CRICOS PROVIDER 00123M

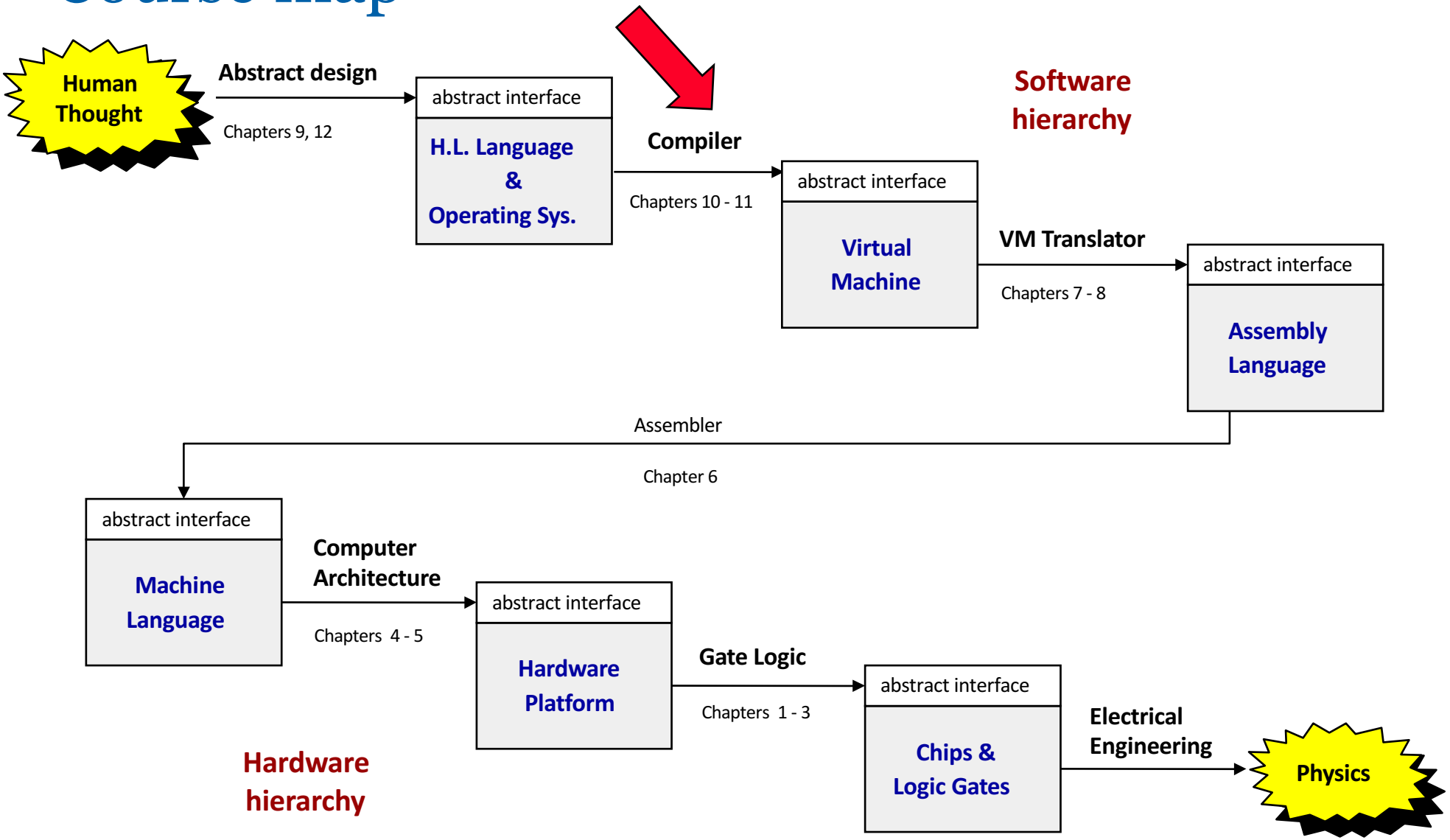
School of Computer Science

COMP SCI 2000 Computer Systems
Lecture 19

adelaide.edu.au

seek LIGHT

Course map


Human Thought
Abstract design
Chapters 9, 12
abstract interface
H.L. Language & Operating Sys.
Compiler
Chapters 10 - 11
abstract interface
Virtual Machine
Software hierarchy
VM Translator
Chapters 7 - 8
abstract interface
Assembly Language
Assembler
Chapter 6
abstract interface
Machine Language
Computer Architecture
Chapters 4 - 5
abstract interface
Hardware Platform
Gate Logic
Chapters 1 - 3
abstract interface
Chips & Logic Gates
Electrical Engineering
Physics
Hardware hierarchy

Review

- Syntax analysis
  - Code generation
  - Parsing

Preview

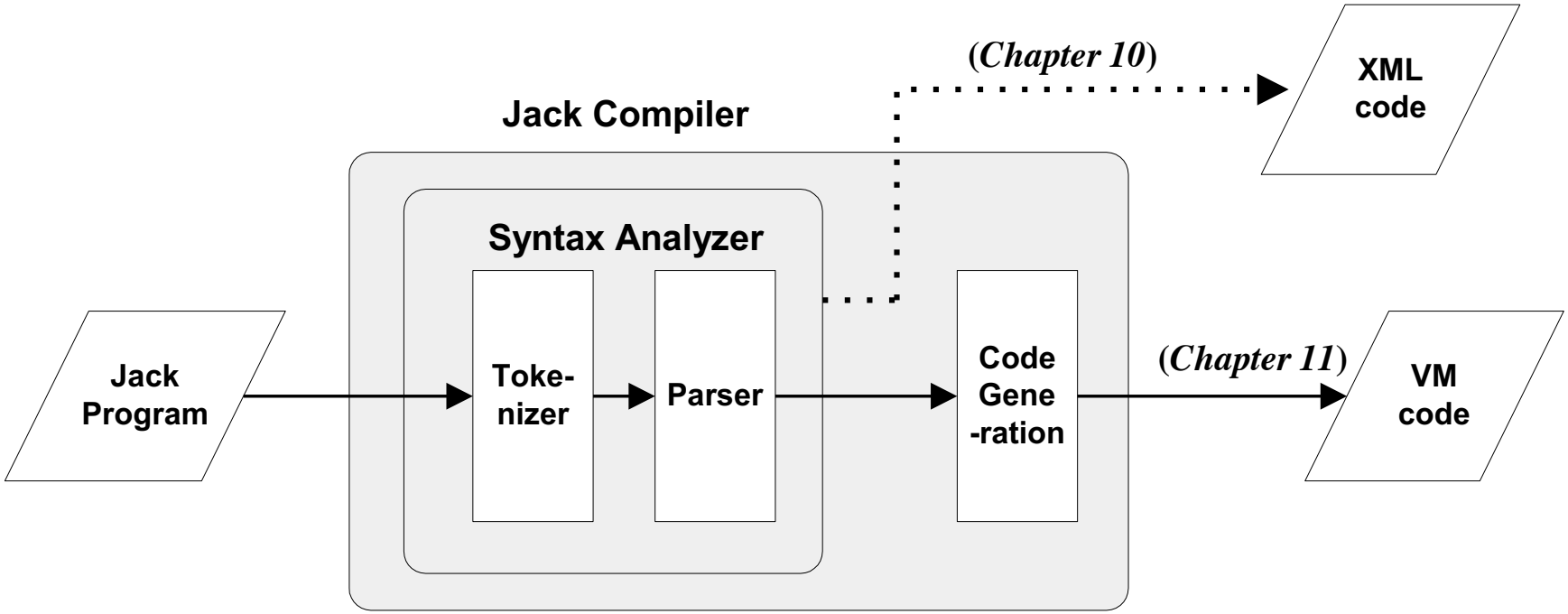
- Code generation
  - Process outline
  - Examples:
    - Variables
    - Methods
    - Classes
    - Objects
    - Arrays
    - Expressions
    - Program Flow
- Chapter 11 of textbook.

We cover variables in this lecture

The big picture

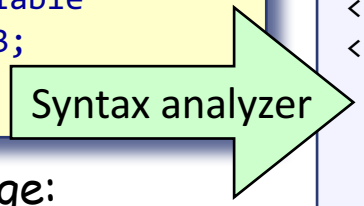
1. Syntax analysis: extracting the semantics from the source code — Last Week

2. Code generation: expressing the semantics using the target language — This week



Syntax analysis (review)

```
class Bar
{
    method Fraction foo(int y)
   {
       var int temp; // a variable
       let temp = (xxx+12)*-63;
       ...
   ...
```




```
<vardec>
  <keyword> var </keyword>
  <type><keyword> int </keyword></type>
  <varName><identifier> temp </identifier></varName>
  <symbol> ; </symbol>
</vardec>
<statements>
  <statement>
    <letstatement>
      <keyword> let </keyword>
      <varName><identifier> temp </identifier></varName>
      <symbol> = </symbol>
      <expression>
       <term>
        <symbol> ( </symbol>
        <expression>
         <term>
           <varName><identifier> xxx </identifier></varName>
         </term>
         <op><symbol> + </symbol></op>
         <term>
           <integerConstant> 12 </integerConstant>
          </term>
        </expression>
        <symbol> ) </symbol>
        <op><symbol> * </symbol></op>
        <expression>
           <term>
            <unaryOp><symbol > - </symbol></unaryOp>
                ...
```

The code generation challenge:

- Program = a series of operations that manipulate data
- Compiler: converts each "understood" (parsed) source operation and data item into corresponding operations and data items in the target language
- Thus, we have to generate code for
  - handling data and handling operations
- Our approach: morph the syntax analyzer into a full-blown compiler: instead of generating XML, we'll make it generate VM code.

Memory segments (review)

VM memory Commands:

pop *segment i*

push *segment i*

Where *i* is a non-negative integer and *segment* is one of the following:

`static:` holds values of global variables, shared by all functions in the same class

`argument:` holds values of the argument variables of the current function

`local:` holds values of the local variables of the current function

`this:` holds values of the private ("object") variables of the current object

`that:` holds memory address to access, typically array elements (silly name, sorry)

`constant:` holds all the constants in the range `0` ... `32767` (pseudo memory segment)

`pointer:` holds values `this` and `that` so programs can change the segment locations

`temp:` fixed 8-entry segment that holds temporary variables for general use; Shared by all VM functions in the program.

Code generation example

```
method int foo()
{
  var int x;
  let x = x + 1;
  ...
```

Syntax analysis

```
<letstatement>
  <keyword> let </keyword>
    <varName><identifier> x </identifier></varName>
  <symbol> = </symbol>
  <expression>
    <term>
      <varName><identifier> x </identifier></varName>
    </term>
    <op><symbol> + </symbol></op>
    <term>
        <integerConstant> 1 </integerConstant>
     </term>
  </expression>
</letstatement>
```

Code generation

```
push local 0
push constant 1
add
pop local 0
```

(note that x is the first local variable declared in the method)

Handling variables

When the compiler encounters a variable, say x, in the source code, it has to know:

What is x's *data type*?

Primitive, or ADT (class name) ?

(Need to know in order to properly allocate RAM resources for its representation)

What *kind* of variable is x?

`local, static, field, argument` ?

(We need to know in order to properly allocate it to the right memory segment;
this also indicates the variable's life cycle).

Do worksheet question 1

Handling variables: mapping them on memory segments

```
class BankAccount {
   // Class variables
   static int nAccounts;
   static int bankCommission;
   // account properties
   field int id;
   field String owner;
   field int balance;

   method void transfer(int sum, BankAccount from, Date when) {
      var int i, j;    // Some local variables
      var Date due;    // Date is a user-defined type
      let balance = (balance + sum) - commission(sum * 5);
      // More code ...
   }
```

- The target language uses 8 memory segments
- Each memory segment, e.g. `static`, is an indexed sequence of 16-bit values that can be referred to as `static 0`, `static 1`, `static 2`, etc.

When compiling this class, we have to create the following mappings:

The class variables	`nAccounts` , `bankCommission`	are mapped onto	`static 0,1`
The object fields	`id`, `owner`, `balance`	are mapped onto	`this 0,1,2`
The argument variables	`sum`, `bankAccount`, `when`	are mapped onto	`argument 1,2,3`
The local variables	`i`, `j`, `due`	are mapped ono	`local 0,1,2`

Handling variables: symbol tables

```
class BankAccount {
   // Class variables
   static int nAccounts;
   static int bankCommission;
   // account properties
   field int id;
   field String owner;
   field int balance;

   method void transfer(int sum, BankAccount from, Date when) {
      var int i, j;    // Some local variables
      var Date due;    // Date is a user-defined type
      let balance = (balance + sum) - commission(sum * 5);
      // More code ...
   }
```

Class-scope symbol table

Name	Type	Kind	#
nAccounts	int	static	0
bankCommission	int	static	1
id	int	field	0
owner	String	field	1
balance	int	field	2

Do worksheet question 2

Method-scope (transfer) symbol table

Name	Type	Kind	#
this	BankAccount	argument	0
sum	int	argument	1
from	BankAccount	argument	2
when	Date	argument	3
i	int	var	0
j	int	var	1
due	Date	var	2

How the compiler uses symbol tables:

- The compiler builds and maintains a linked list of symbol tables, each reflecting a single scope nested within the next one in the list
- Identifier lookup works from the current symbol table back to the list's head

  (a classical implementation).

Handling variables: managing their life cycle

Class-scope symbol table

Name	Type	Kind	#
nAccounts	int	static	0
bankCommission	int	static	1
id	int	field	0
owner	String	field	1
balance	int	field	2

Method-scope (transfer) symbol table

Name	Type	Kind	#
this	BankAccount	argument	0
sum	int	argument	1
from	BankAccount	argument	2
when	Date	argument	3
i	int	var	0
j	int	var	1
due	Date	var	2

Variables life cycle

`static` variables: single copy must be kept alive throughout the program duration

`field` variables: a different copy must be kept for each object, stored in segment `this`

`var` variables: created on subroutine entry, stored in segment `local`

`argument` variables: created during subroutine entry, stored in segment `argument`.

Good news: the VM implementation already handles all these details !

Review

- In this lecture we looked at
  - The basics and context of code generation
  - The translation of variables into VM code
- Next lecture we will look at
  - The translation of
    - Methods
    - Classes
    - Objects
    - Arrays
    - Expressions
    - Program Flow