CRICOS PROVIDER 00123M

School of Computer Science

COMP SCI 2000 Computer Systems
Lecture 20B

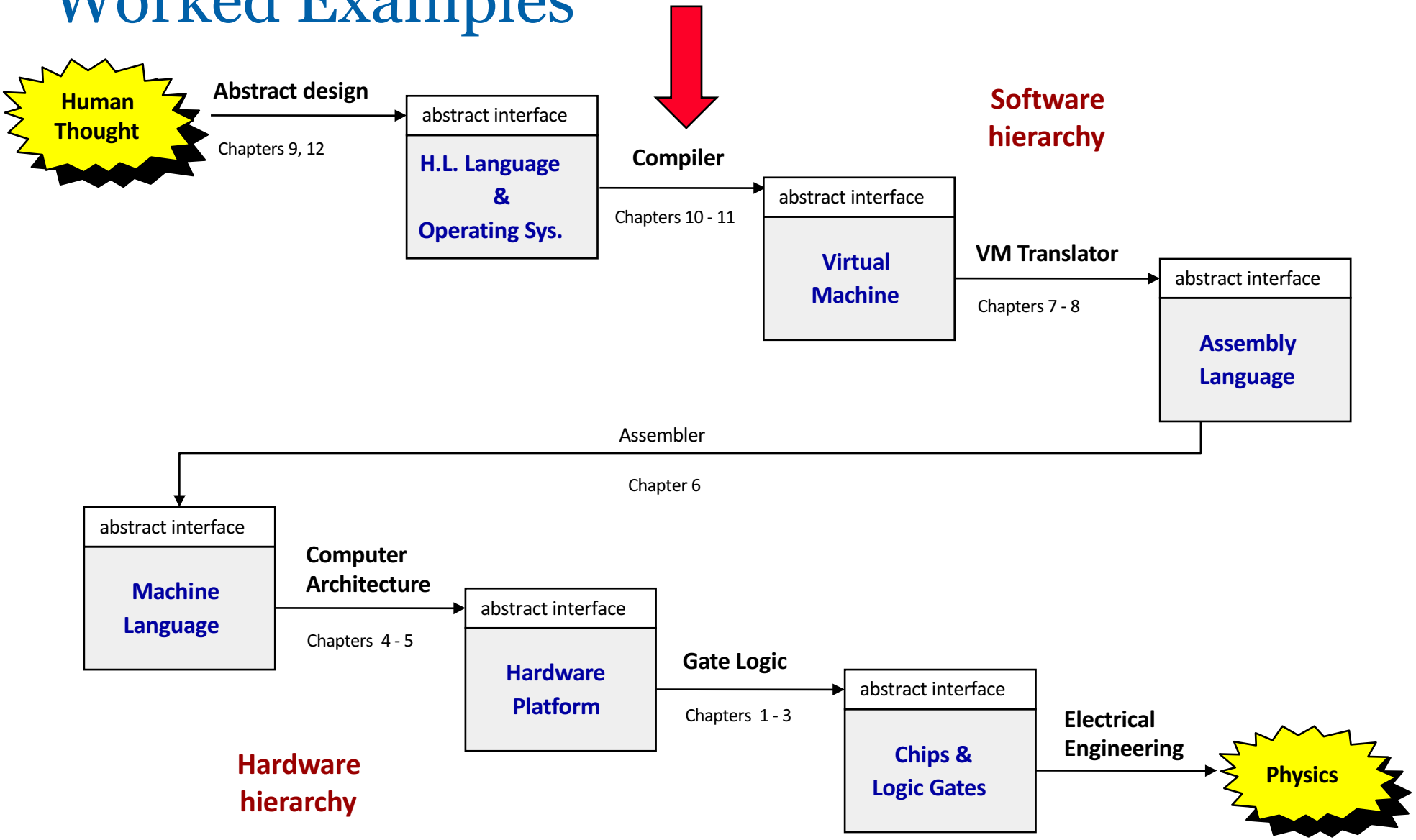
adelaide.edu.au

seekLIGHT

Lecture Plan

- Tokeniser Worked Example
- Parser Worked Example
- Code Generation Worked Example

Worked Examples



Tokeniser

- Tokeniser
  - the language will have a grammar for it tokens (symbols/atoms)
  - the next character should identify the kind of token
  - an additional check may be required to choose the token kind
  - always read the next character before returning a token
- Assume we are implementing the following:

```
int ch ;                    // last character read, a char is an int
void nextch()               // read another character
{
     ch = cin.get() ;       // read from standard input
}

string tokenvalue ;         // last token read
string tokenclass ;         // class of last token read

string next_token()         // returns tokenvalue or tokenclass
{ ... }
```

Tokeniser

- Six cases to consider
  - End of File
  - Characters that we ignore, eg space, tab, newline, carriage return
  - Comments that we ignore, eg // ..., /* ... */, ...
  - Single characters that start a token, eg +, -, /, ...
  - Range of characters that start a token, eg letters|digits|...
  - None of the above – an error!

- Nice to have for each kind of complex token
  - a function to tell if a character can start the token
  - a function to tell if a character can be part of a token
  - a function to tell which token class the token belongs too

Tokeniser

```
string next_token()     // returns tokenvalue or tokenclass
{
    if ( tokenvalue == "?" ) return "?" ;

    tokenvalue = "" ;
    while ( true )
    {
        switch(ch)
        {
        case EOF: tokenvalue = "?" ; tokenclass = "?" ; return "?" ;
        ...
        case '-':
                  tokenvalue += ch ;
                  tokenclass = "symbol" ;
                  nextch() ;
                  return tokenvalue ;
        ...
        case ' ': nextch() ;
                  break ;
```

Tokeniser

```
            ...
            default:
                    if ( startToken(ch) )
                    {
                        while ( inToken(ch) )
                        {
                            tokenvalue += ch ;
                            nextch() ;
                        }
                        tokenclass = lookupClass(tokenvalue);
                        return tokenclass ;
                    } else
                    ...
                    // handle errors by pretending to be at End of File
                    tokenvalue = "?" ; tokenclass = "?" ; return "?" ;
            }
        }
    }
```

Attempt Worksheet Question 1

Parsing

- Recursive Descent Parsing
  - one function per non-terminal (rule)
  - next token indicates the next rule to parse
- Nice to have
  - `mustbe()` for tokens that must be present
  - `have()` for tokens that are optional

- **If Statements**

```
ifStatement ::= 'if' '(' expression ')' '{' statements '}' \
                ( 'else' '{' statements '}' )?
```

Parsing – If Statements

```
ifStatement ::= 'if' '(' expression ')' '{' statements '}' \
                ( 'else' '{' statements '}' )?
```

```
void parseIfStatement()       // assumes "if" read by parseStatement()
{
    // handles all details of any expression
    mustbe("(") ; parseExpression() ; mustbe(")") ;

    // handles all details of any statements
    mustbe("{") ; parseStatements() ; mustbe("}") ;

    // optional else part
    if ( have("else") )
    {
        // handles all details of any statements
        mustbe("{") ; parseStatements() ; mustbe("}") ;
    }
}
```

Attempt Worksheet Question 2

Code Generation

- Code Generation Rules
  - each non-terminal may have an associated rule
  - recursive calls handle details of any sub-rules

- **If Statements**

```
ifStatement ::= 'if' '(' expression ')' '{' statements '}' \
                ( 'else' '{' statements '}' )?

ifStatement ::= expression
                if-goto IF_TRUEn
                goto IF_FALSEn
                label IF_TRUEn
                statements
                goto IF_ENDn
                label IF_FALSEn
                ( statements )?
                label IF_ENDn
```

Code Generation – If Statements

```
ifStatement ::= 'if' '(' expression ')' '{' statements '}' ( 'else' '{' statements '}' )?
```

```
void parseIfStatement()
{
    mustbe("(") ; parseExpression() ; mustbe(")") ;  // pushes TRUE or FALSE

    int n = ifcount++ ;
    cout << "if-goto IF_TRUE" << n << endl ;
    cout << "goto IF_FALSE" << n << endl ;
    cout << "label IF_TRUE" << n << endl ;

    mustbe("{") ; parseStatements() ; mustbe("}") ; // parse/code gen the statements

    cout << "goto IF_END" << n << endl ;
    cout << "label IF_FALSE" << n << endl ;

    if ( have("else") )
    {
        mustbe("{") ; parseStatements() ; mustbe("}") ;// parse/code gen the statements
    }

    cout << "label IF_END" << n << endl ;
}
```

Attempt Worksheet Question 3

Summary

- Tokeniser
  - can mechanically translate token rules into code
- Recursive Descent Parsing
  - can mechanically translate LL(1) grammar rules into code
- Code Generation
  - can mechanically translate code generation rules into code

Next Lecture

- Jack Operating System
  - Jack OS Libraries
- Optimisation
  - Processor power consumption
  - The length of the processor clock cycle
  - The effect of adding new instructions