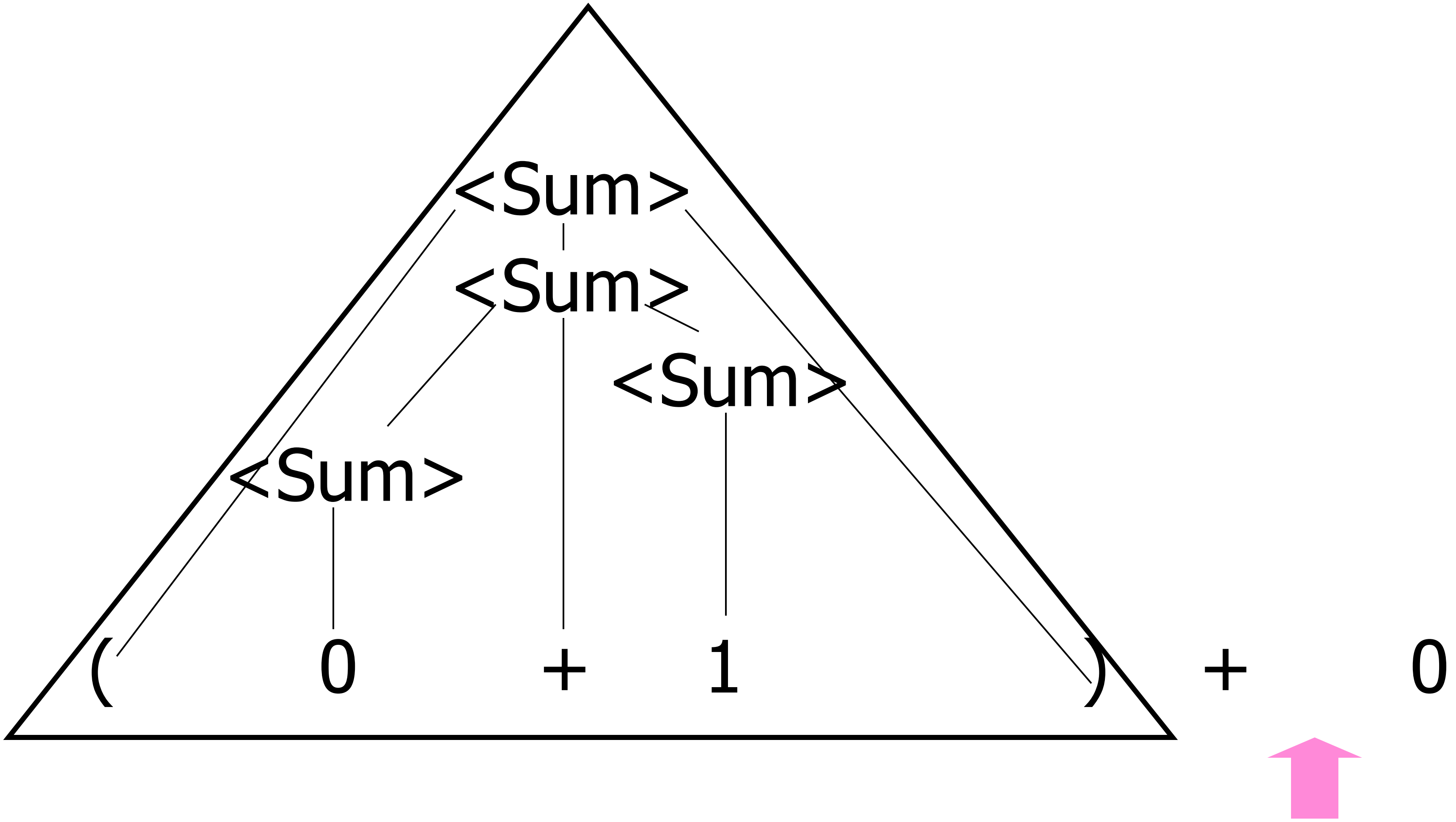
Example



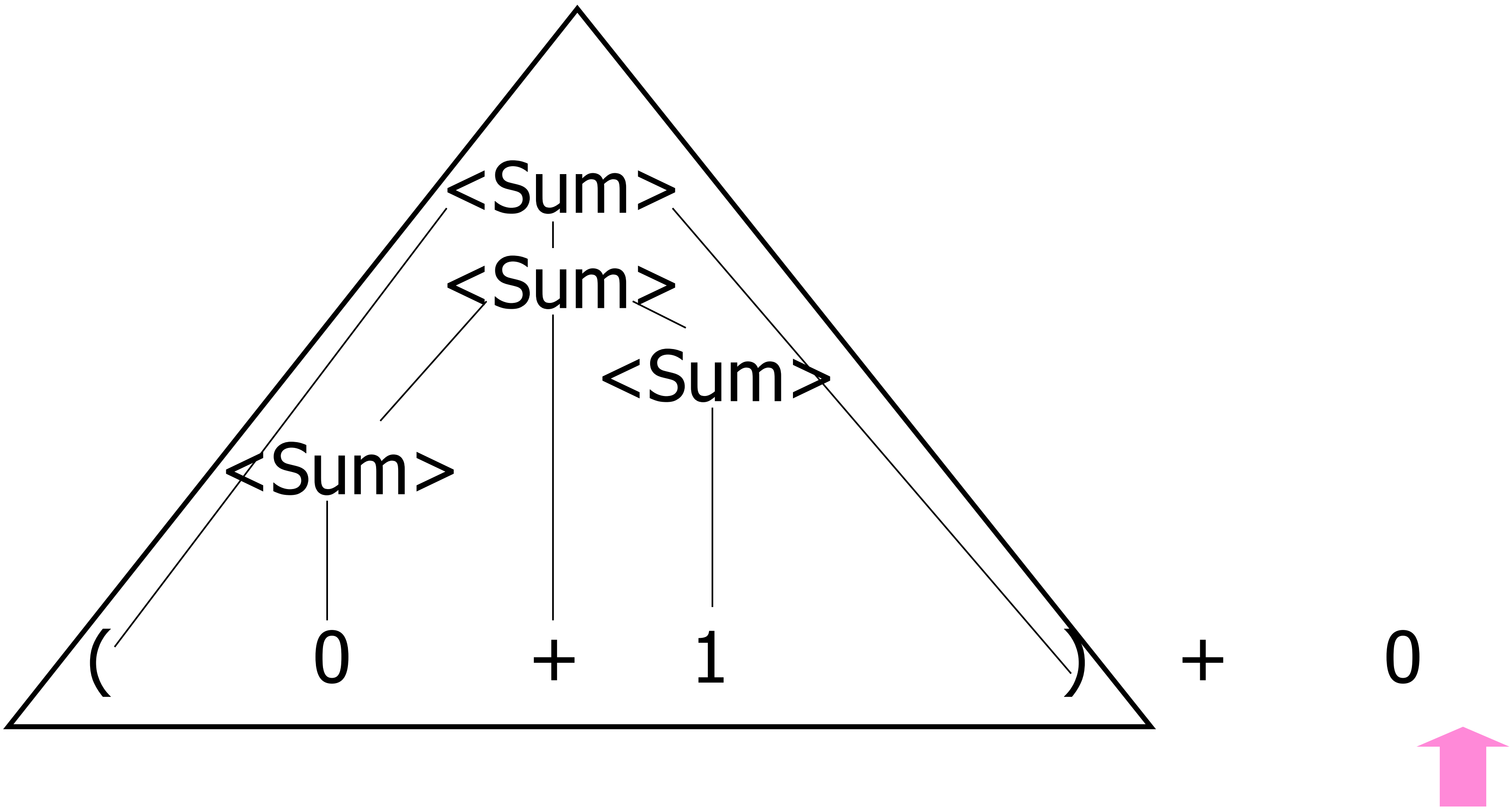
Example



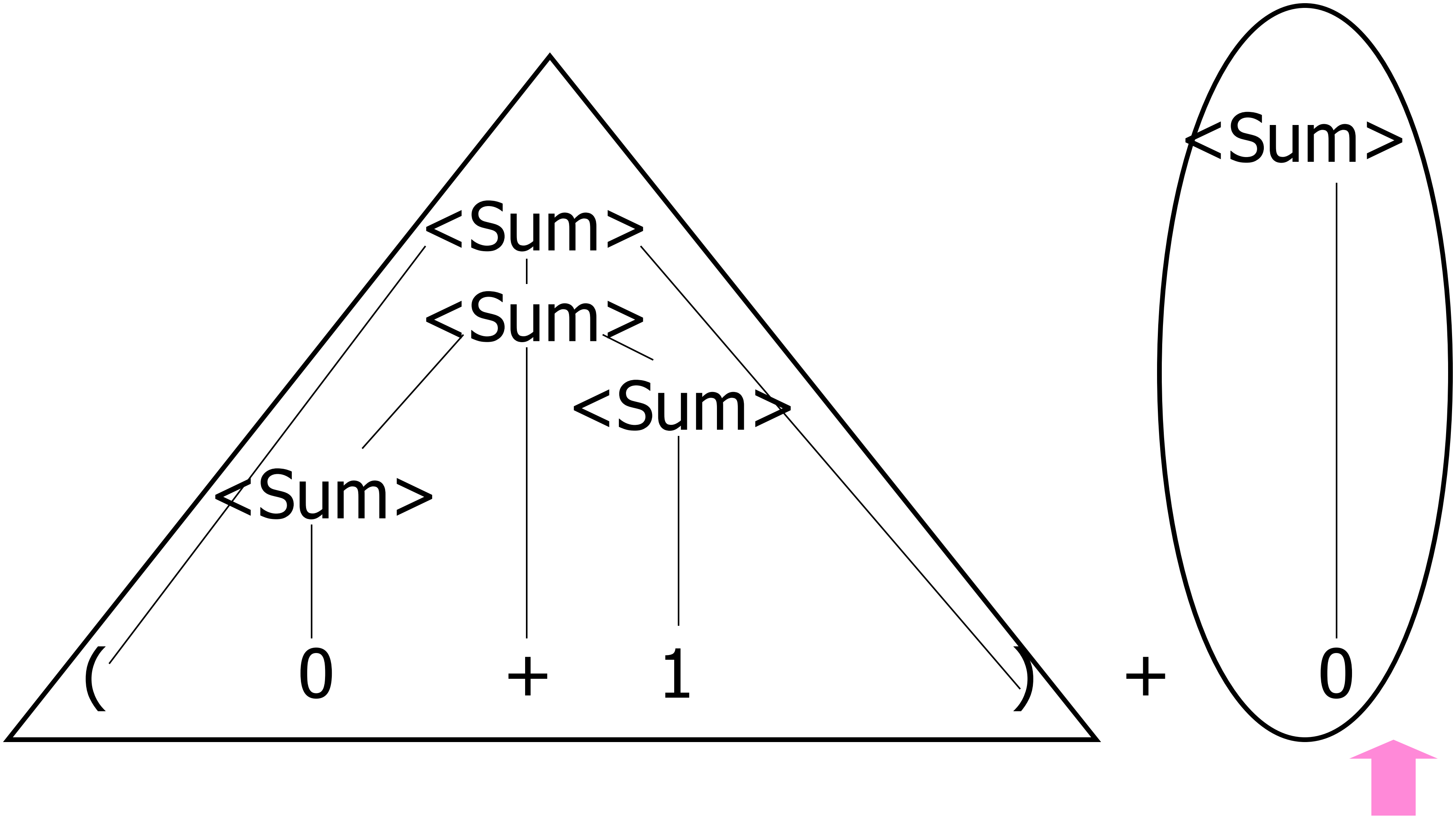
Example



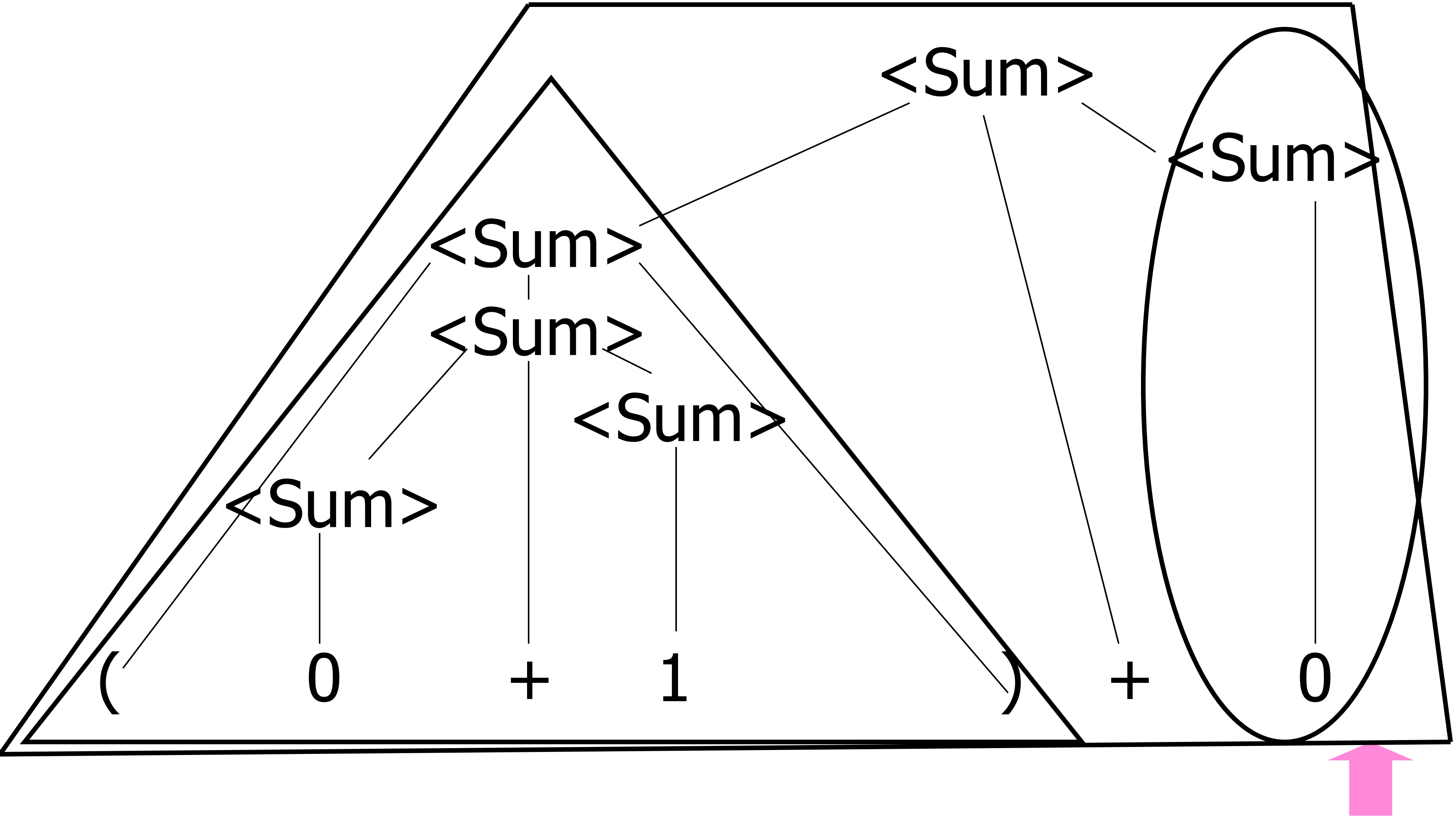
Example



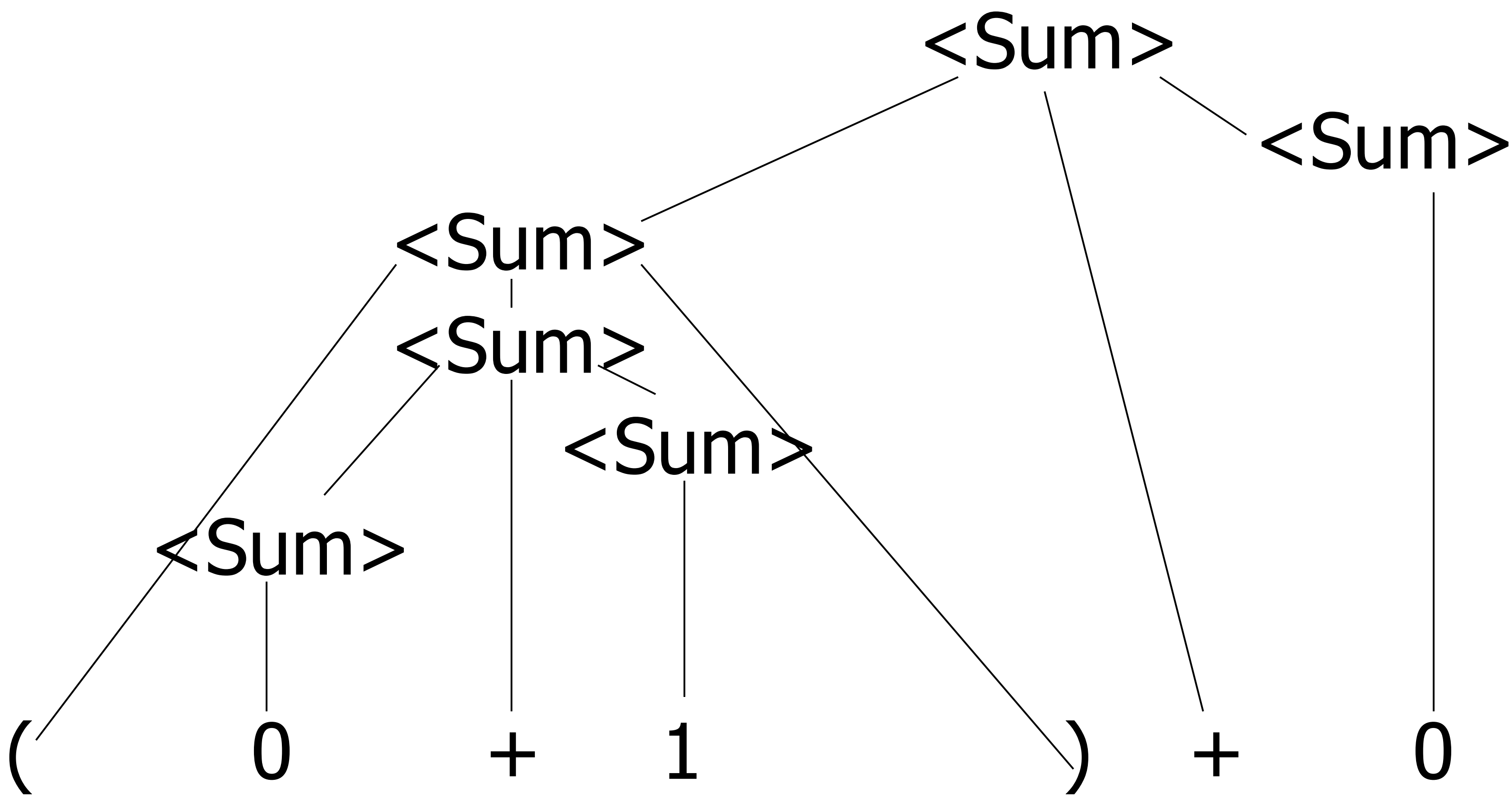
Example



Example



Example



LR Parsing Tables

- Build a pair of tables, Action and Goto, from the grammar
  - This is the hardest part, we omit here
  - Rows labeled by states
  - For Action, columns labeled by terminals and "end-of-tokens" marker
    - (more generally strings of terminals of fixed length)
  - For Goto, columns labeled by non-terminals

Action and Goto Tables

- Given a state and the next input, Action table says either
  - **shift** and go to state *n*, or
  - **reduce** by production *k* (explained in a bit)
  - **accept** or **error**
- Given a state and a non-terminal, Goto table says
  - go to state *m*

LR(i) Parsing Algorithm

- Based on push-down automata
- Uses states and transitions (as recorded in Action and Goto tables)
- Uses a stack containing states, terminals and non-terminals

LR(i) Parsing Algorithm

0. Insure token stream ends in special “end-of-tokens” symbol
1. Start in state 1 with an empty stack
2. Push **state**(1) onto stack
3. Look at next *i* tokens from token stream (*toks*) (don’t remove yet)
4. If top symbol on stack is **state**(*n*), look up action in Action table at (*n*, *toks*)

LR(i) Parsing Algorithm

5. If action = **shift** *m*,
   a) Remove the top token from token stream and push it onto the stack
   b) Push **state**(*m*) onto stack
   c) Go to step 3

LR(i) Parsing Algorithm

6. If action = **reduce** *k* where production *k* is E ::= u

   a) Remove 2 * length(u) symbols from stack (u and all the interleaved states)

   b) If new top symbol on stack is **state**(*m*), look up new state *p* in Goto(*m*,E)

   c) Push E onto the stack, then push **state**(*p*) onto the stack

   d) Go to step 3

LR(i) Parsing Algorithm

7. If action = **accept**
   - Stop parsing, return success
8. If action = **error**,
   - Stop parsing, return failure

Adding Synthesized Attributes

- Add to each **reduce** a rule for calculating the new synthesized attribute from the component attributes
- Add to each non-terminal pushed onto the stack, the attribute calculated for it
- When performing a **reduce**,
  - gather the recorded attributes from each non-terminal popped from stack
  - Compute new attribute for non-terminal pushed onto stack

Shift-Reduce Conflicts

- **Problem**: can’t decide whether the action for a state and input character should be **shift** or **reduce**
- Caused by ambiguity in grammar
- Usually caused by lack of associativity or precedence information in grammar

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

● 0 + 1 + 0	shift
-> 0 ● + 1 + 0	reduce
-> <Sum> ● + 1 + 0	shift
-> <Sum> + ● 1 + 0	shift
-> <Sum> + 1 ● + 0	reduce
-> <Sum> + <Sum> ● + 0	

Example - cont

- **Problem:** shift or reduce?

- You can shift-shift-reduce-reduce or reduce-shift-shift-reduce

- Shift first - right associative
- Reduce first- left associative

Reduce - Reduce Conflicts

- **Problem:** can’t decide between two different rules to reduce by
- Again caused by ambiguity in grammar
- **Symptom:** RHS of one production suffix of another
- Requires examining grammar and rewriting it
- Harder to solve than shift-reduce errors

Example

- S ::= A | aB   A ::= abc   B ::= bc

● abc	shift
a ● bc	shift
ab ● c	shift
abc ●	

- Problem: reduce by B ::= bc then by S ::= aB, or by A::= abc then S::A?

Using Ocamlyacc

- Input attribute grammar is put in file <*grammar*>.mly
- Execute

  ocamlyacc <*grammar*>.mly
- Produces code for parser in

  <*grammar*>.ml

  and interface (including type declaration for tokens) in

  <*grammar*>.mli

Parser Code

- <*grammar*>.ml defines one parsing function per entry point
- Parsing function takes a lexing function (lexer buffer to token) and a lexer buffer as arguments
- Returns semantic attribute of corresponding entry point

Ocamlyacc Input

- File format:

%{

<*header*>

%}

<*declarations*>

%%

<*rules*>

%%

<*trailer*>

Ocamlyacc <*header*>

- Contains arbitrary Ocaml code
- Typically used to give types and functions needed for the semantic actions of rules and to give specialized error recovery
- May be omitted
- <*footer*> similar. Possibly used to call parser

Ocamlyacc <declarations>

- %token *symbol … symbol*
- Declare given symbols as tokens
- %token <*type*> *symbol … symbol*
- Declare given symbols as token constructors, taking an argument of type <*type*>
- %start *symbol … symbol*
- Declare given symbols as entry points; functions of same names in <*grammar*>.ml

Ocamlyacc <*declarations*>

- %type <*type*> *symbol … symbol*

  Specify type of attributes for given symbols. Mandatory for start symbols
- %left *symbol … symbol*
- %right *symbol … symbol*
- %nonassoc *symbol … symbol*

  Associate precedence and associativity to given symbols. Same line,same precedence; earlier line, lower precedence (broadest scope)

Ocamlyacc <*rules*>

- *nonterminal* :

  symbol ... *symbol* { *semantic_action* }

  | ...

  | *symbol* ... *symbol* { *semantic_action* }

  ;
- Semantic actions are arbitrary Ocaml expressions
- Must be of same type as declared (or inferred) for *nonterminal*
- Access semantic attributes (values) of symbols by position: $1 for first symbol, $2 to second …

Example - Base types

```
(* File: expr.ml *)
type expr =
    Term_as_Expr of term
  | Plus_Expr of (term * expr)
  | Minus_Expr of (term * expr)
and term =
    Factor_as_Term of factor
  | Mult_Term of (factor * term)
  | Div_Term of (factor * term)
and factor =
    Id_as_Factor of string
  | Parenthesized_Expr_as_Factor of expr
```

Example - Lexer (exprlex.mll)

```
{ (*open Exprparse*) }
let numeric = ['0' - '9']
let letter =['a' - 'z' 'A' - 'Z']
rule token = parse
  | "+" {Plus_token}
  | "-"  {Minus_token}
  | "*"  {Times_token}
  | "/"  {Divide_token}
  | "("  {Left_parenthesis}
  | ")"  {Right_parenthesis}
  | letter (letter|numeric|"_")* as id  {Id_token id}
  | [' ' '\t' '\n'] {token lexbuf}
  | eof {EOL}
```

Example - Parser (exprparse.mly)

```
%{ open Expr
%}
%token <string> Id_token
%token Left_parenthesis Right_parenthesis
%token Times_token Divide_token
%token Plus_token Minus_token
%token EOL
%start main
%type <expr> main
%%
```

Example - Parser (exprparse.mly)

```
expr:
   term
      { Term_as_Expr $1 }
 | term Plus_token expr
      { Plus_Expr ($1, $3) }
 | term Minus_token expr
      { Minus_Expr ($1, $3) }
```

Example - Parser (exprparse.mly)

```
term:
   factor
       { Factor_as_Term $1 }
 | factor Times_token term
       { Mult_Term ($1, $3) }
 | factor Divide_token term
       { Div_Term ($1, $3) }
```

Example - Parser (exprparse.mly)

```
factor:
   Id_token
      { Id_as_Factor $1 }
 | Left_parenthesis expr Right_parenthesis
      {Parenthesized_Expr_as_Factor $2 }
main:
 | expr EOL
      { $1 }
```

Example - Using Parser

```
# #use "expr.ml";;
…
# #use "exprparse.ml";;
…
# #use "exprlex.ml";;
…
# let test s =
  let lexbuf = Lexing.from_string (s^"\n") in
      main token lexbuf;;
```

Example - Using Parser

```
# test "a + b";;
- : expr =
Plus_Expr
 (Factor_as_Term (Id_as_Factor "a"),
  Term_as_Expr (Factor_as_Term
   (Id_as_Factor "b")))
```

Classic Synchronization Problems (Reader-Writer Problem) Midterm Review Topics

Lecture 19

Klara Nahrstedt

CS241 Administrative

- Read Chapter 5.6 in Stallings
- Homework 1 due 3/2 4pm in Anda Ohlsson's Office
- MIDTERM – MONDAY, March 5, 11am (will talk about Midterm at the end of lecture)

Outline

- Readers/ Writers Problem

First Reader-Writer Problem

- readers: read data
- writers: write data

	Reader	Writer
Reader	OK	No
Writer	NO	No

First Readers-Writers Problem
First Readers-Writers Problem
(Readers have priority)

Let processes reading do so concurrently
Let processes reading do so concurrently

Let processes writing do so one at a time

Introduce semaphores

Semaphore mutex = 1;

Semaphore wrt = 1;

while (TRUE) {

section*/

section*/

lock(&

wrt);

do writing

unlock(&

}

wrt);

}

mutex

do reading);

lock(&

do reading

mutex

lock(&mutex);

readcount = readcount+1;

if readcount == 0 then

unlock(&

readcount=readcount-1;

wrt

if readcount == 0 then

unlock(&wrt);

unlock(&mutex

Unlock(&mutex);

Use read data

Use read data }

Does it work? What if?
Does it work? What if?

Problem with this solution

[redacted]

int readcount; // shared and initialized to 0

// Writer

[redacted]

Lock(&wrt);

.....

writing performed

.....

writing performed

.....

[redacted]

Lock(&wrt);

// Reader

[redacted]

Lock[redacted]

[redacted]

[redacted] == 1) lock(&wrt);

unlock(&m);

reading performed

[redacted]ed

lock[redacted];

[redacted]

[redacted]count == 0) unlock(&wrt);

Unlock(&m);

Midterm Review

- Review
  - Lecture Notes
  - All quizzes until today
    - SMP Quizzes and Regular Quizzes
  - Review homework 1 solutions
  - Review Stallings and R&R book material
- **Midterm** - Monday 11-11:50am, 1404 SC
- **Review Session** – Sunday 2-3:30pm in 1404 SC
  - You need to bring questions, TAs will respond
- Midterm
  - closed book, closed notes,
  - NO calculator or other calculating electronic devices
  - No cell-phones

Topics: Hardware/OS Overview

- Chapter 1.1-1.7 (Stallings)
- Chapter 2.1-2.2 (Stallings)
- Chapter 1 (R&R
- Keywords Need to know
  - Processors – registers
  - Interrupts and Interrupt Handling
  - Polling and Programmed I/O
  - Basic Memory principles
  - Kernel mode, user mode
  - Multiprogramming, uni-programming
  - Time sharing
  - Buffer Overflow and security

Topics: Processes

- Chapter 3.1-3.4 (Stallings)
- Chapter 2 and 3 (R&R)
- Keywords need to know
  - What is process?
  - What is the difference between process and program?
  - What is the program image layout?
  - Understand argument arrays
  - What does it mean to have a thread-safe function?
  - What is the difference between static and dynamic variables?
  - What are the major process states?
  - What is the difference between dispatcher and scheduler?
  - What is PCB?
  - What happens when process switches from running to ready state?
  - What is process chain, process fan?

Topics: Threads

- Chapter 12.1-12.5 (R&R)
- Chapter 4.1, 4.5 (Stallings)
- Keywords need to know
  - What is the difference between processes and threads?
  - What is the difference between user-level threads and kernel-level threads?
  - Detaching and joining threads
  - What happens if you if you call exit(1) in a thread?
  - What is a graceful way to exit a thread without causing process termination?

Topics: Concurrency (Mutual Exclusion)

- Chapter 14.1-14.3 (R&R)
- Chapter 5.1-5.3 (Stallings – and don't forget Appendix A about the Software Solutions)
- Keywords need to know
  - What are the four conditions to provide appropriate synchronization and enter critical region?
  - What is the difference between counting semaphore and mutex?
  - What do sem_wait and sem_post do?
  - How can counting semaphores be implemented using binary semaphores?
  - How can test_and_set be used for synchronization?
  - How can you make a function atomic?
  - Consider increment (i++) and decrement function (i--). How do you ensure that race condition does not occur on the shared variable 'i' when two processes use them at the same time?

Topics: Thread Synchronization

- Chapter 13.1-13.2 (R&R)
- Keywords to know
  - What are mutex locks?
  - How do you initialize mutex locks?
  - When would you use mutex instead of counting semaphore?
  - When would you use counting semaphore instead of mutex?
  - Are mutex functions interrupted by signals?

Topics: Scheduling

- Chapter 9.1-9.2 Scheduling (Stallings)
- Keywords need to know
  - Scheduling policies
  - FCFC, SJF, Round Robin, Priority Scheduling
  - Preemptive vs. Non-Preemptive Scheduling
  - Queues in Process management – what is ready queue? How are process states related to process management queues?
  - What is average waiting time?
  - What is the difference between process waiting time and turn-around time?

Topics: Signals

- Chapter 8.1-8.5 (R&R
- Keywords need to know
  - Signal basic concepts – generating signals, blocked, pending signals, delivered signals, ignored signals, …
  - What is signal mask and what are the operations to modify signal mask?
  - What is signal handler?
  - What is the role of sigaction?
  - How do you wait for signals?

Topics: Timers

- Chapter 9.1-9.3 (R&R)
- Keywords need to know
  - Understand what the various time functions are for
  - Gettimeofday
  - Understand the different clock resolutions
  - Sleep function
  - What are time intervals? What are they great for?

Topics: Classical Sync Problems

- Chapter 5.3 and 6.6 (Stallings)
- Keywords need to know
  - What is the producer-consumer problem?
  - What are the various semaphores in the producer/ consumer solution for?
  - What is the dining philosopher problem?
  - What is the danger of a simple solution for dining philosopher problem?

Summary

Good Luck with the Exam!!!

Please, come to class little earlier (10:55am) so that we can start at 11:00 exactly.

Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Ocamlyacc Input

- File format:

%{

<*header*>

%}

<*declarations*>

%%

<*rules*>

%%

<*trailer*>

Ocamlyacc <*header*>

- Contains arbitrary Ocaml code
- Typically used to give types and functions needed for the semantic actions of rules and to give specialized error recovery
- May be omitted
- <*footer*> similar. Possibly used to call parser

Ocamlyacc <declarations>

- %token *symbol … symbol*
- Declare given symbols as tokens
- %token <*type*> *symbol … symbol*
- Declare given symbols as token constructors, taking an argument of type <*type*>
- %start *symbol … symbol*
- Declare given symbols as entry points; functions of same names in <*grammar*>.ml

Ocamlyacc <*declarations*>

- %type <*type*> *symbol … symbol*

  Specify type of attributes for given symbols. Mandatory for start symbols
- %left *symbol … symbol*
- %right *symbol … symbol*
- %nonassoc *symbol … symbol*

  Associate precedence and associativity to given symbols. Same line,same precedence; earlier line, lower precedence (broadest scope)

Ocamlyacc <*rules*>

- *nonterminal* :

  symbol ... *symbol* { *semantic_action* }

  | ...

  | *symbol* ... *symbol* { *semantic_action* }

  ;
- Semantic actions are arbitrary Ocaml expressions
- Must be of same type as declared (or inferred) for *nonterminal*
- Access semantic attributes (values) of symbols by position: $1 for first symbol, $2 to second …

Example - Base types

```
(* File: expr.ml *)
type expr =
    Term_as_Expr of term
  | Plus_Expr of (term * expr)
  | Minus_Expr of (term * expr)
and term =
    Factor_as_Term of factor
  | Mult_Term of (factor * term)
  | Div_Term of (factor * term)
and factor =
    Id_as_Factor of string
  | Parenthesized_Expr_as_Factor of expr
```

Example - Lexer (exprlex.mll)

```
{ (*open Exprparse*) }
let numeric = ['0' - '9']
let letter =['a' - 'z' 'A' - 'Z']
rule token = parse
  | "+" {Plus_token}
  | "-"  {Minus_token}
  | "*"  {Times_token}
  | "/"  {Divide_token}
  | "("  {Left_parenthesis}
  | ")"  {Right_parenthesis}
  | letter (letter|numeric|"_")* as id  {Id_token id}
  | [' ' '\t' '\n'] {token lexbuf}
  | eof {EOL}
```

Example - Parser (exprparse.mly)

```
%{ open Expr
%}
%token <string> Id_token
%token Left_parenthesis Right_parenthesis
%token Times_token Divide_token
%token Plus_token Minus_token
%token EOL
%start main
%type <expr> main
%%
```

Example - Parser (exprparse.mly)

```
expr:
  term
     { Term_as_Expr $1 }
 | term Plus_token expr
     { Plus_Expr ($1, $3) }
 | term Minus_token expr
     { Minus_Expr ($1, $3) }
```

Example - Parser (exprparse.mly)

```
term:
   factor
      { Factor_as_Term $1 }
 | factor Times_token term
      { Mult_Term ($1, $3) }
 | factor Divide_token term
      { Div_Term ($1, $3) }
```

Example - Parser (exprparse.mly)

```
factor:
   Id_token
       { Id_as_Factor $1 }
 | Left_parenthesis expr Right_parenthesis
       {Parenthesized_Expr_as_Factor $2 }
main:
 | expr EOL
        { $1 }
```

Example - Using Parser

```
# #use "expr.ml";;
…
# #use "exprparse.ml";;
…
# #use "exprlex.ml";;
…
# let test s =
  let lexbuf = Lexing.from_string (s^"\n") in
      main token lexbuf;;
```

Example - Using Parser

```
# test "a + b";;
- : expr =
Plus_Expr
 (Factor_as_Term (Id_as_Factor "a"),
  Term_as_Expr (Factor_as_Term
   (Id_as_Factor "b")))
```

Ambiguous Grammars and Languages

- A BNF grammar is *ambiguous* if its language contains strings for which there is more than one parse tree
- If all BNF's for a language are ambiguous then the language is *inherently ambiguous*
- Your job: *disambiguate given grammar*
  - Write a new grammar that is **not** ambiguous that generates the **same** language

Two Major Sources of Ambiguity

- Lack of determination of operator precedence
- Lack of determination of operator assoicativity

- Not the only sources of ambiguity

How to Enforce Associativity

- Have at most one recursive call per production

- When two or more recursive calls would be natural leave right-most one for right assoicativity, left-most one for left assoiciativity

Example

- <Sum> ::= 0 | 1 | <Sum> + <Sum>
  | (<Sum>)
- Becomes
  - <Sum> ::= <Num> | <Num> + <Sum>
  - <Num> ::= 0 | 1 | (<Sum>)

Operator Precedence

- Operators of highest precedence evaluated first (bind more tightly).
- Precedence for infix binary operators given in following table
- Needs to be reflected in grammar

Predence in Grammar

- Higher precedence translates to longer derivation chain
- Example:

```
<exp> ::= <id> | <exp> + <exp>
          | <exp> * <exp>
```

- Becomes

```
<exp> ::= <mult_exp>
          | <exp> + <mult_exp>
<mult_exp> ::= <id> | <mult_exp> * <id>
```

Recursive Descent Parsing

- Recursive descent parsers are a class of parsers derived fairly directly from BNF grammars

- A recursive descent parser traces out a parse tree in top-down order, corresponding to a left-most derivation (LL - left-to-right scanning, leftmost derivation)

Recursive Descent Parsing

- Each nonterminal in the grammar has a subprogram associated with it; the subprogram parses all phrases that the nonterminal can generate

- Each nonterminal in right-hand side of a rule corresponds to a recursive call to the associated subprogram

Recursive Descent Parsing

- Each subprogram must be able to decide how to begin parsing by looking at the left-most character in the string to be parsed
  - May do so directly, or indirectly by calling another parsing subprogram
- Recursive descent parsers, like other top-down parsers, cannot be built from left-recursive grammars
  - Sometimes can modify grammar to suit

Sample Grammar

<expr> ::= <term> | <term> + <expr>
| <term> - <expr>

<term> ::= <factor> | <factor> * <term>
| <factor> / <term>

<factor> ::= <id> | (<expr>)

Tokens as OCaml Types

- \+ - * / ( ) <id>
- Becomes an OCaml datatype

```
type token =
    Id_token of string
  | Left_parenthesis | Right_parenthesis
  | Times_token | Divide_token
  | Plus_token | Minus_token
```

Parse Trees as Datatypes

<expr> ::= <term> | <term> + <expr>
| <term> - <expr>

```
type expr =
    Term_as_Expr of term
  | Plus_Expr of (term * expr)
  | Minus_Expr of (term * expr)
```

Parse Trees as Datatypes

<term> ::= <factor> | <factor> * <term>
| <factor> / <term>

```
and term =
    Factor_as_Term of factor
  | Mult_Term of (factor * term)
  | Div_Term of (factor * term)
```

Parse Trees as Datatypes

<factor> ::= <id> | (<expr>)

```
and factor =
    Id_as_Factor of string
  | Parenthesized_Expr_as_Factor of expr
```

Parsing Lists of Tokens

- Will create three mutually recursive functions:
  - expr : token list -> (expr * token list)
  - term : token list -> (term * token list)
  - factor : token list -> (factor * token list)
- Each parses what it can and gives back parse and remaining tokens

Parsing an Expression

<expr> ::= <term> [(+ | -) <expr>]

```
let rec expr tokens =
  (match term tokens
    with ( term_parse , tokens_after_term) ->
      (match tokens_after_term
        with( Plus_token  :: tokens_after_plus) ->
```

Parsing an Expression

<expr> ::= <term> [(+ | -) <expr>]

```
let rec expr tokens =
  (match term tokens
    with ( term_parse , tokens_after_term) ->
     (match tokens_after_term
      with ( Plus_token  :: tokens_after_plus) ->
```

Parsing a Plus Expression

<expr> ::= <term> [(+ | -) <expr>]

```
let rec expr tokens =
  (match term tokens
    with ( term_parse , tokens_after_term) ->
      (match tokens_after_term
       with ( Plus_token  :: tokens_after_plus) ->
```

Parsing a Plus Expression

<expr> ::= <term> [(+ | -) <expr>]

```
let rec expr tokens =
  (match term tokens
    with ( term_parse , tokens_after_term) ->
      (match tokens_after_term
        with ( Plus_token  :: tokens_after_plus) ->
```

Parsing a Plus Expression

<expr> ::= <term> [(+ | -) <expr>]

```
let rec expr tokens =
  (match term tokens
    with ( term_parse , tokens_after_term) ->
      (match tokens_after_term
       with ( Plus_token  :: tokens_after_plus) ->
```

Parsing a Plus Expression

<expr> ::= <term> + <expr>

(match expr tokens_after_plus

with (expr_parse , tokens_after_expr) ->

(Plus_Expr (term_parse , expr_parse),

tokens_after_expr))

Parsing a Plus Expression

<expr> ::= <term> + <expr>

```
(match expr tokens_after_plus
    with ( expr_parse , tokens_after_expr) ->
    ( Plus_Expr ( term_parse , expr_parse ),
     tokens_after_expr))
```

Building Plus Expression Parse Tree

<expr> ::= <term> + <expr>

```
(match expr tokens_after_plus
    with ( expr_parse , tokens_after_expr) ->
   ( Plus_Expr ( term_parse , expr_parse ),
     tokens_after_expr))
```

Parsing a Minus Expression

<expr> ::= <term> - <expr>

```
| ( Minus_token  :: tokens_after_minus) ->
   (match expr tokens_after_minus
   with ( expr_parse  , tokens_after_expr) ->
  ( Minus_Expr  ( term_parse  ,  expr_parse  ),
   tokens_after_expr))
```

Parsing a Minus Expression

<expr> ::= <term> - <expr>

```
| ( Minus_token :: tokens_after_minus) ->
   (match expr tokens_after_minus
   with ( expr_parse , tokens_after_expr) ->
  ( Minus_Expr ( term_parse , expr_parse ),
  tokens_after_expr))
```

Parsing an Expression as a Term

<expr> ::= <term>

| _ -> (Term_as_Expr term_parse , tokens_after_term)))

- Code for **term** is same except for replacing addition with multiplication and subtraction with division

Parsing Factor as Id

<factor> ::= <id>

```
and factor  tokens =
 (match tokens
  with (Id_token id_name :: tokens_after_id) =
   ( Id_as_Factor  id_name, tokens_after_id)
```

Parsing Factor as Parenthesized Expression

<factor> ::= (<expr>)

| factor (Left_parenthesis :: tokens) =

(match expr tokens

with (expr_parse , tokens_after_expr) ->

Parsing Factor as Parenthesized Expression

<factor> ::= (<expr>)

(match tokens_after_expr

with Right_parenthesis :: tokens_after_rparen ->

(Parenthesized_Expr_as_Factor expr_parse ,
tokens_after_rparen)

Error Cases

- What if no matching right parenthesis?

  | _ -> raise (Failure "No matching rparen")))

- What if no leading id or left parenthesis?

| _ -> raise (Failure "No id or lparen"));;

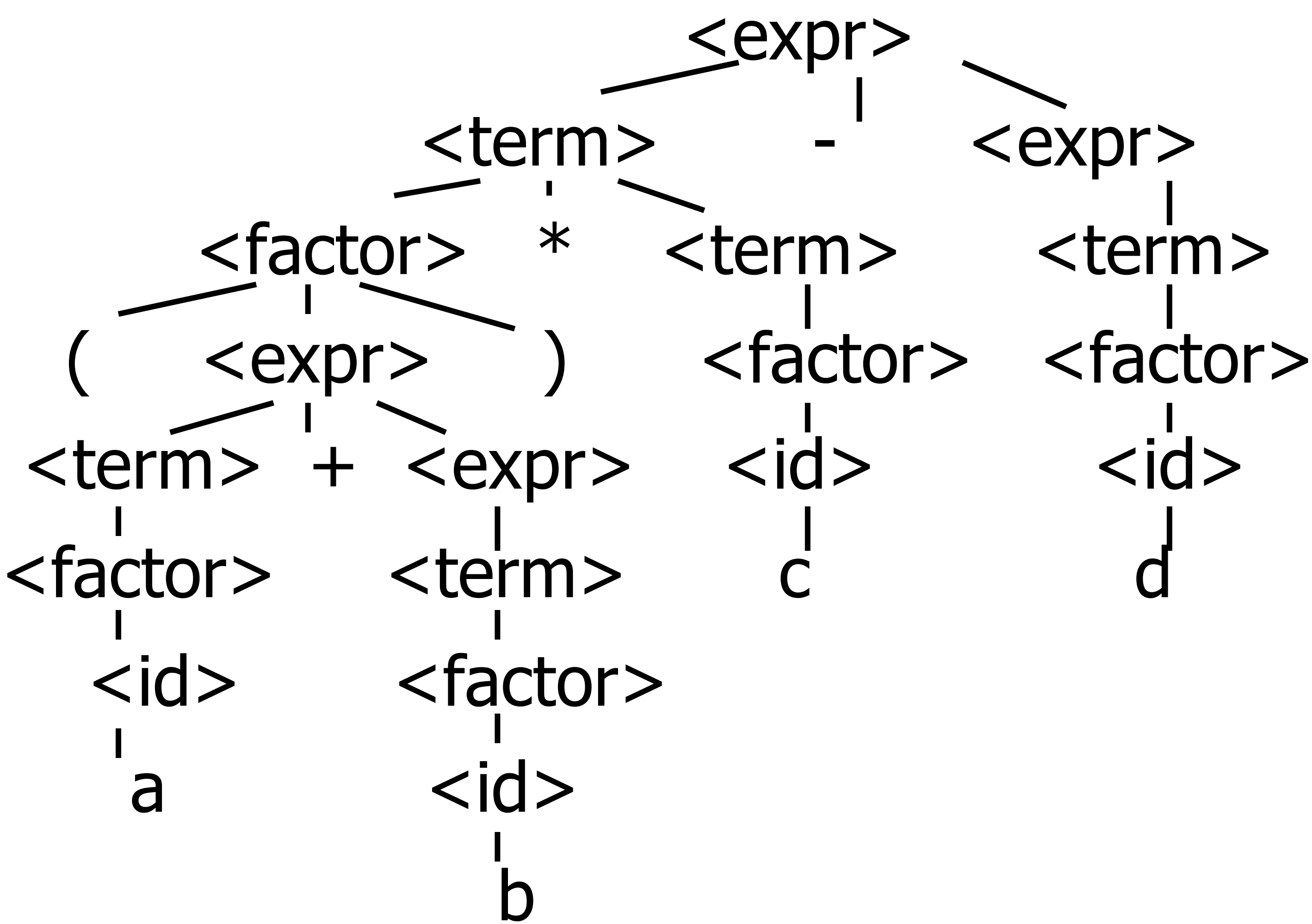
(a + b) * c - d

expr [Left_parenthesis; Id_token "a”; Plus_token; Id_token "b”; Right_parenthesis; Times_token; Id_token "c”; Minus_token; Id_token "d"];;

(a + b) * c - d

```
- : expr * token list =
(Minus_Expr
  (Mult_Term
    (Parenthesized_Expr_as_Factor
      (Plus_Expr
        (Factor_as_Term (Id_as_Factor "a"),
         Term_as_Expr (Factor_as_Term
   (Id_as_Factor "b")))),
     Factor_as_Term (Id_as_Factor "c")),
   Term_as_Expr (Factor_as_Term (Id_as_Factor
   "d"))),
 [])
```

(a + b) * c – d


<expr>
<term>
-
<expr>
<factor>
*
<term>
<term>
(
<expr>
)
<factor>
<factor>
<term>
+
<expr>
<id>
<id>
<factor>
<term>
c
d
<id>
<factor>
a
<id>
b

a + b * c – d

expr [Id_token "a"; Plus_token; Id_token "b"; Times_token; Id_token "c"; Minus_token; Id_token "d"];;

- : expr * token list =

(Plus_Expr

(Factor_as_Term (Id_as_Factor "a"),

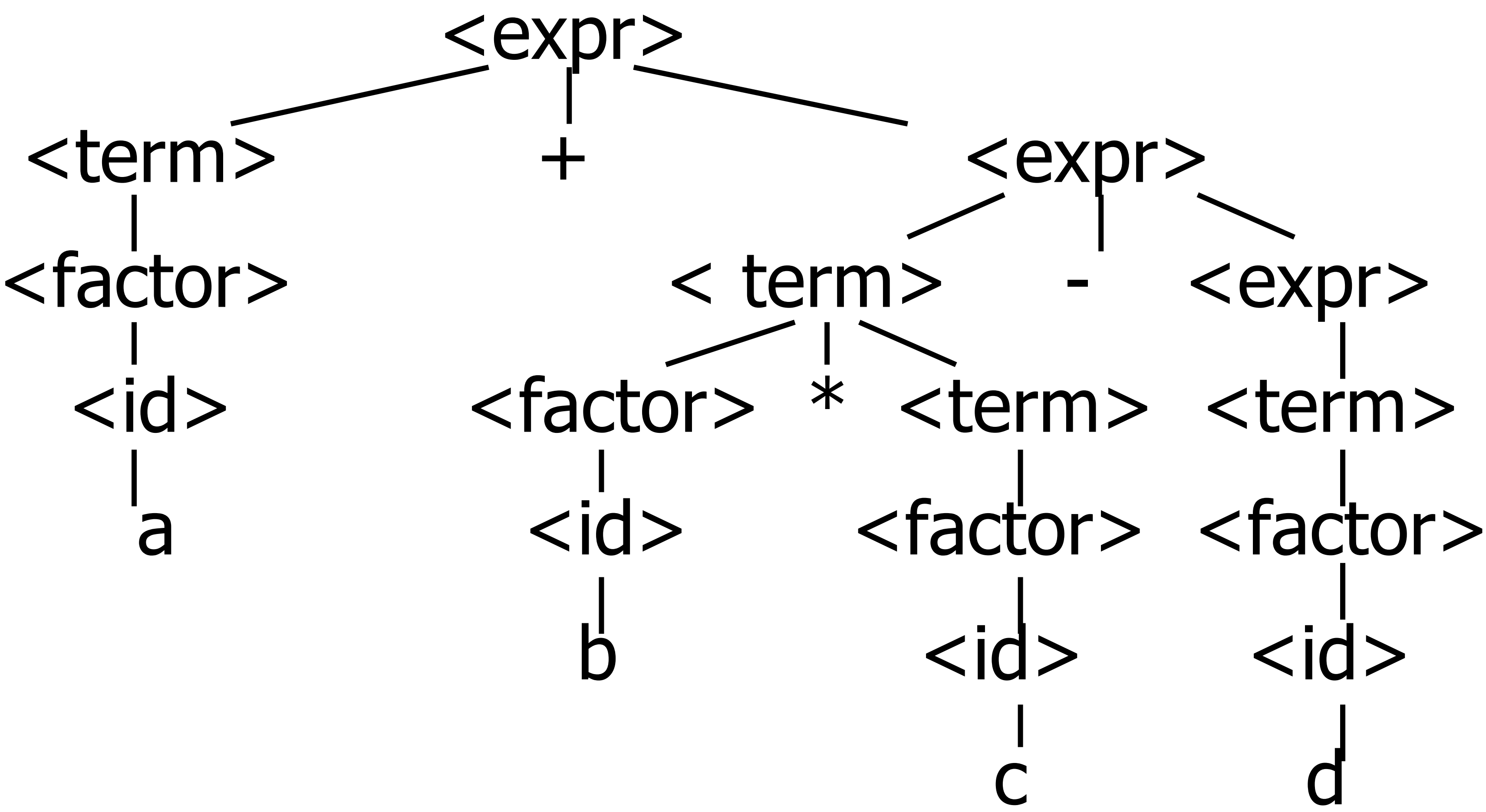
Minus_Expr

(Mult_Term (Id_as_Factor "b", Factor_as_Term (Id_as_Factor "c")),

Term_as_Expr (Factor_as_Term (Id_as_Factor "d")))),

[])

a + b * c – d


<expr>
<term>
+
<expr>
<factor>
< term>
-
<expr>
<id>
<factor>
*
<term>
<term>
a
<id>
<factor>
<factor>
b
<id>
<id>
c
d

(a + b * c - d

expr [Left_parenthesis; Id_token "a”; Plus_token; Id_token "b”; Times_token; Id_token "c”; Minus_token; Id_token "d"];;

Exception: Failure "No matching rparen".

Can’t parse because it was expecting a right parenthesis but it got to the end without finding one

a + b) * c - d *)

expr [Id_token "a”; Plus_token; Id_token "b”; Right_parenthesis; Times_token; Id_token "c”; Minus_token; Id_token "d"];;

- : expr * token list =

(Plus_Expr

(Factor_as_Term (Id_as_Factor "a"),

Term_as_Expr (Factor_as_Term (Id_as_Factor "b"))),

[Right_parenthesis; Times_token; Id_token "c"; Minus_token; Id_token "d"])

Parsing Whole String

- Q: How to guarantee whole string parses?
- A: Check returned tokens empty

```
let parse tokens =
    match expr tokens
     with (expr_parse, []) -> expr_parse
     | _ -> raise (Failure “No parse");;
```

- Fixes <expr> as start symbol

Streams in Place of Lists

- More realistically, we don't want to create the entire list of tokens before we can start parsing
- We want to generate one token at a time and use it to make one step in parsing
- Will use (token * (unit -> token)) or (token * (unit -> token option))

  in place of token list

Problems for Recursive-Descent Parsing

- Left Recursion:

  A ::= Aw

  translates to a subroutine that loops forever
- Indirect Left Recursion:

  A ::= Bw

  B ::= Av

  causes the same problem

Problems for Recursive-Descent Parsing

- Parser must always be able to choose the next action based only only the very next token

- Pairwise Disjointedness Test: Can we always determine which rule (in the non-extended BNF) to choose based on just the first token

Pairwise Disjointedness Test

- For each rule

  $A ::= y$

Calculate

FIRST (y) =

$\{a \mid y =>^* aw\} \cup \{\varepsilon \mid \text{if } y =>^* \varepsilon\}$

- For each pair of rules $A ::= y$ and $A ::= z$, require FIRST(y) $\cap$ FIRST(z) = { }

Example

Grammar:

<S> ::= <A> a <B> b

<A> ::= <A> b | b

<B> ::= a <B> | a

FIRST (<A> b) = {b}

FIRST (b) = {b}

Rules for <A> not pairwise disjoint

Eliminating Left Recursion

- Rewrite grammar to shift left recursion to right recursion
  - Changes associativity
- Given

<expr> ::= <expr> + <term> and

<expr> ::= <term>

- Add new non-terminal <e> and replace above rules with

<expr> ::= <term><e>

<e> ::= + <term><e> | ε

Factoring Grammar

- Test too strong: Can’t handle

  <expr> ::= <term> [(+ | -) <expr>]

- Answer: Add new non-terminal and replace above rules by

  <expr> ::= <term><e>

  <e> ::= + <term><e>

  <e> ::= - <term><e>

  <e> ::= ε

- You are delaying the decision point

Example

Both <A> and <B> have problems:

<S> ::= <A> a <B> b

<A> ::= <A> b | b

<B> ::= a <B> | a

Transform grammar to:

<S> ::= <A> a <B> b

<A> ::-= b<A1>

<A1> :: b<A1> | ε

<B> ::= a<B1>

<B1> ::= a<B1> | ε

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Recursive Descent Parsing

- Recursive descent parsers are a class of parsers derived fairly directly from BNF grammars
- A recursive descent parser traces out a parse tree in top-down order, corresponding to a left-most derivation (LL - left-to-right scanning, leftmost derivation)

Recursive Descent Parsing

- Each nonterminal in the grammar has a subprogram associated with it; the subprogram parses all phrases that the nonterminal can generate

- Each nonterminal in right-hand side of a rule corresponds to a recursive call to the associated subprogram

Recursive Descent Parsing

- Each subprogram must be able to decide how to begin parsing by looking at the left-most character in the string to be parsed
  - May do so directly, or indirectly by calling another parsing subprogram
- Recursive descent parsers, like other top-down parsers, cannot be built from left-recursive grammars
  - Sometimes can modify grammar to suit

Sample Grammar

<expr> ::= <term> | <term> + <expr>
| <term> - <expr>

<term> ::= <factor> | <factor> * <term>
| <factor> / <term>

<factor> ::= <id> | (<expr>)

Tokens as OCaml Types

- \+ - * / ( ) <id>
- Becomes an OCaml datatype

```
type token =
    Id_token of string
  | Left_parenthesis | Right_parenthesis
  | Times_token | Divide_token
  | Plus_token | Minus_token
```

Parse Trees as Datatypes

<expr> ::= <term> | <term> + <expr>
| <term> - <expr>

```
type expr =
    Term_as_Expr of term
  | Plus_Expr of (term * expr)
  | Minus_Expr of (term * expr)
```

Parse Trees as Datatypes

<term> ::= <factor> | <factor> * <term>
| <factor> / <term>

```
and term =
    Factor_as_Term of factor
  | Mult_Term of (factor * term)
  | Div_Term of (factor * term)
```

Parse Trees as Datatypes

<factor> ::= <id> | (<expr>)

```
and factor =
    Id_as_Factor of string
  | Parenthesized_Expr_as_Factor of expr
```

Parsing Lists of Tokens

- Will create three mutually recursive functions:
  - expr : token list -> (expr * token list)
  - term : token list -> (term * token list)
  - factor : token list -> (factor * token list)
- Each parses what it can and gives back parse and remaining tokens

Parsing an Expression

<expr> ::= <term> [(+ | -) <expr>]

```
let rec expr tokens =
  (match term tokens
    with ( term_parse , tokens_after_term) ->
      (match tokens_after_term
        with( Plus_token  :: tokens_after_plus) ->
```

Parsing an Expression

<expr> ::= <term> [(+ | -) <expr>]

```
let rec expr tokens =
  (match term tokens
    with ( term_parse , tokens_after_term) ->
      (match tokens_after_term
        with ( Plus_token  :: tokens_after_plus) ->
```

Parsing a Plus Expression

<expr> ::= <term> [(+ | -) <expr>]

```
let rec expr tokens =
  (match term tokens
    with ( term_parse , tokens_after_term) ->
      (match tokens_after_term
        with ( Plus_token  :: tokens_after_plus) ->
```

Parsing a Plus Expression

<expr> ::= <term> [(+ | -) <expr>]

```
let rec expr tokens =
  (match term tokens
    with ( term_parse , tokens_after_term) ->
      (match tokens_after_term
        with ( Plus_token  :: tokens_after_plus) ->
```

Parsing a Plus Expression

<expr> ::= <term> [(+ | -) <expr>]

```
let rec expr tokens =
  (match term tokens
    with ( term_parse , tokens_after_term) ->
      (match tokens_after_term
        with ( Plus_token  :: tokens_after_plus) ->
```

Parsing a Plus Expression

<expr> ::= <term> + <expr>

(match expr tokens_after_plus

with (expr_parse , tokens_after_expr) ->

(Plus_Expr (term_parse , expr_parse),

tokens_after_expr))

Parsing a Plus Expression

<expr> ::= <term> + <expr>

(match expr tokens_after_plus

with (expr_parse , tokens_after_expr) ->

(Plus_Expr (term_parse , expr_parse),

tokens_after_expr))

Building Plus Expression Parse Tree

<expr> ::= <term> + <expr>

```
(match expr tokens_after_plus
    with ( expr_parse , tokens_after_expr) ->
   ( Plus_Expr ( term_parse , expr_parse ),
    tokens_after_expr))
```

Parsing a Minus Expression

<expr> ::= <term> - <expr>

```
| ( Minus_token  :: tokens_after_minus) ->
   (match expr tokens_after_minus
    with ( expr_parse  , tokens_after_expr) ->
  ( Minus_Expr  ( term_parse  ,  expr_parse  ),
   tokens_after_expr))
```

Parsing a Minus Expression

<expr> ::= <term> - <expr>

| (Minus_token :: tokens_after_minus) ->
 (match expr tokens_after_minus
 with (expr_parse , tokens_after_expr) ->
 (Minus_Expr (term_parse , expr_parse),
 tokens_after_expr))

Parsing an Expression as a Term

<expr> ::= <term>

| _ -> (Term_as_Expr term_parse ,
tokens_after_term)))

- Code for **term** is same except for replacing addition with multiplication and subtraction with division

Parsing Factor as Id

<factor> ::= <id>

```
and factor  tokens =
 (match tokens
  with (Id_token id_name :: tokens_after_id) =
   ( Id_as_Factor  id_name, tokens_after_id)
```

Parsing Factor as Parenthesized Expression

<factor> ::= (<expr>)

| factor (Left_parenthesis :: tokens) =

(match expr tokens

with (expr_parse , tokens_after_expr) ->

Parsing Factor as Parenthesized Expression

<factor> ::= (<expr>)

(match tokens_after_expr

with Right_parenthesis :: tokens_after_rparen ->

(Parenthesized_Expr_as_Factor expr_parse ,
tokens_after_rparen)

Error Cases

- What if no matching right parenthesis?

  | _ -> raise (Failure "No matching rparen")))

- What if no leading id or left parenthesis?

| _ -> raise (Failure "No id or lparen"));;

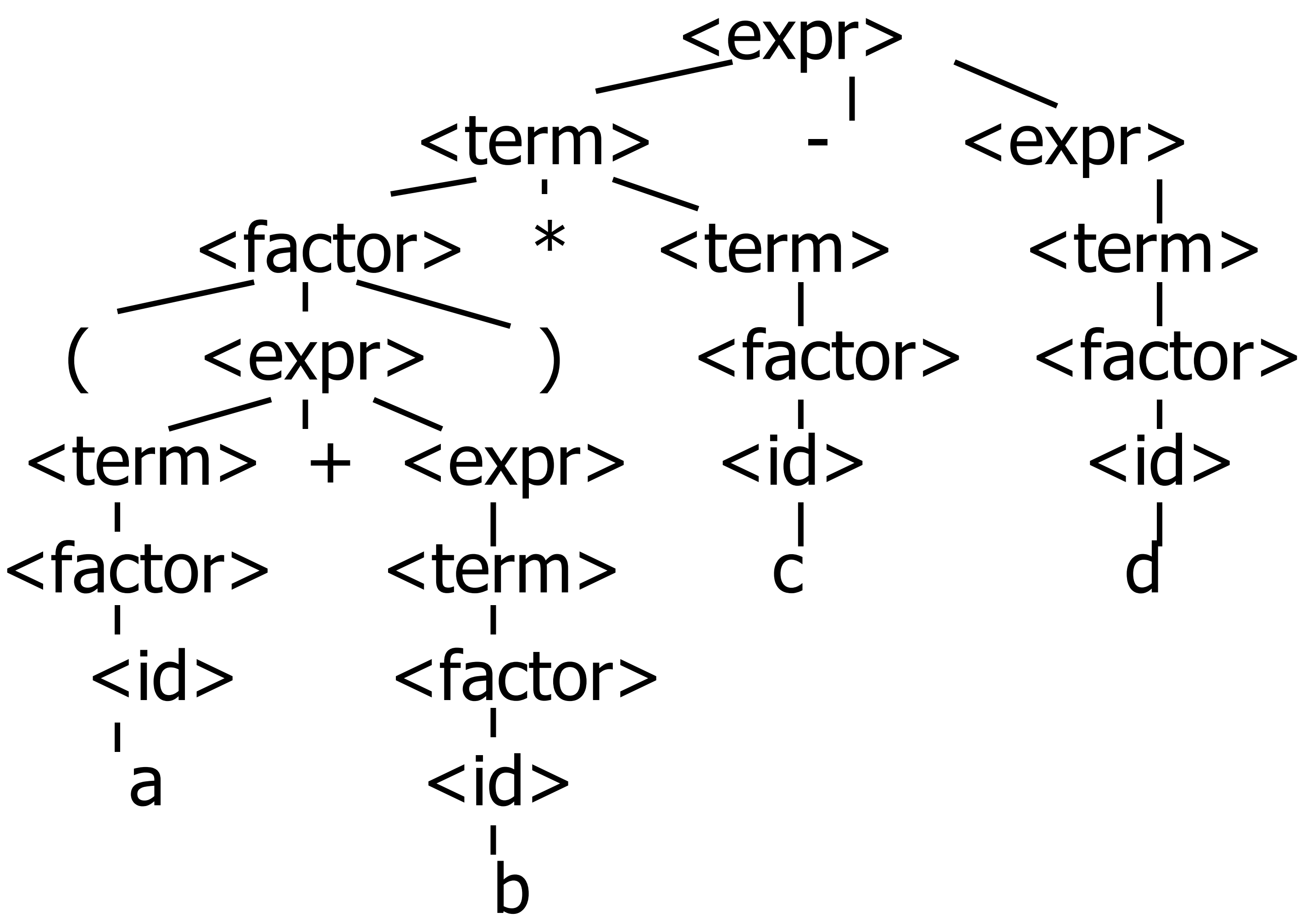
(a + b) * c - d

expr [Left_parenthesis; Id_token "a”; Plus_token; Id_token "b”; Right_parenthesis; Times_token; Id_token "c”; Minus_token; Id_token "d"];;

(a + b) * c - d

- : expr * token list =
(Minus_Expr
 (Mult_Term
  (Parenthesized_Expr_as_Factor
   (Plus_Expr
    (Factor_as_Term (Id_as_Factor "a"),
     Term_as_Expr (Factor_as_Term
 (Id_as_Factor "b")))),
  Factor_as_Term (Id_as_Factor "c")),
 Term_as_Expr (Factor_as_Term (Id_as_Factor
 "d"))),
[])

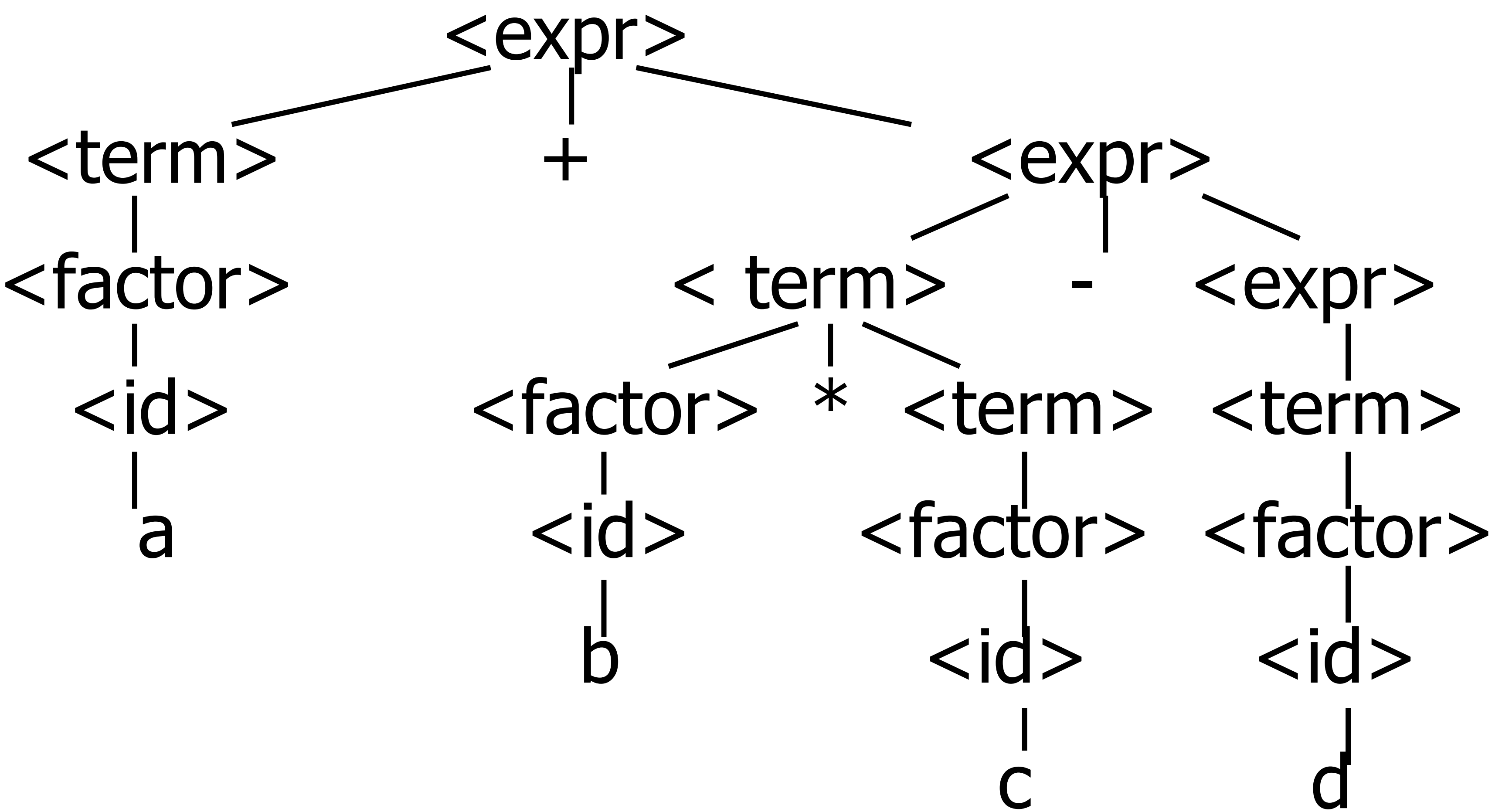
(a + b) * c – d


<expr>
<term>
-
<expr>
<factor>
*
<term>
<term>
(
<expr>
)
<factor>
<factor>
<term>
+
<expr>
<id>
<id>
<factor>
<term>
c
d
<id>
<factor>
a
<id>
b

a + b * c – d

```
# expr [Id_token "a”; Plus_token; Id_token "b”;
   Times_token; Id_token "c”; Minus_token;
        Id_token "d"];;
- : expr * token list =
(Plus_Expr
  (Factor_as_Term (Id_as_Factor "a"),
   Minus_Expr
    (Mult_Term (Id_as_Factor "b", Factor_as_Term
   (Id_as_Factor "c")),
     Term_as_Expr (Factor_as_Term (Id_as_Factor
   "d")))),
 [])
```

a + b * c – d


<expr>
<term>
+
<expr>
<factor>
< term>
-
<expr>
<id>
<factor>
*
<term>
<term>
a
<id>
<factor>
<factor>
b
<id>
<id>
c
d

(a + b * c - d

expr [Left_parenthesis; Id_token "a”; Plus_token; Id_token "b”; Times_token; Id_token "c”; Minus_token; Id_token "d"];;

Exception: Failure "No matching rparen".

Can’t parse because it was expecting a right parenthesis but it got to the end without finding one

a + b) * c - d *)

expr [Id_token "a”; Plus_token; Id_token "b”; Right_parenthesis; Times_token; Id_token "c”; Minus_token; Id_token "d"];;

- : expr * token list =

(Plus_Expr

(Factor_as_Term (Id_as_Factor "a"),

Term_as_Expr (Factor_as_Term (Id_as_Factor "b"))),

[Right_parenthesis; Times_token; Id_token "c"; Minus_token; Id_token "d"])

Parsing Whole String

- Q: How to guarantee whole string parses?
- A: Check returned tokens empty

```
let parse tokens =
    match expr tokens
     with (expr_parse, []) -> expr_parse
     | _ -> raise (Failure "No parse");;
```

- Fixes <expr> as start symbol

Streams in Place of Lists

- More realistically, we don't want to create the entire list of tokens before we can start parsing
- We want to generate one token at a time and use it to make one step in parsing
- Will use (token * (unit -> token)) or (token * (unit -> token option))

  in place of token list

Problems for Recursive-Descent Parsing

- Left Recursion:

  A ::= Aw

  translates to a subroutine that loops forever
- Indirect Left Recursion:

  A ::= Bw

  B ::= Av

  causes the same problem

Problems for Recursive-Descent Parsing

- Parser must always be able to choose the next action based only only the very next token

- Pairwise Disjointedness Test: Can we always determine which rule (in the non-extended BNF) to choose based on just the first token

Pairwise Disjointedness Test

- For each rule

  $A ::= y$

Calculate

FIRST (y) =

$\{a \mid y =>^* aw\} \cup \{\varepsilon \mid \text{if } y =>^* \varepsilon\}$

- For each pair of rules $A ::= y$ and $A ::= z$, require $\text{FIRST}(y) \cap \text{FIRST}(z) = \{\ \}$

Example

Grammar:

<S> ::= <A> a <B> b

<A> ::= <A> b | b

<B> ::= a <B> | a

FIRST (<A> b) = {b}

FIRST (b) = {b}

Rules for <A> not pairwise disjoint

Eliminating Left Recursion

- Rewrite grammar to shift left recursion to right recursion
  - Changes associativity
- Given

<expr> ::= <expr> + <term> and

<expr> ::= <term>

- Add new non-terminal <e> and replace above rules with

<expr> ::= <term><e>

<e> ::= + <term><e> | ε

Factoring Grammar

- Test too strong: Can't handle

  <expr> ::= <term> [(+ | -) <expr>]

- Answer: Add new non-terminal and replace above rules by

  <expr> ::= <term><e>

  <e> ::= + <term><e>

  <e> ::= - <term><e>

  <e> ::= ε

- You are delaying the decision point

Example

Both <A> and <B> have problems:

<S> ::= <A> a <B> b

<A> ::= <A> b | b

<B> ::= a <B> | a

Transform grammar to:

<S> ::= <A> a <B> b

<A> ::-= b<A1>

<A1> :: b<A1> | ε

<B> ::= a<B1>

<B1> ::= a<B1> | ε

Semantics

- Expresses the meaning of syntax
- Static semantics
  - Meaning based only on the form of the expression without executing it
  - Usually restricted to type checking / type inference

Dynamic semantics

- Method of describing meaning of executing a program
- Several different types:
  - Operational Semantics
  - Axiomatic Semantics
  - Denotational Semantics

Dynamic Semantics

- Different languages better suited to different types of semantics
- Different types of semantics serve different purposes

Operational Semantics

- Start with a simple notion of machine
- Describe how to execute (implement) programs of language on virtual machine, by describing how to execute each program statement (ie, following the *structure* of the program)
- Meaning of program is how its execution changes the state of the machine
- Useful as basis for implementations

Axiomatic Semantics

- Also called Floyd-Hoare Logic
- Based on formal logic (first order predicate calculus)
- Axiomatic Semantics is a logical system built from *axioms* and *inference rules*
- Mainly suited to simple imperative programming languages

Axiomatic Semantics

- Used to formally prove a property (*post-condition*) of the *state* (the values of the program variables) after the execution of program, assuming another property (*pre-condition*) of the state before execution
- Written :

  {Precondition} Program {Postcondition}
- Source of idea of *loop invariant*

Denotational Semantics

- Construct a function $\mathcal{M}$ assigning a mathematical meaning to each program construct
- Lambda calculus often used as the range of the meaning function
- Meaning function is compositional: meaning of construct built from meaning of parts
- Useful for proving properties of programs

Natural Semantics

- Aka Structural Operational Semantics, aka "Big Step Semantics"
- Provide value for a program by rules and derivations, similar to type derivations
- Rule conclusions look like

$$(C, m) \Downarrow m'$$

or

$$(E, m) \Downarrow v$$

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Semantics

- Expresses the meaning of syntax
- Static semantics
  - Meaning based only on the form of the expression without executing it
  - Usually restricted to type checking / type inference

Dynamic semantics

- Method of describing meaning of executing a program
- Several different types:
  - Operational Semantics
  - Axiomatic Semantics
  - Denotational Semantics

Dynamic Semantics

- Different languages better suited to different types of semantics
- Different types of semantics serve different purposes

Operational Semantics

- Start with a simple notion of machine
- Describe how to execute (implement) programs of language on virtual machine, by describing how to execute each program statement (ie, following the *structure* of the program)
- Meaning of program is how its execution changes the state of the machine
- Useful as basis for implementations

Axiomatic Semantics

- Also called Floyd-Hoare Logic
- Based on formal logic (first order predicate calculus)
- Axiomatic Semantics is a logical system built from *axioms* and *inference rules*
- Mainly suited to simple imperative programming languages

Axiomatic Semantics

- Used to formally prove a property (*post-condition*) of the *state* (the values of the program variables) after the execution of program, assuming another property (*pre-condition*) of the state before execution
- Written :

  {Precondition} Program {Postcondition}
- Source of idea of *loop invariant*

Denotational Semantics

- Construct a function $\mathcal{M}$ assigning a mathematical meaning to each program construct
- Lambda calculus often used as the range of the meaning function
- Meaning function is compositional: meaning of construct built from meaning of parts
- Useful for proving properties of programs

Natural Semantics

- Aka Structural Operational Semantics, aka "Big Step Semantics"
- Provide value for a program by rules and derivations, similar to type derivations
- Rule conclusions look like

$$(C, m) \Downarrow m'$$

or

$$(E, m) \Downarrow v$$

Simple Imperative Programming Language

- $I \in Identifiers$
- $N \in Numerals$
- B ::= true | false | B & B | B or B | not B | $E < E$ | $E = E$
- E::= N | I | $E + E$ | $E * E$ | $E - E$ | $- E$
- C::= skip | C;C | I ::= E | if B then C else C fi | while B do C od

Natural Semantics of Atomic Expressions

- Identifiers: $(I,m) \Downarrow m(I)$
- Numerals are values: $(N,m) \Downarrow N$
- Booleans: $(\text{true},m) \Downarrow \text{true}$

  $(\text{false},m) \Downarrow \text{false}$

Booleans:

$$\frac{(B, m) \Downarrow \text{false}}{(B \,\&\, B', m) \Downarrow \text{false}} \qquad \frac{(B, m) \Downarrow \text{true} \quad (B', m) \Downarrow b}{(B \,\&\, B', m) \Downarrow b}$$

$$\frac{(B, m) \Downarrow \text{true}}{(B \text{ or } B', m) \Downarrow \text{true}} \qquad \frac{(B, m) \Downarrow \text{false} \quad (B', m) \Downarrow b}{(B \text{ or } B', m) \Downarrow b}$$

$$\frac{(B, m) \Downarrow \text{true}}{(\text{not } B, m) \Downarrow \text{false}} \qquad \frac{(B, m) \Downarrow \text{false}}{(\text{not } B, m) \Downarrow \text{true}}$$

Relations

$$\frac{(E, m) \Downarrow U \qquad (E', m) \Downarrow V \quad U \sim V = b}{(E \sim E', m) \Downarrow b}$$

- By $U \sim V = b$, we mean does (the meaning of) the relation ~ hold on the meaning of U and V
- May be specified by a mathematical expression/equation or rules matching U and V

Arithmetic Expressions

$$\frac{(E, m) \Downarrow U \quad (E', m) \Downarrow V \quad U \; op \; V = N}{(E \; op \; E', m) \Downarrow N}$$

where N is the specified value for $U \; op \; V$

Commands

Skip: $(\text{skip}, m) \Downarrow m$

Assignment:
$$\frac{(E, m) \Downarrow V}{(I ::= E, m) \Downarrow m[I <\!\!-\!- V]}$$

Sequencing:
$$\frac{(C, m) \Downarrow m' \quad (C', m') \Downarrow m''}{(C;C', m) \Downarrow m''}$$

If Then Else Command

$$\frac{(B,m) \Downarrow \text{true} \quad (C,m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$$

$$\frac{(B,m) \Downarrow \text{false} \quad (C',m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$$

While Command

$$\frac{(B,m) \Downarrow \text{false}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m}$$

$$\frac{(B,m)\Downarrow\text{true} \quad (C,m)\Downarrow m' \quad (\text{while } B \text{ do } C \text{ od}, m')\Downarrow m''}{(\text{while } B \text{ do C od}, m) \Downarrow m''}$$

Example: If Then Else Rule

$$\frac{}{(\text{if } x > 5 \text{ then } y{:}= 2 + 3 \text{ else } y{:}=3 + 4 \text{ fi}, \{x \to 7\}) \Downarrow\ ?}$$

Example: If Then Else Rule

$$\frac{\overline{\text{(x > 5, \{x -> 7\})} \Downarrow ?}}{\text{(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi, \{x -> 7\})} \Downarrow ?}$$

Example: Arith Relation

$$? > ? = ?$$

$$\dfrac{\dfrac{(x,\{x\text{->}7\})\Downarrow ? \qquad (5,\{x\text{->}7\})\Downarrow ?}{(x > 5,\ \{x \text{ -> } 7\})\Downarrow ?}}{\begin{array}{c}(\text{if } x > 5 \text{ then } y{:}= 2 + 3 \text{ else } y{:}=3 + 4 \text{ fi},\\ \{x \text{ -> } 7\}) \Downarrow ?\end{array}}$$

Example: Identifier(s)

$$\dfrac{\dfrac{7 > 5 = \text{true}}{(x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5}}{(x > 5,\ \{x \text{ -> } 7\})\Downarrow ?}$$

$$\dfrac{(x > 5,\ \{x \text{ -> } 7\})\Downarrow ?}{(\text{if } x > 5 \text{ then } y{:}= 2 + 3 \text{ else } y{:}=3 + 4 \text{ fi},\ \{x \text{ -> } 7\}) \Downarrow ?}$$

Example: Arith Relation

$$\dfrac{\dfrac{7 > 5 = \text{true} \qquad}{(x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5}}{(x > 5,\ \{x \text{ -> } 7\})\Downarrow \text{true}}$$

$$\dfrac{(x > 5,\ \{x \text{ -> } 7\})\Downarrow \text{true}}{(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi},\ \{x \text{ -> } 7\}) \Downarrow\ ?}$$

Example: If Then Else Rule

$$\dfrac{\dfrac{7 > 5 = \text{true}}{\dfrac{(x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5}{(x > 5, \{x \text{->} 7\})\Downarrow \text{true}}} \qquad \dfrac{\overline{\quad}}{(y{:}= 2 + 3, \{x\text{->} 7\}) \Downarrow ?}}{(\text{if } x > 5 \text{ then } y{:}= 2 + 3 \text{ else } y{:}=3 + 4 \text{ fi}, \{x \text{->} 7\}) \Downarrow ?}$$

Example: Assignment

$$\dfrac{\dfrac{7 > 5 = \text{true} \qquad (x,\{x->7\})\Downarrow 7 \quad (5,\{x->7\})\Downarrow 5}{(x > 5, \{x -> 7\})\Downarrow \text{true}} \qquad \dfrac{\dfrac{}{(2+3, \{x->7\})\Downarrow ?}}{(y:= 2 + 3, \{x-> 7\})\Downarrow ?}}{(\text{if } x > 5 \text{ then } y:= 2 + 3 \text{ else } y:=3 + 4 \text{ fi}, \{x -> 7\}) \Downarrow ?}$$

Example: Arith Op

$$\dfrac{\dfrac{\begin{array}{c}7 > 5 = \text{true}\\ (\text{x},\{\text{x->7}\})\Downarrow 7 \quad (5,\{\text{x->7}\})\Downarrow 5\end{array}}{(\text{x} > 5, \{\text{x -> 7}\})\Downarrow \text{true}} \qquad \dfrac{\dfrac{\begin{array}{c}? + ? = ?\\ (2,\{\text{x->7}\})\Downarrow ? \quad (3,\{\text{x->7}\})\Downarrow ?\end{array}}{(2+3, \{\text{x->7}\})\Downarrow ?}}{\begin{array}{c}(\text{y:= 2 + 3}, \{\text{x-> 7}\}\\ \Downarrow ? \qquad .\end{array}}}{\begin{array}{c}(\text{if x > 5 then y:= 2 + 3 else y:=3 + 4 fi},\\ \{\text{x -> 7}\}) \Downarrow ?\end{array}}$$

Example: Numerals

$$\dfrac{\dfrac{(x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5 \qquad 7 > 5 = \text{true}}{(x > 5, \{x \text{->} 7\})\Downarrow \text{true}} \qquad \dfrac{\dfrac{(2,\{x\text{->}7\})\Downarrow 2 \quad (3,\{x\text{->}7\})\Downarrow 3 \qquad 2 + 3 = 5}{(2+3, \{x\text{->}7\})\Downarrow ?}}{(y{:=}\ 2 + 3, \{x\text{->}\ 7\})\Downarrow ?}}{(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \{x \text{->}\ 7\}) \Downarrow ?}$$

Example: Arith Op

$$2 + 3 = 5$$

$$\frac{(2,\{x\text{->}7\})\Downarrow 2 \quad (3,\{x\text{->}7\})\Downarrow 3}{(2+3,\{x\text{->}7\})\Downarrow 5}$$

$$7 > 5 = \text{true}$$

$$\frac{(x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5}{(x > 5,\ \{x \text{->} 7\})\Downarrow \text{true}} \qquad \frac{(2+3,\ \{x\text{->}7\})\Downarrow 5}{(y{:=}\ 2 + 3,\ \{x\text{->}\ 7\})\Downarrow ?}$$

$$\frac{(x > 5,\ \{x \text{->} 7\})\Downarrow \text{true} \qquad (y{:=}\ 2 + 3,\ \{x\text{->}\ 7\})\Downarrow ?}{(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi},\ \{x \text{->} 7\}) \Downarrow\ ?}$$

Example: Assignment

$$
\dfrac{
\dfrac{\begin{array}{c} 7 > 5 = \text{true} \\ (x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5 \end{array}}{(x > 5,\ \{x\ \text{->}\ 7\})\Downarrow \text{true}}
\qquad
\dfrac{\dfrac{\begin{array}{c} 2 + 3 = 5 \\ (2,\{x\text{->}7\})\Downarrow 2 \quad (3,\{x\text{->}7\})\Downarrow 3 \end{array}}{(2+3,\ \{x\text{->}7\})\Downarrow 5}}{\begin{array}{c}(y{:=}\ 2 + 3,\ \{x\text{->}\ 7\}) \\ \Downarrow \{x\text{->}7,\ y\text{->}5\}\end{array}}
}{
\begin{array}{c}(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \\ \{x\ \text{->}\ 7\})\ \Downarrow ?\end{array}
}
$$

Example: If Then Else Rule

$$\dfrac{\dfrac{\begin{array}{c}7 > 5 = \text{true}\\[4pt] (x,\{x\text{->}7\})\Downarrow 7 \qquad (5,\{x\text{->}7\})\Downarrow 5\end{array}}{(x > 5,\ \{x \text{->}\ 7\})\Downarrow \text{true}} \qquad \dfrac{\dfrac{\begin{array}{c}2 + 3 = 5\\[4pt] (2,\{x\text{->}7\})\Downarrow 2 \qquad (3,\{x\text{->}7\})\Downarrow 3\end{array}}{(2+3,\ \{x\text{->}7\})\Downarrow 5}}{(y{:=}\ 2 + 3,\ \{x\text{->}\ 7\})\ \Downarrow \{x\text{->}7,\ y\text{->}5\}}}{(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi},\ \{x \text{->}\ 7\})\Downarrow \{x\text{->}7,\ y\text{->}5\}}$$

Let in Command

$$\frac{(E,m) \Downarrow v \quad (C, m[I \text{<-} v]) \Downarrow m'}{(\text{let } I = E \text{ in } C,\ m) \Downarrow m''}$$

Where $m''(y) = m'(y)$ for $y \neq I$ and $m''(I) = m(I)$ if $m(I)$ is defined, and $m''(I)$ is undefined otherwise

Example

$$\frac{(5,\{x\text{->}17\}) \Downarrow 5 \qquad \dfrac{\dfrac{(x,\{x\text{->}5\}) \Downarrow 5 \quad (3,\{x\text{->}5\}) \Downarrow 3}{(x+3,\{x\text{->}5\}) \Downarrow 8}}{(x:=x+3,\{x\text{->}5\}) \Downarrow \{x\text{->}8\}}}{(\text{let } x = 5 \text{ in } (x:=x+3),\ \{x \text{ -> } 17\}) \Downarrow \ ?}$$

Example

$$\frac{(5,\{x\text{->}17\}) \Downarrow 5 \qquad \dfrac{\dfrac{(x,\{x\text{->}5\}) \Downarrow 5 \quad (3,\{x\text{->}5\}) \Downarrow 3}{(x+3,\{x\text{->}5\}) \Downarrow 8}}{(x{:=}x+3,\{x\text{->}5\}) \Downarrow \{x\text{->}8\}}}{(\text{let } x = 5 \text{ in } (x{:=}x+3),\ \{x \text{ -> } 17\}) \Downarrow \{x\text{->}17\}}$$

Comment

- Simple Imperative Programming Language introduces variables *implicitly* through assignment
- The let-in command introduces scoped variables *explictly*
- Clash of constructs apparent in awkward semantics

Interpretation Versus Compilation

- A **compiler** from language L1 to language L2 is a program that takes an L1 program and for each piece of code in L1 generates a piece of code in L2 of same meaning
- An **interpreter** of L1 in L2 is an L2 program that executes the meaning of a given L1 program
- Compiler would examine the body of a loop once; an interpreter would examine it every time the loop was executed

Interpreter

- An *Interpreter* represents the operational semantics of a language L1 (source language) in the language of implementation L2 (target language)
- Built incrementally
  - Start with literals
  - Variables
  - Primitive operations
  - Evaluation of expressions
  - Evaluation of commands/declarations

Interpreter

- Takes abstract syntax trees as input
  - In simple cases could be just strings
- One procedure for each syntactic category (nonterminal)
  - eg one for expressions, another for commands
- If Natural semantics used, tells how to compute final value from code
- If Transition semantics used, tells how to compute next “state”
  - To get final value, put in a loop

Natural Semantics Example

- compute_exp (Var(v), m) = look_up v m
- compute_exp (Int(n), _) = Num (n)
- ...
- compute_com(IfExp(b,c1,c2),m) =

  if compute_exp (b,m) = Bool(true)

  then compute_com (c1,m)

  else compute_com (c2,m)

Natural Semantics Example

- compute_com(While(b,c), m) =
  if compute_exp (b,m) = Bool(false)
  then m
  else compute_com
  (While(b,c), compute_com(c,m))

- May fail to terminate - exceed stack limits
- Returns no useful information then

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Interpreter

- Takes abstract syntax trees as input
  - In simple cases could be just strings
- One procedure for each syntactic category (nonterminal)
  - eg one for expressions, another for commands
- If Natural semantics used, tells how to compute final value from code
- If Transition semantics used, tells how to compute next "state"
  - To get final value, put in a loop

Natural Semantics Example

- compute_exp (Var(v), m) = look_up v m
- compute_exp (Int(n), _) = Num (n)
- ...
- compute_com(IfExp(b,c1,c2),m) =

  if compute_exp (b,m) = Bool(true)

  then compute_com (c1,m)

  else compute_com (c2,m)

Natural Semantics Example

- compute_com(While(b,c), m) =

  if compute_exp (b,m) = Bool(false)

  then m

  else compute_com

  (While(b,c), compute_com(c,m))

- May fail to terminate - exceed stack limits
- Returns no useful information then

Transition Semantics

- Form of operational semantics
- Describes how each program construct transforms machine state by *transitions*
- Rules look like

  (C, *m*) --> (C', *m'*)   or   (C,*m*) --> *m'*
- *C, C'* is code remaining to be executed
- m, m' represent the state/store/memory/ environment
  - Partial mapping from identifiers to values
  - Sometimes *m* (or *C*) not needed
- Indicates exactly one step of computation

Expressions and Values

- *C, C'* used for commands; *E, E' for* expressions; *U,V* for values
- Special class of expressions designated as *values*
  - Eg 2, 3 are values, but 2+3 is only an expression
- Memory only holds values
  - Other possibilities exist

Evaluation Semantics

- Transitions successfully stops when E/*C* is a value/memory
- Evaluation fails if no transition possible, but not at value/memory
- Value/memory is the final *meaning* of original expression/command (in the given state)
- Coarse semantics: final value / memory
- More fine grained: whole transition sequence

Simple Imperative Programming Language

- $I \in Identifiers$
- $N \in Numerals$
- B ::= true | false | B & B | B or B | not B | $E < E$ | $E = E$
- $E ::= N \mid I \mid E + E \mid E * E \mid E - E \mid - E$
- C::= skip | $C;C$ | I ::= E
  | if B then C else C fi | while B do C od

Transitions for Expressions

- Numerals are values
- Boolean values = {true, false}
- Identifiers: $(I,m) \dashrightarrow (m(I), m)$

Boolean Operations:

- Operators: (short-circuit)

(false & *B*, *m*) --> (false,*m*)

$$\frac{(B, m) \dashrightarrow (B'', m)}{(B \,\&\, B', m) \dashrightarrow (B'' \,\&\, B', m)}$$

(true & *B*, *m*) --> (*B*,*m*)

(true or *B*, *m*) --> (true,*m*)

$$\frac{(B, m) \dashrightarrow (B'', m)}{(B \text{ or } B', m) \dashrightarrow (B'' \text{ or } B', m)}$$

(false or *B*, *m*) --> (*B*,*m*)

(not true, m) --> (false,*m*)

(not false, m) --> (true,*m*)

$$\frac{(B, m) \dashrightarrow (B', m)}{(\text{not } B, m) \dashrightarrow (\text{not } B', m)}$$

Relations

$$\frac{(E, m) \dashrightarrow (E'', m)}{(E \sim E', m) \dashrightarrow (E'' \sim E', m)}$$

$$\frac{(E, m) \dashrightarrow (E', m)}{(V \sim E, m) \dashrightarrow (V \sim E', m)}$$

$(U \sim V, m)$ --> (true,m) or (false,m) depending on whether $U \sim V$ holds or not

Arithmetic Expressions

$$\frac{(E, m) \dashrightarrow (E'', m)}{(E \; op \; E', m) \dashrightarrow (E'' \; op \; E', m)}$$

$$\frac{(E, m) \dashrightarrow (E', m)}{(V \; op \; E, m) \dashrightarrow (V \; op \; E', m)}$$

$(U \; op \; V, m) \dashrightarrow (N, m)$ where N is the specified value for $U \; op \; V$

Commands - in English

- skip means done evaluating
- When evaluating an assignment, evaluate the expression first
- If the expression being assigned is already a value, update the memory with the new value for the identifier
- When evaluating a sequence, work on the first command in the sequence first
- If the first command evaluates to a new memory (ie completes), evaluate remainder with new memory

Commands

$$(\text{skip}, m) \dashrightarrow m$$

$$\frac{(E,m) \text{ --> } (E',m)}{(I::=E,m) \text{ --> } (I::=E',m)}$$

$$(I::=V,m) \text{ --> } m[I \text{ <-- } V]$$

$$\frac{(C,m) \text{ --> } (C'',m')}{(C;C', m) \text{ --> } (C'';C',m')} \qquad \frac{(C,m) \text{ --> } m'}{(C;C', m) \text{ --> } (C',m')}$$

If Then Else Command - in English

- If the boolean guard in an if_then_else is true, then evaluate the first branch
- If it is false, evaluate the second branch
- If the boolean guard is not a value, then start by evaluating it first.

If Then Else Command

$$(\text{if true then } C \text{ else } C' \text{ fi}, m) \dashrightarrow (C, m)$$

$$(\text{if false then } C \text{ else } C' \text{ fi}, m) \dashrightarrow (C', m)$$

$$\frac{(B, m) \dashrightarrow (B', m)}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \dashrightarrow (\text{if } B' \text{ then } C \text{ else } C' \text{ fi}, m)}$$

While Command

(while *B* do *C* od, *m*)

--> (if *B* then *C*; while *B* do *C* od else skip fi, m)

In English: Expand a While into a test of the boolean guard, with the true case being to do the body and then try the while loop again, and the false case being to stop.

Example Evaluation

- First step:

$$\frac{}{\text{(if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi, } \{x \to 7\})}$$

--> ?

Example Evaluation

- First step:

$$\frac{(x > 5, \{x \text{ -> } 7\}) \text{ --> } ?}{\begin{array}{c}(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi},\\ \{x \text{ -> } 7\})\\ \text{--> } ?\end{array}}$$

Example Evaluation

- First step:

$$\dfrac{\dfrac{(x,\{x \text{ -> } 7\}) \text{ --> } (7, \{x \text{ -> } 7\})}{(x > 5, \{x \text{ -> } 7\}) \text{ --> } ?}}{\begin{array}{c}(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi},\\ \{x \text{ -> } 7\})\\ \text{--> } ?\end{array}}$$

Example Evaluation

- First step:

$$\frac{\dfrac{(x,\{x \text{ -> } 7\}) \text{ --> } (7, \{x \text{ -> } 7\})}{(x > 5, \{x \text{ -> } 7\}) \text{ --> } (7 > 5, \{x \text{ -> } 7\})}}{(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \{x \text{ -> } 7\}) \text{ --> } ?}$$

Example Evaluation

- First step:

$$\cfrac{\cfrac{(x,\{x \text{ -> } 7\}) \text{ --> } (7, \{x \text{ -> } 7\})}{(x > 5, \{x \text{ -> } 7\}) \text{ --> } (7 > 5, \{x \text{ -> } 7\})}}{(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \{x \text{ -> } 7\})}$$

$$\text{--> } (\text{if } 7 > 5 \text{ then } y{:=}2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \{x \text{ -> } 7\})$$

Example Evaluation

- Second Step:

(7 > 5, {x -> 7}) --> (true, {x -> 7})

(if 7 > 5 then y:=2 + 3 else y:=3 + 4 fi,
{x -> 7})
--> (if true then y:=2 + 3 else y:=3 + 4 fi,
{x -> 7})

- Third Step:

(if true then y:=2 + 3 else y:=3 + 4 fi, {x -> 7})
-->(y:=2+3, {x->7})

Example Evaluation

- Fourth Step:

$$\frac{(2+3,\ \{x\text{-}>\ 7\})\ \text{-->}\ (5,\ \{x\ \text{-}>\ 7\})}{(y{:=}2+3,\ \{x\text{-}>7\})\ \text{-->}\ (y{:=}5,\ \{x\text{-}>7\})}$$

- Fifth Step:

$$(y{:=}5,\ \{x\text{-}>7\})\ \text{-->}\ \{y\ \text{-}>\ 5,\ x\ \text{-}>\ 7\}$$

Example Evaluation

- Bottom Line:

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7})

--> (if 7 > 5 then y:=2 + 3 else y:=3 + 4 fi,
{x -> 7})

-->(if true then y:=2 + 3 else y:=3 + 4 fi,
{x -> 7})

-->(y:=2+3, {x->7})

--> (y:=5, {x->7}) --> {y -> 5, x -> 7}

Transition Semantics Evaluation

- A sequence of steps with trees of justification for each step

$(C_1,m_1) \dashrightarrow (C_2,m_2) \dashrightarrow (C_3,m_3) \dashrightarrow \ldots \dashrightarrow m$

- Let -->* be the transitive closure of -->
- Ie, the smallest transitive relation containing -->

Adding Local Declarations

- Add to expressions:
- E ::= … | let I = E in E' | fun I -> E | E E'
- fun I -> E is a value
- Could handle local binding using state, but have assumption that evaluating expressions doesn't alter the environment
- We will use substitution here instead
- **Notation**: $E\,[E' / I\,]$ means replace all free occurrence of I by E' in E

Call-by-value (Eager Evaluation)

$$(\text{let } I = V \text{ in } E, m) \dashrightarrow (E[V/I], m)$$

$$\frac{(E, m) \dashrightarrow (E'', m)}{(\text{let } I = E \text{ in } E', m) \dashrightarrow (\text{let } I = E'' \text{ in } E')}$$

$$((\text{fun } I \to E)\ V, m) \dashrightarrow (E[V/I], m)$$

$$\frac{(E', m) \dashrightarrow (E'', m)}{((\text{fun } I \to E)\ E', m) \dashrightarrow ((\text{fun } I \to E)\ \text{E}'', m)}$$

Call-by-name (Lazy Evaluation)

- (let $I = E$ in E', m) --> (E' [E / I],m)

- ((fun I -> E') E, m) --> (E' [E / I],m)

- Question: Does it make a difference?
- It can depending on the language

Church-Rosser Property

- Church-Rosser Property: If E-->* E_1 and E-->* E_2, if there exists a value V such that E_1 -->* V, then E_2 -->* V
- Also called **confluence** or **diamond property**
- Example: E= 2 + 3 + 4

E_1 = 5 + 4 E_2= 2 + 7

V =9

Does It always Hold?

- No. Languages with side-effects tend not be Church-Rosser with the combination of call-by-name and call-by-value
- Alonzo Church and Barkley Rosser proved in 1936 the λ-calculus does have it
- Benefit of Church-Rosser: can check equality of terms by evaluating them (Given evaluation strategy might not terminate, though)

Transition Semantics for λ-Calculus

- Application (version 1)

$$(\lambda\, x\, .\, E)\ E' \dashrightarrow E[E'/x]$$

- Application (version 2)

$$(\lambda\, x\, .\, E)\ V \dashrightarrow E[V/x]$$

$$\frac{E' \dashrightarrow E''}{(\lambda\, x\, .\, E)\ E' \dashrightarrow (\lambda\, x\, .\, E)\ E''}$$

Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

If Then Else Command

$$(\text{if true then } C \text{ else } C' \text{ fi}, m) \dashrightarrow (C, m)$$

$$(\text{if false then } C \text{ else } C' \text{ fi}, m) \dashrightarrow (C', m)$$

$$\frac{(B,m) \dashrightarrow (B',m)}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \dashrightarrow (\text{if } B' \text{ then } C \text{ else } C' \text{ fi}, m)}$$

While Command

(while B do C od, m)
--> (if B then C; while B do C od else skip fi, m)

In English: Expand a While into a test of the boolean guard, with the true case being to do the body and then try the while loop again, and the false case being to stop.

Example Evaluation

- First step:

$$\frac{}{\text{(if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi, } \{x \to 7\})}$$

--> ?

Example Evaluation

- First step:

$$\frac{(x > 5, \{x \to 7\}) \dashrightarrow\ ?}{(\text{if } x > 5 \text{ then } y:= 2 + 3 \text{ else } y:=3 + 4 \text{ fi}, \{x \to 7\}) \dashrightarrow\ ?}$$

Example Evaluation

- First step:

$$\dfrac{\dfrac{(x,\{x \text{ -> } 7\}) \text{ --> } (7, \{x \text{ -> } 7\})}{(x > 5, \{x \text{ -> } 7\}) \text{ --> } ?}}{\begin{array}{c}(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \\ \{x \text{ -> } 7\}) \\ \text{ --> } ?\end{array}}$$

Example Evaluation

- First step:

$$\dfrac{\dfrac{(x,\{x \text{ -> } 7\}) \text{ --> } (7, \{x \text{ -> } 7\})}{(x > 5, \{x \text{ -> } 7\}) \text{ --> } (7 > 5, \{x \text{ -> } 7\})}}{(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \{x \text{ -> } 7\}) \text{ --> } ?}$$

Example Evaluation

- First step:

$$\dfrac{\dfrac{\text{(x,\{x -> 7\}) --> (7, \{x -> 7\})}}{\text{(x > 5, \{x -> 7\}) --> (7 > 5, \{x -> 7\})}}}{\text{(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,} \\ \text{\{x -> 7\})}}$$

$$\text{--> (if 7 > 5 then y:=2 + 3 else y:=3 + 4 fi,} \\ \text{\{x -> 7\})}$$

Example Evaluation

- Second Step:

(7 > 5, {x -> 7}) --> (true, {x -> 7})

(if 7 > 5 then y:=2 + 3 else y:=3 + 4 fi,
{x -> 7})
--> (if true then y:=2 + 3 else y:=3 + 4 fi,
{x -> 7})

- Third Step:

(if true then y:=2 + 3 else y:=3 + 4 fi, {x -> 7})
-->(y:=2+3, {x->7})

Example Evaluation

- Fourth Step:

$$\frac{(2+3, \{x\text{->}\ 7\})\ \text{-->}\ (5, \{x\ \text{->}\ 7\})}{(y{:=}2+3, \{x\text{->}7\})\ \text{-->}\ (y{:=}5, \{x\text{->}7\})}$$

- Fifth Step:

(y:=5, {x->7}) --> {y -> 5, x -> 7}

Example Evaluation

- Bottom Line:

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi, {x -> 7})

--> (if 7 > 5 then y:=2 + 3 else y:=3 + 4 fi, {x -> 7})

-->(if true then y:=2 + 3 else y:=3 + 4 fi, {x -> 7})

-->(y:=2+3, {x->7})

--> (y:=5, {x->7}) --> {y -> 5, x -> 7}

Transition Semantics Evaluation

- A sequence of steps with trees of justification for each step

$(C_1,m_1) \text{-->} (C_2,m_2) \text{-->} (C_3,m_3) \text{-->} \ldots \text{-->} m$

- Let -->* be the transitive closure of -->
- Ie, the smallest transitive relation containing -->

Adding Local Declarations

- Add to expressions:
- E ::= … | let I = E in E' | fun I -> E | E E'
- fun I -> E is a value
- Could handle local binding using state, but have assumption that evaluating expressions doesn't alter the environment
- We will use substitution here instead
- **Notation**: $E\,[E'\,/\,I\,]$ means replace all free occurrence of I by E' in E

Call-by-value (Eager Evaluation)

$$(\text{let } I = V \text{ in } E, m) \dashrightarrow (E[V/I], m)$$

$$\frac{(E, m) \dashrightarrow (E'', m)}{(\text{let } I = E \text{ in } E', m) \dashrightarrow (\text{let } I = E'' \text{ in } E')}$$

$$((\text{fun } I \to E)\ V, m) \dashrightarrow (E[V/I], m)$$

$$\frac{(E', m) \dashrightarrow (E'', m)}{((\text{fun } I \to E)\ E', m) \dashrightarrow ((\text{fun } I \to E)\ \mathrm{E}'', m)}$$

Call-by-name (Lazy Evaluation)

- (let I = E in E', m) --> (E' [E / I],m)
- ((fun I -> E') E, m) --> (E' [E / I],m)
- Question: Does it make a difference?
- It can depending on the language

Church-Rosser Property

- Church-Rosser Property: If E-->* E_1 and E-->* E_2, if there exists a value V such that E_1 -->* V, then E_2 -->* V
- Also called **confluence** or **diamond property**
- Example: E= 2 + 3 + 4

E_1 = 5 + 4 E_2= 2 + 7

V =9

Does It always Hold?

- No. Languages with side-effects tend not be Church-Rosser with the combination of call-by-name and call-by-value
- Alonzo Church and Barkley Rosser proved in 1936 the λ-calculus does have it
- Benefit of Church-Rosser: can check equality of terms by evaluating them (Given evaluation strategy might not terminate, though)

Lambda Calculus - Motivation

- Aim is to capture the essence of functions, function applications, and evaluation
- $\lambda-$calculus is a theory of computation
- "The Lambda Calculus: Its Syntax and Semantics". H. P. Barendregt. North Holland, 1984

Lambda Calculus - Motivation

- All *sequential programs* may be viewed as functions from input (initial state and input values) to output (resulting state and output values).
- λ-calculus is a mathematical formalism of functions and functional computations
- Two flavors: typed and untyped

Untyped λ-Calculus

- Only three kinds of expressions:
  - Variables: x, y, z, w, …
  - Abstraction: λ x. e

    (Function creation, think fun x -> e)
  - Application: $e_1\ e_2$

Untyped λ-Calculus Grammar

- Formal BNF Grammar:
  - <expression> ::= <variable>
    | <abstraction>
    | <application>
    | (<expression>)
  - <abstraction>
    ::= λ<variable>.<expression>
  - <application>
    ::= <expression> <expression>

Untyped λ-Calculus Terminology

- Occurrence: a location of a subterm in a term
- Variable binding: λ x. e is a binding of x in e
- Bound occurrence: all occurrences of x in λ x. e
- Free occurrence: one that is not bound
- Scope of binding: in λ x. e, all occurrences in e not in a subterm of the form λ x. e′ (same x)
- Free variables: all variables having free occurrences in a term

Example

- Label occurrences and scope:

$$(\lambda\ x.\ \lambda\ y.\ y\ (\lambda\ x.\ x\ y)\ x)\ x$$

Example

- Label occurrences and scope:

$$(\lambda\, x.\ y\ \lambda\, y.\ y\ (\lambda\, x.\ x\ y)\ x)\ x$$

Untyped λ-Calculus

- How do you compute with the λ-calculus?
- Roughly speaking, by substitution:

- $(\lambda\ x.\ e_1)\ e_2 \Rightarrow^* e_1\ [e_2\ /\ x]$

- * Modulo all kinds of subtleties to avoid free variable capture

Transition Semantics for λ-Calculus

$$\frac{E \dashrightarrow E''}{E\,E' \dashrightarrow E''\,E'}$$

- Application (version 1 - Lazy Evaluation)

$$(\lambda\, x\, .\, E)\; E' \dashrightarrow E[E'/x]$$

- Application (version 2 - Eager Evaluation)

$$\frac{E' \dashrightarrow E''}{(\lambda\, x\, .\, E)\; E' \dashrightarrow (\lambda\, x\, .\, E)\; E''}$$

$$\frac{}{(\lambda\, x\, .\, E)\; V \dashrightarrow E[V/x]}$$

V - variable or abstraction (value)

How Powerful is the Untyped λ-Calculus?

- The untyped λ-calculus is Turing Complete
  - Can express any sequential computation
- Problems:
  - How to express basic data: booleans, integers, etc?
  - How to express recursion?
  - Constants, if_then_else, etc, are conveniences; can be added as syntactic sugar

Typed vs Untyped λ-Calculus

- The *pure* λ-calculus has no notion of type: (f f) is a legal expression
- Types restrict which applications are valid
- Types are not syntactic sugar! They disallow some terms
- Simply typed λ-calculus is less powerful than the untyped λ-Calculus: NOT Turing Complete (no recursion)

Uses of λ-Calculus

- Typed and untyped λ-calculus used for theoretical study of sequential programming languages
- Sequential programming languages are essentially the λ-calculus, extended with predefined constructs, constants, types, and syntactic sugar
- Ocaml is close to the λ-Calculus:

fun x -> exp --> λ x. exp

let x = e_1 in e_2 --> (λ x. e_2)e_1

α Conversion

- α-conversion:

  λ x. exp --α--> λ y. (exp [y/x])
- Provided that
  1. y is not free in exp
  2. No free occurrence of x in exp becomes bound in exp when replaced by y

α Conversion Non-Examples

1. Error: y is not free in termsecond

$$\lambda x.\ x\ y\ \cancel{\text{--}\alpha\text{-->}}\ \lambda y.\ y\ y$$

2. Error: free occurrence of x becomes bound in wrong way when replaced by y

$$\lambda x.\ \underbrace{\lambda y.\ x\ y}_{\text{exp}}\ \cancel{\text{--}\alpha\text{-->}}\ \lambda y.\ \underbrace{\lambda y.\ y\ y}_{\text{exp[y/x]}}$$

But $\lambda x.\ (\lambda y.\ y)\ x \text{ --}\alpha\text{--> } \lambda y.\ (\lambda y.\ y)\ y$

And $\lambda y.\ (\lambda y.\ y)\ y \text{ --}\alpha\text{--> } \lambda x.\ (\lambda y.\ y)\ x$

Congruence

- Let ~ be a relation on lambda terms. ~ is a congruence if
- it is an equivalence relation
- If $e_1 \sim e_2$ then
  - $(e\ e_1) \sim (e\ e_2)$ and $(e_1 e) \sim (e_2\ e)$
  - $\lambda\ x.\ e_1 \sim \lambda\ x.\ e_2$

α Equivalence

- α equivalence is the smallest congruence containing α conversion
- One usually treats α-equivalent terms as equal - i.e. use α equivalence classes of terms

Example

Show: λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y

- λ x. (λ y. y x) x --α--> λ z. (λ y. y z) z so
  λ x. (λ y. y x) x ~α~ λ z. (λ y. y z) z
- (λ y. y z) --α--> (λ x. x z) so
  (λ y. y z) ~α~ (λ x. x z) so
  λ z. (λ y. y z) z ~α~ λ z. (λ x. x z) z
- λ z. (λ x. x z) z --α--> λ y. (λ x. x y) y so
  λ z. (λ x. x z) z ~α~ λ y. (λ x. x y) y
- λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y

Substitution

- Defined on α-equivalence classes of terms
- P [N / x] means replace every free occurrence of x in P by N
  - P called *redex*; N called *residue*
- Provided that no variable free in P becomes bound in P [N / x]
  - Rename bound variables in P to avoid capturing free variables of N

Substitution

- $x\ [N / x] = N$
- $y\ [N / x] = y$ if $y \neq x$
- $(e_1\ e_2)\ [N / x] = ((e_1\ [N / x])\ (e_2\ [N / x]))$
- $(\lambda\ x.\ e)\ [N / x] = (\lambda\ x.\ e)$
- $(\lambda\ y.\ e)\ [N / x] = \lambda\ y.\ (e\ [N / x])$ provided $y \neq x$ and y not free in N
  - Rename y in redex if necessary

Example

$$(\lambda\ y.\ y\ z)\ [(\lambda\ x.\ x\ y)\ /\ z] = ?$$

- Problems?
  - z in redex in scope of y binding
  - y free in the residue
- $(\lambda\ y.\ y\ z)\ [(\lambda\ x.\ x\ y)\ /\ z]$ --α-->
  $(\lambda\ w.w\ z)\ [(\lambda\ x.\ x\ y)\ /\ z] =$
  $\lambda\ w.\ w\ (\lambda\ x.\ x\ y)$

Example

- Only replace free occurrences
- $(\lambda\ y.\ y\ z\ (\lambda\ z.\ z))\ [(\lambda\ x.\ x)\ /\ z] =$

$$\lambda\ y.\ y\ (\lambda\ x.\ x)\ (\lambda\ z.\ z)$$

Not

$$\lambda\ y.\ y\ (\lambda\ x.\ x)\ (\lambda\ z.\ (\lambda\ x.\ x))$$

β reduction

- β Rule: $(\lambda\ x.\ P)\ N$ --β--> $P\ [N\ /x]$
- Essence of computation in the lambda calculus
- Usually defined on α-equivalence classes of terms

Example

- $(\lambda\, z.\ (\lambda\, x.\ x\ y)\ z)\ (\lambda\, y.\ y\ z)$

  $\text{--}\beta\text{-->}\ (\lambda\, x.\ x\ y)\ (\lambda\, y.\ y\ z)$

  $\text{--}\beta\text{-->}\ (\lambda\, y.\ y\ z)\ y\ \text{--}\beta\text{-->}\ y\ z$

- $(\lambda\, x.\ x\ x)\ (\lambda\, x.\ x\ x)$

  $\text{--}\beta\text{-->}\ (\lambda\, x.\ x\ x)\ (\lambda\, x.\ x\ x)$

  $\text{--}\beta\text{-->}\ (\lambda\, x.\ x\ x)\ (\lambda\, x.\ x\ x)\ \text{--}\beta\text{-->}\ \ldots.$

α β Equivalence

- α β equivalence is the smallest congruence containing α equivalence and β reduction
- A term is in *normal form* if no subterm is α equivalent to a term that can be β reduced
- Hard fact (Church-Rosser): if e_1 and e_2 are $\alpha\beta$-equivalent and both are normal forms, then they are α equivalent

Order of Evaluation

- Not all terms reduce to normal forms
- Not all reduction strategies will produce a normal form if one exists

Lazy evaluation:

- Always reduce the left-most application in a top-most series of applications (i.e. Do not perform reduction inside an abstraction)

- Stop when term is not an application, or left-most application is not an application of an abstraction to a term

Example 1

- $(\lambda\ z.\ (\lambda\ x.\ x))\ ((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y))$
- Lazy evaluation:
- Reduce the left-most application:
- $(\lambda\ z.\ (\lambda\ x.\ x))\ ((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y))$
  --β--> $(\lambda\ x.\ x)$

Eager evaluation

- (Eagerly) reduce left of top application to an abstraction
- Then (eagerly) reduce argument
- Then β-reduce the application

Example 1

- $(\lambda\ z.\ (\lambda\ x.\ x))((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y))$
- Eager evaluation:
- Reduce the rator of the top-most application to an abstraction: Done.
- Reduce the argument:
- $(\lambda\ z.\ (\lambda\ x.\ x))((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y))$

--β--> $(\lambda\ z.\ (\lambda\ x.\ x))((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y))$

--β--> $(\lambda\ z.\ (\lambda\ x.\ x))((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y))$…

Example 2

- $(\lambda\ x.\ x\ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$
- Lazy evaluation:

$(\lambda\ x.\ x\ \ x\ )((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$ --β-->

Example 2

- $(\lambda\ x.\ x\ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$
- Lazy evaluation:

$(\lambda\ x.\ \boxed{x}\ \boxed{x}\ )\underline{((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))}$ --β-->

Example 2

- $(\lambda\ x.\ x\ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$
- Lazy evaluation:

$(\lambda\ x.\ \boxed{x}\ \boxed{x}\ )(\underline{(\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))}$ --β-->

$\boxed{((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))}\ \boxed{((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))}$

Example 2

- $(\lambda$ x. x x$)((\lambda$ y. y y$)$ $(\lambda$ z. z$))$
- Lazy evaluation:

$(\lambda$ x. x x $)((\lambda$ y. y y$)$ $(\lambda$ z. z$))$ --β-->

$\boxed{((\lambda \text{ y. y y }) (\lambda \text{ z. z}))}$ $((\lambda$ y. y y $)$ $(\lambda$ z. z$)$

Example 2

- $(\lambda\, x.\ x\ x)((\lambda\, y.\ y\ y)\ (\lambda\, z.\ z))$
- Lazy evaluation:

$(\lambda\, x.\ x\ \ x\ )((\lambda\, y.\ y\ y)\ (\lambda\, z.\ z))$ --β-->

$((\lambda\, y.\ \boxed{y}\ \ \boxed{y}\ )\ \underline{(\lambda\, z.\ z)})\ ((\lambda\, y.\ y\ \ y\ )\ (\lambda\, z.\ z))$

Example 2

- $(\lambda\, x.\ x\ x)((\lambda\, y.\ y\ y)\ (\lambda\, z.\ z))$
- Lazy evaluation:

$(\lambda\, x.\ x\ \ x)((\lambda\, y.\ y\ y)\ (\lambda\, z.\ z))$ --β-->

$((\lambda\, y.\ \boxed{y}\ \ \boxed{y})\ \underline{(\lambda\, z.\ z)})\ ((\lambda\, y.\ y\ \ y)\ (\lambda\, z.\ z))$

--β--> $(\boxed{(\lambda\, z.\ z)}\ \boxed{(\lambda\, z.\ z)})((\lambda\, y.\ y\ \ y)\ (\lambda\, z.\ z))$

Example 2

- $(\lambda\, x.\ x\ x)((\lambda\, y.\ y\ y)\ (\lambda\, z.\ z))$
- Lazy evaluation:

$(\lambda\, x.\ x\ \ x)((\lambda\, y.\ y\ y)\ (\lambda\, z.\ z))$ --β-->

$((\lambda\, y.\ y\ \ y)\ (\lambda\, z.\ z))\ ((\lambda\, y.\ y\ \ y)\ (\lambda\, z.\ z))$

--β--> $\boxed{((\lambda\, z.\ z)\ (\lambda\, z.\ z))}((\lambda\, y.\ y\ \ y)\ (\lambda\, z.\ z))$

Example 2

- $(\lambda\ x.\ x\ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$
- Lazy evaluation:

$(\lambda\ x.\ x\ \ x\ )((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$ --β-->

$((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))\ ((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

--β--> $((\lambda\ z.\ \boxed{z}\ )\ \underline{(\lambda\ z.\ z)})((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

Example 2

- $(\lambda\, x.\ x\ x)((\lambda\, y.\ y\ y)\ (\lambda\, z.\ z))$
- Lazy evaluation:

$(\lambda\, x.\ x\ \ x\ )((\lambda\, y.\ y\ y)\ (\lambda\, z.\ z))$ --β-->

$((\lambda\, y.\ y\ \ y\ )\ (\lambda\, z.\ z))\ ((\lambda\, y.\ y\ \ y\ )\ (\lambda\, z.\ z))$

--β--> $((\lambda\, z.\ \boxed{z}\ )\ \underline{(\lambda\, z.\ z)})((\lambda\, y.\ y\ \ y\ )\ (\lambda\, z.\ z))$

--β--> $\boxed{(\lambda\, z.\ z\ )}\ ((\lambda\, y.\ y\ \ y\ )\ (\lambda\, z.\ z))$

Example 2

- $(\lambda\ x.\ x\ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$
- Lazy evaluation:

$(\lambda\ x.\ x\ \ x\ )((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$ --β-->

$((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))\ ((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

--β--> $((\lambda\ z.\ z\ )\ (\lambda\ z.\ z))((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

--β--> $(\lambda\ z.\ \boxed{z}\ )\ \underline{((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))}$ --β-->

$(\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z)$

Example 2

- $(\lambda\, x.\ x\ x)((\lambda\, y.\ y\ y)\ (\lambda\, z.\ z))$
- Lazy evaluation:

$(\lambda\, x.\ x\ \ x)((\lambda\, y.\ y\ y)\ (\lambda\, z.\ z))$ --β-->

$(\boxed{(\lambda\, y.\ y\ \ y)\ (\lambda\, z.\ z)})\ ((\lambda\, y.\ y\ \ y)\ (\lambda\, z.\ z))$

--β--> $\boxed{((\lambda\, z.\ z)\ (\lambda\, z.\ z))}((\lambda\, y.\ y\ \ y)\ (\lambda\, z.\ z))$

--β--> $\boxed{(\lambda\, z.\ z)}\ ((\lambda\, y.\ y\ \ y)\ (\lambda\, z.\ z))$ --β-->

$\boxed{(\lambda\, y.\ y\ \ y)\ (\lambda\, z.\ z)}$

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. y y) (λ z. z))) ((λ y. y y) (λ z. z))

--β--> ((λ z. z) (λ z. z))((λ y. y y) (λ z. z))

--β--> (λ z. z) ((λ y. y y) (λ z. z)) --β-->

(λ y. y y) (λ z. z) ~β~ λ z. z

Example 2

- $(\lambda\ x.\ x\ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$
- Eager evaluation:

$(\lambda\ x.\ x\ \ x)\ \boxed{((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))}$ --β-->

$(\lambda\ x.\ x\ \ x)\ \boxed{((\ \lambda\ z.\ z\ )\ (\lambda\ z.\ z))}$ --β-->

$(\lambda\ x.\ x\ \ x)\ \boxed{(\lambda\ z.\ z)}$ --β-->

$(\lambda\ z.\ z)\ (\lambda\ z.\ z)$ --β--> $\lambda\ z.\ z$

η (Eta) Reduction

- η Rule: λ x. f x --η--> f if x not free in f
  - Can be useful in each direction
  - Not valid in Ocaml
    - recall lambda-lifting and side effects
  - Not equivalent to (λ x. f) x --> f (inst of β)

- Example: λ x. (λ y. y) x --η--> λ y. y

CS241 File Systems

Files & Directories
… a practical primer for LMP1

Lawrence Angrave

Robbins Ch4.1-4.3, 4.6, 5.2
Stallings Ch12

?

API... open? stat?

Why so many options in open?

How do I make my code robust?

What concepts underpin the POISX filesystem API?

What exactly is a file, directory...? I-node?

Unix concept of a FILE

Byte-orientated & sequential

So what?

Unicode file

Typed Fields and Records

Key indices

Unified. Get the contents of a file as bytes

```
#include <unistd.h>
ssize_t read(int dec, void *buf, size_t nbyte);
```

read

Don't need to read entire file

... Grab bytes into your buffer

... Next call to read will read next unread byte

Can even read the bytes of a 'directory'

read() boundary cases
to trip you up

May not read anything

- signal interruption
- end of file
- asynchronous non-blocking mode
- hardware issue

Need to check # bytes actually read

It usually works as you might assume

Upon successful completion, read(), readv(), and pread() return the number of bytes actually read and placed in the buffer.

The system guarantees to read the number of bytes requested if the descriptor references a normal file that has that many bytes left before the end-of-file, but in no other case.

read() boundary cases
Practical Guide

Return 0: End of File

Return -1: Check errno and act accordingly

e.g. EINTR; restart

Return +N: # bytes placed into buffer

Sequential bytes != C string

Doesn't know or care about strings and NUL bytes

E*nsure NUL termination before using printf debug statements contents!*

A good alternative is to use write…

```
write(2,buf,nbytes)
```

What are we reading from?

Get and maintain a reference to a file

Integer file descriptor

POSIX open()

Let's dig deeper...

... Errors and options

... Directory organization & implementation

... How does O/S Manage file descriptors?

open – read – close

```
fd=open("/tmp/1.txt", options);

read(fd,buffer,sizeof(buffer));

close(fd);
```

open –... – read – ... – close

fd=open("/tmp/1.txt", *options*);

if(fd <1) *error (e.g. File doesn't exist)*

r=read(fd,buffer,sizeof(buffer))

handle special cases (eof,restart, media errors)

close(fd);

free up resources

What can go wrong with open?

EACCES The requested access to the file is not allowed, or search permission is denied for one of the directories in the path prefix of pathname, or the file did not exist yet and write access to the parent directory is not allowed. (See also path_resolution(2).)

Just open the file. Please.

EEXIST pathname already exists and O_CREAT and O_EXCL were used.

EFAULT pathname points outside your accessible address space.

EISDIR pathname refers to a directory and the access requested involved writing (that is, O_WRONLY or O_RDWR is set).

ELOOP Too many symbolic links were encountered in resolving pathname, or O_NOFOLLOW was specified but pathname was a symbolic link.

EMFILE The process already has the maximum number of files open.

ENAMETOOLONG pathname was too long.

ENFILE The system limit on the total number of open files has been reached.

ENODEV pathname refers to a device special file and no corresponding device exists. (This is a Linux kernel bug; in this situation ENXIO must be returned.)

ENOENT O_CREAT is not set and the named file does not exist. Or, a directory component in pathname does not exist or is a dangling symbolic link.

ENOMEM Insufficient kernel memory was available.

ENOSPC pathname was to be created but the device containing pathname has no room for the new file.

ENOTDIR A component used as a directory in pathname is not, in fact, a directory, or O_DIRECTORY was specified and pathname was not a directory.

ENXIO O_NONBLOCK | O_WRONLY is set, the named file is a FIFO and no process has the file open for reading. Or, the file is a device special file and no corresponding device exists.

EOVERFLOW pathname refers to a regular file, too large to be opened; see O_LARGEFILE above.

EPERM The O_NOATIME flag was specified, but the effective user ID of the caller did not match the owner of the file and the caller was not privileged (CAP_FOWNER).

EROFS pathname refers to a file on a read-only filesystem and write access was requested.

ETXTBSY pathname refers to an executable image which is currently being executed and write access was requested.

EWOULDBLOCK The O_NONBLOCK flag was specified, and an incompatible lease was held on the file (see fcntl(2)).

EACCES

EEXIST

EFAULT

EISDIR

ELOOP

EMFILE

ENAMETOOLONG

ENFILE

ENODEV

ENOENT

ENOMEM

ENOSPC

ENOTDIR

ENXIO

EOVERFLOW

EPERM

EROFS

ETXTBSY

EWOULDBLOCK

EACCES
EEXIST
EFAULT
EISDIR
ELOOP
EMFILE
ENAMETOOLONG
ENFILE
ENODEV
ENOENT

ENOENT :

O_CREAT is not set and the named file does not exist. Or, a directory component in pathname does not exist or is a dangling symbolic link.

Options and modes

R vs. R/W vs. Write only

Truncate an existing file before writing

Read-Write-Execute Permissions

open("/tmp/1.txt", ...)

Pathnames -> travserse directories

Imposes a hierarchy on files

Files can be referenced from more than one directory

Organization of files useful for security

stat

Meta-information about a file

modification and access time

Kind of file (e.g. Directory | regular file?)

Support for symbolic links

Three flavors

```
int stat(const char *path, struct stat *buf);
int fstat(int filedes, struct stat *buf);
```

Info about link (more on this later)

```
int lstat(const char *path, struct stat *buf);
```

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;  /* number of blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */};
```

Stat macros (st_mode)

S_ISREG(m) is it a regular file?

S_ISDIR(m) directory?

S_ISCHR(m) character device?

S_ISBLK(m) block device?

S_ISFIFO(m) FIFO (named pipe)?

S_ISLNK(m) symbolic link?*

S_ISSOCK(m) socket?*

*(Not in POSIX.1-1996.)

open –... – stat – ... – close

fd=open("/tmp/1.txt", *options*);

if(fd <1) *error (e.g. File doesn't exist)*

r=fstat(fd,&buffer)

handle special cases (eof,restart, media errors)S_ISDIR(buffer.st_mode)

close(fd);

free up resources

Directories Robbins Ch.5.2

```
struct dirent *entry;
 // add error handing!
DIR *dirp = opendir(".");

while((entry = readdir(dirp)) != NULL)
  printf("%s\n",direntp->name);

closedir(dirp);
```

Directories

readdir will return "." and ".."

readdir returns a pointer to a static structure

i.e. not threadsafe, not recursive-safe

Can't call opendir recursively!?!?

Don't forget to closedir!

System Perspective

File Descriptors

File Descriptors

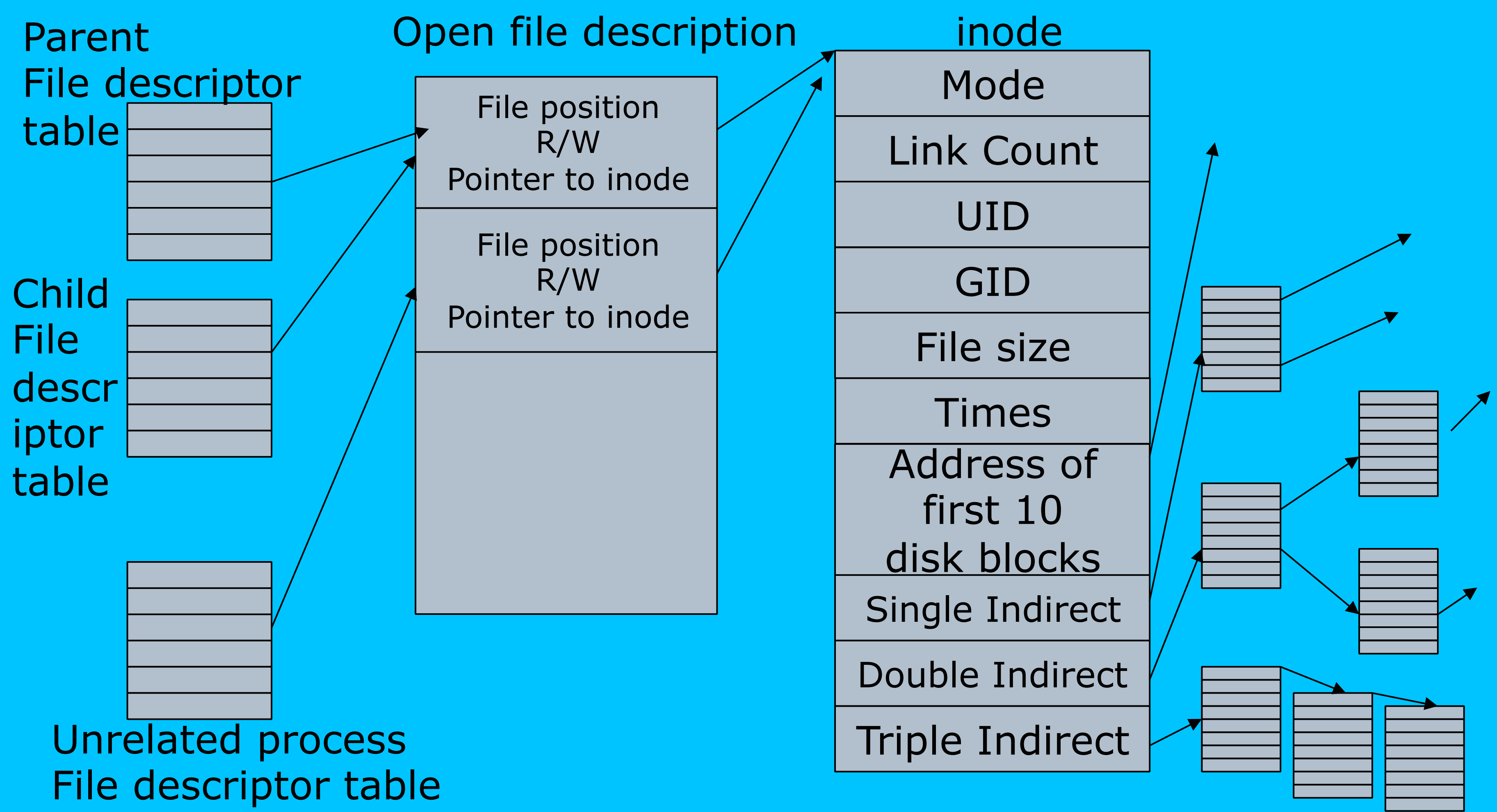
Specific to each process

Stored

fork() child inherits a copy of the parent's file descriptors

File descriptors to I-Nodes

UNIX file structure implementation



LMP1

Be lazy … Read & use restart library
Read man pages for error codes
Useful code examples in Robbins Ch4,5

Understand Bonnie

Restart library makes programs simpler!

```
#include <unistd.h>
#include "restart.h"
#define BLKSIZE 1024
int copyfile(int fromfd, int tofd) {
   char buf[BLKSIZE];
   int bytesread, byteswritten;
   int totalbytes = 0;
   for (  ;  ;  ) {
      if ((bytesread = r_read(fromfd, buf, BLKSIZE)) <= 0)
         break;
      if ((byteswritten = r_write(tofd, buf, bytesread)) == -1)
         break;
      totalbytes += byteswritten;
   }
   return totalbytes;
}
```

Use the restart library for read/write – e.g. r_write

```
ssize_t r_write(int fd, void *buf, size_t size) {
   char *bufp;
   size_t bytestowrite;
   ssize_t byteswritten;
   size_t totalbytes;
   for (bufp = buf, bytestowrite = size, totalbytes = 0;
        bytestowrite > 0;
        bufp += byteswritten, bytestowrite -= byteswritten) {
      byteswritten = write(fd, bufp, bytestowrite);
      if ((byteswritten) == -1 && (errno != EINTR))
         return -1;
      if (byteswritten == -1)
         byteswritten = 0;
      totalbytes += byteswritten;
   }
   return totalbytes;
}
```

LMP1

Staged

Test-driven development

Future LMPs build on LMP1

Bonnie

read,write,directory

benchmarks

Light reading for spring break?

The Diamond Age (Young Lady's Illustrated Primer) ,Neal Stephenson

Cryptonomicon,Neal Stephenson

Victorian Internet, Tom Standage

The man who mistook his wife for a hat (and other clinical tales) ,Oliver Sacks

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

α Conversion

- α-conversion:

  λ x. exp --α--> λ y. (exp [y/x])

- Provided that

  1. y is not free in exp
  2. No free occurrence of x in exp becomes bound in exp when replaced by y

α Conversion Non-Examples

1. Error: y is not free in termsecond

$$\lambda\, x.\ x\ y \ \cancel{\text{--}\alpha\text{-->}}\ \lambda\, y.\ y\ y$$

2. Error: free occurrence of x becomes bound in wrong way when replaced by y

$$\lambda\, x.\ \underbrace{\lambda\, y.\ x\ y}_{\text{exp}} \ \cancel{\text{--}\alpha\text{-->}}\ \lambda\, y.\ \underbrace{\lambda\, y.\ y\ y}_{\text{exp[y/x]}}$$

But $\lambda\, x.\ (\lambda\, y.\ y)\ x$ --α--> $\lambda\, y.\ (\lambda\, y.\ y)\ y$

And $\lambda\, y.\ (\lambda\, y.\ y)\ y$ --α--> $\lambda\, x.\ (\lambda\, y.\ y)\ x$

Congruence

- Let ~ be a relation on lambda terms. ~ is a congruence if
- it is an equivalence relation
- If $e_1 \sim e_2$ then
  - $(e\ e_1) \sim (e\ e_2)$ and $(e_1 e) \sim (e_2\ e)$
  - $\lambda\ x.\ e_1 \sim \lambda\ x.\ e_2$

α Equivalence

- α equivalence is the smallest congruence containing α conversion
- One usually treats α-equivalent terms as equal - i.e. use α equivalence classes of terms

Example

Show: $\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \sim\alpha\sim \lambda\ y.\ (\lambda\ x.\ x\ y)\ y$

- $\lambda\ x.\ (\lambda\ y.\ y\ x)\ x$ --α--> $\lambda\ z.\ (\lambda\ y.\ y\ z)\ z$ so
  $\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \sim\alpha\sim \lambda\ z.\ (\lambda\ y.\ y\ z)\ z$
- $(\lambda\ y.\ y\ z)$ --α--> $(\lambda\ x.\ x\ z)$ so
  $(\lambda\ y.\ y\ z) \sim\alpha\sim (\lambda\ x.\ x\ z)$ so
  $\lambda\ z.\ (\lambda\ y.\ y\ z)\ z \sim\alpha\sim \lambda\ z.\ (\lambda\ x.\ x\ z)\ z$
- $\lambda\ z.\ (\lambda\ x.\ x\ z)\ z$ --α--> $\lambda\ y.\ (\lambda\ x.\ x\ y)\ y$ so
  $\lambda\ z.\ (\lambda\ x.\ x\ z)\ z \sim\alpha\sim \lambda\ y.\ (\lambda\ x.\ x\ y)\ y$
- $\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \sim\alpha\sim \lambda\ y.\ (\lambda\ x.\ x\ y)\ y$

Substitution

- Defined on α-equivalence classes of terms
- P [N / x] means replace every free occurrence of x in P by N
  - P called *redex*; N called *residue*
- Provided that no variable free in P becomes bound in P [N / x]
  - Rename bound variables in P to avoid capturing free variables of N

Substitution

- $x\ [N / x] = N$
- $y\ [N / x] = y$ if $y \neq x$
- $(e_1\ e_2)\ [N / x] = ((e_1\ [N / x]\ )\ (e_2\ [N / x]\ ))$
- $(\lambda\ x.\ e)\ [N / x] = (\lambda\ x.\ e)$
- $(\lambda\ y.\ e)\ [N / x] = \lambda\ y.\ (e\ [N / x]\ )$ provided $y \neq x$ and y not free in N
  - Rename y in redex if necessary

Example

$$(\lambda\ y.\ y\ z)\ [(\lambda\ x.\ x\ y)\ /\ z] = ?$$

- Problems?
  - z in redex in scope of y binding
  - y free in the residue
- $(\lambda\ y.\ y\ z)\ [(\lambda\ x.\ x\ y)\ /\ z]$ --α--> $(\lambda\ w.w\ z)\ [(\lambda\ x.\ x\ y)\ /\ z] =$ $\lambda\ w.\ w\ (\lambda\ x.\ x\ y)$

Example

- Only replace free occurrences
- $(\lambda\ y.\ y\ z\ (\lambda\ z.\ z))\ [(\lambda\ x.\ x)\ /\ z] =$

$$\lambda\ y.\ y\ (\lambda\ x.\ x)\ (\lambda\ z.\ z)$$

Not

$$\lambda\ y.\ y\ (\lambda\ x.\ x)\ (\lambda\ z.\ (\lambda\ x.\ x))$$

β reduction

- β Rule: $(\lambda$ x. P) N --β--> P [N /x]

- Essence of computation in the lambda calculus
- Usually defined on α-equivalence classes of terms

Example

- $(\lambda\ z.\ (\lambda\ x.\ x\ y)\ z)\ (\lambda\ y.\ y\ z)$

  $\text{--}\beta\text{-->}\ (\lambda\ x.\ x\ y)\ (\lambda\ y.\ y\ z)$

  $\text{--}\beta\text{-->}\ (\lambda\ y.\ y\ z)\ y\ \text{--}\beta\text{-->}\ y\ z$

- $(\lambda\ x.\ x\ x)\ (\lambda\ x.\ x\ x)$

  $\text{--}\beta\text{-->}\ (\lambda\ x.\ x\ x)\ (\lambda\ x.\ x\ x)$

  $\text{--}\beta\text{-->}\ (\lambda\ x.\ x\ x)\ (\lambda\ x.\ x\ x)\ \text{--}\beta\text{-->}\ \ldots.$

α β Equivalence

- α β equivalence is the smallest congruence containing α equivalence and β reduction
- A term is in *normal form* if no subterm is α equivalent to a term that can be β reduced
- Hard fact (Church-Rosser): if e_1 and e_2 are $\alpha\beta$-equivalent and both are normal forms, then they are α equivalent

Order of Evaluation

- Not all terms reduce to normal forms
- Not all reduction strategies will produce a normal form if one exists

Lazy evaluation:

- Always reduce the left-most application in a top-most series of applications (i.e. Do not perform reduction inside an abstraction)

- Stop when term is not an application, or left-most application is not an application of an abstraction to a term

Example 1

- $(\lambda\ z.\ (\lambda\ x.\ x))\ ((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y))$
- Lazy evaluation:
- Reduce the left-most application:
- $(\lambda\ z.\ (\lambda\ x.\ x))\ ((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y))$ --β--> $(\lambda\ x.\ x)$

Eager evaluation

- (Eagerly) reduce left of top application to an abstraction
- Then (eagerly) reduce argument
- Then β-reduce the application

Example 1

- (λ z. (λ x. x))((λ y. y y) (λ y. y y))
- Eager evaluation:
- Reduce the rator of the top-most application to an abstraction: Done.
- Reduce the argument:
- (λ z. (λ x. x))((λ y. y y) (λ y. y y))

--β--> (λ z. (λ x. x))((λ y. y y) (λ y. y y))

--β--> (λ z. (λ x. x))((λ y. y y) (λ y. y y))…

Example 2

- $(\lambda\ x.\ x\ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$
- Lazy evaluation:

$(\lambda\ x.\ x\ \ x\ )((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$ --β-->

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. $\boxed{x}$ $\boxed{x}$)$\underline{((λ y. y y) (λ z. z))}$ --β-->

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. $\boxed{x}$ $\boxed{x}$)((λ y. y y) (λ z. z)) --β-->

$\boxed{((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))}$ $\boxed{((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))}$

Example 2

- $(\lambda\ x.\ x\ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$
- Lazy evaluation:

$(\lambda\ x.\ x\ \ x\ )((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$ --β-->

$\boxed{((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))}\ ((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z)$

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. $\boxed{y}$ $\boxed{y}$) (λ z. z)) ((λ y. y y) (λ z. z))

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. $\boxed{y}$ $\boxed{y}$) (λ z. z)) ((λ y. y y) (λ z. z))

--β--> ($\boxed{(λ z. z)}$ $\boxed{(λ z. z)}$)((λ y. y y) (λ z. z))

Example 2

- $(\lambda\ x.\ x\ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$
- Lazy evaluation:

$(\lambda\ x.\ x\ \ x\ )((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$ --β-->

$((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))\ ((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

--β--> $\boxed{((\lambda\ z.\ z\ )\ (\lambda\ z.\ z))}((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. y y) (λ z. z)) ((λ y. y y) (λ z. z))

--β--> ((λ z. $\boxed{z}$) (λ z. z))((λ y. y y) (λ z. z))

Example 2

- $(\lambda\ x.\ x\ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$
- Lazy evaluation:

$$(\lambda\ x.\ x\ \ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))\ \text{--}\beta\text{-->}$$

$$((\lambda\ y.\ y\ \ y)\ (\lambda\ z.\ z))\ ((\lambda\ y.\ y\ \ y)\ (\lambda\ z.\ z))$$

$$\text{--}\beta\text{-->}\ ((\lambda\ z.\ \boxed{z})\ \underline{(\lambda\ z.\ z)})((\lambda\ y.\ y\ \ y)\ (\lambda\ z.\ z))$$

$$\text{--}\beta\text{-->}\ \boxed{(\lambda\ z.\ z)}\ ((\lambda\ y.\ y\ \ y)\ (\lambda\ z.\ z))$$

Example 2

- $(\lambda\ x.\ x\ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$
- Lazy evaluation:

$(\lambda\ x.\ x\ \ x\ )((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$ --β-->

$((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))\ ((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

--β--> $((\lambda\ z.\ z\ )\ (\lambda\ z.\ z))((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

--β--> $(\lambda\ z.\ \boxed{z}\ )\ \underline{((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))}$ --β-->

$(\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z)$

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. y y) (λ z. z)) ((λ y. y y) (λ z. z))

--β--> ((λ z. z) (λ z. z))((λ y. y y) (λ z. z))

--β--> (λ z. z) ((λ y. y y) (λ z. z)) --β-->

(λ y. y y) (λ z. z)

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. y y) (λ z. z)) ((λ y. y y) (λ z. z))

--β--> ((λ z. z) (λ z. z))((λ y. y y) (λ z. z))

--β--> (λ z. z) ((λ y. y y) (λ z. z)) --β-->

(λ y. y y) (λ z. z) ~β~ λ z. z

Example 2

- $(\lambda\ x.\ x\ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$
- Eager evaluation:

$(\lambda\ x.\ x\ \ x)\ \boxed{((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))}$ --β-->

$(\lambda\ x.\ x\ \ x)\ \boxed{((\ \lambda\ z.\ z\ )\ (\lambda\ z.\ z))}$ --β-->

$(\lambda\ x.\ x\ \ x)\ \boxed{(\lambda\ z.\ z)}$ --β-->

$(\lambda\ z.\ z)\ (\lambda\ z.\ z)$ --β--> $\lambda\ z.\ z$

η (Eta) Reduction

- η Rule: λ x. f x --η--> f if x not free in f
  - Can be useful in each direction
  - Not valid in Ocaml
    - recall lambda-lifting and side effects
  - Not equivalent to (λ x. f) x --> f (inst of β)

- Example: λ x. (λ y. y) x --η--> λ y. y

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Untyped λ-Calculus

- Only three kinds of expressions:
  - Variables: x, y, z, w, …
  - Abstraction: $\lambda\, x.\, e$
    (Function creation)
  - Application: $e_1\ e_2$

How to Represent (Free) Data Structures (First Pass - Enumeration Types)

- Suppose τ is a type with *n* constructors: $C_1,\ldots,C_n$ (no arguments)
- Represent each term as an abstraction:
- Let $C_i \rightarrow \lambda\, x_1 \ldots x_n.\, x_i$
- Think: you give me what to return in each case (think match statement) and I'll return the case for the *i*'th constructor

How to Represent Booleans

- bool = True | False
- True $\rightarrow \lambda\, x_1.\, \lambda\, x_2.\, x_1 \quad \equiv_\alpha \quad \lambda\, x.\, \lambda\, y.\, x$
- False $\rightarrow \lambda\, x_1.\, \lambda\, x_2.\, x_2 \quad \equiv_\alpha \quad \lambda\, x.\, \lambda\, y.\, y$

- Notation
  - Will write

  $\lambda\, x_1 \ldots x_n.\, e$ for $\lambda\, x_1.\, \ldots\, \lambda x_n.\, e$

  $e_1\, e_2 \ldots e_n$ for $(\ldots(e_1\, e_2)\ldots e_n)$

Functions over Enumeration Types

- Write a “match” function
- match e with C_1 -> x_1

  | …

  | C_n -> x_n

$\rightarrow \quad \lambda\, x_1 \ldots x_n\, e.\ e\, x_1 \ldots x_n$

- Think: give me what to do in each case and give me a case, and I’ll apply that case

Functions over Enumeration Types

- type τ = $C_1|\ldots|C_n$
- match e with C_1 -> x_1

  | …

  | C_n -> x_n

- *match*τ = $\lambda\, x_1 \ldots x_n\, e.\ e\ x_1 \ldots x_n$

- e = expression (single constructor)

  x_i is returned if e = C_i

match for Booleans

- bool = True | False
- True $\rightarrow \lambda\, x_1\, x_2.\, x_1 \quad \equiv_\alpha \quad \lambda\, x\, y.\, x$
- False $\rightarrow \lambda\, x_1\, x_2.\, x_2 \quad \equiv_\alpha \quad \lambda\, x\, y.\, y$

- $\text{match}_{\textit{bool}} = ?$

match for Booleans

- $\text{bool} = \text{True} \mid \text{False}$
- $\text{True} \rightarrow \lambda\, x_1\, x_2.\, x_1 \quad \equiv_\alpha \quad \lambda\, x\, y.\, x$
- $\text{False} \rightarrow \lambda\, x_1\, x_2.\, x_2 \quad \equiv_\alpha \quad \lambda\, x\, y.\, y$

- $\text{match}_{\mathit{bool}} = \lambda\, x_1\, x_2\, e.\, e\, x_1\, x_2$
  $\equiv_\alpha \quad \lambda\, x\, y\, b.\, b\, x\, y$

How to Write Functions over Booleans

- if b then x_1 else x_2 $\rightarrow$
- if_then_else b x_1 x_2 = b x_1 x_2
- if_then_else $\equiv \lambda$ b x_1 x_2 . b x_1 x_2

How to Write Functions over Booleans

- Alternately:
- if b then x_1 else x_2 =
  match b with True -> x_1 | False -> x_2 $\rightarrow$
  $\text{match}_{\text{bool}}\ x_1\ x_2\ b =$
  $(\lambda\ x_1\ x_2\ b\ .\ b\ x_1\ x_2\ )\ x_1\ x_2\ b = b\ x_1\ x_2$
- if_then_else
  $\equiv \lambda\ b\ x_1\ x_2.\ (\text{match}_{\text{bool}}\ x_1\ x_2\ b)$
  $= \lambda\ b\ x_1\ x_2.\ (\lambda\ x_1\ x_2\ b\ .\ b\ x_1\ x_2\ )\ x_1\ x_2\ b$
  $= \lambda\ b\ x_1\ x_2.\ b\ x_1\ x_2$

Example:

not b

= match b with True -> False | False -> True

$\rightarrow$ $(\text{match}_{\text{bool}})$ False True b

= $(\lambda\, x_1\, x_2\, b\,.\, b\, x_1\, x_2\,)\,(\lambda\, x\, y.\, y)\,(\lambda\, x\, y.\, x)\, b$

= $b\,(\lambda\, x\, y.\, y)(\lambda\, x\, y.\, x)$

- not $\equiv \lambda\, b.\, b\,(\lambda\, x\, y.\, y)(\lambda\, x\, y.\, x)$
- Try and, or

and or

How to Represent (Free) Data Structures (Second Pass - Union Types)

- Suppose τ is a type with *n* constructors: type $\tau = C_1\, t_{11} \ldots t_{1k} \mid \ldots \mid C_n\, t_{n1} \ldots t_{nm},$
- Represent each term as an abstraction:
- $C_i\, t_{i1} \ldots t_{ij,} \rightarrow \lambda\, \mathrm{x}_1 \ldots \mathrm{x}_\mathrm{n}.\ \mathrm{x}_\mathrm{i}\, t_{i1} \ldots t_{ij,}$
- $C_i \rightarrow \lambda\, t_{i1} \ldots t_{ij,}\, \mathrm{x}_1 \ldots \mathrm{x}_\mathrm{n}\ .\ \mathrm{x}_\mathrm{i}\, t_{i1} \ldots t_{ij,}$
- Think: you need to give each constructor its arguments fisrt

How to Represent Pairs

- Pair has one constructor (comma) that takes two arguments
- type (α,β)pair = (,) α β
- (a , b) --> λ x . x a b
- (_ , _) --> λ a b x . x a b

Functions over Union Types

- Write a “match” function
- match e with $C_1\ y_1 \ldots y_{m1} \text{ -> } f_1\ y_1 \ldots y_{m1}$

  $\mid \ldots$

  $\mid C_n\ y_1 \ldots y_{mn} \text{ -> } f_n\ y_1 \ldots y_{mn}$

- *match*$\tau \rightarrow \ \lambda\, f_1 \ldots f_n\, e.\ e\ f_1 \ldots f_n$
- Think: give me a function for each case and give me a case, and I’ll apply that case to the appropriate fucntion with the data in that case

Functions over Pairs

- $\text{match}_{\text{pair}} = \lambda$ f p. p f

- fst p = match p with (x,y) -> x
- fst $\rightarrow \lambda$ p. $\text{match}_{\text{pair}}$ (λ x y. x)
  = (λ f p. p f) (λ x y. x) = λ p. p (λ x y. x)

- snd $\rightarrow \lambda$ p. p (λ x y. y)

How to Represent (Free) Data Structures (Third Pass - Recursive Types)

- Suppose τ is a type with n constructors:

  type τ $= C_1\ t_{11}\ \ldots\ t_{1k}\ |\ \ldots\ |\ C_n\ t_{n1}\ \ldots\ t_{nm,}$
- Suppose $t_{ih} : \tau$ (ie. is recursive)
- In place of a value t_{ih} have a function to compute the recursive value $r_{ih}\ x_1 \ldots x_n$
- $C_i\ t_{i1}\ \ldots\ r_{ih}\ \ldots t_{ij} \rightarrow\ \lambda\, x_1 \ldots x_n\,.\ x_i\ t_{i1}\ \ldots\ (r_{ih}\ x_1 \ldots x_n)\ \ldots\ t_{ij}$
- $C_i \rightarrow\ \lambda\ t_{i1}\ \ldots\ r_{ih}\ \ldots t_{ij}\ x_1 \ldots x_n\ .x_i\ t_{i1}\ \ldots\ (r_{ih}\ x_1 \ldots x_n)\ \ldots\ t_{ij,}$

How to Represent Natural Numbers

- nat = Suc nat | 0
- $\overline{\text{Suc}} = \lambda\, n\, f\, x.\ f\ (n\ f\ x)$
- $\overline{\text{Suc}\ n} = \lambda\, f\, x.\ f\ (n\ f\ x)$
- $\overline{0} = \lambda\, f\, x.\ x$
- Such representation called *Church Numerals*

Some Church Numerals

- $\overline{\text{Suc } 0} = (\lambda\, n\, f\, x.\ f\, (n\, f\, x))\ (\lambda\, f\, x.\ x)$ -->

$\lambda\, f\, x.\ f\, ((\lambda\, f\, x.\ x)\ f\, x)$ -->

$\lambda\, f\, x.\ f\, ((\lambda\, x.\ x)\ x)$ --> $\lambda\, f\, x.\ f\, x$

Apply a function to its argument once

Some Church Numerals

- $\overline{\text{Suc(Suc 0)}}$ = (λ n f x. f (n f x)) (Suc 0) -->

(λ n f x. f (n f x)) (λ f x. f x) -->

λ f x. f ((λ f x. f x) f x)) -->

λ f x. f ((λ x. f x) x)) --> λ f x. f (f x)

Apply a function twice

In general $\overline{n}$ = λ f x. f (… (f x)…) with n applications of f

Primitive Recursive Functions

- Write a "fold" function
- fold $f_1 \ldots f_n$ = match e
  with $C_1\ y_1 \ldots y_{m1} \text{ -> } f_1\ y_1 \ldots y_{m1}$
  | …
  | $Ci\ y_1 \ldots r_{ij} \ldots y_{in} \text{ -> } f_n\ y_1 \ldots (\text{fold } f_1 \ldots f_n\ r_{ij}) \ldots y_{mn}$
  | …
  | $C_n\ y_1 \ldots y_{mn} \text{ -> } f_n\ y_1 \ldots y_{mn}$
- $fold\tau \rightarrow \lambda\, f_1 \ldots f_n\, e.\ e\ f_1 \ldots f_n$
- Match in non recursive case a degenerate version of fold

Primitive Recursion over Nat

- fold f z n=
- match n with 0 -> z
- | Suc m -> f (fold f z m)
- $\overline{\text{fold}} \equiv \lambda\, f\, z\, n.\ n\ f\ z$
- $\overline{\text{is\_zero}}\ \ \overline{n} = \overline{\text{fold}}\ (\lambda\, r.\ \overline{\text{False}})\ \overline{\text{True}}\ \overline{n}$
- $= (\lambda\, f\, x.\ f^{\,n}\, x)\ (\lambda\, r.\ \text{False})\ \text{True}$
- $= ((\lambda\, r.\ \text{False})^{\,n}\,)\ \text{True}$
- $\equiv$ if n = 0 then True else False

Adding Church Numerals

- $\overline{n} \equiv \lambda f\, x.\ f^{n}\, x$ and $\overline{m} \equiv \lambda f\, x.\ f^{m}\, x$

- $\overline{n + m} = \lambda f\, x.\ f^{(n+m)}\, x$
  $= \lambda f\, x.\ f^{n}\, (f^{m}\, x) = \lambda f\, x.\ \overline{n}\, f\, (\overline{m}\, f\, x)$

- $\overline{+} \equiv \lambda n\, m\, f\, x.\ n\, f\, (m\, f\, x)$

- Subtraction is harder

Multiplying Church Numerals

- $\overline{n} \equiv \lambda\, f\, x.\ f^{\,n}\, x$ and $\overline{m} \equiv \lambda\, f\, x.\ f^{\,m}\, x$

- $\overline{n * m} = \lambda\, f\, x.\ (f^{\,n * m})\ x = \lambda\, f\, x.\ (f^{\,m})^{n}\ x$
$= \lambda\, f\, x.\ \overline{n}\ (\overline{m}\ f)\ x$

$\overline{*} \equiv \lambda\, n\, m\, f\, x.\ n\ (m\ f)\ x$

Predecessor

- let pred_aux n =
 match n with 0 -> (0,0)
 | Suc m
-> (Suc(fst(pred_aux m)), fst(pred_aux m)
= fold (λ r. (Suc(fst r), fst r)) (0,0) n

- pred ≡ λ n. snd (pred_aux n) n =
λ n. snd (fold (λ r.(Suc(fst r), fst r)) (0,0) n)

Recursion

- Want a λ-term Y such that for all term R we have
- $Y\ R = R\ (Y\ R)$
- Y needs to have replication to “remember” a copy of R
- $Y = \lambda\, y.\ (\lambda\, x.\ y(x\ x))\ (\lambda\, x.\ y(x\ x))$
- $Y\ R = (\lambda\, x.\ R(x\ x))\ (\lambda\, x.\ R(x\ x))$

  $= R\ ((\lambda\, x.\ R(x\ x))\ (\lambda\, x.\ R(x\ x)))$
- Notice: Requires lazy evaluation

Factorial

- Let F = λ f n. if n = 0 then 1 else n * f (n - 1)

Y F 3 = F (Y F) 3

= if 3 = 0 then 1 else 3 * ((Y F)(3 - 1))

= 3 * (Y F) 2 = 3 * (F(Y F) 2)

= 3 * (if 2 = 0 then 1 else 2 * (Y F)(2 - 1))

= 3 * (2 * (Y F)(1)) = 3 * (2 * (F(Y F) 1)) =…

= 3 * 2 * 1 * (if 0 = 0 then 1 else 0*(Y F)(0 -1))

= 3 * 2 * 1 * 1 = 6

Y in OCaml

```
# let rec y f = f (y f);;
val y : ('a -> 'a) -> 'a = <fun>
# let mk_fact =
   fun f n -> if n = 0 then 1 else n * f(n-1);;
val mk_fact : (int -> int) -> int -> int = <fun>
# y mk_fact;;
Stack overflow during evaluation (looping
   recursion?).
```

Eager Eval Y in Ocaml

```
# let rec y f x = f (y f) x;;
val y : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b =
  <fun>
# y mk_fact;;
- : int -> int = <fun>
# y mk_fact 5;;
- : int = 120
```

- Use recursion to get recursion

Some Other Combinators

- For your general exposure

- $I = \lambda\, x\, .\, x$
- $K = \lambda\, x.\ \lambda\, y.\ x$
- $K_* = \lambda\, x.\ \lambda\, y.\ y$
- $S = \lambda\, x.\ \lambda\, y.\ \lambda\, z.\ x\ z\ (y\ z)$

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Axiomatic Semantics

- Also called Floyd-Hoare Logic
- Based on formal logic (first order predicate calculus)
- Axiomatic Semantics is a logical system built from *axioms* and *inference rules*
- Mainly suited to simple imperative programming languages

Axiomatic Semantics

- Used to formally prove a property (*post-condition*) of the *state* (the values of the program variables) after the execution of program, assuming another property (*pre-condition*) of the state holds before execution

Axiomatic Semantics

- Goal: Derive statements of form

  $$\{P\}\ C\ \{Q\}$$

  - P , Q logical statements about state, P precondition, Q postcondition, C program

- Example: $\{x = 1\}\ x := x + 1\ \{x = 2\}$

Axiomatic Semantics

- *Approach:* For each type of language statement, give an axiom or inference rule stating how to derive assertions of form

$$\{P\}\ C\ \{Q\}$$

where C is a statement of that type

- Compose axioms and inference rules to build proofs for complex programs

Axiomatic Semantics

- An expression {P} C {Q} is a *partial correctness* statement
- For *total correctness* must also prove that C terminates (i.e. doesn't run forever)
  - Written: [P] C [Q]
- Will only consider partial correctness here

Language

- We will give rules for simple imperative language

<command>
 ::= <variable> := <term>
 | <command>; … ;<command>
 | if <statement> then <command> else <command>
 | while <statement> do <command>

- Could add more features, like for-loops

Substitution

- Notation: P[e/v] (sometimes P[v <- e])
- Meaning: Replace every v in P by e
- Example:

$$(x + 2)\,[y\text{-}1/x] = ((y - 1) + 2)$$

The Assignment Rule

$$\frac{}{\{P\,[e/x]\}\; x := e\; \{P\}}$$

Example:

$$\frac{}{\{\quad ? \quad\}\; x := y\; \{x = 2\}}$$

The Assignment Rule

$$\frac{}{\{P\,[e/x]\}\ x := e\ \{P\}}$$

Example:

$$\frac{}{\{\boxed{\_}= 2\}\ x := y\ \{\boxed{x} = 2\}}$$

The Assignment Rule

$$\frac{\qquad}{\{P\ [e/x]\}\ x := e\ \{P\}}$$

Example:

$$\frac{\qquad}{\{y = 2\}\ x := y\ \{x = 2\}}$$

The Assignment Rule

$$\overline{\{P\,[e/x]\,\}\ x := e\ \{P\}}$$

Examples:

$$\overline{\{y = 2\}\ x := y\ \{x = 2\}}$$

$$\overline{\{y = 2\}\ x := 2\ \{y = x\}}$$

$$\overline{\{x + 1 = n + 1\}\ x := x + 1\ \ \{x = n + 1\}}$$

$$\overline{\{2 = 2\}\ x := 2\ \{x = 2\}}$$

The Assignment Rule – Your Turn

- What is the weakest precondition of

$$x := x + y \ \{x + y = w - x\}?$$

$$\{ \qquad\qquad ? \qquad\qquad \}$$

$$x := x + y$$

$$\{x + y = w - x\}$$

The Assignment Rule – Your Turn

- What is the weakest precondition of

$$x := x + y \ \{x + y = w - x\}?$$

$$\{(x + y) + y = w - (x + y)\}$$

$$x := x + y$$

$$\{x + y = w - x\}$$

Precondition Strengthening

$$\frac{P \rightarrow P' \qquad \{P'\}\ C\ \{Q\}}{\{P\}\ C\ \{Q\}}$$

- Meaning: If we can show that P implies P' ($P \rightarrow P'$) and we can show that $\{P'\}\ C\ \{Q\}$, then we know that $\{P\}\ C\ \{Q\}$
- P is *stronger* than P' means $P \rightarrow P'$

Precondition Strengthening

- Examples:

$$\frac{x = 3 \rightarrow x < 7 \quad \{x < 7\}\ x := x + 3\ \{x < 10\}}{\{x = 3\}\ x := x + 3\ \{x < 10\}}$$

$$\frac{\text{True} \rightarrow 2 = 2 \quad \{2 = 2\}\ x := 2\ \{x = 2\}}{\{\text{True}\}\ x := 2\ \{x = 2\}}$$

$$\frac{x=n \rightarrow x+1=n+1 \quad \{x+1=n+1\}\ x:=x+1\ \{x=n+1\}}{\{x=n\}\ x:=x+1\ \{x=n+1\}}$$

Which Inferences Are Correct?

$$\frac{\{x > 0 \ \&\ x < 5\}\ x := x * x\ \{x < 25\}}{\{x = 3\}\ x := x * x\ \{x < 25\}}$$

$$\frac{\{x = 3\}\ x := x * x\ \{x < 25\}}{\{x > 0 \ \&\ x < 5\}\ x := x * x\ \{x < 25\}}$$

$$\frac{\{x * x < 25\}\ x := x * x\ \{x < 25\}}{\{x > 0 \ \&\ x < 5\}\ x := x * x\ \{x < 25\}}$$

Which Inferences Are Correct?

$$\frac{\{x > 0 \;\&\; x < 5\}\ x := x * x\ \{x < 25\}}{\{x = 3\}\ x := x * x\ \{x < 25\}} \quad \checkmark$$

$$\cancel{\frac{\{x = 3\}\ x := x * x\ \{x < 25\}}{\{x > 0 \;\&\; x < 5\}\ x := x * x\ \{x < 25\}}}$$

$$\frac{\{x * x < 25\}\ x := x * x\ \{x < 25\}}{\{x > 0 \;\&\; x < 5\}\ x := x * x\ \{x < 25\}} \quad \checkmark$$

Sequencing

$$\frac{\{P\}\ C_1\ \{Q\} \qquad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1;\ C_2\ \{R\}}$$

- Example:

$$\frac{\begin{array}{c}\{z = z \ \&\ z = z\}\ x := z\ \{x = z \ \&\ z = z\} \\ \{x = z \ \&\ z = z\}\ y := z\ \{x = z \ \&\ y = z\}\end{array}}{\{z = z \ \&\ z = z\}\ x := z;\ y := z\ \{x = z \ \&\ y = z\}}$$

Sequencing

$$\frac{\{P\}\ C_1\ \{Q\} \qquad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1;\ C_2\ \{R\}}$$

- Example:

$$\frac{\{z = z \ \&\ z = z\}\ x := z\ \{x = z \ \&\ z = z\} \qquad \{x = z \ \&\ z = z\}\ y := z\ \{x = z \ \&\ y = z\}}{\{z = z \ \&\ z = z\}\ x := z;\ y := z\ \{x = z \ \&\ y = z\}}$$

Postcondition Weakening

$$\frac{\{P\}\ C\ \{Q'\} \qquad Q' \rightarrow Q}{\{P\}\ C\ \{Q\}}$$

Example:

$$\frac{\{z = z \,\&\, z = z\}\ x := z;\ y := z\ \{x = z \,\&\, y = z\} \qquad (x = z \,\&\, y = z) \rightarrow (x = y)}{\{z = z \,\&\, z = z\}\ x := z;\ y := z\ \{x = y\}}$$

Rule of Consequence

$$\frac{P \rightarrow P' \qquad \{P'\}\ C\ \{Q'\} \qquad Q' \rightarrow Q}{\{P\}\ C\ \{Q\}}$$

- Logically equivalent to the combination of Precondition Strengthening and Postcondition Weakening
- Uses $P \rightarrow P$ and $Q \rightarrow Q$

If Then Else

$$\frac{\{P \text{ and } B\}\ C_1\ \{Q\} \quad \{P \text{ and } (\text{not } B)\}\ C_2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \{Q\}}$$

- Example: Want

$$\{y=a\}$$
$$\text{if } x < 0 \text{ then } y{:=}\ y-x \text{ else } y{:=}\ y+x$$
$$\{y=a+|x|\}$$

Suffices to show:

(1) $\{y=a \,\&\, x<0\}\ \ y{:=}y-x\ \ \{y=a+|x|\}$ and

(4) $\{y=a \,\&\, \text{not}(x<0)\}\ \ y{:=}y+x\ \ \{y=a+|x|\}$

{y=a&x<0} y:=y-x {y=a+|x|}

$$\begin{array}{ll} (3) & (y=a \,\&\, x<0) \rightarrow y-x=a+|x| \\ (2) & \{y-x=a+|x|\}\ y:=y-x\ \{y=a+|x|\} \\ \hline (1) & \{y=a \,\&\, x<0\}\ y:=y-x\ \{y=a+|x|\} \end{array}$$

(1) Reduces to (2) and (3) by Precondition Strengthening
(2) Follows from assignment axiom
(3) Because $x<0 \rightarrow |x| = -x$

{y=a&not(x<0)} y:=y+x {y=a+|x|}

(6) (y=a&not(x<0))➔(y+x=a+|x|)

(5) {y+x=a+|x|} y:=y+x {y=a+|x}}

---

(4) {y=a&not(x<0)} y:=y+x {y=a+|x|}

(4) Reduces to (5) and (6) by Precondition Strengthening

(5) Follows from assignment axiom

(6) Because not(x<0) ➔ |x| = x

If then else

(1) $\{y=a \& x<0\} y:=y-x \{y=a+|x|\}$

(4) $\{y=a \& not(x<0)\} y:=y+x \{y=a+|x|\}$

$$\frac{\begin{array}{c}\{y=a \& x<0\}\ y:=y-x\ \{y=a+|x|\} \\ \{y=a \& \mathrm{not}(x<0)\}\ y:=y+x\ \{y=a+|x|\}\end{array}}{\{y=a\}\ \text{if } x<0 \text{ then } y:=y-x \text{ else } y:=y+x\ \{y=a+|x|\}}$$

By the if_then_else rule

While

- We need a rule to be able to make assertions about **while** loops.
  - Inference rule because we can only draw conclusions if we know something about the body
  - Let’s start with:

$$\frac{\{\ ?\ \}\ C\ \{\ ?\ \}}{\{\ ?\ \}\ \textbf{while}\ B\ \textbf{do}\ C\ \{\ P\ \}}$$

While

- The loop may never be executed, so if we want P to hold after, it had better hold before, so let's try:

$$\frac{\{\ ?\ \}\ C\ \{\ ?\ \}}{\{\ P\ \}\ \textbf{while}\ B\ \textbf{do}\ C\ \{\ P\ \}}$$

While

- If all we know is P when we enter the **while** loop, then we all we know when we enter the body is (P and B)
- If we need to know P when we finish the **while** loop, we had better know it when we finish the loop body:

$$\frac{\{\,P \text{ and } B\}\;\; C\;\; \{\,P\,\}}{\{\,P\,\}\;\; \textbf{while}\;\; B\;\; \textbf{do}\;\; C\;\; \{\,P\,\}}$$

While

- We can strengthen the previous rule because we also know that when the loop is finished, **not B** also holds
- Final **while** rule:

$$\frac{\{\ P \text{ and } B\ \}\ \ C\ \ \{\ P\ \}}{\{\ P\ \}\ \textbf{while}\ \ B\ \ \textbf{do}\ \ C\ \ \{\ P \text{ and not } B\ \}}$$

While

$$\frac{\{\ P \text{ and } B\ \}\ \ C\ \ \{\ P\ \}}{\{\ P\ \}\ \textbf{while}\ \ B\ \ \textbf{do}\ \ C\ \ \{\ P \text{ and not } B\ \}}$$

- P satisfying this rule is called a *loop invariant* because it must hold before and after the each iteration of the loop

While

- **While** rule generally needs to be used together with precondition strengthening and postcondition weakening
- There is NO algorithm for computing the correct P; it requires intuition and an understanding of why the program works

Example

- Let us prove

  {x>= 0 and x = a}

  fact := 1;

  while x > 0 do (fact := fact * x; x := x –1)

  {fact = a!}

Example

- We need to find a condition P that is true both before and after the loop is executed, and such that

$$(P \text{ and not } x > 0) \rightarrow (fact = a!)$$

Example

- First attempt:

$$\{a! = fact * (x!)\}$$

- Motivation:
- What we want to compute: **a!**
- What we have computed: **fact**

  which is the sequential product of **a** down through **(x + 1)**
- What we still need to compute: **x!**

Example

By post-condition weakening suffices to show

1. {x>=0 and x = a}

   fact := 1;

   while x > 0 do (fact := fact * x; x := x –1)

   {a! = fact * (x!) and not (x > 0)}

and

2. {a! = fact * (x!) and not (x > 0) } $\rightarrow$ {fact = a!}

Problem

2. {a! = fact * (x!) and not (x > 0)} ➔ {fact = a!}

- Don't know this if x < 0
- Need to know that x = 0 when loop terminates
- Need a new loop invariant
- Try adding x >= 0
- Then will have x = 0 when loop is done

Example

Second try, combine the two:

$$P = \{a! = fact * (x!) \text{ and } x >= 0\}$$

Again, suffices to show

1. {x>=0 and x = a}
   fact := 1;
   while x > 0 do (fact := fact * x; x := x –1)
   {P and not x > 0}

and

2. {P and not x > 0} ➔ {fact = a!}

Example

- For 2, we need

  {a! = fact * (x!) and x >=0 and not (x > 0)} ➔ {fact = a!}

  But {x >=0 and not (x > 0)} ➔ {x = 0} so

  fact * (x!) = fact * (0!) = fact

  Therefore

  {a! = fact * (x!) and x >=0 and not (x > 0)} ➔ {fact = a!}

Example

- For 1, by the sequencing rule it suffices to show

3. {x>=0 and x = a}
   fact := 1
   {a! = fact * (x!) and x >=0 }

And

4. {a! = fact * (x!) and x >=0}
   while x > 0 do
   (fact := fact * x; x := x –1)
   {a! = fact * (x!) and x >=0 and not (x > 0)}

Example

- Suffices to show that

  {a! = fact * (x!) and x >= 0}

  holds before the while loop is entered and that if

  {(a! = fact * (x!)) and x >= 0 and x > 0}

  holds before we execute the body of the loop, then

  {(a! = fact * (x!)) and x >= 0}

  holds after we execute the body

Example

By the assignment rule, we have

{a! = 1 * (x!) and x >= 0}
fact := 1
{a! = fact * (x!) and x >= 0}

Therefore, to show (3), by precondition strengthening, it suffices to show

(x>= 0 and x = a) → (a! = 1 * (x!) and x >= 0)

Example

$$(x >= 0 \text{ and } x = a) \rightarrow$$
$$(a! = 1 * (x!) \text{ and } x >= 0)$$

holds because $x = a \rightarrow x! = a!$

Have that {a! = fact * (x!) and x >= 0} holds at the start of the while loop

Example

To show (4):

{a! = fact * (x!) and x >=0}
while x > 0 do
(fact := fact * x; x := x –1)
{a! = fact * (x!) and x >=0 and not (x > 0)}

we need to show that

{(a! = fact * (x!)) and x >= 0}

is a loop invariant

Example

We need to show:

{(a! = fact * (x!)) and x >= 0 and x > 0}

(fact = fact * x; x := x – 1)

{(a! = fact * (x!)) and x >= 0}

We will use assignment rule, sequencing rule and precondition strengthening

Example

By the assignment rule, we have

{(a! = fact * ((x-1)!)) and x – 1 >= 0}

x := x – 1

{(a! = fact * (x!)) and x >= 0}

By the sequencing rule, it suffices to show

{(a! = fact * (x!)) and x >= 0 and x > 0}

fact = fact * x

{(a! = fact * ((x-1)!)) and x – 1 >= 0}

Example

By the assignment rule, we have that

{(a! = (fact * x) * ((x-1)!)) and x – 1 >= 0}

fact = fact * x

{(a! = fact * ((x-1)!)) and x – 1 >= 0}

By Precondition strengthening, it suffices to show that

((a! = fact * (x!)) and x >= 0 and x > 0) $\rightarrow$

((a! = (fact * x) * ((x-1)!)) and x – 1 >= 0)

Example

However

$$fact * x * (x - 1)! = fact * x$$

and $(x > 0) \rightarrow x - 1 >= 0$

since x is an integer,so

$$\{(a! = fact * (x!)) \text{ and } x >= 0 \text{ and } x > 0\} \rightarrow$$

$$\{(a! = (fact * x) * ((x-1)!)) \text{ and } x - 1 >= 0\}$$

Example

Therefore, by precondition strengthening

{(a! = fact * (x!)) and x >= 0 and x > 0}

fact = fact * x

{(a! = fact * ((x-1)!)) and x – 1 >= 0}

This finishes the proof