COMP36511 - Compilers
Lecture 1: Introduction

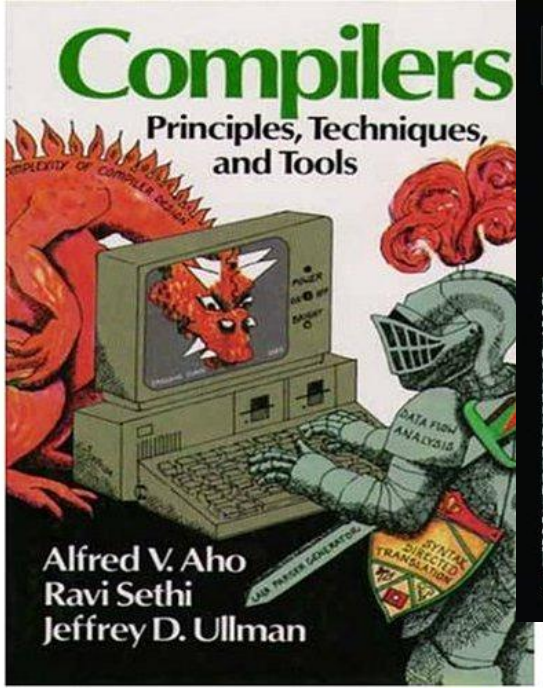
- Module Aims:
  - Any program written in a programming language must be translated before it can be executed. This translation is typically accomplished by a software system called compiler. This module aims to introduce students to the principles and techniques used to perform this translation and the issues that arise in the construction of a compiler.

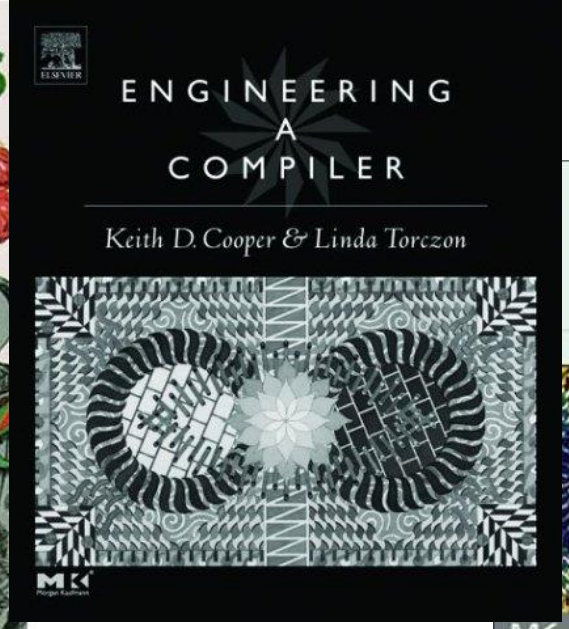
COMP36511 - Compilers (cont.)

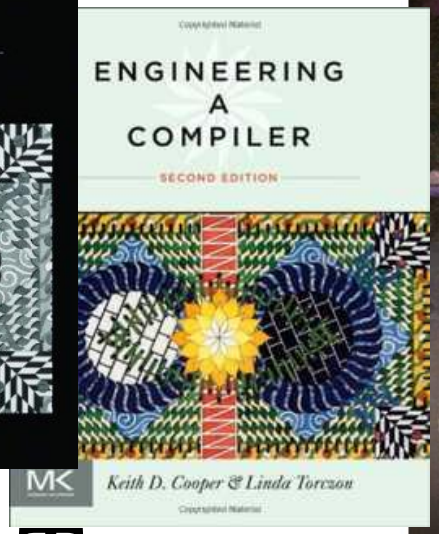
- Learning Outcomes:
  - A student successfully completing this module should be able to:
    - understand the principles governing all phases of the compilation process.
    - understand the role of each of the basic components of a standard compiler.
    - show awareness of the problems of and methods and techniques applied to each phase of the compilation process.
    - apply standard techniques to solve basic problems that arise in compiler construction.
    - understand how the compiler can take advantage of particular processor characteristics to generate good code.

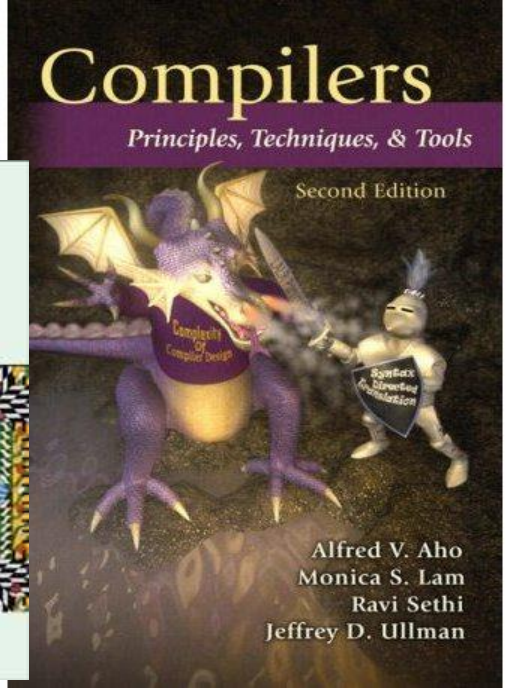
Course Lecturer - and Lectures

- Who am I?
  - Rizos Sakellariou – **rizos@manchester.ac.uk**
  - Research: Distributed & Large-Scale Software Systems (including scheduling, performance optimisation, etc), and (in the distant past) parallelising compilers
- Material
  - Check Blackboard for handouts – also visit the legacy website:
  - **http://studentnet.cs.manchester.ac.uk/ugt/COMP36512/**
- How to study:
  - Read, ask yourself, find material (online or through book pointers)
  - Post a question on **BlackBoard discussion board**
  - Online discussion: Tuesdays 15:00-17:00 using zoom
- Keep a flexible approach in mind.
    - This is engineering: there are tradeoffs/constraints/optimisation
- Assessment: online exam









Course Texts

- **Aho, Lam, Sethi, Ullman. "Compilers: Principles, Techniques and Tools", 2nd edition**. (Aho2) The **1st edition** (by Aho, Sethi, Ullman – Aho1), the "Dragon Book", has been a classic for over 20 years.
- **Cooper & Torczon. "Engineering a Compiler"** – an earlier draft has been consulted when preparing this module. The 2nd edition is now available and being assessed (pointers will be provided to the 1st and hopefully to the 2nd edition).
- Other books:
  - Hunter *et al*. "The essence of Compilers" (Prentice-Hall)
  - Grune *et al*. "Modern Compiler Design" (Wiley)

Syllabus (11 weeks)

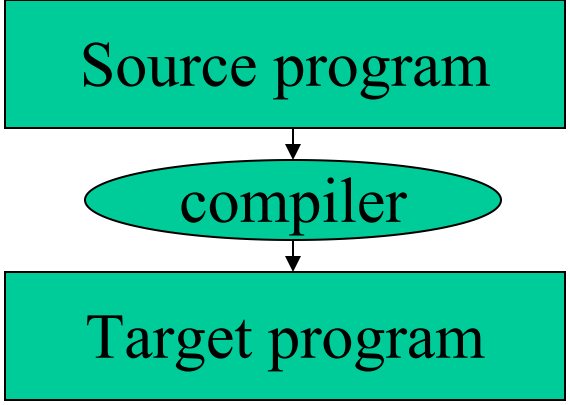
- 2 Introduction
- 3 Lexical Analysis (scanning)
- 1 – Exercises (on your own – optional)
- 3 Syntax Analysis (parsing)
- 1 – Exercises (on your own – optional)
- 1 Semantic Analysis
- 1 Intermediate Representations
- 1 Storage Management
- 1 Exercises (on your own – optional)
- 4 Code Generation
- 1 Code Optimisation
- 3 Exam preparation, exercises - Conclusion

Definitions

(compile: collect material into a list, volume)

- What is a compiler?
  - A program that accepts as input a program text in a certain language and produces as output a program text in another language, while preserving the meaning of that text (Grune *et al*, 2000).
  - A program that reads a program written in one language (source language) and translates it into an equivalent program in another language (target language) (Aho *et al*)
- What is an interpreter?
  - A program that reads a source program and produces the results of executing this source.
- *We deal with compilers! Many of these issues arise with interpreters!*

Examples



- C is typically compiled
- Lisp is typically interpreted
- Java is compiled to bytecodes, which are then interpreted

Also:

- C++ to Intel Core 2/.../Assembly
- C++ to C
- High Performance Fortran (HPF) to Fortran (parallelising compiler)
- C to C (or any language to itself)

In the general sense:

- What is LaTeX?
- What is ghostview? (PostScript is a language for describing images)

Qualities of a Good Compiler

What qualities would you want in a compiler?

- generates correct code (first and foremost!)
- generates fast code
- conforms to the specifications of the input language
- copes with essentially arbitrary input size, variables, etc.
- compilation time (linearly)proportional to size of source
- good diagnostics
- consistent optimisations
- works well with the debugger

Principles of Compilation

The compiler must:

- *preserve the meaning of the program being compiled.*
- *"improve" the source code in some way.*

Other issues (depending on the setting):

- Speed (of compiled code)
- Space (size of compiled code)
- Feedback (information provided to the user)
- Debugging (transformations obscure the relationship source code vs target)
- Compilation time efficiency (fast or slow compiler?)

Why study Compilation Technology?

- Success stories (one of the earliest branches in CS)
  - Applying theory to practice (scanning, parsing, static analysis)
  - Many practical applications have embedded languages (eg, tags)
- Practical algorithmic & engineering issues:
  - Approximating really hard (and interesting!) problems
  - Emphasis on efficiency and scalability
  - Small issues can be important!
- Ideas from different parts of computer science are involved:
    - AI: Heuristic search techniques; greedy algorithms - Algorithms: graph algorithms - Theory: pattern matching - Also: Systems, Architecture
- Compiler construction can be challenging and fun:
  - new architectures always create new challenges; success requires mastery of complex interactions; results are useful; opportunity to achieve performance.

Uses of Compiler Technology

- Most common use: translate a high-level program to object code
  - Program Translation: binary translation, hardware synthesis, …
- Optimizations for computer architectures:
  - Improve program performance, take into account hardware parallelism, etc…
- Automatic parallelisation or vectorisation
- Performance instrumentation: e.g., -pg option of cc or gcc
- Interpreters: e.g., Python, Ruby, Perl, Matlab, sh, …
- Software productivity tools
  - Debugging aids: e.g, purify
- Security: Java VM uses compiler analysis to prove "safety" of Java code.
- Text formatters, just-in-time compilation for Java, power management, global distributed computing, …

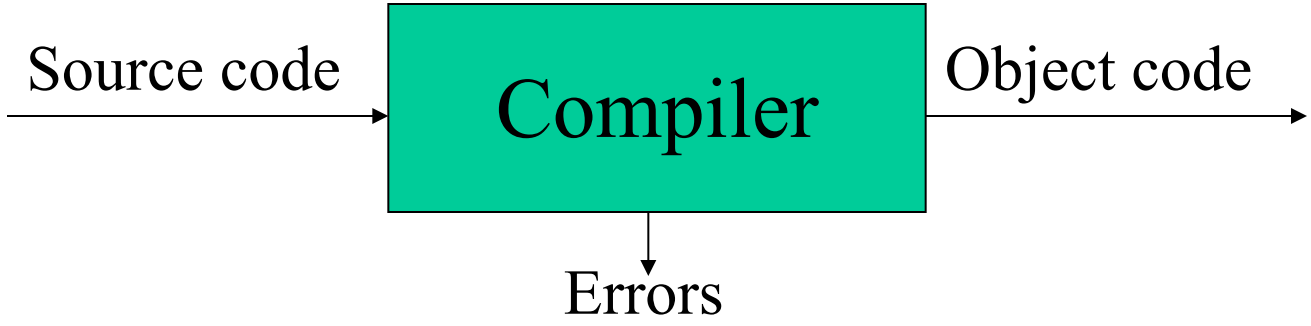
Key: Ability to extract properties of a source program (analysis) and transform it to construct a target program (synthesis)

Summary

- A compiler is a program that converts some input text in a source language to output in a target language.
- Compiler construction poses some of the most challenging problems in computer science.
- Reading:
  - Aho2, 1.1, 1.5; Aho1 1.1; Cooper1 1.1-1.3;
  - Grune 1.1, 1.5

- Next lecture: structure of a typical compiler.

Lecture 2: General Structure of a Compiler



(from last lecture) The compiler:-

- must generate correct code.
- must recognise errors.
- analyses and synthesises.

In today's lecture:

more details about the compiler's structure.

Conceptual Structure:two major phases

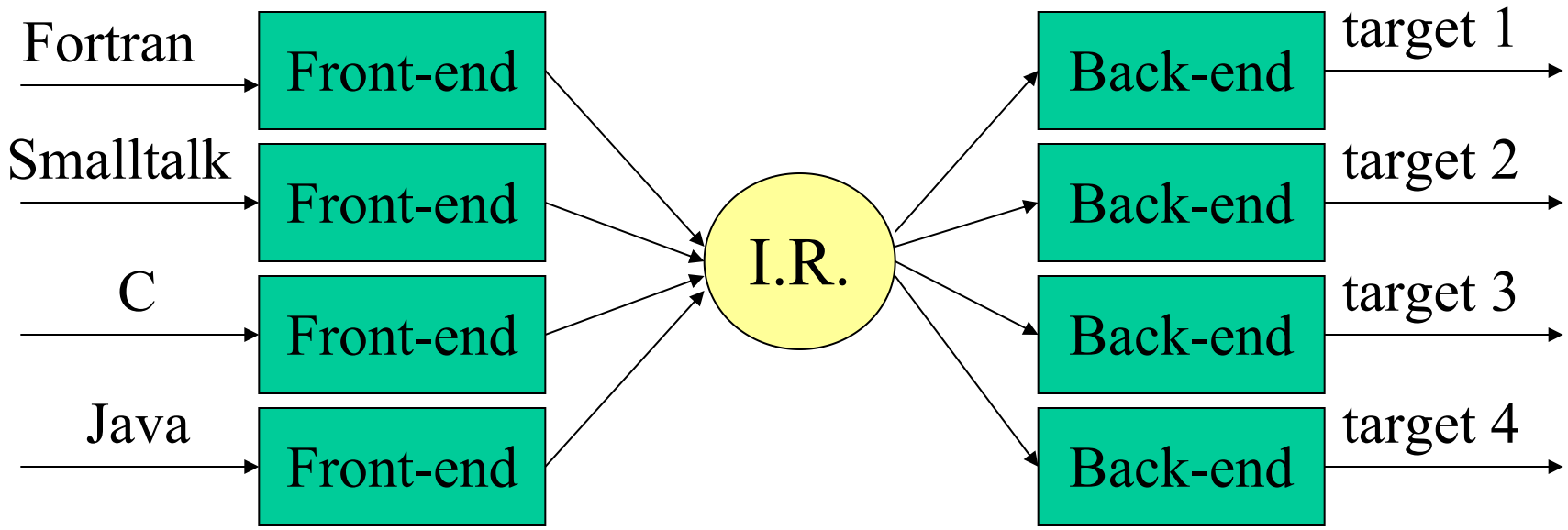
- **Front-end** performs the **analysis** of the source language:
  - Recognises legal and illegal programs and reports errors.
  - "understands" the input program and collects its semantics in an IR.
  - Produces IR and shapes the code for the back-end.
  - Much can be automated.
- **Back-end** does the target language **synthesis**:
  - Chooses instructions to implement each IR operation.
  - Translates IR into target code.
  - Needs to conform with system interfaces.
  - Automation has been less successful.

A problem which we don't know how to solve in less than exponential time

- Typically front-end is **O(n),** while back-end is **NP-complete**.

What is the implication of this separation (front-end: analysis; back-end:synthesis) in building a compiler for, say, a new language?

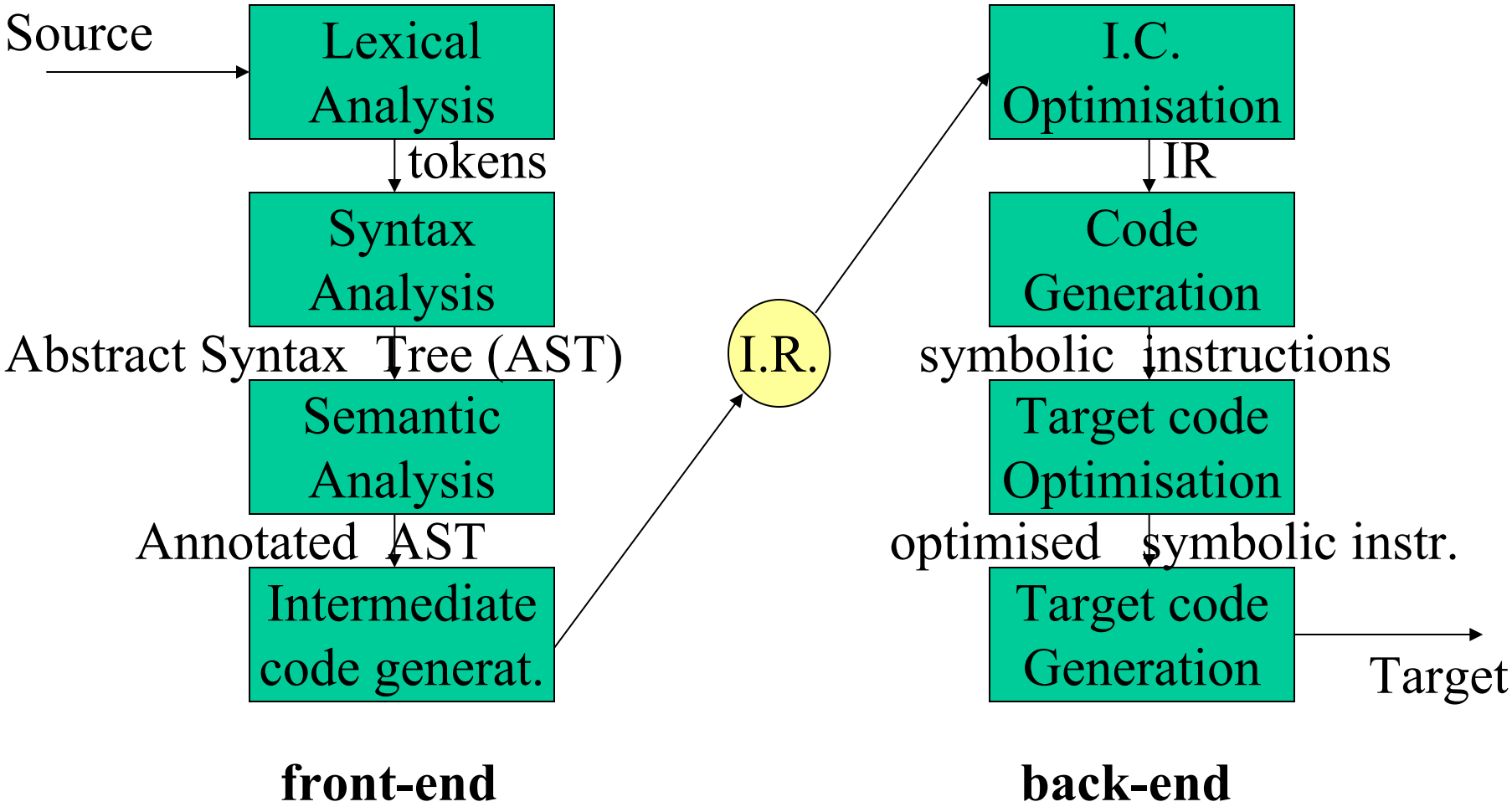
m×n compilers with m+n components!



- All language specific knowledge must be encoded in the front-end
- All target specific knowledge must be encoded in the back-end

But: in practice, this strict separation is not free of charge.

General Structure of a compiler



Lexical Analysis (Scanning)

- Reads characters in the source program and groups them into words (basic unit of syntax)
- Produces words and recognises what sort they are.
- The output is called token and is a pair of the form <*type, lexeme*> or <*token_class, attribute*>
- E.g.: **`a=b+c`** becomes <id,**`a`**> <=,> <id,**`b`**> <+,> <id,**`c`**>
- Needs to record each id attribute: keep a **symbol table**.
- Lexical analysis eliminates white space, etc…
- Speed is important - use a specialised tool: e.g., flex - a tool for generating **scanners**: programs which recognise lexical patterns in text; for more info: **`% man flex`**

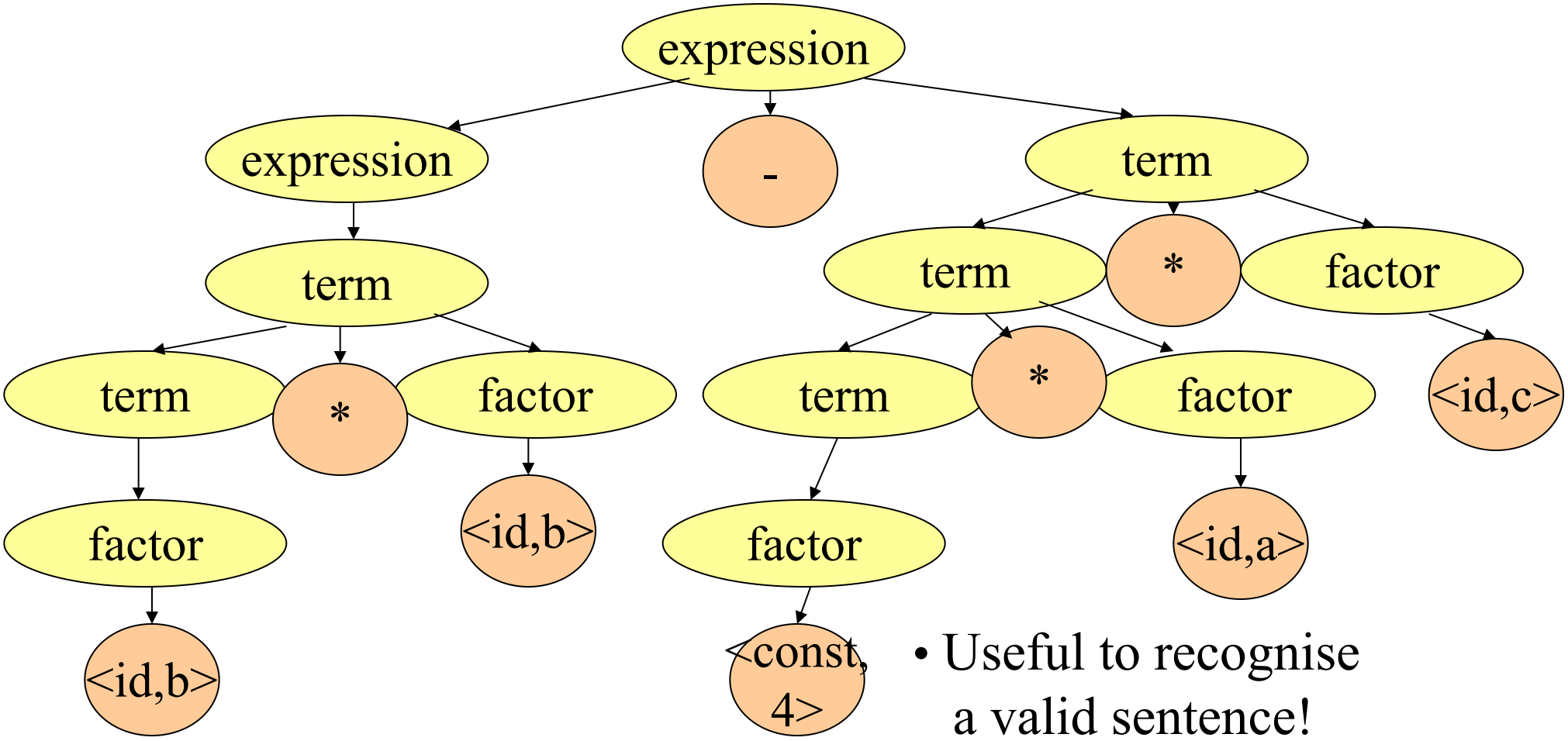
Syntax (or syntactic) Analysis (Parsing)

- Imposes a hierarchical structure on the token stream.
- This hierarchical structure is usually expressed by recursive rules.
- Context-free grammars formalise these recursive rules and guide syntax analysis.
- Example:

```
expression → expression '+' term | expression '-' term | term
term → term '*' factor | term '/' factor | factor
factor → identifier | constant | '(' expression ')'
```

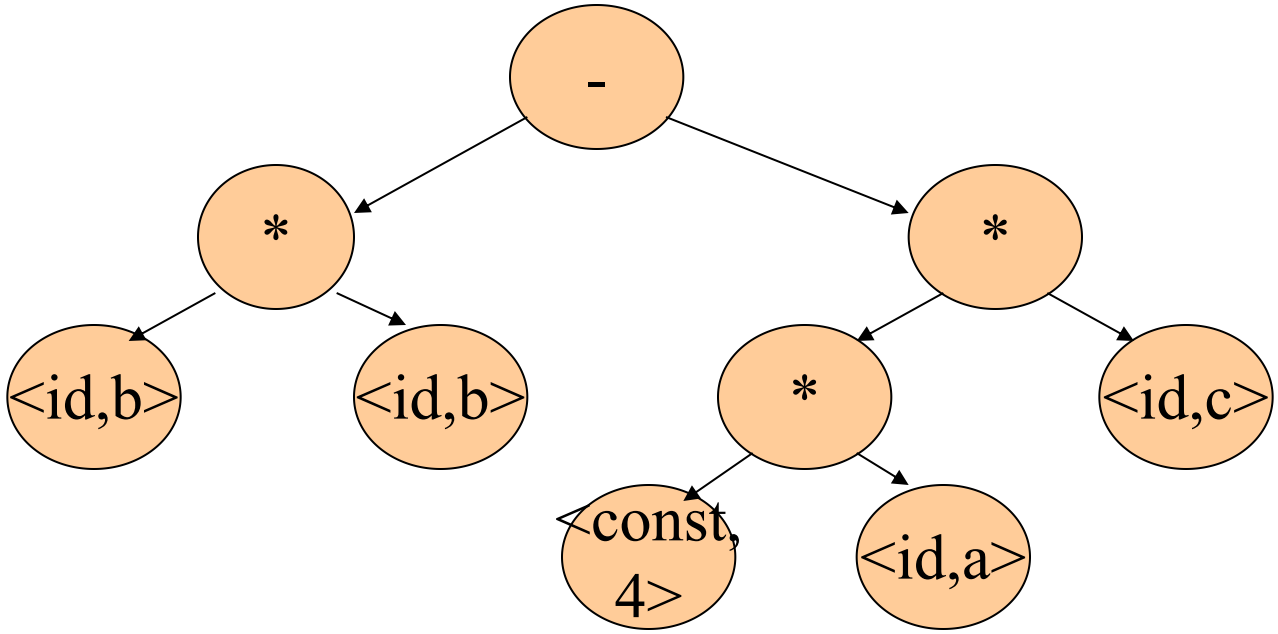
(this grammar defines simple algebraic expressions)

Parsing: parse tree for *b*b-4*a*c*



- Useful to recognise a valid sentence!
- Contains a lot of unneeded information!

AST for *b*b-4*a*c*



- An Abstract Syntax Tree (AST) is a more useful data structure for internal representation. It is a compressed version of the parse tree (summary of grammatical structure without details about its derivation)
- ASTs are one form of IR

Semantic Analysis (context handling)

- Collects context (semantic) information, checks for semantic errors, and annotates nodes of the tree with the results.
- Examples:
  - type checking: report error if an operator is applied to an incompatible operand.
  - check flow-of-controls.
  - uniqueness or name-related checks.

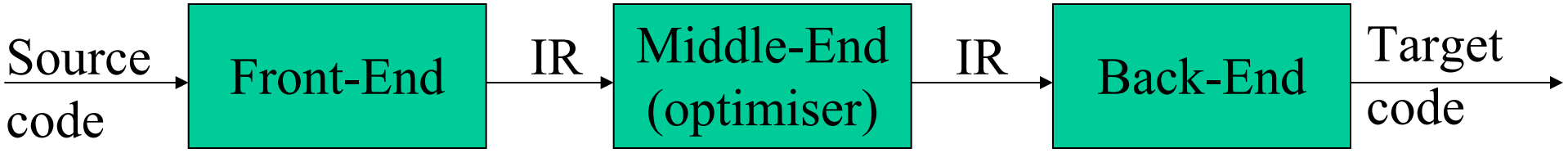
Intermediate code generation

- Translate language-specific constructs in the AST into more general constructs.
- A criterion for the level of "generality": it should be straightforward to generate the target code from the intermediate representation chosen.
- Example of a form of IR (3-address code):

```
tmp1=4
tmp2=tmp1*a
tmp3=tmp2*c
tmp4=b*b
tmp5=tmp4-tmp3
```

Code Optimisation

- The goal is to improve the intermediate code and, thus, the effectiveness of code generation and the performance of the target code.
- Optimisations can range from trivial (e.g. constant folding) to highly sophisticated (e.g, in-lining).
- For example: replace the first two statements in the example of the previous slide with: **`tmp2=4*a`**
- Modern compilers perform such a range of optimisations, that one could argue for:



Code Generation Phase

- Map the AST onto a linear list of target machine instructions in a symbolic form:
  - Instruction selection: a pattern matching problem.
  - Register allocation: each value should be in a register when it is used (but there is only a limited number): NP-Complete problem.
  - Instruction scheduling: take advantage of multiple functional units: NP-Complete problem.
- Target, machine-specific properties may be used to optimise the code.
- Finally, machine code and associated information required by the Operating System are generated.

Some historical notes...

Emphasis of compiler construction research:

- 1945-1960: code generation
  - need to "prove" that high-level programming can produce efficient code ("automatic programming").
- 1960-1975: parsing
  - proliferation of programming languages
  - study of formal languages reveals powerful techniques.
- 1975-...: code generation and code optimisation

Knuth (1962) observed that "*in this field there has been an unusual amount of parallel discovery of the same technique by people working independently*"

Historical Notes:
the Move to Higher-Level Programming Languages

- Machine Languages (1st generation)
- Assembly Languages (2nd generation) – early 1950s
- High-Level Languages (3rd generation) – later 1950s
- 4th generation higher level languages (SQL, Postscript)
- 5th generation languages (logic based, eg, Prolog)
- Other classifications:
  - Imperative (how); declarative (what)
  - Object-oriented languages
  - Scripting languages

See exercise 1.3.1 in Aho2

Finally...

Parts of a compiler can be generated automatically using generators based on formalisms. E.g.:

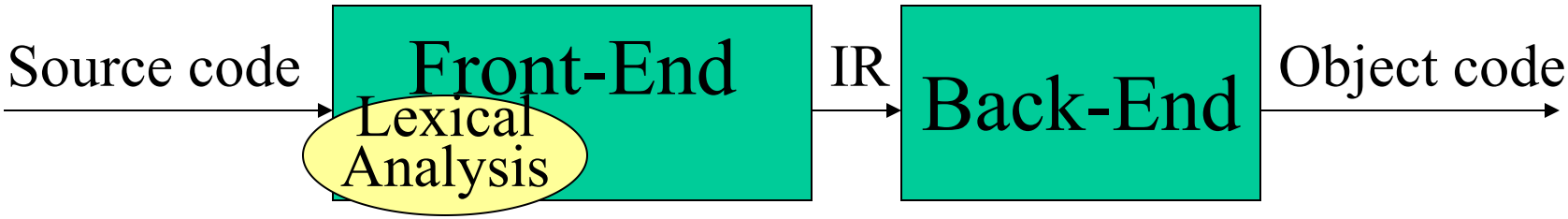
- Scanner generators: flex
- Parser generators: bison

Summary: the structure of a typical compiler was described.

Next time: Introduction to lexical analysis.

Reading: Aho2, Sections 1.2, 1.3; Aho1, pp. 1-24; Hunter, pp. 1-15 (try the exercises); Grune [rest of Chapter 1 up to Section 1.8] (try the exercises); Cooper & Torczon (1st edition), Sections 1.4, 1.5.

Lecture 3: Introduction to Lexical Analysis



(from last lecture) Lexical Analysis:

- reads characters and produces sequences of tokens.

Today's lecture:

Towards automated Lexical Analysis.

The Big Picture

First step in any translation: determine whether the text to be translated is well constructed in terms of the input language. Syntax is specified with parts of speech - syntax checking matches parts of speech against a grammar.

In natural languages, mapping words to part of speech is idiosyncratic.

In formal languages, mapping words to part of speech is syntactic:

- based on denotation
- makes this a matter of syntax
- reserved keywords are important

What does lexical analysis do?

Recognises the language's parts of speech.

Some Definitions

- A vocabulary (alphabet) is a finite set of symbols.
- A string is any finite sequence of symbols from a vocabulary.
- A language is any set of strings over a fixed vocabulary.
- A grammar is a finite way of describing a language.
- A context-free grammar, *G*, is a 4-tuple, *G=(S,N,T,P)*, where:
  - *S*: starting symbol
  - *N*: set of non-terminal symbols
  - *T*: set of terminal symbols
  - *P*: set of production rules
- A language is the set of all terminal productions of *G*.
- Example (thanks to Keith Cooper for inspiration):
  - *S*=CatWord; *N*={CatWord}; *T*={miau};
  - *P*={CatWord $\rightarrow$ CatWord miau | miau}

Example

(A simplified version from Lecture2, Slide 6):

$S=E;\ N=\{E,T,F\};\ T=\{+,*,(,),x\}$

$P=\{E \rightarrow T|E+T,\ T \rightarrow F|T*F,\ F \rightarrow (E)|x\}$

By repeated substitution we derive *sentential forms*:

$\underline{E} \Rightarrow \underline{E}+T \Rightarrow \underline{T}+T \Rightarrow \underline{F}+T \Rightarrow x+\underline{T} \Rightarrow x+\underline{T}*F \Rightarrow x+\underline{F}*F$
$\Rightarrow x+x*\underline{F} \Rightarrow x+x*x$

This is an example of a *leftmost derivation* (at each step the leftmost non-terminal is expanded).

To recognise a valid sentence we reverse this process.

- Exercise: what language is generated by the (non-context free) grammar:
  $S=S;\ N=\{A,B,S\};\ T=\{a,b,c\};$
  $P=\{S \rightarrow abc|aAbc,\ Ab \rightarrow bA,\ Ac \rightarrow Bbcc,\ bB \rightarrow Bb,\ aB \rightarrow aa|aaA\}$
  (for the curious: read about Chomsky's Hierarchy)

Why all this?

- Why study lexical analysis?
  - To avoid writing lexical analysers (scanners) by hand.
  - To simplify specification and implementation.
  - To understand the underlying techniques and technologies.
- We want to specify **lexical patterns** (to derive tokens):
  - Some parts are easy:
    - *WhiteSpace* $\rightarrow$ *blank* | *tab* | *combination_of_blank_and_tab*
    - Keywords and operators (if, then, =, +)
    - Comments (/* followed by */ in C, // in C++, % in latex, ...)
  - Some parts are more complex:
    - Identifiers (letter followed by - up to *n* - alphanumerics…)
    - Numbers

We need a notation that could lead to an implementation!

Regular Expressions

Patterns form a regular language. A regular expression is a way of specifying a regular language. It is a formula that describes a possibly infinite set of strings.

(*Have you ever tried* **`ls [x-z]*`** *?*)

Regular Expression (RE) (over a vocabulary V):

- ε is a RE denoting the empty set $\{\varepsilon\}$.
- If $a \in$ V then a is a RE denoting $\{a\}$.
- If r_1, r_2 are REs then:
  - r_1* denotes zero or more occurrences of r_1;
  - $r_1 r_2$ denotes concatenation;
  - $r_1 \mid r_2$ denotes either r_1 or r_2;
- **Shorthands**: *[a-d]* for $a \mid b \mid c \mid d$; r^+ for rr*; *r?* for $r \mid \varepsilon$

Describe the languages denoted by the following REs

a; *a* | *b*; *a**; *(a | b)**; *(a | b)(a | b)*; *(a*b*)**; (*a* | *b*)**baa*;

(*What about* **`ls [x-z]*`** *above? Hmm… not a good example?*)

Examples

- *integer* → *(+ | – | ε) (0 | 1 | 2 | ... | 9)+*
- *integer* → *(+ | – | ε) (0 | 1 | 2 | ... | 9) (0 | 1 | 2 | ... | 9)**
- *decimal* → *integer.(0 | 1 | 2 | ... | 9)**
- *identifier* → *[a-zA-Z] [a-zA-Z0-9]**

- Real-life application (perl regular expressions):
  - `[+-]?(\d+\.\d+|\d+\.|\.\d+)`
  - `[+-]?(\d+\.\d+|\d+\.|\.\d+|\d+)([eE][+-]?\d+)?`

    (for more information read: `% man perlre`)

(Not all languages can be described by regular expressions. But, we don't care for now).

Building a Lexical Analyser by hand

Based on the specifications of tokens through regular expressions we can write a lexical analyser. One approach is to check case by case and split into smaller problems that can be solved *ad hoc*. Example:

```
void get_next_token() {
   c=input_char();
   if (is_eof(c)) { token ← (EOF,"eof"); return}
   if (is_letter(c)) {recognise_id()}
   else if (is_digit(c)) {recognise_number()}
        else if (is_operator(c))||is_separator(c))
             {token ← (c,c)}  //single char assumed
             else {token ← (ERROR,c)}
   return;
}
...
do {
   get_next_token();
   print(token.class, token.attribute);
} while (token.class != EOF);
```

Can be efficient; but requires a lot of work and may be difficult to modify!

Building Lexical Analysers "automatically"

Idea: try the regular expressions one by one and find the longest match:

```
set (token.class, token.length) ←(NULL, 0)
// first
find max_length such that input matches T1→RE1
  if max_length > token.length
      set (token.class, token.length) ←(T1, max_length)
// second
find max_length such that input matches T2→RE2
  if max_length > token.length
      set (token.class, token.length) ←(T2, max_length)
…
// n-th
find max_length such that input matches Tn→REn
  if max_length > token.length
      set (token.class, token.length) ←(Tn, max_length)
// error
if (token.class == NULL) { handle no_match }
```

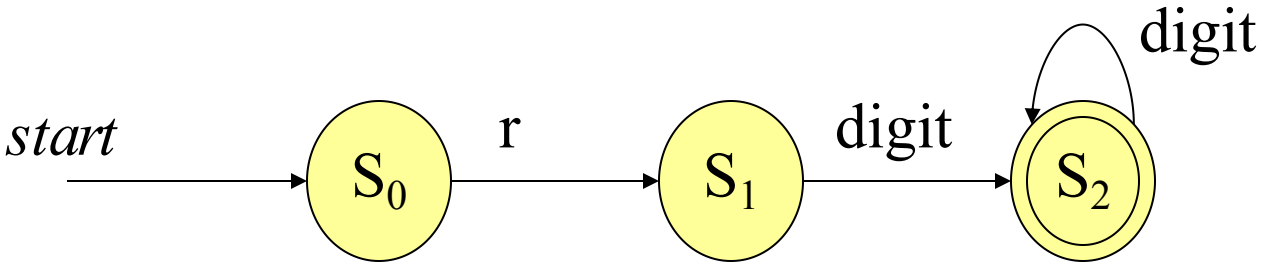
Disadvantage: linearly dependent on number of token classes and requires restarting the search for each regular expression.

We study REs to **automate** scanner construction!

Consider the problem of recognising register names starting with r and requiring at least one digit:

Register → *r (0|1|2|...|9) (0|1|2|...|9)** (or, *Register* → *r Digit Digit**)

The RE corresponds to a **transition diagram**:



Depicts the actions that take place in the scanner.

- A circle represents a state; S0: start state; S2: final state (double circle)
- An arrow represents a transition; the label specifies the cause of the transition.

A string is accepted if, going through the transitions, ends in a final state (for example, r345, r0, r29, as opposed to a, r, rab)

Towards Automation (finally!)

An easy (computerised) implementation of a transition diagram is a **transition table**: a column for each input symbol and a row for each state. An entry is a set of states that can be reached from a state on some input symbol. E.g.:

state	'r'	digit
0	1	-
1	-	2
2(final)	-	2

If we know the transition table and the final state(s) we can build directly a recogniser that detects acceptance:

```
char=input_char();
state=0;    // starting state
while (char != EOF) {
    state ← table(state,char);
    if (state == '-') return failure;
    word=word+char;
    char=input_char();
}
if (state == FINAL) return acceptance; else return failure;
```

The Full Story!

The generalised transition diagram is a **finite automaton**. It can be:

- **Deterministic**, DFA; as in the example
- **Non-Deterministic**, NFA; more than 1 transition out of a state may be possible on the same input symbol: think about: *(a | b)* abb*

Every regular expression can be converted to a DFA !

Summary: an introduction to lexical analysis was given.

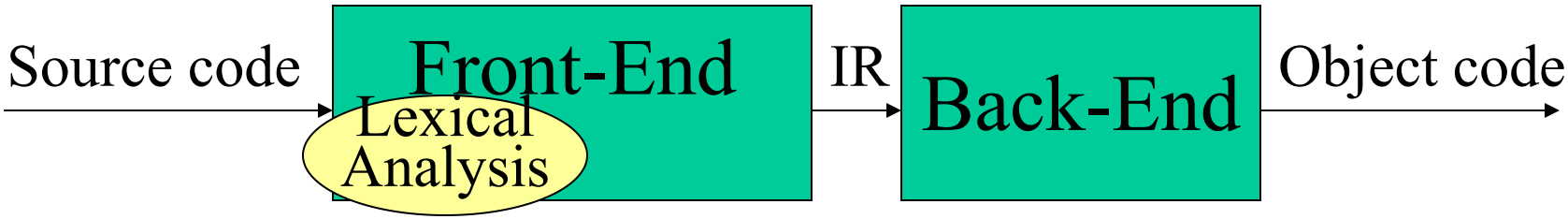
Next time: More on finite automata and conversions.

Exercise: Produce the DFA for the RE (Q: what is it for?):

Register $\rightarrow$ *r ((0|1|2) (Digit|ε) | (4|5|6|7|8|9) | (3|30|31))*

Reading: Aho2, Sections 2.2, 3.1-3.4. Aho1, pp. 25-29; 84-87; 92-105. Hunter, Chapter 2 (too detailed); Sec. 3.1 -3.3 (too condensed). Grune 1.9; 2.1-2.5. Cooper, Sections 2.1-2.3

Lecture 4: Lexical Analysis II: From REs to DFAs



(from last lecture) Lexical Analysis:

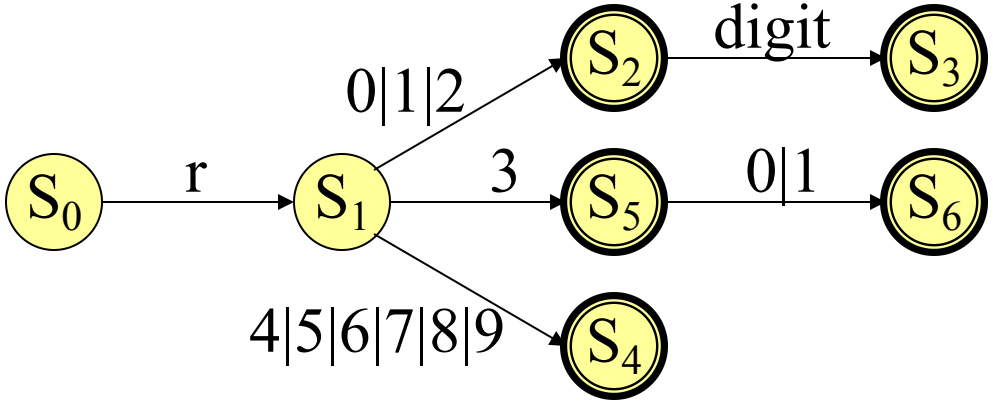
- Regular Expressions (REs) are formulae to describe a (regular) language.
- Every RE can be converted to a Deterministic Finite Automaton (DFA).
- DFAs can automate the construction of lexical analysers.

Today's lecture:

Algorithms to derive a DFA from a RE.

An Example (recognise r0 through r31)

Register → *r ((0|1|2) (Digit|ε) | (4|5|6|7|8|9) | (3|30|31))*

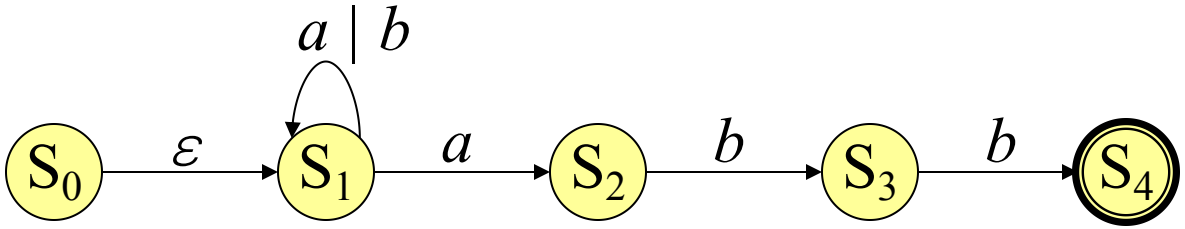


State	'r'	0,1	2	3	4,5,…,9
0	1	-	-	-	-
1	-	2	2	5	4
2(final)	-	3	3	3	3
3(final)	-	-	-	-	-
4(final)	-	-	-	-	-
5(final)	-	6	-	-	-
6(final)	-	-	-	-	-

- Same code skeleton (Lecture 3, slide 11) can be used!
- Different (bigger) transition table.
- Our Deterministic Finite Automaton (DFA) recognises only r0 through r31.

Non-deterministic Finite Automata

*What about a RE such as (a | b)*abb?*



- This is a Non-deterministic Finite Automaton (NFA):
  - S_0 has a transition on ε; S_1 has two transitions on *a* (not possible for a DFA).
- A DFA is a special case of an NFA:
  - for each state and each transition there is at most one rule.
- A DFA can be simulated with an NFA (obvious!)
- A NFA can be simulated with a DFA (less obvious).
  - Simulate sets of possible states.

Why study NFAs? DFAs can lead to faster recognisers than NFAs but may be much bigger. Converting a RE into an NFA is more direct.

The Big Picture:
Automatic Lexical Analyser Construction

To convert a specification into code:

- Write down the RE for the input language.
- Convert the RE to a NFA (Thompson's construction)
- Build the DFA that simulates the NFA (subset construction)
- Shrink the DFA (Hopcroft's algorithm)

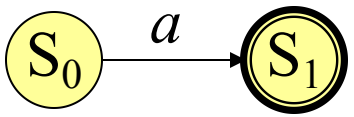
(for the curious: there is a full cycle - DFA to RE construction is all pairs, all paths)

Lexical analyser generators:

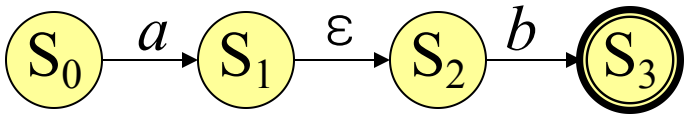
- lex or flex work along these lines.
- Algorithms are well-known and understood.
- Key issue is the interface to parser.

RE to NFA using Thompson's construction

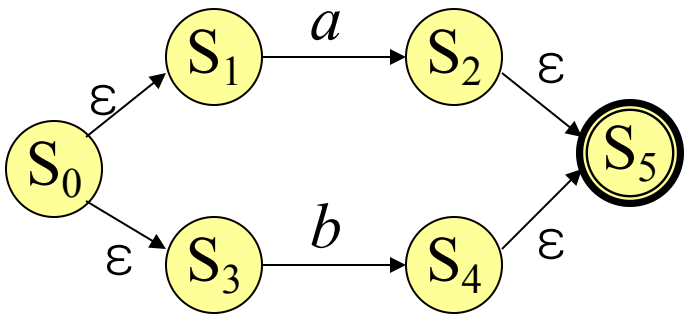
Key idea (Ken Thompson; CACM, 1968): NFA pattern for each symbol and/or operator: join them in precedence order.



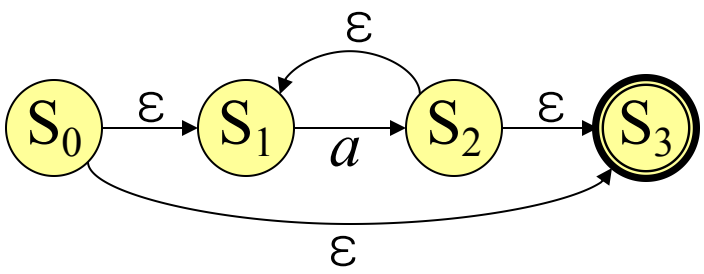
NFA for *a*



NFA for *ab*



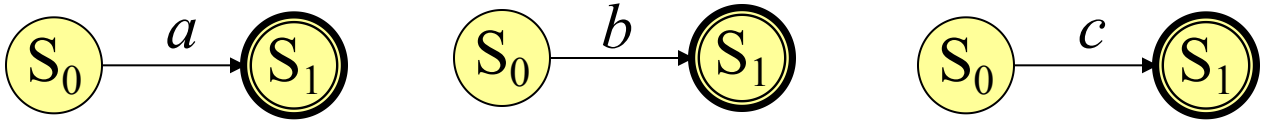
NFA for *a* | *b*

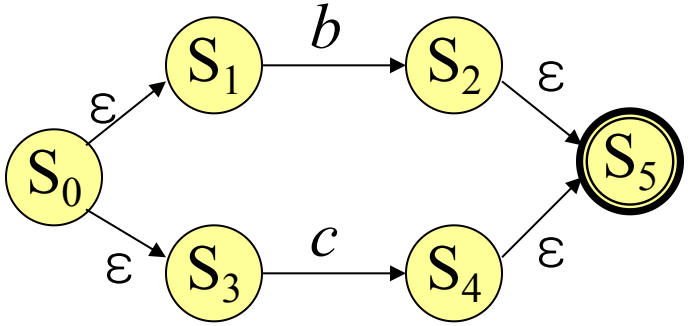


NFA for *a**

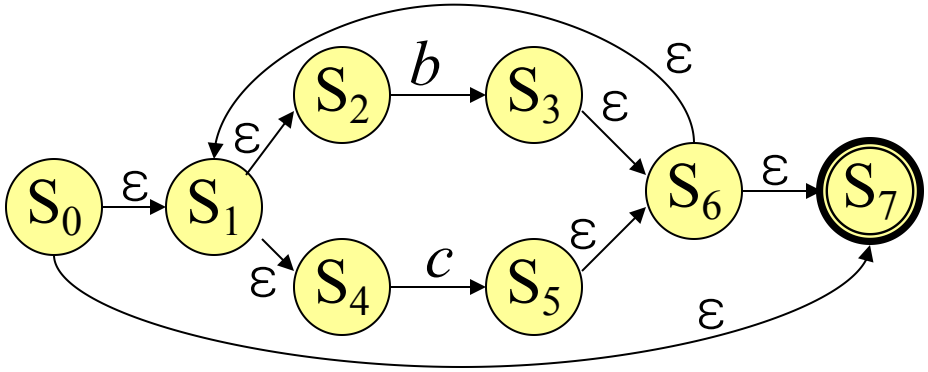
Example: Construct the NFA of *a (b|c)**

First: NFAs for *a, b, c*



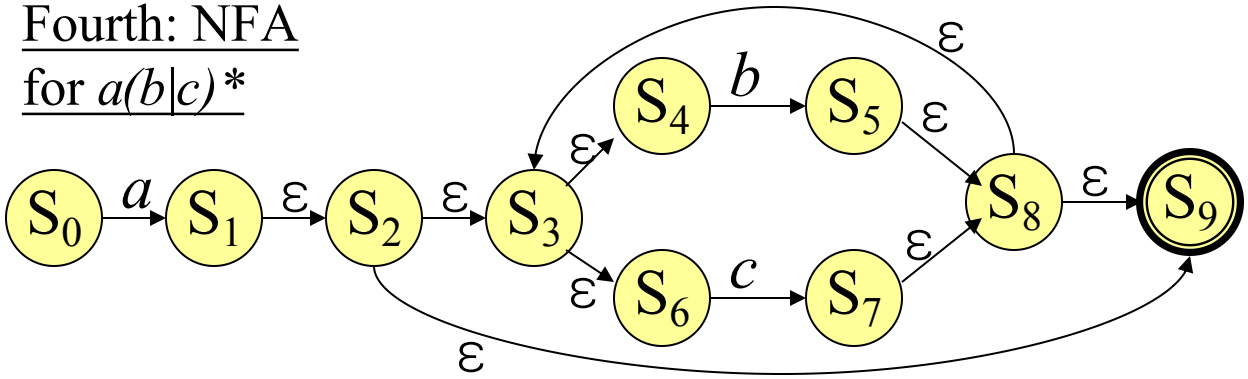


Second: NFA for *b|c*

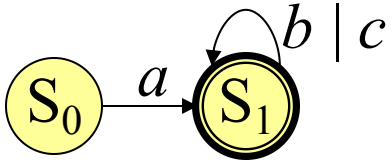


Third: NFA for *(b|c)**

Fourth: NFA for *a(b|c)**



Of course, a human would design a simpler one… But, we can automate production of the complex one...

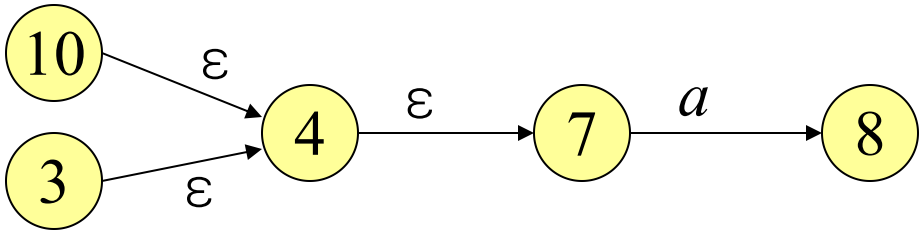


NFA to DFA: two key functions

- **move(s_i,a):** the (union of the) set of states to which there is a transition on input symbol **a** from state $\mathbf{s_i}$
- **ε-closure(s_i):** the (union of the) set of states reachable by **ε** from $\mathbf{s_i}$.

Example (see the diagram below):

- ε-closure(3)={3,4,7}; ε-closure({3,10})={3,4,7,10};
- move(ε-closure({3,10}),*a*)=8;



The Algorithm:

- start with the ε-closure of s_0 from NFA.
- Do for each unmarked state until there are no unmarked states:
  - for each symbol take their ε-closure(move(state,symbol))

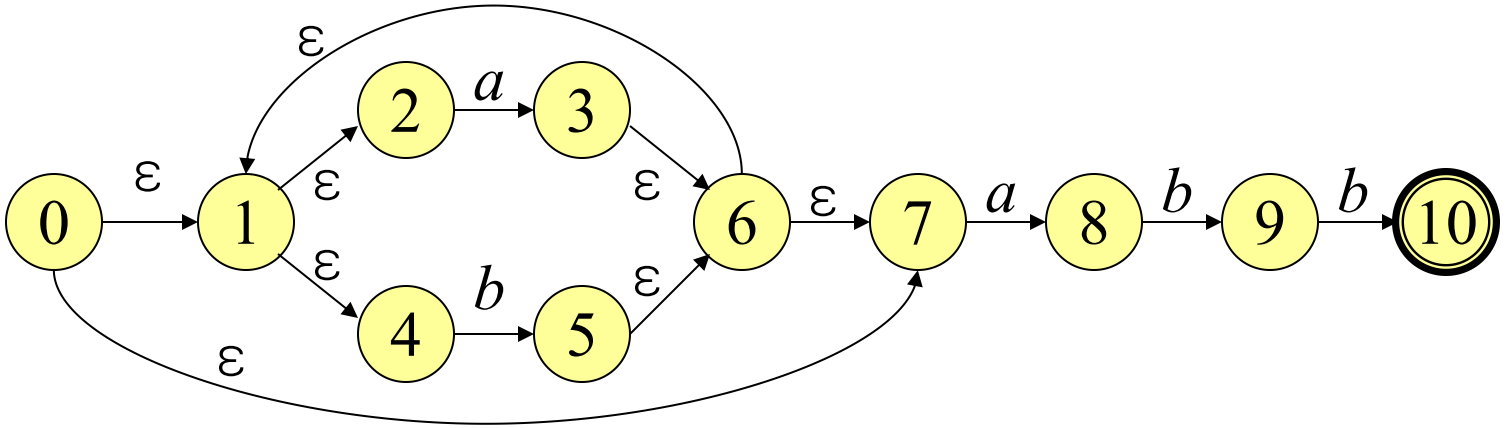
NFA to DFA with subset construction

Initially, ε-closure is the only state in Dstates and it is unmarked.
while there is an unmarked state T in Dstates
    mark T
    for each input symbol a
        U:=ε-closure(move(T,a))
        if U is not in Dstates then add U as unmarked to Dstates
        Dtable[T,a]:=U

- Dstates (set of states for DFA) and Dtable form the DFA.
- Each state of DFA corresponds to a set of NFA states that NFA could be in after reading some sequences of input symbols.
- This is a fixed-point computation.

It sounds more complex than it actually is!

Example: NFA for *(a | b)*abb*

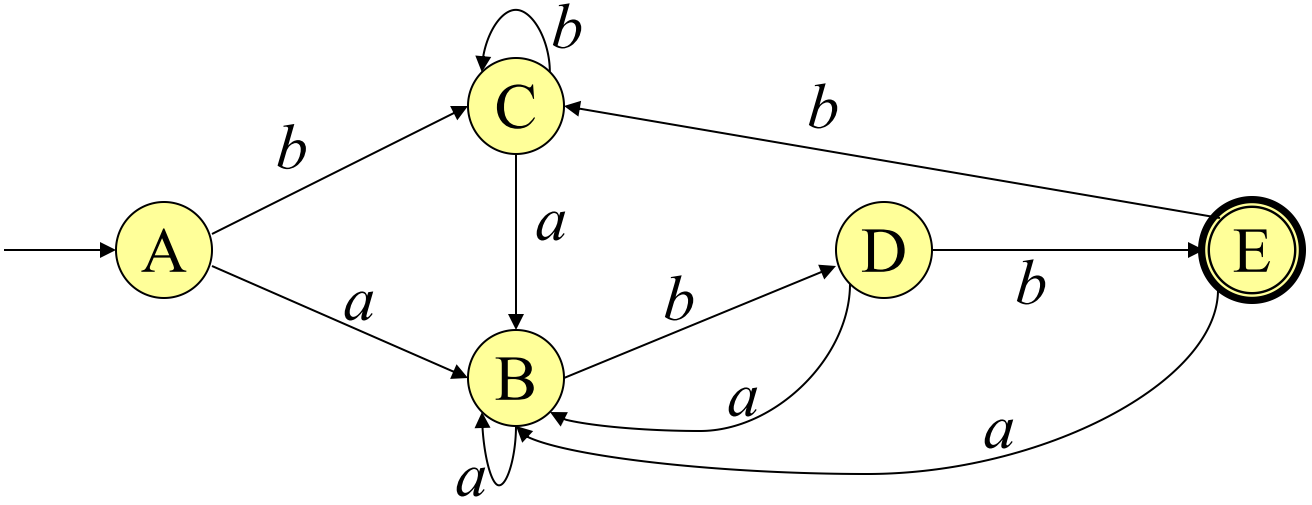


- A=ε-closure(0)={0,1,2,4,7}
- for each input symbol (that is, *a* and *b*):
  - B=ε-closure(move(A,*a*))=ε-closure({3,8})={1,2,3,4,6,7,8}
  - C=ε-closure(move(A,*b*))=ε-closure({5})={1,2,4,5,6,7}
  - Dtable[A,*a*]=B; Dtable[A,*b*]=C
- B and C are unmarked. Repeating the above we end up with:
  - C={1,2,4,5,6,7}; D={1,2,4,5,6,7,9}; E={1,2,4,5,6,7,10}; and
  - Dtable[B,*a*]=B; Dtable[B,*b*]=D; Dtable[C,*a*]=B; Dtable[C,*b*]=C; Dtable[D,*a*]=B; Dtable[D,*b*]=E; Dtable[E,*a*]=B; Dtable[E,*b*]=C; no more unmarked sets at this point!

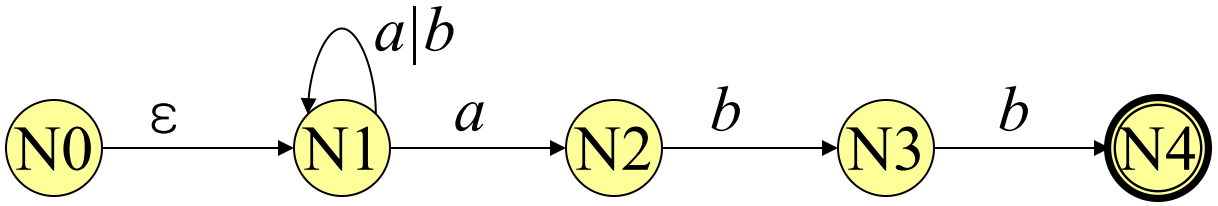
Result of applying subset construction

Transition table:

state	*a*	*b*
A	B	C
B	B	D
C	B	C
D	B	E
E(final)	B	C



Another NFA version of the same RE



Apply the subset construction algorithm:

Iteration	State	Contains	ε-closure(move(s,*a*))	ε-closure(move(s,*b*))
0	A	N0,N1	N1,N2	N1
1	B	N1,N2	N1,N2	N1,N3
	C	N1	N1,N2	N1
2	D	N1,N3	N1,N2	N1,N4
3	E	N1,N4	N1,N2	N1

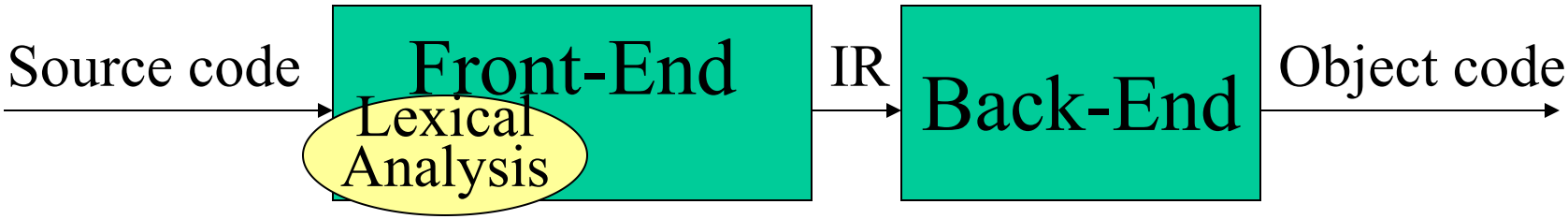
Note:

- iteration 3 adds nothing new, so the algorithm stops.
- state E contains N4 (final state)

Enough theory… Let's conclude!

- We presented algorithms to construct a DFA from a RE.
- The DFA is not necessarily the smallest possible.
- Using an (automatically generated) transition table and the standard code skeleton (Lecture 3, slide 11) we can build a lexical analyser from regular expressions automatically. But, the size of the table can be large...
- Next time:
  - DFA minimisation; Practical considerations; Lexical Analysis wrap-up.
- Reading: Aho2 Sections 3.6-3.7; Aho1 pp. 113-125; Grune 2.1.6.1-2.1.6.6 (different style); Hunter 3.3 (very condensed); Cooper1 2.4-2.4.3

Lecture 5: Lexical Analysis III: The final bits



(from the last 2 lectures) Lexical Analysis:

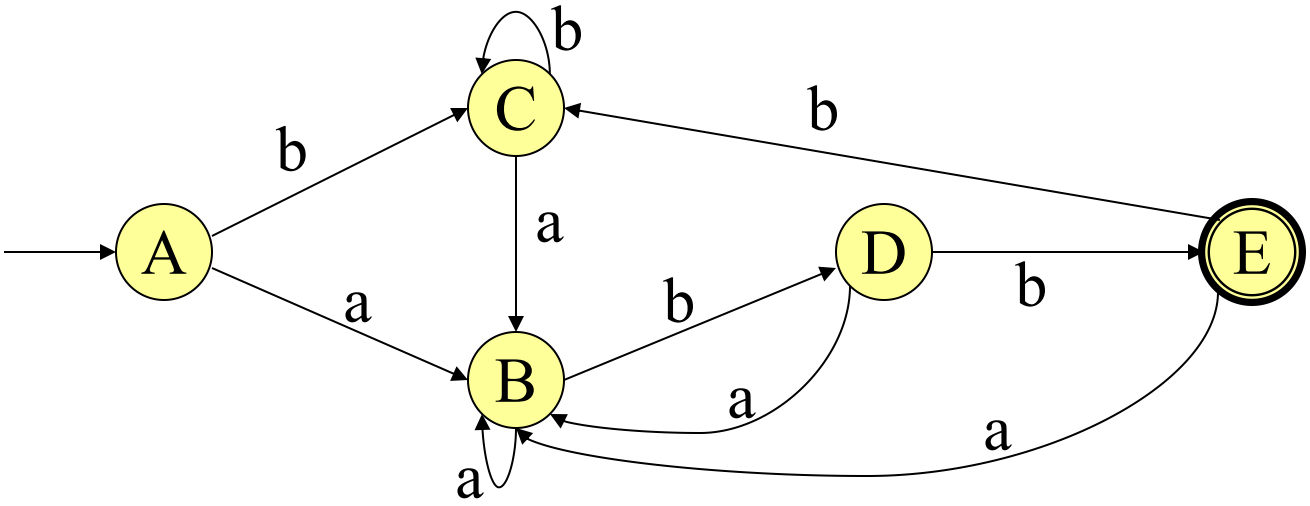
- Regular Expressions (REs) are formulae to describe a (regular) language.
- Every RE can be converted to a Deterministic Finite Automaton (DFA).
- DFAs can automate the construction of lexical analysers.
- Algorithms to construct a DFA from a RE (through a NFA) were given.

Today's lecture:

How to minimise the (resulting) DFA

Practical Considerations; lex; wrap-up

DFA Minimisation: the problem



- Problem: can we minimise the number of states?
- Answer: yes, if we can find groups of states where, for each input symbol, every state of such a group will have transitions to the same group.

DFA minimisation: the algorithm

(Hopcroft's algorithm: simple version)

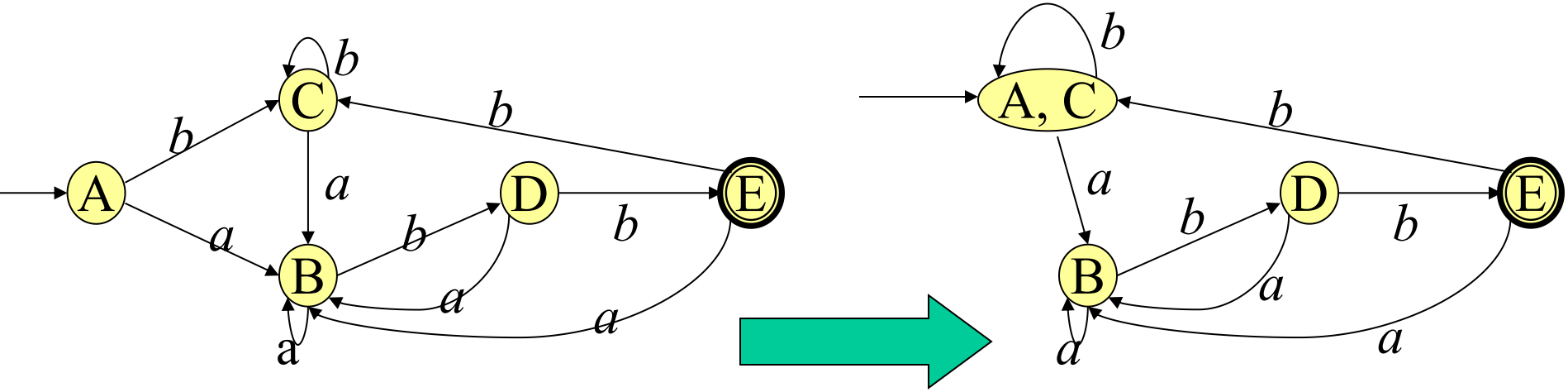
Divide the states of the DFA into two groups: those containing final states and those containing non-final states.
while there are group changes
    for each group
        for each input symbol
            if for any two states of the group and a given input symbol, their transitions do not lead to the same group, these states must belong to different groups.

For the curious, there is an alternative approach: create a graph in which there is an edge between each pair of states which cannot coexist in a group because of the conflict above. Then use a graph colouring algorithm to find the minimum number of colours needed so that any two nodes connected by an edge do not have the same colour (we'll examine graph colouring algorithms later on, in register allocation)

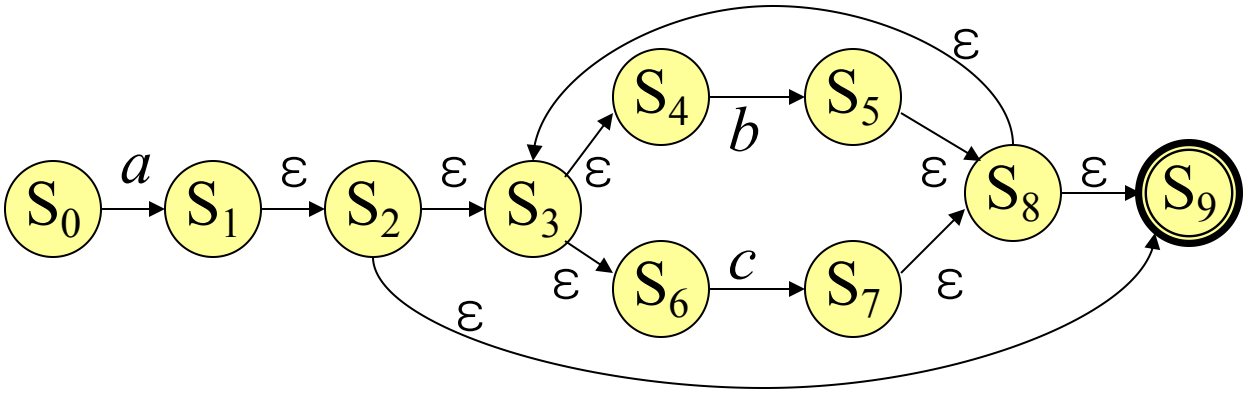
How does it work? Recall *(a | b)* abb*

Iteration	Current groups	Split on a	Split on b
0	{E}, {A,B,C,D}	None	{A,B,C}, {D}
1	{E}, {D}, {A,B,C}	None	{A,C}, {B}
2	{E}, {D}, {B}, {A, C}	None	None

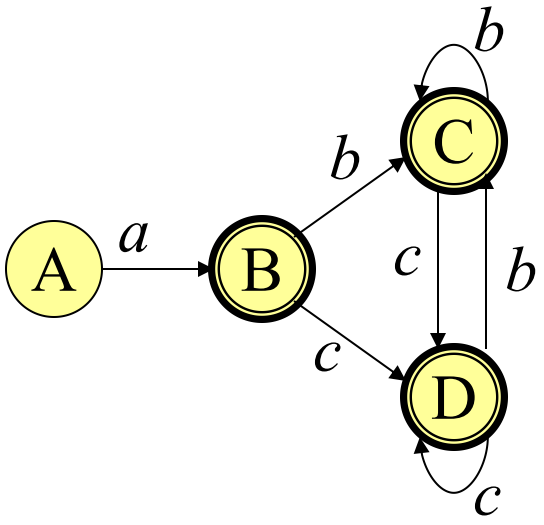
In each iteration, we consider any non-single-member groups and we consider the partitioning criterion for all pairs of states in the group.



From NFA to minimized DFA: recall *a (b | c)**



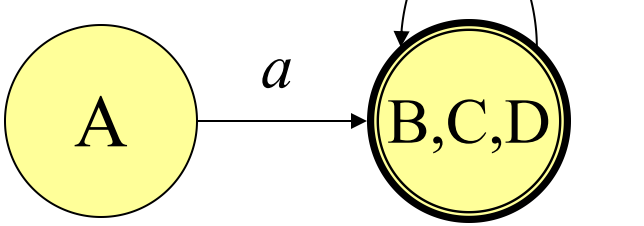
DFA states	NFA states	ε-closure(move(s,*))		
		a	*b*	*c*
A	S0	S1,S2,S3, S4,S6,S9	None	None
B	S1,S2,S3, S4,S6,S9	None	S5,S8,S9, S3,S4,S6	S7,S8,S9, S3,S4,S6
C	S5,S8,S9, S3,S4,S6	None	S5,S8,S9, S3,S4,S6	S7,S8,S9, S3,S4,S6
D	S7,S8,S9, S3,S4,S6	None	S5,S8,S9, S3,S4,S6	S7,S8,S9, S3,S4,S6

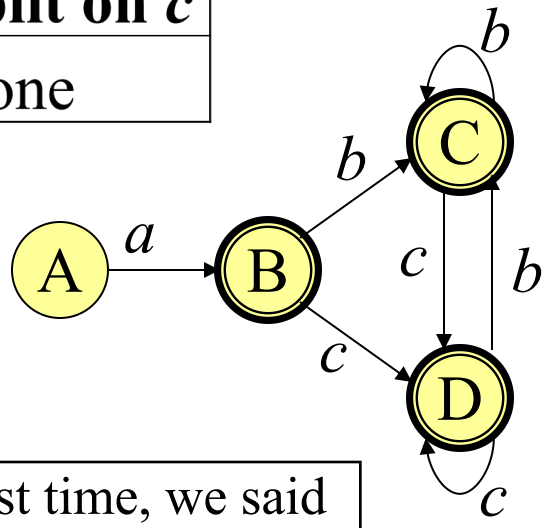


DFA minimisation: recall *a (b | c)**

Apply the minimisation algorithm to produce the minimal DFA:

	Current groups	**Split on *a***	**Split on *b***	**Split on *c***
0	{B, C, D} {A}	None	None	None





Remember, last time, we said that a human could construct a simpler automaton than Thompson's construction? Well, algorithms can produce the same DFA!

Building fast scanners...

From a RE to a DFA; why have we been doing all this?

- If we know the transition table and the final state(s) we can build directly a recogniser that detects acceptance (recall from Lecture 3):

```
char=input_char();
state=0;    // starting state
while (char != EOF) {
    state ← table(state,char);
    if (state == '-') return failure;
    word=word+char;
    char=input_char();
}
if (state == FINAL) return acceptance; else return failure;
```

But, table-driven recognisers can waste a lot of effort.

- Can we do better? Encode states and actions in the code.

How to deal with reserved keywords?

- Some compilers recognise keywords as identifiers and check them in a table.
- Simpler to include them as REs in the lexical analyser's specification.

Direct coding from the transition table

Recall *Register* $\rightarrow$ *r digit digit** (Lecture 3, slides 10-11)

```
        word=''; char=''; goto s0;
    s0: word=word+char;
        char=input_char();
        if (char == 'r') then goto s1 else goto error;
    s1: word=word+char;
        char=input_char();
        if ('0' ≤ char ≥ '9') then goto s2 else goto error;
    s2: word=word+char;
        char=input_char();
        if ('0' ≤ char ≥ '9') then goto s2
           else if (char== EOF) return acceptance
                else goto error
error: print "error"; return failure;
```

- fewer operations
- avoids memory operations (esp. important for large tables)
- added complexity may make the code ugly and difficult to understand

Practical considerations

- Poor language design may complicate lexical analysis:
  - **`if then then = else; else else = then`** (PL/I)
  - **`DO5I=1,25`** vs **`DO5I=1.25`** (Fortran: urban legend has it that an error like this caused a crash of an early NASA mission)
  - the development of a sound theoretical basis has influenced language design positively.
- Template syntax in C++:
  - **`aaaa<mytype>`**
  - **`aaaa<mytype<int>>`** (>> is an operator for writing to the output stream)
  - The lexical analyser treats the >> operator as two consecutive > symbols. The confusion will be resolved by the parser (by matching the <, >)

Lex/Flex: Generating Lexical Analysers

Flex is a tool for generating scanners: programs which recognised lexical patterns in text (from the online manual pages: `% man flex`)

- Lex input consists of 3 sections:
  - regular expressions;
  - pairs of regular expressions and C code;
  - auxiliary C code.
- When the lex input is compiled, it generates as output a C source file lex.yy.c that contains a routine yylex(). After compiling the C file, the executable will start isolating tokens from the input according to the regular expressions, and, for each token, will execute the code associated with it. The array char yytext[] contains the representation of a token.

flex Example

```
%{
#define ERROR -1
int line_number=1;
%}
whitespace    [ \t]
letter        [a-zA-Z]
digit         [0-9]
integer       ({digit}+)
l_or_d        ({letter}|{digit})
identifier    ({letter}{l_or_d}*)
operator      [-+*/]
separator     [;,(){}]
%%
{integer}     {return 1;}
{identifier}  {return 2;}
{operator}|{separator}  {return (int)yytext[0];}
{whitespace}  {}
\n            {line_number++;}
.             {return ERROR;}
%%
int yywrap(void) {return 1;}
int main() {
    int token;
    yyin=fopen("myfile","r");
    while ((token=yylex())!=0)
       printf("%d %s \n", token, yytext);
    printf("lines %d \n",line_number);
}
```

Input file ("myfile")

```
123+435+34=aaaa
329*45/a-34*(45+23)**3
bye-bye
```

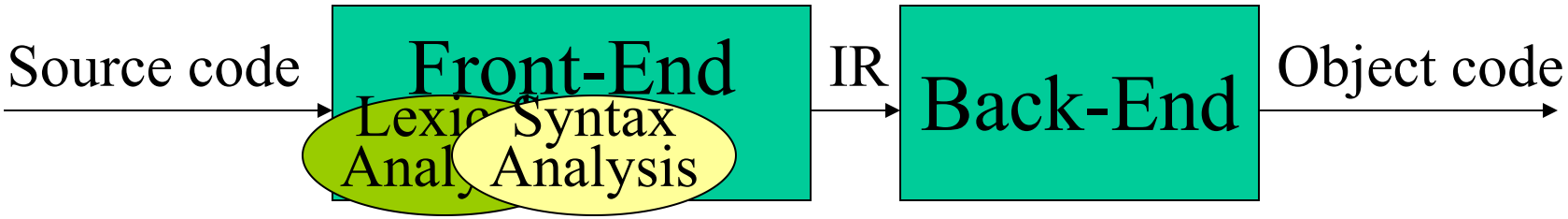
Output:

```
 1 123
43 +
 1 435
43 +
 1 34
-1 =
 2 aaaa
 1 329
42 *
 1 45
47 /
 2 a
45 -
 1 34
42 *
40 (
 1 45
43 +
 1 23
41 )
42 *
42 *
 1 3
 2 bye
45 -
 2 bye
lines 4
```

Conclusion

- Lexical Analysis turns a stream of characters in to a stream of tokens:
  - a largely automatic process.
    - REs are powerful enough to specify scanners.
    - DFAs have good properties for an implementation.
- Next time: Introduction to Parsing
- Reading: Aho2 3.9.6, 3.5 (lex); Aho1, pp. 141-144, 105-111 (lex); Grune pp.86-96; Hunter pp. 53-62; Cooper1 pp.55-72.
- Exercises: Aho2, p.166, p.186; Aho1 pp.146-150; Hunter pp. 71; Grune pp.185-187.

Lecture 7: Introduction to Parsing (Syntax Analysis)



Lexical Analysis:

- Reads characters of the input program and produces tokens.

  But: Are they syntactically correct? Are they valid sentences of the input language?

Today's lecture:

context-free grammars, derivations, parse trees, ambiguity

Not all languages can be described by Regular Expressions!!

(Lecture 3, Slide 7)

The descriptive power of regular expressions has limits:

- REs cannot be used to describe balanced or nested constructs: E.g., set of all strings of balanced parentheses {(), (()), ((())), …}, or the set of all 0s followed by an equal number of 1s, {01, 0011, 000111, ...}.
- In regular expressions, a non-terminal symbol cannot be used before it has been fully defined.

Chomsky's hierarchy of Grammars:

- 1. Phrase structured.
- 2. Context Sensitive

  number of Left Hand Side Symbols ≤ number of Right Hand Side Symbols
- 3. Context-Free

  The Left Hand Side Symbol is a non-terminal
- 4. Regular

  Only rules of the form: $A \rightarrow \varepsilon$, $A \rightarrow a$, $A \rightarrow pB$ are allowed.

Regular Languages $\subset$ Context-Free Languages $\subset$ Cont.Sens.Ls $\subset$ Phr.Str.Ls

Expressing Syntax

- Context-free syntax is specified with a context-free grammar.

  Recall (Lect.3, slide 3): A grammar, G, is a 4-tuple G={S,N,T,P}, where:

  S is a starting symbol; N is a set of non-terminal symbols;
  T is a set of terminal symbols; P is a set of production rules.

- Example:

  CatNoise → *CatNoise miau* **rule 1**

  | *miau* **rule 2**

  – We can use the CatNoise grammar to create sentences: E.g.:

Rule	**Sentential Form**
-	*CatNoise*
1	*CatNoise miau*
2	*miau miau*

  – Such a sequence of rewrites is called a derivation

The process of discovering a derivation for some sentence is called parsing!

Derivations and Parse Trees

Derivation: a sequence of derivation steps:

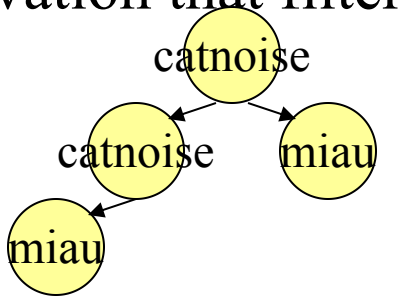
- At each step, we choose a non-terminal to replace.
- Different choices can lead to different derivations.

Two derivations are of interest:

- Leftmost derivation: at each step, replace the leftmost non-terminal.
- Rightmost derivation: at each step, replace the rightmost non-terminal

(we don't care about randomly-ordered derivations!)

A parse tree is a graphical representation for a derivation that filters out the choice regarding the replacement order.



Construction:

start with the starting symbol (root of the tree);

for each sentential form:

- *add children nodes (for each symbol in the right-hand-side of the production rule that was applied) to the node corresponding to the left-hand-side symbol.*

The leaves of the tree (read from left to right) constitute a sentential form (fringe, or yield, or frontier, or ...)

Find leftmost, rightmost derivation & parse tree for: x-2*y

1. Goal → Expr
2. Expr → Expr op Expr
3. | number
4. | id
5. Op → +
6. | –
7. | *
8. | /

Derivations and Precedence

- The leftmost and the rightmost derivation in the previous slide give rise to different parse trees. Assuming a standard way of traversing, the former will evaluate to *x – (2*y)*, but the latter will evaluate to *(x – 2)*y*.
- The two derivations point out a problem with the grammar: it has no notion of precedence (or implied order of evaluation).
- To add precedence: force parser to recognise high-precedence subexpressions first.

Ambiguity

A grammar that produces more than one parse tree for some sentence is ambiguous. Or:

- If a grammar has more than one leftmost derivation for a single sentential form, the grammar is ambiguous.
- If a grammar has more than one rightmost derivation for a single sentential form, the grammar is ambiguous.

Example:

- Stmt $\rightarrow$ if Expr then Stmt | if Expr then Stmt else Stmt | …other…
- What are the derivations of:
  - if E1 then if E2 then S1 else S2

Eliminating Ambiguity

- Rewrite the grammar to avoid the problem
- Match each else to innermost unmatched if:
  - 1. Stmt → IfwithElse
    2. | IfnoElse
  - 3. IfwithElse → if Expr then IfwithElse else IfwithElse
    4. | … other stmts…
  - 5. IfnoElse → if Expr then Stmt
    6. | if Expr then IfwithElse else IfnoElse

Stmt
(2) IfnoElse
(5) if Expr then Stmt
(?) if E1 then Stmt
(1) if E1 then IfwithElse
(3) if E1 then if Expr then IfwithElse else IfwithElse
(?) if E1 then if E2 then IfwithElse else IfwithElse
(4) if E1 then if E2 then S1 else IfwithElse
(4) if E1 then if E2 then S1 else S2

Deeper Ambiguity

- Ambiguity usually refers to confusion in the CFG
- Overloading can create deeper ambiguity
  - E.g.: a=b(3) : b could be either a function or a variable.
- Disambiguating this one requires context:
  - An issue of type, not context-free syntax
  - Needs values of declarations
  - Requires an extra-grammatical solution
- Resolving ambiguity:
  - if context-free: rewrite the grammar
  - context-sensitive ambiguity: check with other means: needs knowledge of types, declarations, … This is a language design problem
- Sometimes the compiler writer accepts an ambiguous grammar: parsing techniques may do the “right thing”.

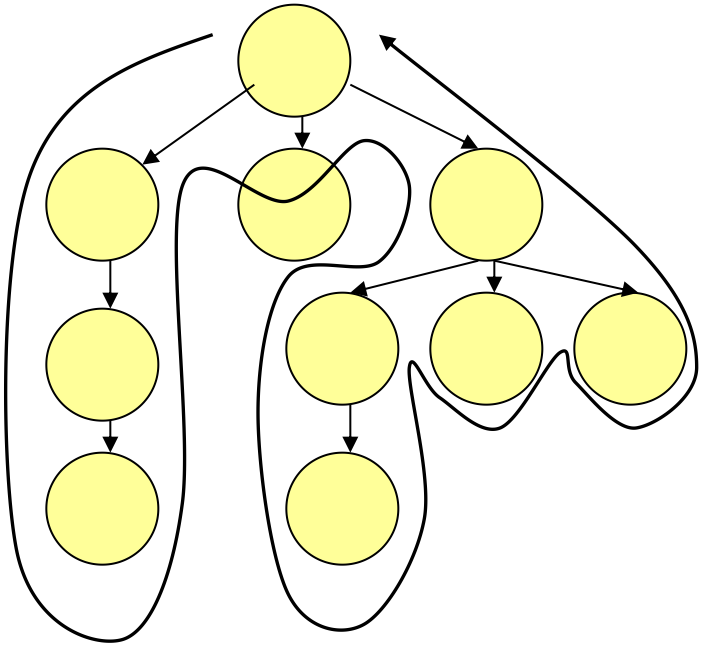
Parsing techniques

- Top-down parsers:
  - Construct the top node of the tree and then the rest in pre-order. (depth-first)
  - Pick a production & try to match the input; if you fail, backtrack.
  - Essentially, we try to find a **leftmost** derivation for the input string (which we scan left-to-right).
  - some grammars are backtrack-free (predictive parsing).
- Bottom-up parsers:
  - Construct the tree for an input string, beginning at the leaves and working up towards the top (root).
  - Bottom-up parsing, using left-to-right scan of the input, tries to construct a **rightmost** derivation in reverse.
  - Handle a large class of grammars.

Top-down vs …

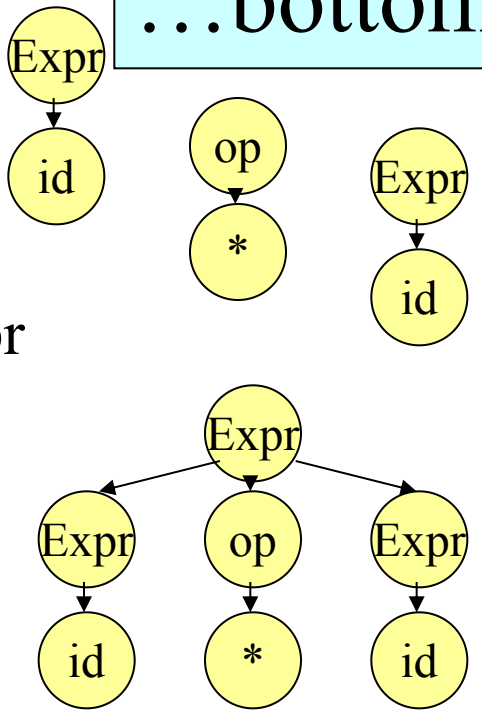
Has an analogy with two special cases of depth-first traversals:

- Pre-order: first traverse node x and then x's subtrees in left-to-right order. (action is done when we first visit a node)
- Post-order: first traverse node x's subtrees in left-to-right order and then node x. (action is done just before we leave a node for the last time)



…bottom-up!

id * id
Expr * id
Expr op id
Expr op Expr
Expr



Top-Down Recursive-Descent Parsing

- 1. Construct the root with the starting symbol of the grammar.
- 2. Repeat until the fringe of the parse tree matches the input string:
  - Assuming a node labelled A, select a production with A on its left-hand-side and, for each symbol on its right-hand-side, construct the appropriate child.
  - When a terminal symbol is added to the fringe and it doesn't match the fringe, backtrack.
  - Find the next node to be expanded.

The key is picking the right production in the first step: that choice should be guided by the input string.

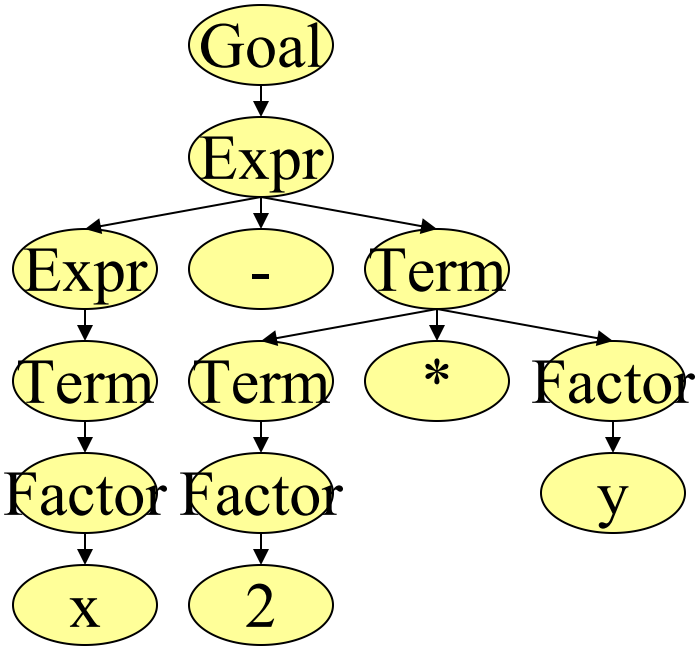
Example:

1. *Goal* → *Expr*
2. *Expr* → *Expr* + *Term*
3. | *Expr* – *Term*
4. | *Term*
5. *Term* → *Term* * *Factor*
6. | *Term* / *Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

Example: Parse *x-2*y*

Steps (one scenario from many)

Rule	Sentential Form	Input
-	*Goal*	| x – 2*y
1	*Expr*	| x – 2*y
2	*Expr + Term*	| x – 2*y
4	*Term + Term*	| x – 2*y
7	*Factor + Term*	| x – 2*y
9	*id + Term*	| x – 2*y
Fail	*id + Term*	x | – 2*y
Back	*Expr*	| x – 2*y
3	*Expr – Term*	| x – 2*y
4	*Term – Term*	| x – 2*y
7	*Factor – Term*	| x – 2*y
9	*id – Term*	| x – 2*y
Match	*id – Term*	x – | 2*y
7	*id – Factor*	x – | 2*y
9	*id – num*	x – | 2*y
Fail	*id – num*	x – 2 | *y
Back	*id – Term*	x – | 2*y
5	*id – Term * Factor*	x – | 2*y
7	*id – Factor * Factor*	x – | 2*y
8	*id – num * Factor*	x – | 2*y
match	*id – num * Factor*	x – 2* | y
9	*id – num * id*	x – 2* | y
match	*id – num * id*	x – 2*y |



Other choices for expansion are possible:

Rule	Sentential Form	Input
-	*Goal*	| x – 2*y
1	*Expr*	| x – 2*y
2	*Expr + Term*	| x – 2*y
2	*Expr + Term + Term*	| x – 2*y
2	*Expr + Term + Term + Term*	| x – 2*y
2	*Expr + Term + Term + … + Term*	| x – 2*y

- Wrong choice leads to non-termination!
- This is a bad property for a parser!
- Parser must make the right choice!

Conclusion

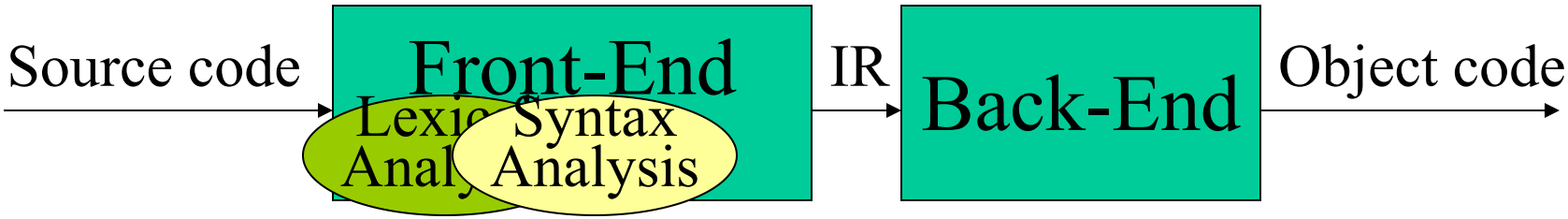
- The parser's task is to analyse the input program as abstracted by the scanner.

- Next time: Top-Down Parsing

- Reading: Aho2, Sections 4.1, 4.2, 4.3.1, 4.3.2, (see also pp.56-60); Aho1, pp. 160-175; Grune pp.34-40, 110-115; Hunter pp. 21-44; Cooper pp.73-89.

- Exercises: Aho1 267-268; Hunter pp. 44-46.

Lecture 8: Top-Down Parsing



Parsing:

- Context-free syntax is expressed with a context-free grammar.
- The process of discovering a derivation for some sentence.

Today's lecture:

Top-down parsing

Recursive-Descent Parsing

- 1. Construct the root with the starting symbol of the grammar.
- 2. Repeat until the fringe of the parse tree matches the input string:
  - Assuming a node labelled A, select a production with A on its left-hand-side and, for each symbol on its right-hand-side, construct the appropriate child.
  - When a terminal symbol is added to the fringe and it doesn't match the fringe, backtrack.
  - Find the next node to be expanded.

The key is picking the right production in the first step: that choice should be guided by the input string.

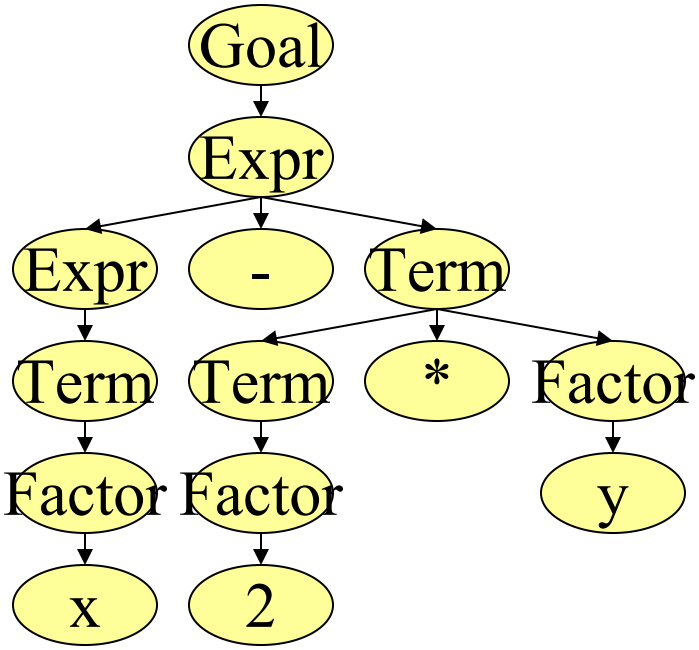
Example:

1. *Goal* → *Expr*
2. *Expr* → *Expr* + *Term*
3. | *Expr* – *Term*
4. | *Term*
5. *Term* → *Term* * *Factor*
6. | *Term* / *Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

Example: Parse *x-2*y*

Steps (one scenario from many)

Rule	Sentential Form	Input
-	*Goal*	| x – 2*y
1	*Expr*	| x – 2*y
2	*Expr + Term*	| x – 2*y
4	*Term + Term*	| x – 2*y
7	*Factor + Term*	| x – 2*y
9	*id + Term*	| x – 2*y
Fail	*id + Term*	x | – 2*y
Back	*Expr*	| x – 2*y
3	*Expr – Term*	| x – 2*y
4	*Term – Term*	| x – 2*y
7	*Factor – Term*	| x – 2*y
9	*id – Term*	| x – 2*y
Match	*id – Term*	x – | 2*y
7	*id – Factor*	x – | 2*y
9	*id – num*	x – | 2*y
Fail	*id – num*	x – 2 | *y
Back	*id – Term*	x – | 2*y
5	*id – Term * Factor*	x – | 2*y
7	*id – Factor * Factor*	x – | 2*y
8	*id – num * Factor*	x – | 2*y
match	*id – num * Factor*	x – 2* | y
9	*id – num * id*	x – 2* | y
match	*id – num * id*	x – 2*y |



Other choices for expansion are possible:

Rule	Sentential Form	Input
-	*Goal*	| x – 2*y
1	*Expr*	| x – 2*y
2	*Expr + Term*	| x – 2*y
2	*Expr + Term + Term*	| x – 2*y
2	*Expr + Term + Term + Term*	| x – 2*y
2	*Expr + Term + Term + ... + Term*	| x – 2*y

- Wrong choice leads to non-termination!
- This is a bad property for a parser!
- Parser must make the right choice!

Left-Recursive Grammars

- **Definition**: A grammar is left-recursive if it has a non-terminal symbol A, such that there is a derivation $A \Rightarrow Aa$, for some string a.
- A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop.
- **Eliminating left-recursion**: In many cases, it is sufficient to replace $A \rightarrow Aa \mid b$ with $A \rightarrow bA'$ and $A' \rightarrow aA' \mid \varepsilon$
- **Example**:

  $Sum \rightarrow Sum + number \mid number$

  would become:

  $Sum \rightarrow number \; Sum'$

  $Sum' \rightarrow +number \; Sum' \mid \varepsilon$

Eliminating Left Recursion

Applying the transformation to the Grammar of the Example in Slide 2 we get:

$Expr \rightarrow Term\ Expr'$

$Expr' \rightarrow +Term\ Expr' \mid -Term\ Expr' \mid \varepsilon$

$Term \rightarrow Factor\ Term'$

$Term' \rightarrow *Factor\ Term' \mid /Factor\ Term' \mid \varepsilon$

($Goal \rightarrow Expr$ and $Factor \rightarrow number \mid id$ remain unchanged)

Non-intuitive, but it works!

General algorithm: works for non-cyclic, no ε-productions grammars

1. Arrange the non-terminal symbols in order: $A_1, A_2, A_3, \ldots, A_n$

2. For i=1 to n do

for j=1 to i-1 do

I) replace each production of the form $A_i \rightarrow A_j \gamma$ with

the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$

where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current A_j productions

II) eliminate the immediate left recursion among the A_i

Where are we?

- We can produce a top-down parser, but:
  - if it picks the wrong production rule it has to backtrack.
- **Idea**: look ahead in input and use context to pick correctly.
- How much lookahead is needed?
  - In general, an arbitrarily large amount.
  - Fortunately, most programming language constructs fall into subclasses of context-free grammars that can be parsed with limited lookahead.

Predictive Parsing

- Basic idea:
  - For any production $A \rightarrow a \mid b$ we would like to have a distinct way of choosing the correct production to expand.
- *FIRST* sets:
  - For any symbol A, *FIRST(A)* is defined as the set of terminal symbols that appear as the first symbol of one or more strings derived from A.

    E.g. (grammar in Slide 5): *FIRST(Expr')={+,-,ε}, FIRST(Term')={*,/,ε}, FIRST(Factor)={number, id}*
- The LL(1) property:
  - If $A \rightarrow a$ and $A \rightarrow b$ both appear in the grammar, we would like to have: $FIRST(a) \cap FIRST(b) = \varnothing$. This would allow the parser to make a correct choice with a lookahead of exactly one symbol!

The Grammar of Slide 5 has this property!

Recursive Descent Predictive Parsing

(a practical implementation of the Grammar in Slide 5)

```
Main()
  token=next_token();
  if (Expr()!=false)
    then <next_compilation_step>
  else return false;

Expr()
  if (Term()==false)
    then result=false
  else if (EPrime()==false)
    then result=false
  else result=true
  return result

EPrime()
  if (token=='+' or '-') then
    token=next_token()
    if (Term()==false)
      then result=false
    elseif (EPrime()==false)
      then result=false
    else result=true
  else result=true /* ε */
  return result

Term()
  if (Factor()==false)
    then result=false
  else if (TPrime()==false)
    then result=false
  else result=true
  return result

TPrime()
  if (token=='*' or '/') then
    token=next_token()
    if (Factor()==false)
      then result=false
    else if (TPrime()==false)
      then result=false
    else result=true
  else result=true
  return result

Factor()
  if (token=='number' or 'id')then
    token=next_token()
    result=true
  else
    report syntax_error
    result=false
  return result
```

No backtracking is needed!

check :-)

Left Factoring

What if my grammar does not have the LL(1) property?

Sometimes, we can transform a grammar to have this property.

Algorithm:

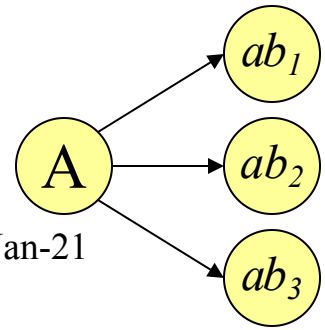
1. For each non-terminal A, find the longest prefix, say a, common to two or more of its alternatives

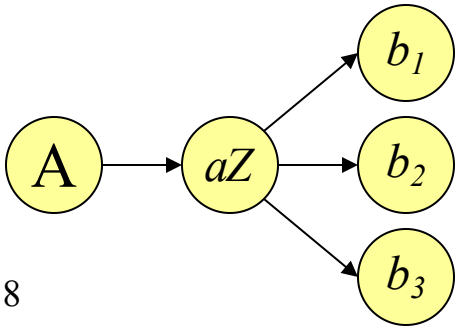
2. if $a \neq \varepsilon$ then replace all the A productions, $A \rightarrow ab_1|ab_2|ab_3|...|ab_n|\gamma$, where γ is anything that does not begin with a, with $A \rightarrow aZ \mid \gamma$ and $Z \rightarrow b_1|b_2|b_3|...|b_n$

Repeat the above until no common prefixes remain

Example: $A \rightarrow ab_1 \mid ab_2 \mid ab_3$ would become $A \rightarrow aZ$ and $Z \rightarrow b_1|b_2|b_3$

Note the graphical representation:





Example

(NB: this is a different grammar from the one in Slide 2)

$Goal \rightarrow Expr$

$\mathbf{Expr \rightarrow Term + Expr \mid Term - Expr \mid Term}$

$\mathbf{Term \rightarrow Factor * Term \mid Factor / Term \mid Factor}$

$Factor \rightarrow number \mid id$

We have a problem with the different rules for *Expr* as well as those for *Term*. In both cases, the first symbol of the right-hand side is the same (*Term* and *Factor*, respectively). E.g.:

$FIRST(Term)=FIRST(Term)\cap FIRST(Term)=\{number, id\}.$
$FIRST(Factor)=FIRST(Factor)\cap FIRST(Factor)=\{number, id\}.$

Applying left factoring:

$Expr \rightarrow Term\ Expr'$
$Expr' \rightarrow + Expr \mid - Expr \mid \varepsilon$

$Term \rightarrow Factor\ Term'$
$Term' \rightarrow * Term \mid / Term \mid \varepsilon$

$FIRST(+)=\{+\}; FIRST(-)=\{-\}; FIRST(\varepsilon)=\{\varepsilon\};$
$FIRST(-)\cap FIRST(+) \cap FIRST(\varepsilon)= =\varnothing$

$FIRST(*)=\{*\}; FIRST(/)=\{/\}; FIRST(\varepsilon)=\{\varepsilon\};$
$FIRST(*)\cap FIRST(/) \cap FIRST(\varepsilon)= =\varnothing$

Example (cont.)

1. *Goal* → *Expr*
2. *Expr* → *Term Expr′*
3. *Expr′* → + *Expr*
4. | - *Expr*
5. | ε
6. *Term* → *Factor Term′*
7. *Term′* → * *Term*
8. | / *Term*
9. | ε
10. *Factor* → *number*
11. | *id*

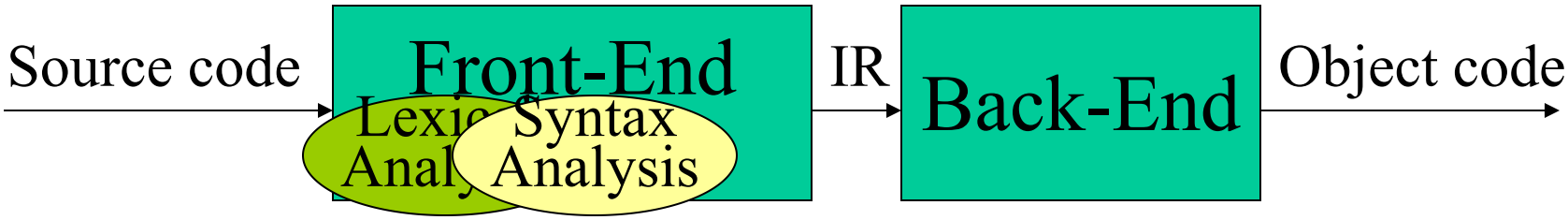
The next symbol determines each choice correctly. No backtracking needed.

Rule	Sentential Form	Input
-	*Goal*	| x – 2*y
1	*Expr*	| x – 2*y
2	*Term Expr′*	| x – 2*y
6	*Factor Term′ Expr′*	| x – 2*y
11	*id Term′ Expr′*	| x – 2*y
Match	*id Term′ Expr′*	x | – 2*y
9	*id* ε *Expr′*	x | – 2*y
4	*id – Expr*	x | – 2*y
Match	*id – Expr*	x – | 2*y
2	*id – Term Expr′*	x – | 2*y
6	*id – Factor Term′ Expr′*	x – | 2*y
10	*id – num Term′ Expr′*	x – | 2*y
Match	*id – num Term′ Expr′*	x – 2 | *y
7	*id – num * Term Expr′*	x – 2 | *y
Match	*id – num * Term Expr′*	x – 2* | y
6	*id – num * Factor Term′ Expr′*	x – 2* | y
11	*id – num * id Term′ Expr′*	x – 2* | y
Match	*id – num * id Term′ Expr′*	x – 2*y |
9	*id – num * id Expr′*	x – 2*y |
5	*id – num * id*	x – 2*y |

Conclusion

- Top-down parsing:
  - recursive with backtracking (not often used in practice)
  - recursive predictive
- Nonrecursive Predictive Parsing is possible too: maintain a stack explicitly rather than implicitly via recursion and determine the production to be applied using a table (Aho, pp.186-190).
- Given a Context Free Grammar that doesn't meet the LL(1) condition, it is undecidable whether or not an equivalent LL(1) grammar exists.
- Next time: Bottom-Up Parsing
- Reading: Aho2, Sections 4.3.3, 4.3.4, 4.4; Aho1, pp. 176-178, 181-185; Grune pp.117-133; Hunter pp. 72-93; Cooper, Section 3.3.

Lecture 9: Bottom-Up Parsing



(from last lecture) Top-Down Parsing:

- Start at the root of the tree and grow towards leaves.
- Pick a production and try to match the input.
- We may need to backtrack if a bad choice is made.
- Some grammars are backtrack-free (predictive parsing).

Today's lecture:

Bottom-Up parsing

Bottom-Up Parsing: What is it all about?

Goal: Given a grammar, G, construct a parse tree for a string (i.e., sentence) by starting at the leaves and working to the root (i.e., by working from the input sentence back toward the start symbol S).

Recall: the point of parsing is to construct a derivation:

$$S \Rightarrow \delta_0 \Rightarrow \delta_1 \Rightarrow \delta_2 \Rightarrow \ldots \Rightarrow \delta_{n-1} \Rightarrow sentence$$

To derive δ_{i-1} from δ_i, we match some *rhs* b in δ_i, then replace b with its corresponding *lhs*, A. This is called a **reduction** (it assumes $A \rightarrow b$).

The **parse tree** is the result of the **tokens** and the **reductions**.

Example: Consider the grammar below and the input string **abbcde**.

```
1.  Goal→aABe
2.     A→Abc
3.       |b
4.     B→d
```

Sentential Form	Production	Position
abbcde	3	2
a A bcde	2	4
a A de	4	3
a A B e	1	4
Goal	-	-

Finding Reductions

- What are we trying to find?
  - A substring *b* that matches the right-side of a production that occurs as one step in the rightmost derivation. Informally, this substring is called a **handle**.
- Formally, a handle of a right-sentential form δ is a pair $<A \rightarrow b, k>$ where $A \rightarrow b \in P$ and *k* is the position in δ of *b*'s rightmost symbol.

  (right-sentential form: a sentential form that occurs in some rightmost derivation).
  - Because δ is a right-sentential form, the substring to the right of a handle contains only terminal symbols. Therefore, the parser doesn't need to scan past the handle.
  - If a grammar is unambiguous, then every right-sentential form has a unique handle (sketch of proof by definition: if unambiguous then rightmost derivation is unique; then there is unique production at each step to produce a sentential form; then there is a unique position at which the rule is applied; hence, unique handle).

    If we can find those handles, we can build a derivation!

Motivating Example

Given the grammar of the left-hand side below, find a rightmost derivation for *x – 2*y* (starting from Goal there is only one, the grammar is not ambiguous!). In each step, identify the handle.

1. *Goal* → *Expr*
2. *Expr* → *Expr* + *Term*
3. | *Expr* – *Term*
4. | *Term*
5. *Term* → *Term* * *Factor*
6. | *Term* / *Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

Production	Sentential Form	Handle
-	*Goal*	-
1	*Expr*	1,1
3	*Expr – Term*	3,3

Problem: given the sentence *x – 2*y*, find the handles!

A basic bottom-up parser

- The process of discovering a handle is called handle pruning.
- To construct a rightmost derivation, apply the simple algorithm:

  for $i=n$ to 1, step -1

  find the handle $<A \rightarrow b,k>_i$ in δ_i

  replace b with A to generate δ_{i-1}

  (needs 2n steps, where n is the length of the derivation)
- One implementation is based on using a stack to hold grammar symbols and an input buffer to hold the string to be parsed. Four operations apply:
  - **shift**: next input is shifted (pushed) onto the top of the stack
  - **reduce**: right-end of the handle is on the top of the stack; locate left-end of the handle within the stack; pop handle off stack and push appropriate non-terminal left-hand-side symbol.
  - **accept**: terminate parsing and signal success.
  - **error**: call an error recovery routine.

Implementing a shift-reduce parser

```
push $ onto the stack
token = next_token()
repeat
    if the top of the stack is a handle A→b
        then /* reduce b to A  */
                pop the symbols of b off the stack
                push A onto the stack
        elseif (token != eof) /* eof: end-of-file = end-of-input */
                then /* shift */
                        push token
                        token=next_token()
                else /* error */
                        call error_handling()
until (top_of_stack == Goal && token==eof)
```

Errors show up: a) when we fail to find a handle, or b) when we hit EOF and we need to shift. The parser needs to recognise syntax errors.

Example: x–2*y

Stack	Input	Handle	Action	
$	id – num * id	None	Shift	
$ id	– num * id	9,1	Reduce 9	
$ Factor	– num * id	7,1	Reduce 7	
$ Term	– num * id	4,1	Reduce 4	
$ Expr	– num * id	None	Shift	!!
$ Expr –	num * id	None	Shift	
$ Expr – num	* id	8,3	Reduce 8	
$ Expr – Factor	* id	7,3	Reduce 7	
$ Expr – Term	* id	None	Shift	!!
$ Expr – Term *	id	None	Shift	
$ Expr – Term * id		9,5	Reduce 9	
$ Expr – Term * Factor		5,5	Reduce 5	
$ Expr – Term		3,3	Reduce 3	
$ Expr		1,1	Reduce 1	
$ Goal		none	Accept	

- 1. Shift until top of stack is the right end of the handle
- 2. Find the left end of the handle and reduce

(5 shifts, 9 reduces, 1 accept)

What can go wrong?

(think about the steps with an exclamation mark in the previous slide)

- **Shift/reduce conflicts**: the parser cannot decide whether to shift or to reduce.

  Example: the dangling-else grammar; usually due to ambiguous grammars.

  Solution: a) modify the grammar; b) resolve in favour of a shift.

- **Reduce/reduce conflicts**: the parser cannot decide which of several reductions to make.

  Example: `id(id,id)`; reduction is dependent on whether the first `id` refers to array or function.

  May be difficult to tackle.

Key to efficient bottom-up parsing: the handle-finding mechanism.

LR(1) grammars

(a beautiful example of applying theory to solve a complex problem in practice)

A grammar is LR(1) if, given a rightmost derivation, we can (I) isolate the handle of each right-sentential form, and (II) determine the production by which to reduce, by scanning the sentential form from left-to-right, going at most 1 symbol beyond the right-end of the handle.

- LR(1) grammars are widely used to construct (automatically) efficient and flexible parsers:
  - Virtually all context-free programming language constructs can be expressed in an LR(1) form.
  - LR grammars are the most general grammars parsable by a non-backtracking, shift-reduce parser (deterministic CFGs).
  - Parsers can be implemented in time proportional to tokens+reductions.
  - LR parsers detect an error as soon as possible in a left-to-right scan of the input.

L stands for left-to-right scanning of the input; R for constructing a rightmost derivation in reverse; 1 for the number of input symbols for lookahead.

LR Parsing: Background

- Read tokens from an input buffer (same as with shift-reduce parsers)
- Add an extra state information after each symbol in the stack. The state summarises the information contained in the stack below it. The stack would look like:

  $\$ \; S_0 \; Expr \; S_1 \; - \; S_2 \; num \; S_3$

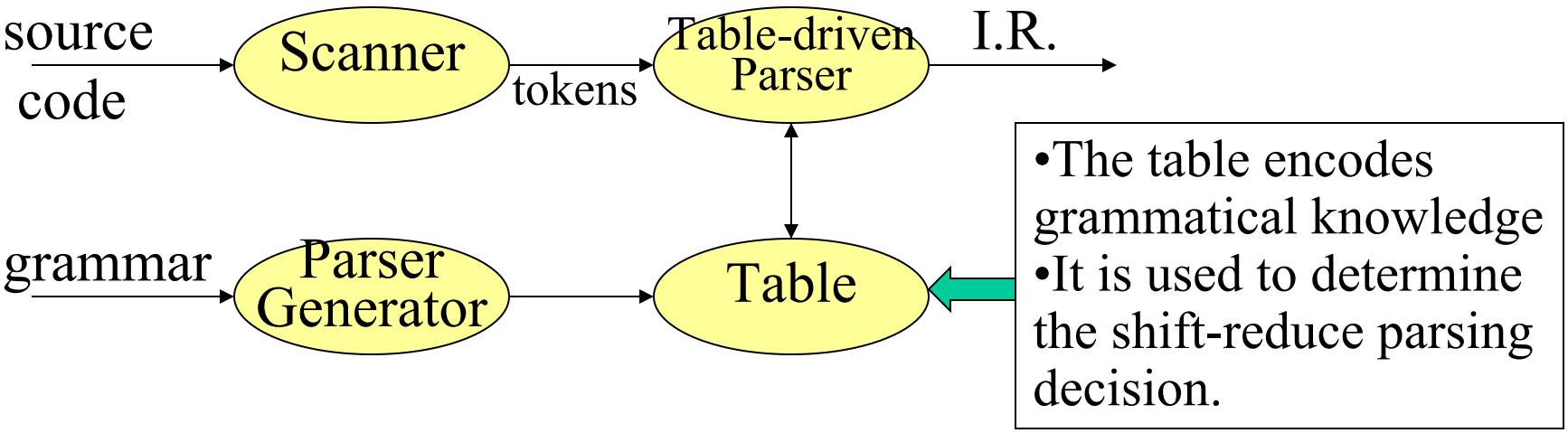
- Use a table that consists of two parts:
  - **action**[state_on_top_of_stack, input_symbol]: returns one of: shift s (push a symbol and a state); reduce by a rule; accept; error.
  - **goto**[state_on_top_of_stack,non_terminal_symbol]: returns a new state to push onto the stack after a reduction.

Skeleton code for an LR Parser

```
Push $ onto the stack
push s0
token=next_token()
repeat
  s=top_of_the_stack  /* not pop! */
  if ACTION[s,token]==‘reduce A→b’
     then  pop 2*(symbols_of_b) off the stack
           s=top_of_the_stack  /* not pop! */
           push A; push GOTO[s,A]
     elseif ACTION[s,token]==‘shift sx’
                then      push token; push sx
                     token=next_token()
                elseif ACTION[s,token]==‘accept’
                     then break
                     else report_error
end repeat
report_success
```

The Big Picture: Prelude to what follows

- LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context for handle recognition.
- They can be built by hand; perfect to automate too!
- Summary: Bottom-up parsing is more powerful!



Next: we will automate table construction!
Reading: Aho2 Section 4.5; Aho1 pp.195-202; Hunter pp.100-103; Grune pp.150-152

Example

Consider the following grammar and tables:

1. *Goal* → *CatNoise*
2. *CatNoise* → *CatNoise miau*
3. | *miau*

STATE	ACTION		GOTO
	eof	miau	CatNoise
0	-	Shift 2	1
1	accept	Shift 3	
2	Reduce 3	Reduce 3	
3	Reduce 2	Reduce 2	

Example 1: (input string miau)

Stack	Input	Action
$ s0	miau eof	Shift 2
$ s0 miau s2	eof	Reduce 3
$ s0 CatNoise s1	eof	Accept

Note that there cannot be a syntax error with CatNoise, because it has only 1 terminal symbol. "miau woof" is a lexical problem, not a syntax error!

Example 2: (input string miau miau)

Stack	Input	Action
$ s0	miau miau eof	Shift 2
$ s0 miau s2	miau eof	Reduce 3
$ s0 CatNoise s1	miau eof	Shift 3
$ s0 CatNoise s1 miau s3	eof	Reduce 2
$ s0 CatNoise s1	eof	accept

eof is a convention for end-of-file (=end of input)

Example: the expression grammar (slide 4)

1. *Goal* → *Expr*
2. *Expr* → *Expr* + *Term*
3. | *Expr* – *Term*
4. | *Term*
5. *Term* → *Term* * *Factor*
6. | *Term* / *Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

STATE	ACTION							GOTO		
	eof	+	–	*	/	num	id	Expr	Term	Factor
0						S 4	S 5	1	2	3
1	Acc	S 6	S 7							
2	R 4	R 4	R 4	S 8	S 9					
3	R 7	R 7	R 7	R 7	R 7					
4	R 8	R 8	R 8	R 8	R 8					
5	R 9	R 9	R 9	R 9	R 9					
6						S 4	S 5		10	3
7						S 4	S 5		11	3
8						S 4	S 5			12
9						S 4	S 5			13
10	R 2	R 2	R 2	S 8	S 9					
11	R 3	R 3	R 3	S 8	S 9					
12	R 5	R 5	R 5	R 5	R 5					
13	R 6	R 6	R 6	R 6	R 6					

*Apply the algorithm in slide 3 to the expression x-2*y*
The result is the rightmost derivation (as in Lect.8, slide 7), but …
…no conflicts now: state information makes it fully deterministic!

1. *Goal* → *Expr*
2. *Expr* → *Expr* + *Term*
3. | *Expr* – *Term*
4. | *Term*
5. *Term* → *Term* * *Factor*
6. | *Term* / *Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

STATE	ACTION							GOTO		
	eof	+	–	*	/	num	id	Expr	Term	Factor
0						S 4	S 5	1	2	3
1	Acc	S 6	S 7							
2	R 4	R 4	R 4	S 8	S 9					
3	R 7	R 7	R 7	R 7	R 7					
4	R 8	R 8	R 8	R 8	R 8					
5	R 9	R 9	R 9	R 9	R 9					
6						S 4	S 5		10	3
7						S 4	S 5		11	3
8						S 4	S 5			12
9						S 4	S 5			13
10	R 2	R 2	R 2	S 8	S 9					
11	R 3	R 3	R 3	S 8	S 9					
12	R 5	R 5	R 5	R 5	R 5					
13	R 6	R 6	R 6	R 6	R 6					

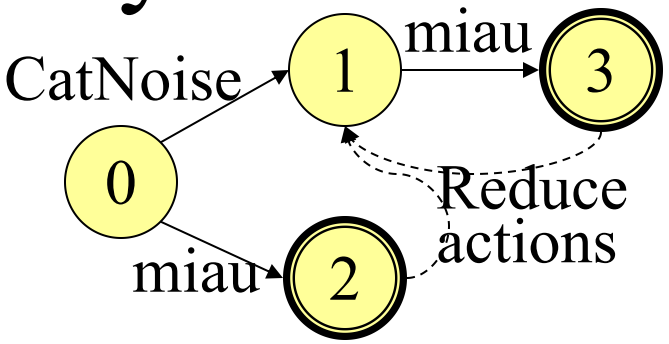
Stack	Input	Action
$s0	x-2*y	Shift 5
$s0 x s5	- 2*y	Reduce 9
$s0 Factor s3	- 2*y	Reduce 7
$s0 Term s2	- 2*y	Reduce 4
$s0 Expr s1	- 2*y	Shift 7
$s0 Expr s1 - s7	2*y	Shift 4
$s0 Expr s1 - s7 2 s4	*y	Reduce 8
$s0 Expr s1 - s7 Factor s3	*y	Reduce 7
$s0 Expr s1 - s7 Term s11	*y	Shift 8
$s0 Expr s1 - s7 Term s11 * s8	y	Shift 5
$s0 Expr s1 - s7 Term s11 * s8 y s5	EOF	Reduce 9
$s0 Expr s1 - s7 Term s11 * s8 Factor s12	EOF	Reduce 5
$s0 Expr s1 - s7 Term s11	EOF	Reduce 3
$s0 Expr s1	EOF	accept

Summary

- **Top-Down Recursive Descent**: Pros: Fast, Good locality, Simple, good error-handling. Cons: Hand-coded, high-maintenance.
- **LR(1):** Pros: Fast, deterministic languages, automatable. Cons: large working sets, poor error messages.
- **What is left to study?**
  - Checking for context-sensitive properties
  - Laying out the abstractions for programs & procedures.
  - Generating code for the target machine.
  - Generating good code for the target machine.
- **Reading**: Aho2 Sections 4.7, 4.10; Aho1 pp.215-220 & 230-236; Cooper 3.4, 3.5; Grune pp.165-170; Hunter 5.1-5.5 (too general).

LR(1) – Table Generation

LR Parsers: How do they work?



- Key: language of handles is regular
  - build a handle-recognising DFA
  - Action and Goto tables encode the DFA
- How do we generate the Action and Goto tables?
  - Use the grammar to build a model of the DFA
  - Use the model to build Action and Goto tables
  - If construction succeeds, the grammar is LR(1).
- Three commonly used algorithms to build tables:
  - LR(1): full set of LR(1) grammars; large tables; slow, large construction.
  - SLR(1): smallest class of grammars; smallest tables; simple, fast construction.
  - LALR(1): intermediate sized set of grammars; smallest tables; very common. (Space used to be an obsession; now it is only a concern)

LR(1) Items

- An LR(1) item is a pair [A,B], where:
  - A is a production $\alpha \rightarrow \beta\gamma\delta$ with a • at some position in the *rhs*.
  - B is a lookahead symbol.
- The • indicates the position of the top of the stack:
  - $[\alpha \rightarrow \beta\gamma \bullet \delta, a]$: the input seen so far (ie, what is in the stack) is consistent with the use of $\alpha \rightarrow \beta\gamma\delta$, and the parser has recognised $\beta\gamma$.
  - $[\alpha \rightarrow \beta\gamma\delta \bullet, a]$: the parser has seen $\beta\gamma\delta$, and a lookahead symbol of a is consistent with reducing to α.
- The production $\alpha \rightarrow \beta\gamma\delta$ with lookahead a, generates:
  - $[\alpha \rightarrow \bullet\beta\gamma\delta, a]$, $[\alpha \rightarrow \beta \bullet \gamma\delta, a]$, $[\alpha \rightarrow \beta\gamma \bullet \delta, a]$, $[\alpha \rightarrow \beta\gamma\delta \bullet, a]$
- ***The set of LR(1) items is finite.***
  - *Sets of LR(1) items represent LR(1) parser states.*

The Table Construction Algorithm

- Table construction:
  - **1.** Build the canonical collection of sets of LR(1) items, S:
    - I) Begin in S_0 with [Goal→•α, eof] and find all equivalent items as **closure**(S_0).
    - II) Repeatedly compute, for each S_k and each symbol α (both terminal and non-terminal), **goto**(S_k,α). If the set is not in the collection add it. This eventually reaches a fixed point.
  - **2.** Fill in the table from the collection of sets of LR(1) items.
- The canonical collection completely encodes the transition diagram for the handle-finding DFA.
- The lookahead is the key in choosing an action:

Remember Expr-Term from Lecture 8 slide 7, when we chose to shift rather than reduce to Expr?

Closure(state)

Closure(s) // s is the state

while (s is still changing)

for each item [α→β•γδ,a] in s

for each production γ→τ

for each terminal b in FIRST(δa)

if [γ→•τ,b] is not in s, then add it.

Recall (Lecture 7, Slide 7): *FIRST(A) is defined as the set of terminal symbols that appear as the first symbol in strings derived from A.*

E.g.: FIRST(Goal) = FIRST(CatNoise) = FIRST(miau) = miau

Example: (using the CatNoise Grammar) S0: {[Goal→•CatNoise,eof], [CatNoise→•CatNoise miau, eof], [CatNoise→•miau, eof], [CatNoise→•CatNoise miau, miau], [CatNoise→•miau, miau]}
(the 1st item by definition; 2nd,3rd are derived from the 1st; 4th,5th are derived from the 2nd)

Goto(s,x)

Goto(s,x)

new=∅

for each item [α→β•xδ,a] in s

add [α→βx•δ,a] to new

return closure(new)

Computes the state that the parser would reach if it recognised an x while in state s.

Example:

S1 (x=CatNoise): [Goal→CatNoise•,eof], [CatNoise→CatNoise• miau, eof], [CatNoise→CatNoise• miau, miau]

S2 (x=miau): [CatNoise→miau•, eof], [CatNoise→miau•, miau]

S3 (from S1): [CatNoise→CatNoise miau•, eof], [CatNoise→CatNoise miau•, miau]

Example (slide 1 of 4)

Simplified expression grammar:

$Goal \rightarrow Expr$

$Expr \rightarrow Term\text{-}Expr$

$Expr \rightarrow Term$

$Term \rightarrow Factor*Term$

$Term \rightarrow Factor$

$Factor \rightarrow id$

$FIRST(Goal)=FIRST(Expr)=FIRST(Term)=FIRST(Factor)=FIRST(id)=id$

$FIRST(-)=-$

$FIRST(*)=*$

Example: first step (slide 2 of 4)

- S0: closure({[Goal→•Expr,eof]})

  {[Goal→•Expr,eof], [Expr→•Term-Expr,eof], [Expr→•Term,eof], [Term→•Factor*Term,eof], [Term→•Factor*Term,-], [Term→•Factor,eof], [Term→•Factor,-], [Factor→•id, eof], [Factor→•id,-], [Factor→•id,*]}
- Next states:
  - Iteration 1:
    - S1: goto(S0,Expr), S2: goto(S0,Term), S3: goto(S0, Factor), S4: goto(S0, id)
  - Iteration 2:
    - S5: goto(S2,-), S6: goto(S3,*)
  - Iteration 3:
    - S7: goto(S5, Expr), S8: goto(S6, Term)

Example: the states (slide 3 of 4)

S1: {[Goal→Expr•,eof]}
S2: {[Goal→Term•-Expr,eof], [Expr→Term•,eof]}
S3: {[Term→Factor•*Term,eof],[Term→Factor•*Term,-],
[Term→Factor•,eof], [Term→Factor•,-]}
S4: {[Factor→id•,eof], [Factor→id•,-], [Factor→id•,*]}
S5: {[Expr→Term-•Expr,eof], [Expr→•Term,eof],
[Term→•Factor*Term,eof], [Term→•Factor*Term,-],
[Term→•Factor,eof], [Term→•Factor,-], [Factor→•id,eof],
[Factor→•id,-], [Factor→•id,-]}
S6: {[Term→Factor*•Term,eof],[Term→Factor*•Term,-],
[Term→•Factor*Term,eof], [Term→•Factor*Term,-],
[Term→•Factor,eof], [Term→•Factor,-], [Factor→•id,eof],
[Factor→•id,-], [Factor→•id,-]}
S7: {[Expr→Term-Expr•,eof]}
S8: {[Term→Factor*Term•,eof], Term→Factor*Term•,-]}

Table Construction

- 1. Construct the collection of sets of LR(1) items.
- 2. State i of the parser is constructed from state j.
  - If [A→α•aβ,b] in state i, and goto(i,a)=j, then set action[i,a] to "shift j".
  - If [A→α•,a] in state i, then set action[i,a] to "reduce A→α".
  - If [Goal→A•,eof] in state i, then set action[i,eof] to "accept".
  - If goto[i,A]=j then set goto[i,A] to j.
- 3. All other entries in action and goto are set to "error".

Example: The Table (slide 4 of 4)

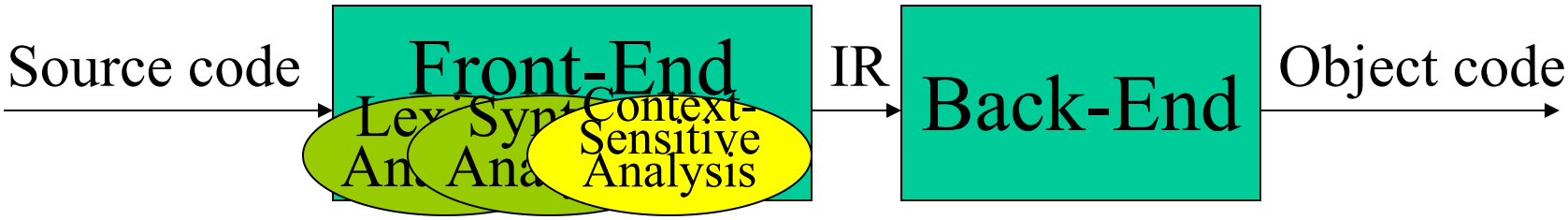
Goal → *Expr*
Expr → *Term* - *Expr*
Expr → *Term*
Term → *Factor* * *Term*
Term → *Factor*
Factor → *id*

STATE	ACTION				GOTO		
	id	-	*	eof	Expr	Term	Factor
0	S 4				1	2	3
1				Accept			
2		S 5		R 3			
3		R 5	S 6	R 5			
4		R 6	R 6	R 6			
5	S 4				7	2	3
6	S 4					8	3
7				R 2			
8		R 4		R 4			

Further remarks

- If the algorithm defines an entry more than once in the ACTION table, then the grammar is not LR(1).
- Other table construction algorithms, such as LALR(1) or SLR(1), produce smaller tables, but at the cost of larger space requirements.
- **`yacc`** can be used to convert a context-free grammar into a set of tables using LALR(1) (see **`% man yacc`**)
- **In practice**: "…the compiler-writer does not really want to concern himself with how parsing is done. So long as the parse is done correctly, …, he can live with almost any reliable technique…" [J.J.Horning from "Compiler Construction: An Advanced Course", Springer-Verlag, 1976]

Lecture 11: Context Sensitive Analysis



(from last lectures) :

- After Lexical & Syntax analysis we have determined that the input program is a valid sentence in the source language.
- The outcome from syntax analysis is a parse tree: a graphical representation of how the start symbol of a grammar derives a string in the language.
- *But, have we finished with analysis & detection of all possible errors?*
- Today's lecture: attribute grammars

What is wrong with the following?

```
foo(a,b,c,d)
  int a,b,c,d;
{ … }

lala()
{
  int f[3],g[0],h,i,j,k;
  char *p;
  foo(h,i,"ab",j,k);
  k=f*i+j;
  h=g[7];
  printf("%s,%s",p,q);
  p=10;
}
```

I can count (at least) 6 errors!

All of which are beyond syntax!

These are not issues for the ***context-free*** grammar!

To generate code, we need to understand its meaning!

Beyond syntax: context sensitive questions

To generate code, the compiler needs to answer questions such as:

- Is **`x`** a scalar, an array or a function? Is **`x`** declared?
- Are there names that are not declared? Declared but not used?
- Which declaration of **`x`** does each use reference?
- Is the expression **`x-2*y`** type-consistent? (**type checking**: the compiler needs to assign a type to each expression it calculates)
- In **`a[i,j,k]`**, is **`a`** declared to have 3 dimensions?
- Who is responsible to allocate space for **`z`**? For how long its value needs to be preserved?
- How do we represent **`15`** in **`f=15`**?
- How many arguments does **`foo()`** take?
- Can **`p`** and **`q`** refer to the same memory location?

These are beyond a context-free grammar!

Type Systems

- A value's **type** is the set of properties associated with it. The set of types in a programming language is called **type system**. **Type checking** refers to assigning/inferring types for expressions and checking that they are used in contexts that are legal.
- Components of a type system:
  - Base or Built-in Types (e.g., numbers, characters, booleans)
  - Rules to: construct new types; determine if two types are equivalent; infer the type of source language expressions.
- Type checking: (depends on the design of the language and the implementation)
  - At compile-time: statically checked
  - At run-time: dynamically checked

+	**int**	**real**	**double**	**complex**
int	int	real	double	complex
real	real	real	double	complex
double	double	double	double	illegal
complex	complex	complex	illegal	complex

- If the language has a 'declare before use' policy, then inference is not difficult. The real challenge is when no declarations are needed.

Context-sensitive questions

The questions in Slide 3 are part of context-sensitive analysis:

- answers depend on values, not syntax.
- questions and answers involve non-local information.
- answers may involve computation.

How can we answer these questions?

- Use formal methods:
  - use context-sensitive grammars: many important questions would be difficult to encode in a Context-Sensitive Grammar and the set of rules would be too large to be manageable (e.g., the issue of declaration before use).
  - attribute grammars.
- Use *ad hoc* techniques:
  - Symbol tables and *ad hoc*, syntax-directed translation using attribute grammars.

In scanning and parsing formalism won; here it is a different story!

Attribute Grammars

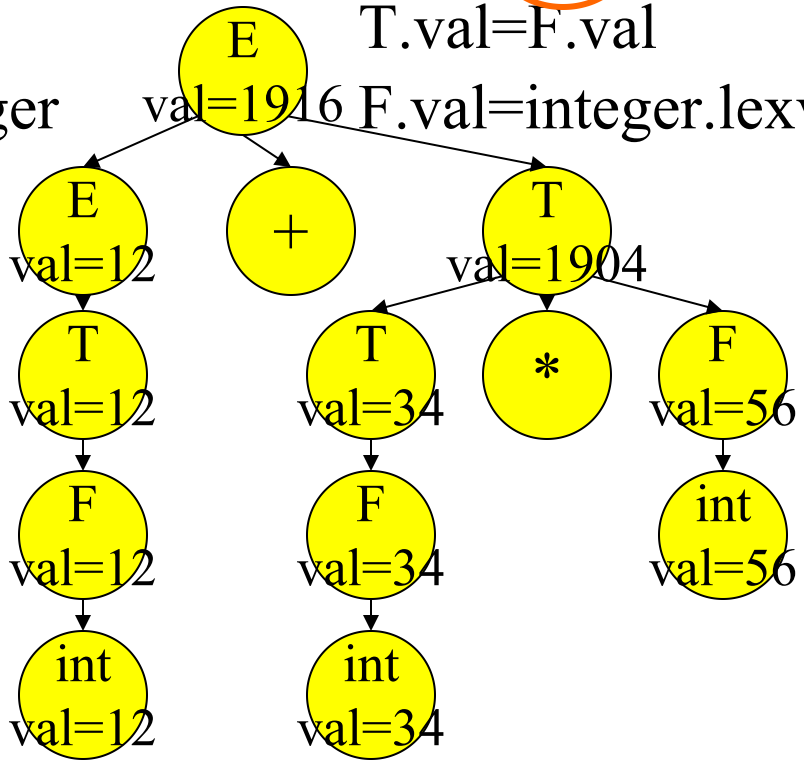
- Idea:
  - annotate each grammar symbol with a set of values or attributes.
  - associate a semantic rule with each production rule that defines the value of each attribute in terms of other attributes.
- Attribute Grammars (a formalisation by Knuth):
  - A context-free grammar augmented with a set of (semantic) rules.
  - Each symbol has a set of values (or attributes).
  - The rules specify how to compute a value for each attribute.

Example

(semantic rules to calculate the value of an expression)

$G \rightarrow E$	print(E.val)
$E \rightarrow E_1+T$	$E.val=E_1.val+T.val$
$\mid T$	E.val=T.val
$T \rightarrow T_1*F$	$T.val=T_1.val \times F.val$
$\mid F$	T.val=F.val
$F \rightarrow integer$	F.val=integer.lexval

NB: we label identical terms uniquely!



Evaluation order:
start from the leaves
and proceed bottom-up!

Attributes

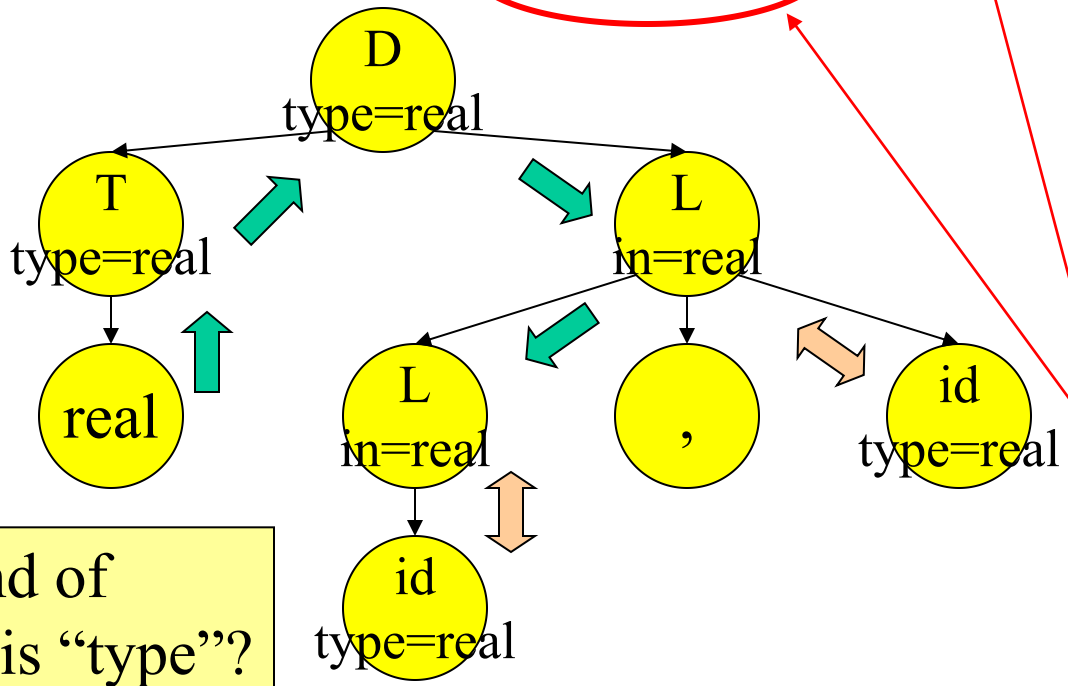
Dependences between attributes:

- Synthesised attributes: derive their value from constants and children.
  - Only synthesised attributes: S-attribute grammars (can be evaluated in one bottom-up pass; cf. with the example before).
- Inherited attributes: derive their value from parent, constants and siblings.
  - Directly express context.

***Issue**: semantic rules of one node of the parse tree define dependencies with semantic rules of other nodes. What about the evaluation order in more complex cases? (we'll come to this later…)*

Example (variable declarations) cf. Aho1, p.283, Ex: 5.3

$D \rightarrow T\ L$	L.in=T.type
$T \rightarrow$ int	T.type=integer
| real	T.type=real
$L \rightarrow L_1$, id	L_1.in=L.in; id.type=L.in
| id	id.type=L.in

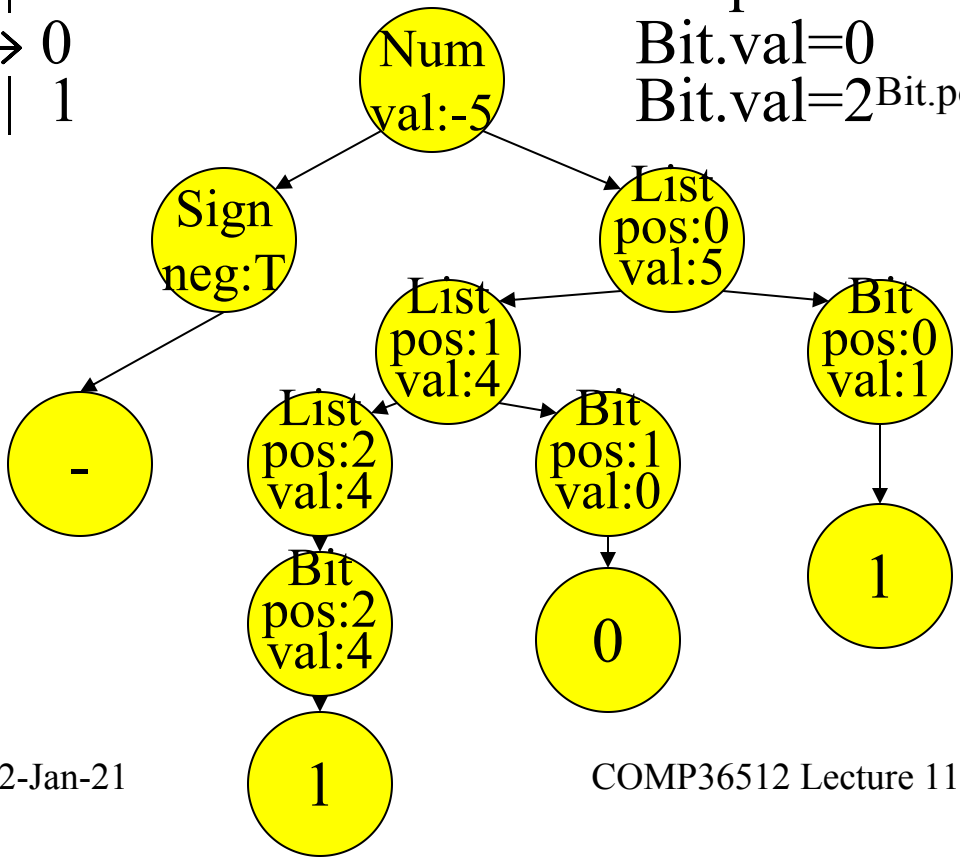


What kind of attribute is "type"? What about "in"?

In practice, we don't need to do all this with variable declarations! We can use a symbol table (for more see lecture 11). There, all we need to do is to add a new entry into a symbol table. E.g.: *addtype(id.type, L.in)*

Another Example (signed binary numbers)

Number→Sign List	List.pos=0; If Sign.neg then Number.val=-List.val else Number.val=List.val
Sign→ +	Sign.neg=false
| −	Sign.neg=true
List→List$_1$ Bit	List$_1$.pos=List.pos+1; Bit.pos=List.pos; List.val= List$_1$.val+Bit.val
| Bit	Bit.pos=List.pos; List.val=Bit.val
Bit → 0	Bit.val=0
| 1	Bit.val=$2^{\text{Bit.pos}}$



- Number and Sign have one attribute each. List and Bit have two attributes each.
- Val and neg are synthesised attributes; pos is an inherited attribute.
- What about an evaluation order? Knuth suggested a data-flow model of evaluation with independent attributes first. Implies a dependency graph!

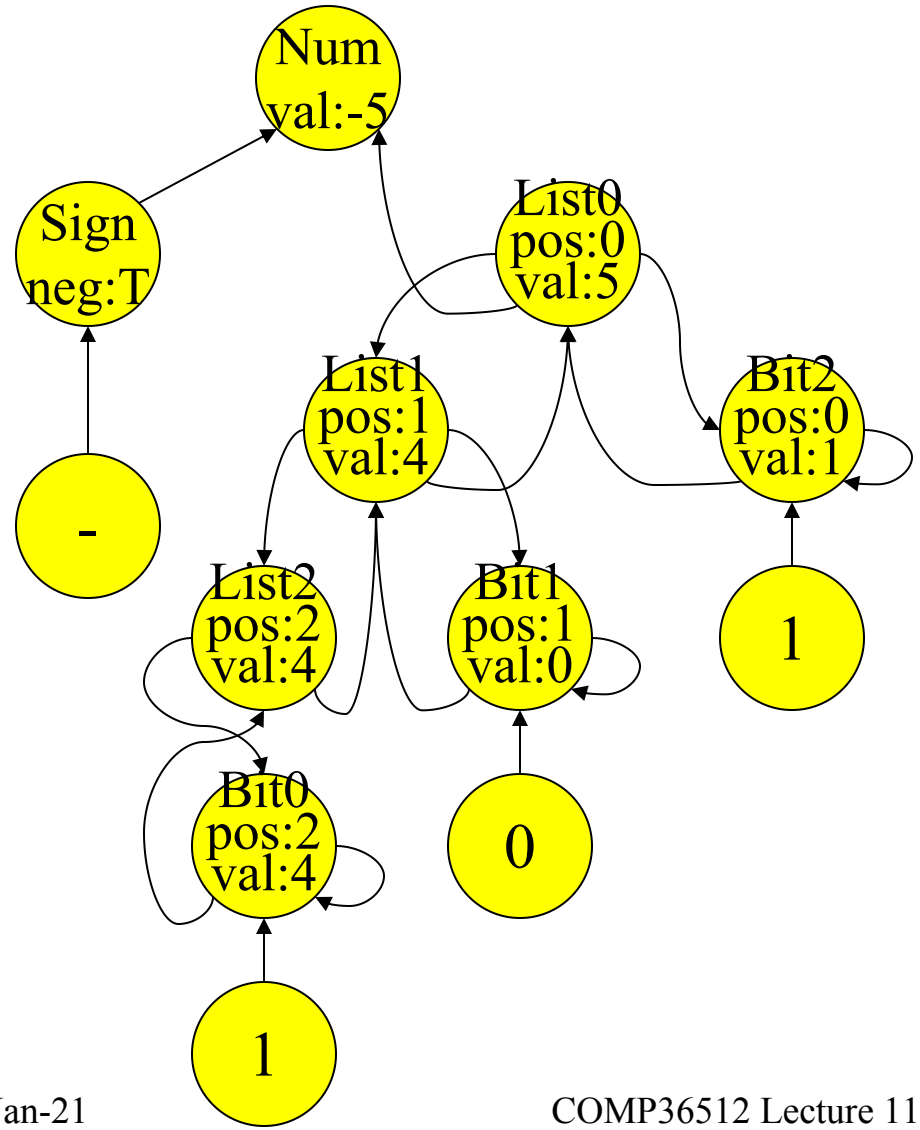
Attribute dependence graph

If an attribute at one node depends on an attribute of another node then the former must be evaluated before the latter.

Attribute Dependence Graph:

- Nodes represent attributes; edges represent the flow of values.
- Graph is specific to the parse tree (can be built alongside and its size is related to the size of the parse tree).
- Evaluation order:
  - Parse tree methods: use a topological sort (any ordering of the nodes such that edges go only from the earlier nodes to the later nodes: sort the graph, find independent values, then walk along graph edges): cyclic graph fails!
  - Rule-based methods: the order is statically predetermined.
  - Oblivious methods: use a convenient approach independent of semantic rules.
- Problem: how to deal with cycles.
- Another issue: complex dependencies.

Binary Numbers example: dependency graph



A topological order:

1. Sign.neg
2. List0.pos
3. List1.pos
4. List2.pos
5. Bit0.pos
6. Bit1.pos
7. Bit2.pos
8. Bit0.val
9. List2.val
10. Bit1.val
11. List1.val
12. Bit2.val
13. List0.val
14. Num.val

Using attribute grammars in practice

- Generic attribute grammars have seen limited practical use:
  - Complex local context handling is easy.
  - Non-local context-sensitive issues need a lot of supporting rules. Naturally, one tends to think about global tables… all this takes time…
- Still, there is some research and they have seen applications in structured editors, representation of structured documents, etc... (e.g, see attribute grammars and XML)
- In practice, a simplified idea is used:

  ***ad hoc* syntax directed translation**:

  - **S-attribute grammars**: only synthesized attributes are used. Values flow in only one direction (from leaves to root). It provides an evaluation method that works well with bottom-up parsers (such as those described in previous lectures).
  - **L-attribute grammars**: both synthesized and inherited attributes can be used, but there are certain rules that limit the use of inherited attributes.

Example (S-attribute grammar) (lect. 9, slide 5)

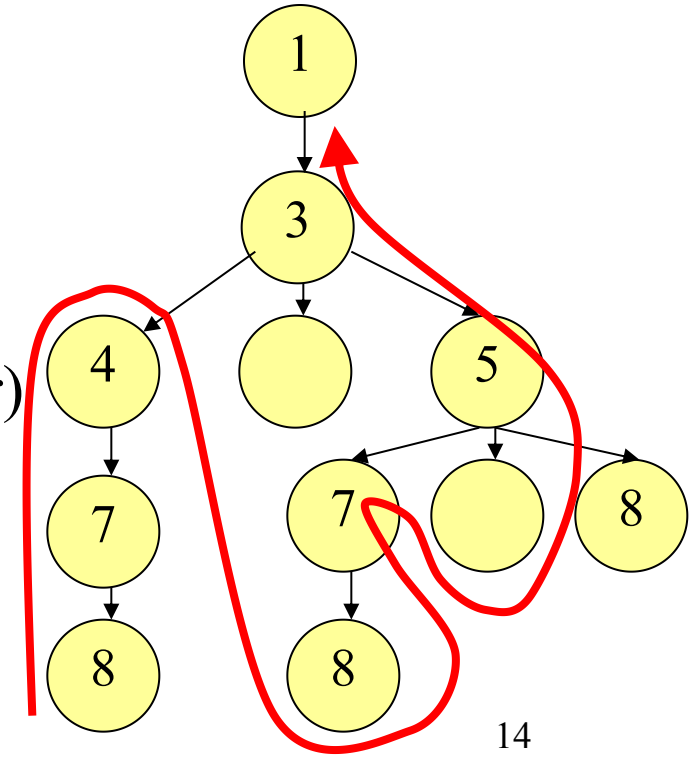
1. *Goal* → *Expr*
2. *Expr* → *Expr* + *Term*
3. | *Expr* – *Term*
4. | *Term*
5. *Term* → *Term* * *Factor*
6. | *Term* / *Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

Using LR parsing (see table in lecture 9, slide 5), the reductions for the string 19-8*2 make use of the rules (in this order): 8, 7, 4, 8, 7, 8, 5, 3, 1. The attributes of the left-hand side are easily computed in this order too!

Assume we specify for each symbol the attribute *val* to compute the value of an expression (semantic rules are omitted, but are obvious).

Parse tree:

The red arrow shows the flow of values, which is the order that rules (cf. number) are discovered!



Finally…

- Another example: (signed binary numbers again)

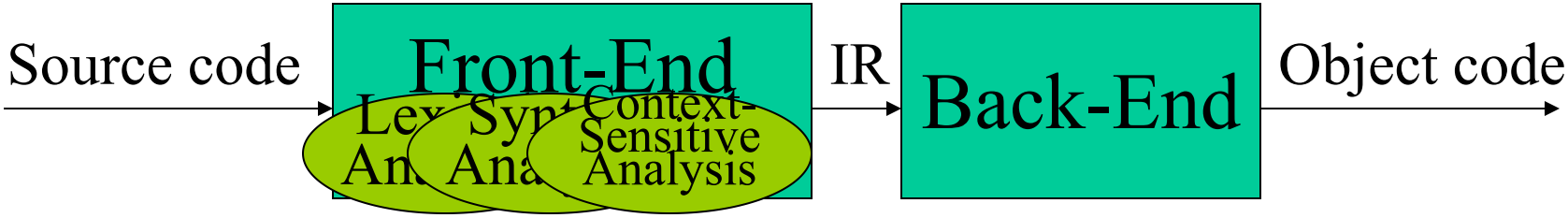
Number→Sign List	Number = Sign.neg*List.val
Sign→ +	Sign.neg=1
| –	Sign.neg=–1
List→List$_1$ Bit	List.val=2*List1.val+Bit.val
| Bit	List.val=Bit.val
Bit → 0	Bit.val=0
| 1	Bit.val=1

- An L-attribute grammar: (see Aho2, example 5.8)

T → F Q	Q.inh = F.val
Q → * F Q$_1$	Q$_1$.inh = Q.inh * F.val

- Much of the (detailed) information in the parse tree is not needed for subsequent phases, so a condensed form (where non-terminal symbols are removed) is used: the **abstract syntax tree**.
- **Reading:** Aho2, Chapter 5; Aho1, pp.279-287; Grune pp.194-210; Hunter pp.147-152 (too general), Cooper Sec.4.1-4.4 (good discussion).

Lecture 12: Intermediate Representations



- (from previous lectures) The Front-End analyses the source code.

Today's lecture: Intermediate Representations (IR):

- The Intermediate Representation encodes all the knowledge that the compiler has derived about the source program.

To follow: Back-End transforms the code, as represented by the IR, into target code.

Recall: Middle-End, may transform the code represented by the IR into equivalent code that may perform more efficiently. Typically:

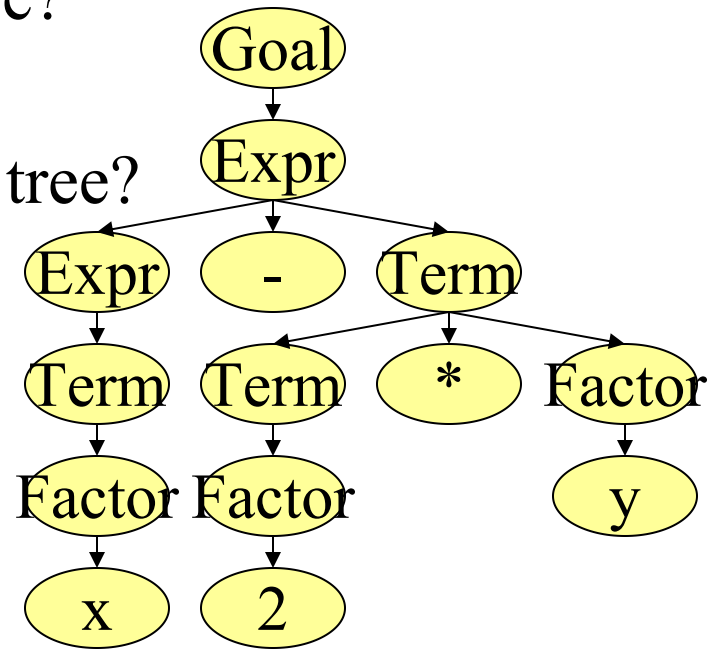
About IRs, taxonomy, etc...

- Why use an intermediate representation?
  - To facilitate retargeting.
  - To enable machine independent-code optimisations or more aggressive code generation strategies.
- Design issues:
  - ease of generation; ease of manipulation; cost of manipulation; level of abstraction; size of typical procedure.
  - Decisions in the IR design have major effects on the speed and effectiveness of compiler.
- A useful distinction:
  - Code representation: AST, 3-address code, stack code, SSA form.
  - Analysis representation (may have several at a time): CFG, ...
- Categories of IRs by structure:
  - Graphical (structural): trees, DAGs; used in source to source translators; node and edge structures tend to be large.
  - Linear: pseudo-code for some abstract machine; large variation.
  - Hybrid: combination of the above.

There is no universally good IR. The right choice depends on the goals of the compiler!

From Parse Trees to Abstract Syntax Trees

- Why we don't want to use the parse tree?
  - Quite a lot of unnecessary information...
- How to convert it to an abstract syntax tree?
  - Traverse in postorder (postfix)
  - Use *mkleaf* and *mknode* where appropriate
  - Match action with grammar rule (Aho1 pp.288-289; Aho2, 5.3.1)

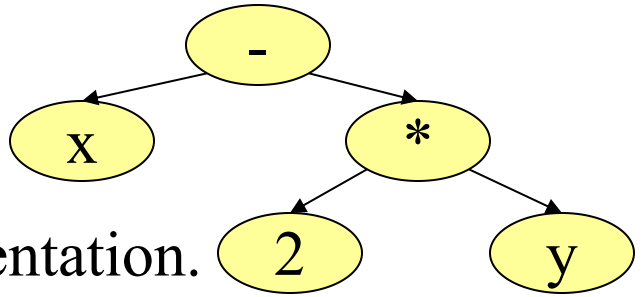


1. *Goal* → *Expr*
2. *Expr* → *Expr* + *Term*
3. | *Expr* – *Term*
4. | *Term*
5. *Term* → *Term* * *Factor*
6. | *Term* / *Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

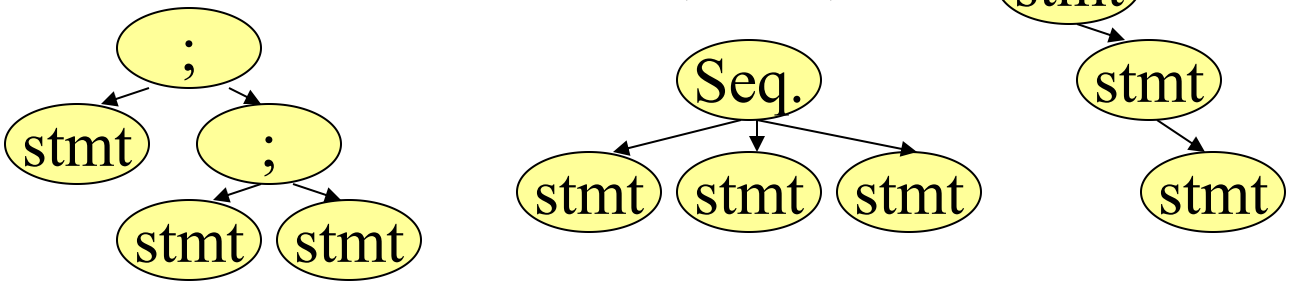
Abstract Syntax Trees

An Abstract Syntax Tree (AST) is the procedure's parse tree with the non-terminal symbols removed.

Example: $x - 2*y$



- The AST is a near source-level representation.
- Source code can be easily generated: perform an *inorder* treewalk - first the left subtree, then the root, then the right subtree - printing each node as visited.
- Issues: traversals and transformations are pointer-intensive; generally memory-intensive.
- Example: *Stmt_sequence* → *stmt; Stmt_sequence* | *stmt*
  - At least 3 AST versions for *stmt; stmt; stmt*



AST real-world example (dHPF)

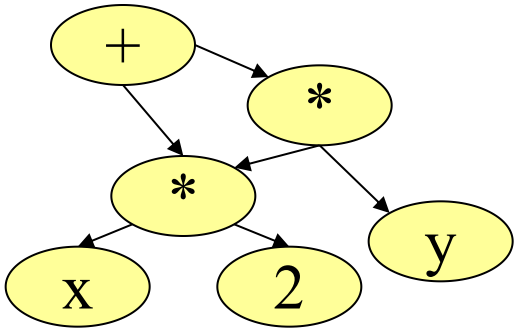
```
      PROGRAM MAIN
      REAL A(100), X
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P

      FORALL (i=1:100) A(i) = X+1
        CALL FOO(A)
      END

      SUBROUTINE FOO(X)
      REAL X(100)
!HPF$ INHERIT X

      IF (X(1).EQ.0) THEN
        X = 1
      ELSE
        X = X + 1
      END IF
      RETURN
      END
```

```
(1[GLOBAL]
   ((2[PROG_HEDR]
      (3[VAR_DECL]
       4[PROCESSORS_STMT]
       5[DISTRIBUTE_DECL]
       (6[FORALL_STMT]
          (7[ASSIGN_STAT]
           8[CONTROL_END]
          )
          NULL
       )
       9[PROC_STAT]
       10[CONTROL_END]
      )
      NULL
   )
   (11[PROC_HEDR]
      (12[VAR_DECL]
       13[INHERIT_DECL]
       (14[LOGIF_NODE]
          (15[ASSIGN_NODE]
           16[CONTROL_END]
          )
          (17[ASSIGN_NODE]
           18[CONTROL_END]
          )
       )
       19[RETURN_STAT]
       20[CONTROL_END]
      )
      NULL
   )
   NULL
)
```

Directed Acyclic Graphs (DAGs)

A DAG is an AST with a unique node for each value

Example: The DAG for x*2+x*2*y



Powerful representation, encodes redundancy; but difficult to transform, not useful for showing control-flow.

Construction:

- Replace constructors used to build an AST with versions that remember each node constructed by using a table.
- Traverse the code in another representation.

Exercise: Construct the DAG for x=2*y+sin(2*x); z=x/2

Auxiliary Graph Representations

The following can be useful for analysis:

- **Control-Flow Graph** (CFG): models the way that the code transfers control between blocks in the procedure.
  - Node: a single basic block (a maximal straight line of code)
  - Edge: transfer of control between basic blocks.
  - (Captures loops, if statements, case, goto).
- **Data Dependence Graph**: encodes the flow of data.
  - Node: program statement
  - Edge: connects two nodes if one uses the result of the other
  - Useful in examining the legality of program transformations
- **Call Graph**: shows dependences between procedures.
  - Useful for interprocedural analysis.

Examples / Exercises

Draw the control-flow graph of the following:

```
Stmtlist1
if (x=y)
   stmtlist2
   stmtlist1
else
   stmtlist3
   stmtlist1
stmtlist4


stmtlist1
while (x<k)
   stmtlist2
stmtlist3
```

Draw the data dependence graph of:

```
1. sum=0
2. done=0
3. while !done do
4. read j
5. if (j>0)
6.       sum=sum+j
7.       if (sum>100)
8.          done=1
9.       else
10.         sum=sum+1
11.      endif
12.endif
13.endwhile
14.print sum

what about:1. Do i=1,n
           2.    A(I)=a(3*I+10)
```

Draw the call graph of:

```
void a() { … b() … c() … f() … }
void b() { … d() … c() … }
void c() { … e() … }
void d() { … }
void e() { … b() … }
void f() { … d() …}
```

Three-address code

A term used to describe many different representations: each statement is a single operator and at most three operands.

Example: **`if (x>y) then z=x-2*y`** becomes:

```
   t1=load x
   t2=load y
   t3=t1>t2
   if not(t3) goto L
   t4=2*t2
   t5=t1-t4
   z=store t5
L: …
```

Array addresses have to be converted to single memory access (see lecture 13), e.g., A[I] will require a 'load I' and then 't1=I*sizeof(I); load A+t1'

Advantages: compact form, makes intermediate values explicit, resembles many machines.

Storage considerations (until recently compile-time space was an issue)

*x–2*y*

```
load  r1,y
loadi r2,2
mult r3,r2,r1
load r4,x
sub r5,r4,r3
```

Quadruples

```
load  1 y -
loadi 2 2 -
mult  3 2 1
load  4 x -
sub   5 4 3
```

(Indirect) Triples

```
load y
loadi 2
mult (1),(2)
load x
sub (4),(3)
```

Other linear representations

- Two address code is more compact: In general, it allows statements of the form x=x <op> y (single operator and at most two operands).

```
load  r1,y
loadi r2,2
mult r2,r2,r1
load r3,x
sub r3,r3,r2
```

- One address code (also called stack machine code) is more compact. Would be useful in environments where space is at a premium (has been used to construct bytecode interpreters for Java):

```
push  2
push  y
multiply
push  x
subtract
```

Some Examples

- 1: Simple back-end compiler:
  - Code: CFG + 3-address code
  - Analysis info: value DAG (represents dataflow in basic block)
- 2. Sun Compilers for SPARC:
  - Code: 2 different IRs
  - Analysis info: CFG+dependence graph+??
  - High-level IRs: linked list of triples
  - Low-level IRs: SPARC assembly like operations
- 3. IBM Compilers for Power, PowerPC:
  - Code: Low-level IR
  - Analysis info: CFG + value graph + dataflow graphs.
- 4. dHPF compiler:
  - Code: AST
  - Analysis info: CFG+SSA+Value DAG+Call Graph
    (**SSA** stands for Static Single Assignment: same as 3-address code, but all variables have a distinct name every time they are defined; if they are defined in different control paths the statement x3=Ø(x1,x2) is used to combine the two definitions)

Remark:

- Many kinds of IR are used in practice. Choice depends...
- ***But… representing code is half the story!***
  - ***Symbol tables, constants table, storage map.***

Symbol Tables: The key idea

- Introduce a central repository of facts:
  - symbol table or sets of symbol tables
- Associate with each production a snippet of code that would execute each time the parser reduces that production - action routines. Examples:
  - Code that checks if a variable is declared prior to use (on a production like *Factor*→*id*)
  - Code that checks that each operator and its operands are type-compatible (on a production like *Term* →*Term***Factor*)
- Allowing arbitrary code provides flexibility.
- Evaluation fits nicely with LR(1) parsing.
- Symbol tables are retained across compilation (carry on for debugging too)

What information is stored in the symbol table?

What items to enter in the symbol table?

- Variable names; defined constants; procedure and function names; literal constants and strings; source text labels; compiler-generated temporaries.

What kind of information might the compiler need about each item:

- textual name, data type, declaring procedure, storage information. Depending on the type of the object, the compiler may want to know list of fields (for structures), number of parameters and types (for functions), etc…

In practice, many different tables may exist.

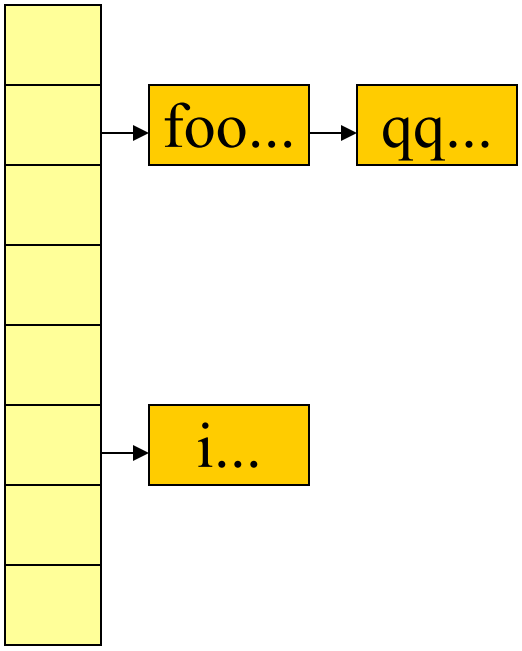
Symbol table information is accessed frequently:
hence, efficiency of access is critical!

Organising the symbol table

- Linear List:
  - Simple approach, has no fixed size; but inefficient: a lookup may need to traverse the entire list: this takes *O(n).*
- Binary tree:
  - An unbalanced tree would have similar behaviour as a linear list (this could arise if symbols are entered in sorted order).
  - A balanced tree (path length is roughly equal to all its leaves) would take $O(log_2 n)$ probes per lookup (worst-case). Techniques exist for dynamically rebalancing trees.
- Hash table:
  - Uses a hash function, *h*, to map names into integers; this is taken as a table index to store information. Potentially *O(1),* but needs inexpensive function, with good mapping properties, and a policy to handle cases when several names map to the same single index.

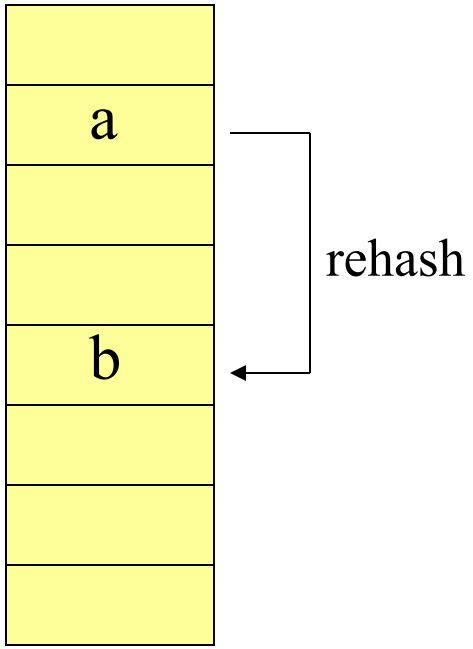
Bucket hashing (open hashing)

- A hash table consisting of a fixed array of *m* pointers to table entries.
- Table entries are organised as separate linked lists called buckets.
- Use the hash function to obtain an integer from *0* to *m-1*.
- As long as *h* distributes names fairly uniformly (and the number of names is within a small constant factor of the number of buckets), bucket hashing behaves reasonably well.



Linear Rehashing (open addressing)

- Use a single large table to hold records. When a collision is encountered, use a simple technique (i.e., add a constant) to compute subsequent indices into the table until an empty slot is found or the table is full. If the constant is relatively prime to the table size, this, eventually, will check every slot in the table.
- Disadvantages: too many collisions may degrade performance. Expanding the table may not be straightforward.



If $h(a)=h(b)$ rehash (say, add 3).

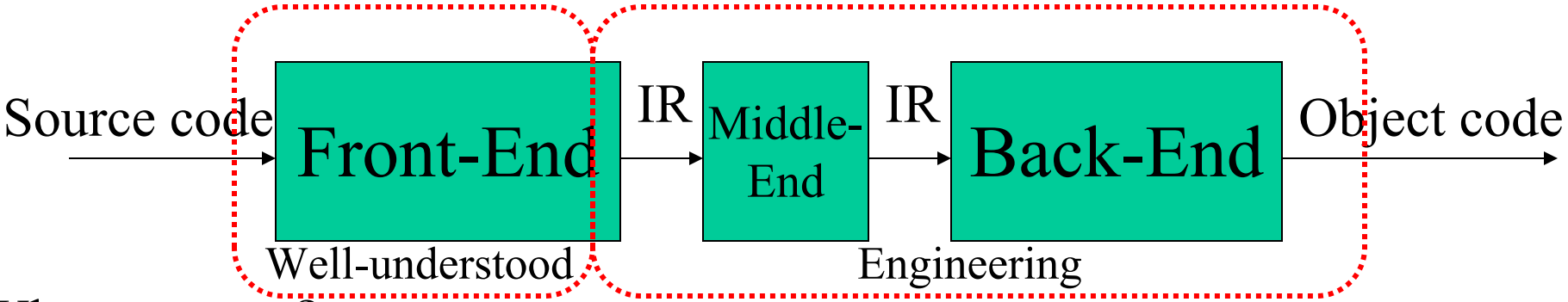
Other issues

- Choosing a good hash function is of paramount importance:
  - take the hash key in four-byte chunks, XOR the chunks together and take this number modulo 2048 (this is the symbol table size). What is the problem with this?
  - See the universal hashing function (Cormen, Leiserson, Rivest), Knuth's multiplicative function… This is one of those cases we should pay attention to theory!
- Lexical scoping:
  - Many languages introduce independent name scopes:
    - C, for example, may have global, static (file), local and block scopes.
    - Pascal: nested procedure declarations
    - C++, Java: class inheritance, nested classes
    - C++, Java, Modula: packages, namespaces, modules, etc…
    - Namespaces allow two different entities to have the same name within the same scope: E.g.: In Java, a class and a method can have the same name (Java has six namespaces: packages, types, fields, methods, local variables, labels)
  - The problems:
    - at point x, which declaration of variable y is current?
    - as parser goes in and out of scopes, how does it track y? allocate and initialise a symbol table for each level!

Conclusion

- Many intermediate representations – there is no universally good one! A combination might be used!
- Representing code is half the story - Hash tables are used to store program names.
- Choice of an appropriate hash function is key to an efficient implementation.
- In a large system it may be worth the effort to create a flexible symbol table.
- Fully qualified names (i.e., file.procedure.scope.x) to deal with scopes is not a good idea (the extra work needed to build qualified names is not worth the effort).
- Reading: Reading: Aho2, pp. 85-100, 357-370; Aho1 pp.287-293; 429-440; pp.463-472. Cooper, Chapter 5 (excellent discussion).

Lecture 13: The Procedure Abstraction; Run-Time Storage Organisation



Where are we?

- We crossed the dividing line between the application of well-understood technology and fundamental issues of design and engineering. The complications of compiling begin to emerge!
- The second half contains more open problems, more challenges, and more gray areas that the first half
  - This is compilation as opposed to parsing or translation (engineering as opposed to theory: imperfection, trade-off, constraints, optimisation)
  - Needs to manage target machine resources
  - This is where legendary compilers are made...

Today's lecture:

- The Procedure Abstraction and Run-Time Storage Organisation

The Procedure

- **Procedures are the key to building large systems; they provide:**
  - Control abstraction: well-defined entries & exits.
  - Name Space: has its own protected name space.
  - External Interface: access is by name & parameters.
- **Requires system wide-compact:**
  - broad agreement on memory layout, protection, etc…
  - must involve compiler, architecture, OS
- **Establishes the need for private context:**
  - create a run-time "record" for each procedure to encapsulate information about control & data abstractions.
- **Separate compilation:**
  - allows us to build large systems; keeps compile-time reasonable

The Procedure: A more abstract view

- A procedure is a collection of fictions.
- Underlying hardware supports little of this fiction:
  - well-defined entries and exits: mostly name-mangling
  - call/return mechanism: often done in software
  - name space, nested scopes: hardware understands integers!
  - interfaces: need to be specified.
- The procedure abstraction is a deliberate deception, produced collaboratively by the OS & the compiler.

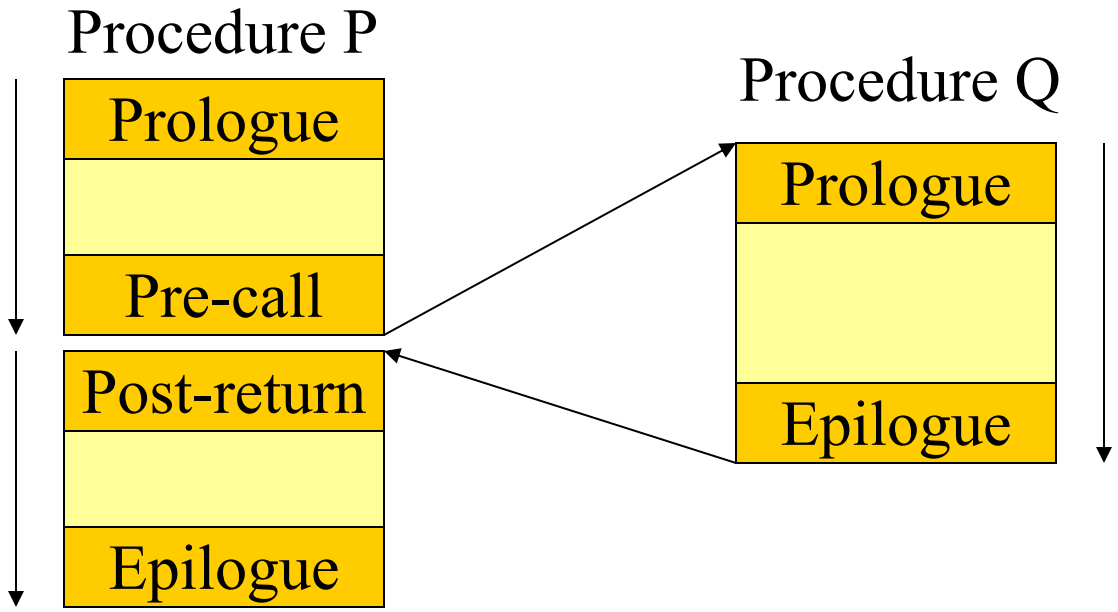
One view holds that computer science is simply the art of realising successive layers of abstraction!

The linkage convention

Procedures have well-defined control-flow behaviour:

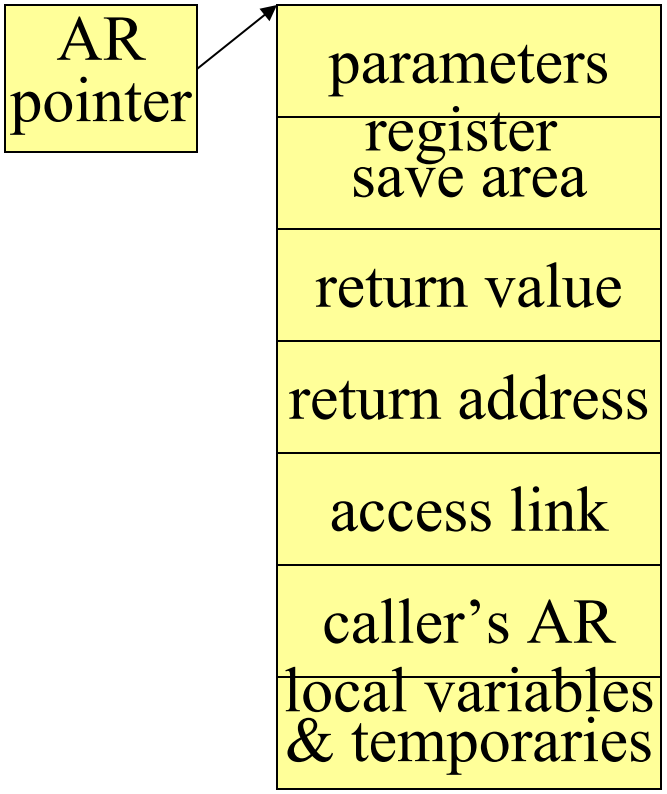
- A protocol for passing values and program control at procedure call and return is needed.
- The linkage convention ensures that procedures inherit a valid run-time environment and that they restore one for their parents.

- Linkages execute at run-time.
- Code to make the linkage is generated at compile-time.



Storage Organisation: Activation Records

- Local variables require storage during the lifetime of the procedure invocation at run-time.
- The compiler arranges to set aside a region of memory for each individual call to a procedure (run-time support): activation record:



Activation Records (ARs here) are also known as stack frames.

In general, the compiler is free to choose any convention for the AR. The manufacturer may want to specify a standard for the architecture.

→ Address to resume caller

→ Help with non-local access

→ Pointer to caller's activation record

Procedure linkages

(the procedure linkage convention is a machine-dependent contract between the compiler, the OS and the target machines to divide clearly responsibility)

Caller (pre-call):
- allocate AR
- evaluate and store parameters
- store return address
- store self's AR pointer
- set AR pointer to child
- jump to child

Callee (prologue):
- save registers, state
- extend AR for local data
- get static data area base address
- initialise local variables
- fall through to code

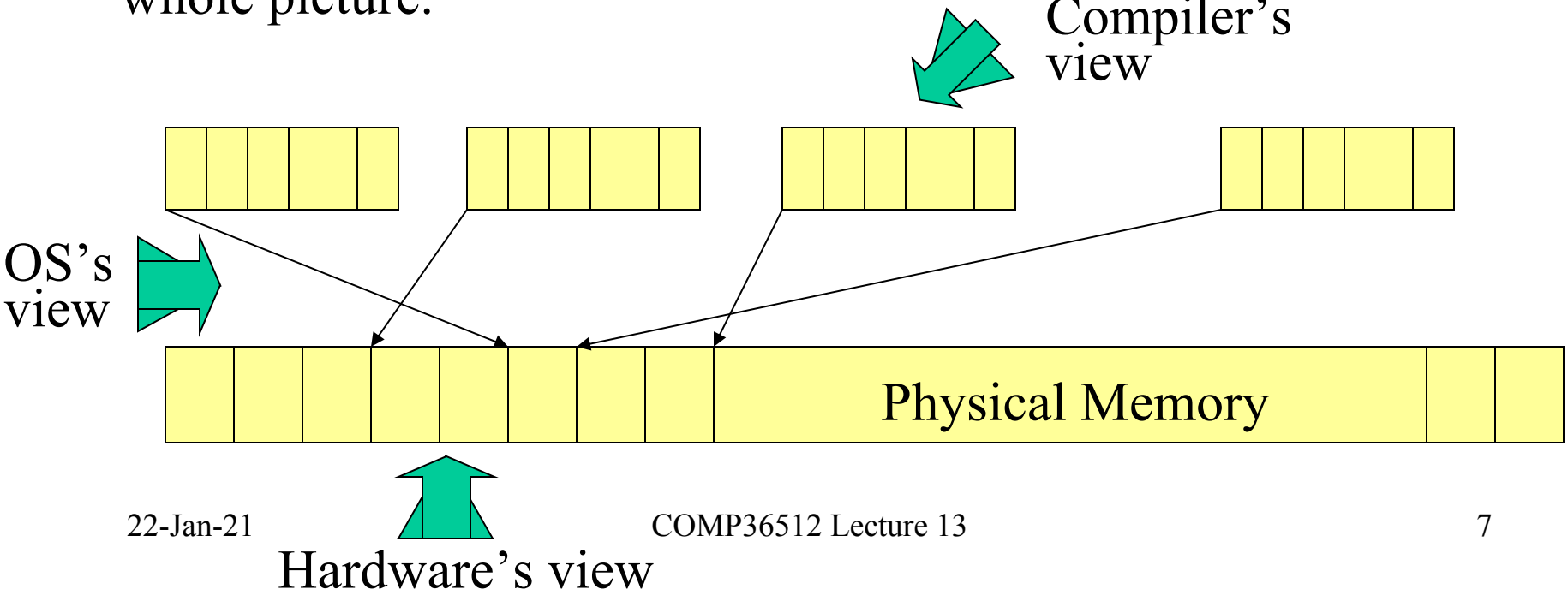
Callee (epilogue):
- store return value
- restore registers, state
- unextend basic frame
- restore parent's AR pointer
- jump to return address

Caller (post-return):
- copy return value
- deallocate callee's AR
- restore parameters (if used for call-by reference)

Placing run-time data structures

Single logical address space:

- Code, static, and global data have known size.
- Heap & stack grow towards each other.
- From the compiler's perspective, the logical address space is the whole picture.



Activation Record Details

- How does the compiler find the variables?
  - They are offsets from the AR pointer.
  - Variable length-data: if AR can be extended, put it below local variables; otherwise put on the heap.
- Where do activation records live?
  - If it makes no calls (leaf procedure - hence, only one can be active at a time), AR can be allocated statically.
  - Place in the heap, if it needs to be active after exit (e.g., may return a pointer that refers to its execution state).
  - Otherwise place in the stack (this implies: lifetime of AR matches lifetime of invocation and code normally executes a “return”).
  - (in decreasing order of efficiency: static, stack, heap)

Run-time storage organisation

The compiler must ensure that each procedure generates an address for each variable that it references:

- Static and Global variables:
  - Addresses are allocated statically (at compile-time). (relocatable)
- Procedure local variables:
  - Put them in the activation record if: sizes are fixed and values are not preserved.
- Dynamically allocated variables:
  - Usually allocated and deallocated explicitly.
  - Handled with pointers.

Establishing addressability

- Local variables of current procedure:
  - If it is in the AR: use AR pointer and load as offset.
  - If in the heap: store in the AR a pointer to the heap (double indirection).
  - (both the above need offset information)
  - If in a register: well, it is there!
- Global and static variables:
  - Use a relocatable (by the OS's loader) label (no need to emit code to determine address at run-time).
- Local variables of other procedures:
  - Need to retrieve information from the "other" procedure's AR.

Addressing non-local data

In a language that supports nested lexical scopes, the compiler must provide a mechanism to map variables onto addresses.

- The compiler knows current level of lexical scope and of variable in question and offset (from the symbol table).
- Needs code to:
  - Track lexical ancestry (not necessarily the caller) among ARs.
  - Interpret difference between levels of lexical scope and offset.
- Two basic mechanisms:
  - Access links
  - Global display.

Access Links

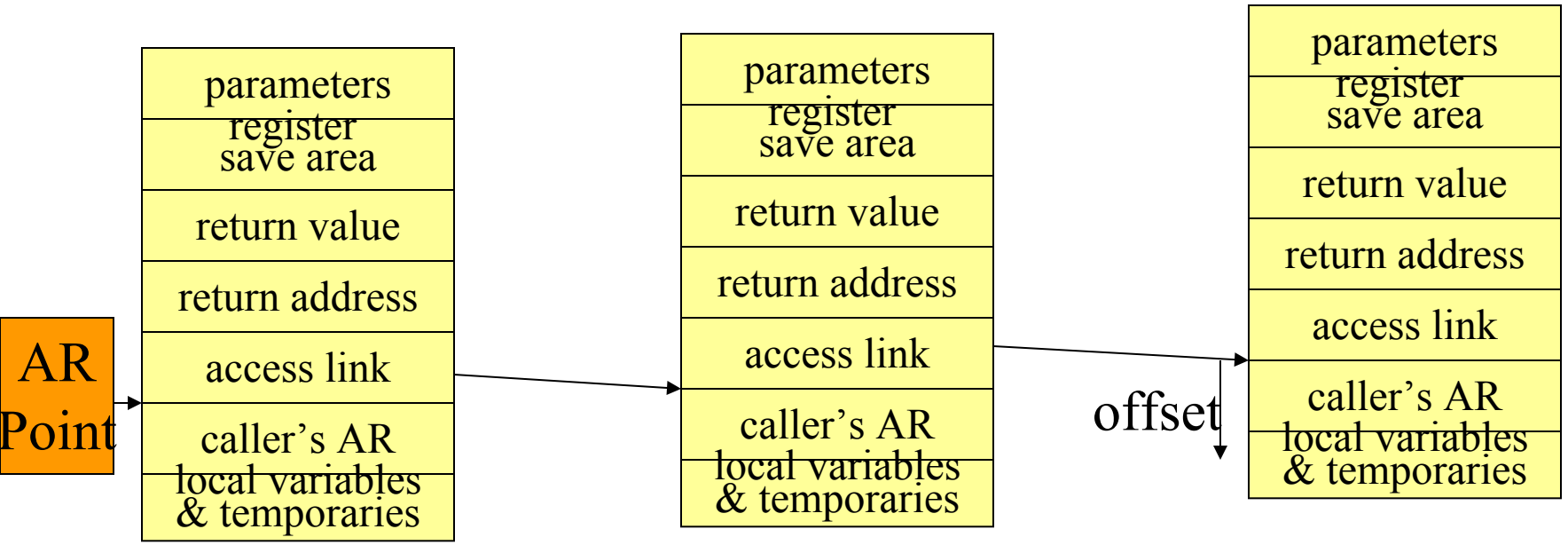
Idea: Each AR contains a pointer to its lexical ancestor.

Compiler needs to emit code to find lexical ancestor (if caller's scope=callee's scope+1 then it is the caller; else walk through the caller's ancestors)

Cost of access depends on depth of lexical nesting. Example:

(current level=2): needs variable at level=0, offset=16:

load r1,(ARP-4); load r1,(r1-4); load r2,(r1+16)



Global Display

Idea: keep a global array to hold ARPs for each level.

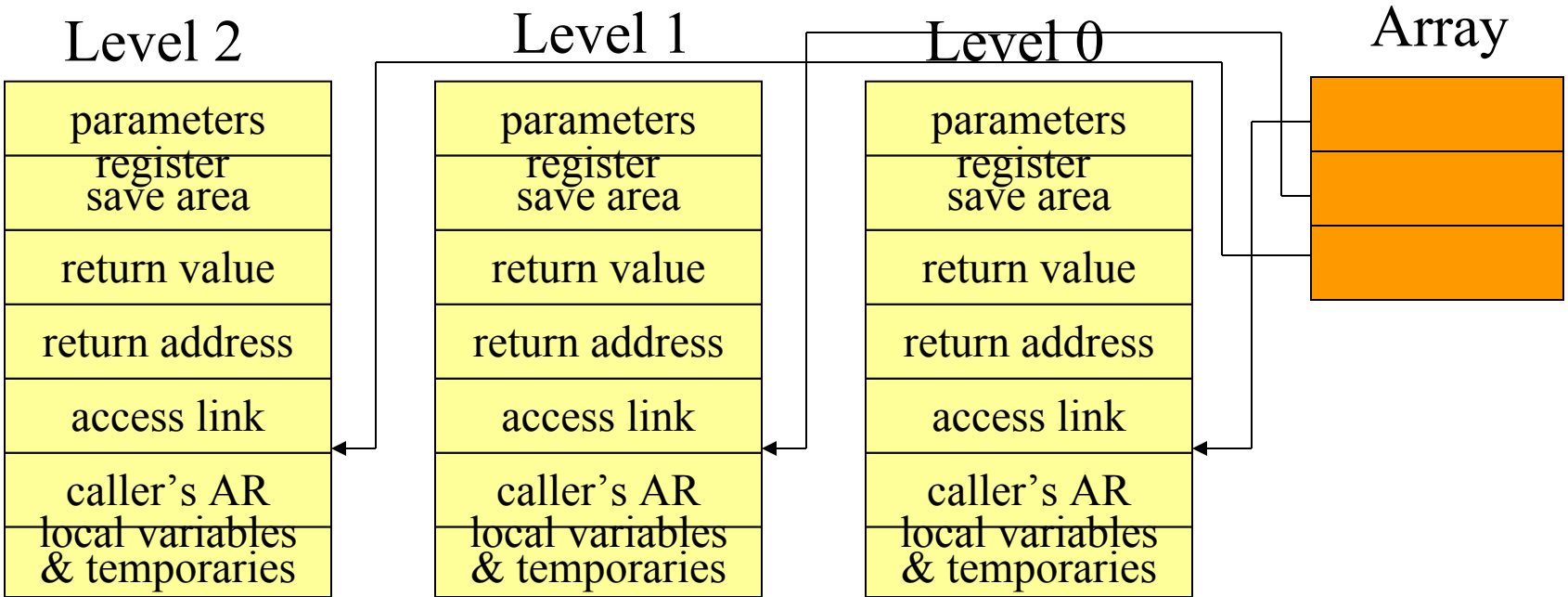
Compiler needs to emit code (when calling and returning from a procedure) to maintain the array.

Cost of access is fixed (table lookup + AR). Example:

(current level=2): needs variable at level=0, offset=16:

load r1,(DISPLAY_BASE+0); load r2,(r1+16)

Display vs access links trade-off. conventional wisdom: use access links when tight on registers; display when lots of registers.



Other (storage related) issues

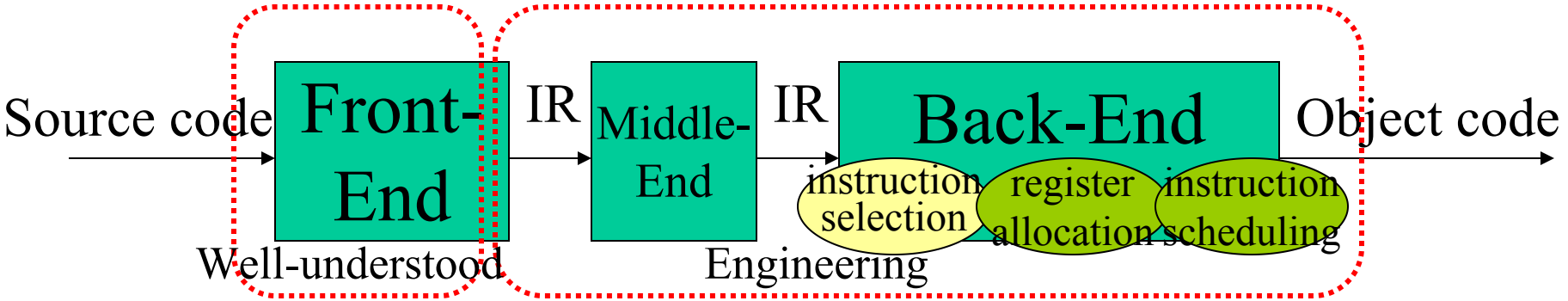
- Target machines may have requirements on where data items can be stored (e.g., 32-bit integers begin on a full word boundary). The compiler should order variables to take into account this.
- Cache performance: if two variables are used in near proximity in the code, the compiler needs to ensure that they can reside in the cache at the same time. Complex to consider more variables.
- Conventional wisdom: tight on registers: use access links; lots of registers: use global display.
- Memory to memory model vs register to register model.
- Managing the heap: first-fit allocation with several pools for common sizes (usually powers of 2 up to page size).
  - Implicit deallocation: reference counting (track the number of outstanding pointers referring to an object); garbage collection (when there is no space, stop execution and discover objects that can be reached from pointers stored in program variables; unreachable space is recycled).
- Object-oriented languages have more complex name spaces.

Finally…

- The compiler needs to emit code for each call to a procedure to take into account (at run-time) procedure linkage.
- The compiler needs to emit code to provide (at run-time) addressability for variables of other procedures.
- **Inlining**: the compiler can avoid some of the problems related to procedures by substituting a procedure call with the actual code for the procedure. There are advantages from doing this, but it may not be always possible (can you see why?) and there are disadvantages too.
- Reading:

  Aho2 Sections 7.1, 7.2, 7.3 (skim through the rest of Chapter 7); Aho1 pp.389-423; Hunter, Chapter 7; Cooper, Chapter 6.
- What is left to discuss:
  - Instruction selection/code generation; register allocation; instruction scheduling.
  - Code optimisations.

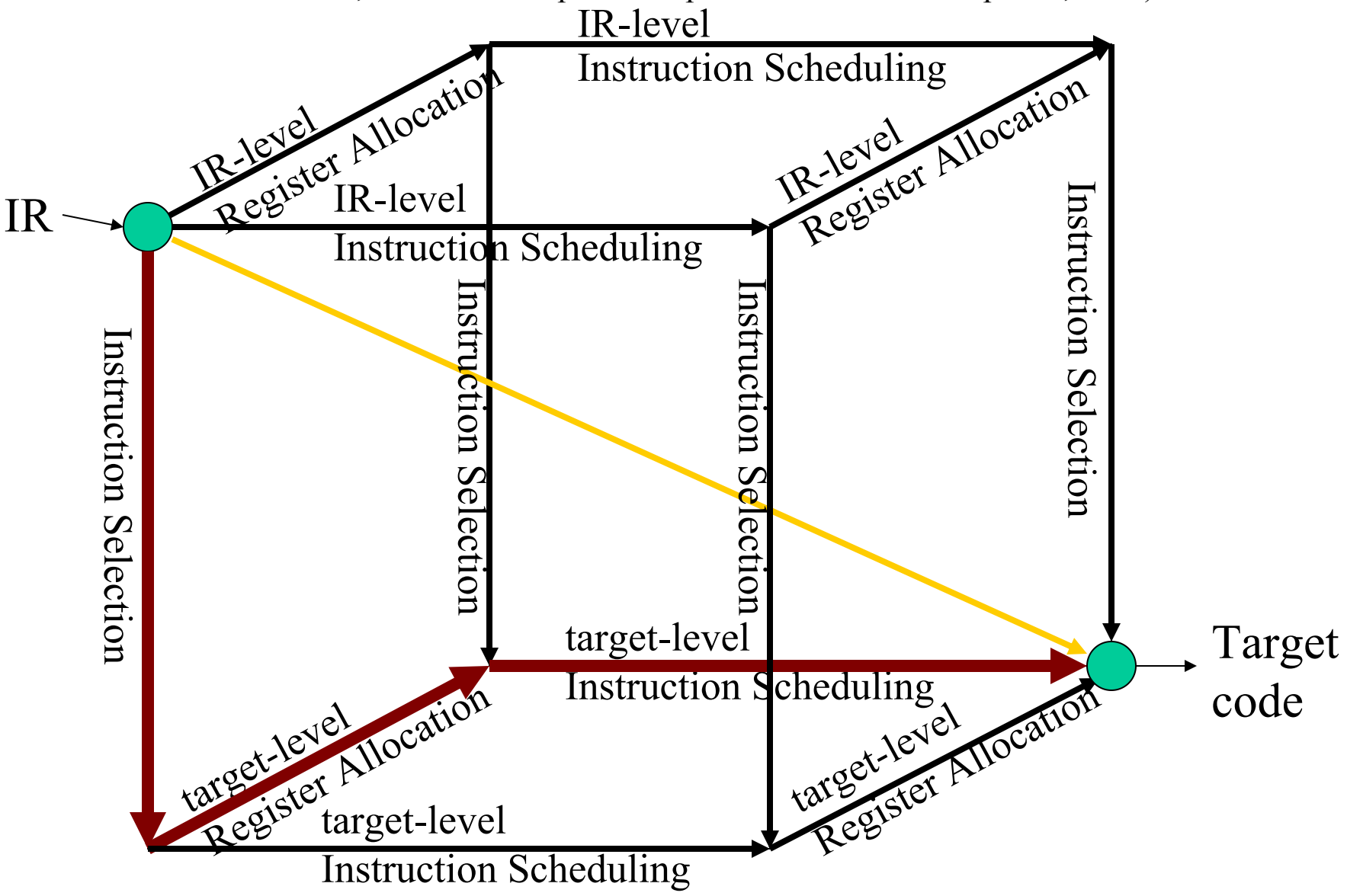
Lecture 15: Code Generation - Instruction Selection



- Last Lecture:
  - The Procedure Abstraction
  - Run-Time Organisation: Activation records for procedures.
- Where are we?
  - Code generation: we assume the following model:
    - instruction selection: mapping IR into assembly code
    - register allocation: decide which values will reside in registers.
    - instruction scheduling: reorder operations to hide latencies…
  - Conventional wisdom says that we lose little by solving these (NP-complete) problems independently.

Code Generation as a phase-decoupled problem

(thanks for inspiration to: C.Kessler, A.Bednarski; Optimal Integrated Code Generation for VLIW architectures; 10th Workshop on Compilers for Parallel Computers, 2003)



Instruction Selection

- Characteristics:
  - use some form of pattern matching
  - can make locally optimal choices, but global optimality is NP-complete
  - assume enough registers.
- Assume a RISC-like target language
- Recall:
  - modern processors can issue multiple instructions at the same time.
  - instructions may have different **latencies**: e.g., add: 1 cycle; load 1-3 cycles; imult: 3-16 cycles, etc
- Instruction Latency: length time elapsed from the time the instruction was issued until the time the results can be used.

Our (very basic) instruction set

- load r1, @a ; load register r1 with the memory value for a
- load r1, [r2] ; load register r1 with the memory value pointed by r2
- mov r1, 3 ; r1=3
- add r1, r2, r3 ; r1=r2+r3

  (same for mult, sub, div)
- shr r1, r1, 1 ; r1=r1>>1 (shift right; r1/2)
- store r1 ; store r1 in memory
- nop ; no operation

Code generation for arithmetic expressions

- Adopt a simple treewalk scheme; emit code in postorder:

```
expr(node)
{ int result, t1, t2;
  switch(type(node))
  { case *,/,+,-:
      t1=expr(left child(node));
      t2=expr(right child(node));
      result=NextRegister();
      emit(op(node),result,t1,t2);
      break;
    case IDENTIFIER:
      t1=base(node); t2=offset(node);
      result=NextRegister();
      emit(…) /* load IDENTIFIER */
      break;
    case NUM:
      result=NextRegister();
      emit(load result, val(node));
      break;
  }
  return result;
}
```

Example: x+y:

```
load r1, @x
load r2, @y
add r3,r1,r2
```

(`load r1,@x` would involve a load from address base+offset)

Issues with arithmetic expressions

- What about values already in registers?
  - Modify the IDENTIFIER case.
- Why the left subtree first and not the right?
  - (cf. 2*y+x; x-2*y; x+(5+y)*7): the most demanding (in registers) subtree should be evaluated first (this leads to the Sethi-Ullman labelling scheme – first proposed by Ershov – see Aho2 §8.10 or Aho1 §9.10).
- 2nd pass to minimise register usage/improve performance.
- The compiler can take advantage of commutativity and associativity to improve code (but not for floating-point operations)

Issues with arithmetic expressions - 2

- Observation: on most processors, the cost of a mult instruction might be several cycles; the cost of shift and add instructions is, typically, 1 cycle.
- Problem: generate code that multiplies an integer with an unknown using only shifts and adds!
- E.g.:

  325*x = 256*x+64*x+4*x+x or (4*x+x)*(64+1) (Sparc)
- For division, say x/3, we could compute 1/3 and perform a multiplication (using shifts and adds)… but this is getting complex!

  [For the curious only: see the "Hacker's delight" and http://www.hackersdelight.org/divcMore.pdf]

Trading register usage with performance

- Example: w=w*2*x*y*z

```
 1. load r1, @w
 2. mov  r2, 2
 6. mult r1,r1,r2
 7. load r2, @x
12. mult r1,r1,r2
13. load r2, @y
18. mult r1,r1,r2
19. load r2, @z
24. mult r1,r1,r2
26. store r1
```

Hardware will stall until result is available!

```
 1. load r1, @w   ; load: 5 cycles
 2. load r2, @x
 3. load r3, @y
 4. load r4, @z
 5. mov  r5, 2    ; mov: 1 cycle
 6. mult r1,r1,r5 ; mult: 2 cycles
 8. mult r1,r1,r2
10. mult r1,r1,r3
12. mult r1,r1,r4
14. store r1      ; store: 5 cycles
```

2 registers, 31 cycles

5 registers, 19 cycles

- Instruction scheduling (the problem): given a code fragment and the latencies for each operation, reorder the operations to minimise execution time (produce correct code; avoid spilling registers)

Array references

- Agree to a storage scheme:
  - Row-major order: layout as a sequence of consecutive rows: A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]
  - Column-major order: layout as a sequence of consecutive columns: A[1,1], A[2,1], A[1,2], A[2,2], …
  - Indirection vectors: vector of pointers to pointers to … values; best storage is application dependent; not amenable to analysis.
- Referencing an array element, where w=sizeof(element):
  - row-major, 2d: base+((i$_1$–low$_1$)*(high$_2$–low$_2$+1)+i$_2$–low$_2$)*w
  - column-major, 2d: base+((i$_2$–low$_2$)*(high$_1$–low$_1$+1)+i$_1$–low$_1$)*w
  - general case for row-major order:
    - ((…(i$_1$n$_2$+i$_2$)n$_3$+i$_3$)…)n$_k$+i$_k$)*w+base - w*((…((low$_1$*n$_2$)+low$_2$)n$_3$+low$_3$)…)n$_k$+low$_k$), where n$_i$=high$_i$–low$_i$+1

Boolean and relational values

expr→ not or-term | or-term
or-term→or-term or and-term | and-term
and-term→and-term and boolean | boolean
boolean→true | false | rel-term
rel-term→rel-term rel-op expr | expr
rel-op→ < | > | == | != | >= | <=

- Evaluate using treewalk-style generation. Two approaches for translation: numerical representation (0/1) or positional encoding.
- B or C and not D: r1=not D; r2=r1 and C; r3=r2 or B.
- if (a<b) then … else… : comp rx,ra,rb; br rx L1, L2; L1: …code for then...; br L3; L2: …code for else...; L3: ...
- Short-circuit evaluation: the C expression (x!=0 && y/x>1) relies on short-circuit evaluation for safety.

Control-Flow statements

- If expr then stmt1 else stmt2
  - 1. Evaluate the expr to true or false
  - 2. If true, fall through to then; branch around else
  - 3. If false, branch to else; fall through to next statement.

    (if not(expr) br L1; stmt1; br L2; L1: stmt2; L2: …)
- While loop; for loop; or do loop:
  - 1. Evaluate expr
  - 2. If false, branch beyond end of loop; if true, fall through
  - 3. At end, re-evaluate expr
  - 4. If true, branch to top of loop body; if false, fall through

    (if not(expr) br L2; L1: loop-body; if (expr) br L1; L2: …)
- Case statement: evaluate; branch to case; execute; branch
- Procedure calls: recall last lecture...

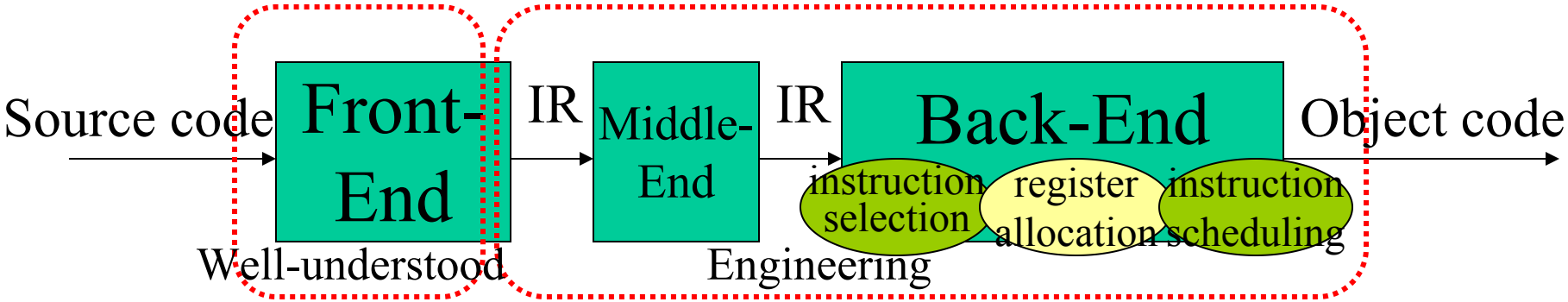
Conclusion

- (Initial) code generation is a pattern matching problem.
- Instruction selection (and register allocation) was the problem of the 70s. With the advent of RISC architectures (followed by superscalar and superpipelined architectures and with the ratio memory access vs CPU speed starting to rise) the problems shifted to register allocation and instruction scheduling.

- Reading: Aho2, Chapter 8 (more information than covered in this lecture but most of it useful for later parts of the module); Aho1 pp.478-500; Hunter, pp.186-198; Cooper, Chapter 11 (NB: Aho treats intermediate and machine code generation as two separate problems but follows a low-level IR; things may vary in reality).

- Next time: Register allocation

Lecture 16: Register Allocation



- Last Lecture: Instruction Selection.
- Register Allocation:
  - We assume a RISC-like (three-address) type of code.
  - The code makes use of an unbounded number of registers (**virtual registers**) but the machine has only a limited number of registers (**physical registers**), say *k*.
  - The task:
    - Produce correct *k* register code.
    - Minimise number of loads and stores (spill code) and their space.
    - The allocator must be efficient (e.g., no backtracking)

Background

- **Basic Block**: a maximal length segment of straight-line (i.e., branch-free) code. (Importance: strongest facts are provable for branch-free code; problems are simpler; strongest techniques.)
- **Local Register Allocation**: within a single basic block.
- **Global Register Allocation**: across an entire procedure (multiple BBs).
- **Allocation**: choose what to keep in registers.
- **Assignment**: choose specific registers for values.
- Modern processors may have multiple register classes:
  - General-purpose, floating-point, branch target, …
  - Problem: interactions between classes - Assume separate allocation for each class.
- **Complexity**: Only simplified cases of local allocation and assignment can be solved in linear time. All the rest (including global allocation – even for 1 register – and most sub-problems) are NP-complete. We need good heuristics!

Real compilers face real problems!

Liveness and Live Ranges

- Problem: What is the number of registers needed in a basic block?
  - Naïve: all occurrences of a variable to the same register.
  - Realistic: Compute a set of live ranges and use their name space.
- A value of a variable is **live** between its definition and its uses:
  - Find definitions (x←…) and uses (…←…x…)
  - From definition to last use is the "live range"
  - Can represent live range as an interval [i,j] in basic block.
- Over all instructions in the basic block, let:
  - MAXLIVE be the maximum number of values live at an instruction
  - k, the number of physical registers available.
    - If MAXLIVE ≤ k, allocation is trivial.
    - If MAXLIVE > k, some values must be spilled to memory.

Example / Exercise

Compute live ranges for all registers and MAXLIVE in the following Basic Blocks:

```
1. load r1,@a
2. load r2,2
3. load r3,@b
4. load r4,@c
5. load r5,@d
6. mult r1,r1,r2
7. mult r1,r1,r3
8. mult r1,r1,r4
9. mult r1,r1,r5
10.store r1
```

r1	[1,6]	*	*	*	*	*	*				
r1	[6,7]						*	*			
r1	[7,8]							*	*		
r1	[8,9]								*	*	
r1	[9,10]									*	*
r2	[2,6]		*	*	*	*	*				
r3	[3,7]			*	*	*	*	*			
r4	[4,8]				*	*	*	*	*		
r5	[5,9]					*	*	*	*	*	

```
1. load r1,@a
2. load r2,@y
3. mult r3,r1,r2
4. load r4,@x
5. sub  r5,r4,r2
6. load r6,@z
7. mult r7,r5,r6
8. sub  r8,r7,r3
9. add  r9,r8,r1
```

r1	[1,9]	*	*	*	*	*	*	*	*	*
r2	[2,5]		*	*	*	*				
r3	[3,8]			*	*	*	*	*	*	
r4	[4,5]				*	*				
r5	[5,7]					*	*	*		
r6	[6,7]						*	*		
r7	[7,8]							*	*	
r8	[8,9]								*	*
r9	[9,9]									*
# of live values		1	2	3	4	5	4	5	4	3
						-1		-1	-1	-1

Top-Down (Local) Allocation

- Allocator must reserve f registers to ensure feasibility (e.g., for use in computations that involve values allocated to memory; 2 to 4 depending on the target processor).
- Idea (frequency count algorithm): keep $k–f$ most frequently used values in the BB in a register; use f for the rest:
  - 1. Count number of uses for each virtual register.
  - 2. Assign top k-f virtual registers to physical registers.
  - 3. Rewrite code: if a virtual register was assigned to a physical register, replace. Else spill: use reserved registers to load before use and store after definition.
- Weakness: a value heavily used in the 1st half of the basic block and unused in the 2nd half, essentially wastes the register for the latter.

Example

- Assume 3 physical registers – two needed for feasibility.

```
1. load r1,@a
2. load r2,@y
3. mult r3,r1,r2
4. load r4,@x
5. sub  r5,r4,r2
6. load r6,@z
7. mult r7,r5,r6
8. sub  r8,r7,r3
9. add  r9,r8,r1
```

r1	2	[1,9]	*	*	*	*	*	*	*	*	*
r2	2	[2,5]		*	*	*	*				
r3	1	[3,8]			*	*	*	*	*	*	
r4	1	[4,5]				*	*				
r5	1	[5,7]					*	*	*		
r6	1	[6,7]						*	*		
r7	1	[7,8]							*	*	
r8	1	[8,9]								*	*
r9	-	[9,9]									*
# of live values			1	2	3	4	4	4	4	3	2

```
1. load r1,@a
2. load r2,@y
3. mult r3,r1,r2
4. store r3 //spill r3
5. load r3,@x
6. sub  r3,r3,r2
7. load r2,@z
8. mult r3,r3,r2
9. load r2, … //spilled value
10.sub  r3,r3,r2
11.add  r3,r3,r1
```

r1 is assigned to the most commonly used value (ok, there is a tie; we choose the first one), and r2 and r3 are used for feasibility!

This assumes that the compiler can realize that **y** is already in register **r2**, hence it is not necessary to do a **load r2,@y** again!

Bottom-Up (Local) Allocation

- Let multiple values occupy a single register – Best's algorithm:

  for each operation, i, 1 to N (op vr3, vr2, vr1)

  ensure that vr1 is in r1
  ensure that vr2 is in r2
  if r1 not needed after i, free(r1)
  if r2 not needed after i, free(r2)
  allocate r3 for vr3
  emit code – op r3,r2,r1

- **ensure**: if a vr is not in a physical register, **allocate** register and make sure that occurrences of vr are tied to this physical register.
- **allocate**: return a free physical register, or select the register that is used farthest in the future, store its value and return it.
- Due to Sheldon Best (1955) – often reinvented. Many have argued for its optimality…
- What does it remind you?

Example

- Assume 3 physical registers

```
1. load r1,@a
2. load r2,@y
3. mult r3,r1,r2
4. load r4,@x
5. sub  r5,r4,r2
6. load r6,@z
7. mult r7,r5,r6
8. sub  r8,r7,r3
9. add  r9,r8,r1
```

By spilling the value here, note that MAXLIVE ≤ 3

r1	2	[1,9]	*	*	*	*	*	*	*	*	*
r2	2	[2,5]		*	*	*	*				
r3	1	[3,8]			*	*	*	*	*	*	
r4	1	[4,5]				*	*				
r5	1	[5,7]					*	*	*		
r6	1	[6,7]						*	*		
r7	1	[7,8]							*	*	
r8	1	[8,9]								*	*
r9	-	[9,9]									*
# of live values			1	2	3	4	4	4	4	3	2

```
1. load r1,@a
2. load r2,@y
3. mult r3,r1,r2
4. store r1 // spill the one used farthest
5. load r1,@x
6. sub  r1,r1,r2
7. load r2,@z
8. mult r1,r1,r2
9. sub  r1,r1,r3
10.load r2, … // load spilled value (load r2,@a)
11.add  r1,r1,r2
```

A 'clever' compiler may recognise that the store may not be needed since the value may be available from memory location @a (needs to guarantee that the value of this location won't change)

Exercise

(register allocation using Best's algorithm and 10 registers)

```
// a really useless program
mov r1,1
// generate 1 to 2^16
shl r2, r1, r1
shl r3, r2, r1
shl r4, r3, r1
shl r5, r4, r1
shl r6, r5, r1
shl r7, r6, r1
shl r8, r7, r1
shl r9, r8, r1
shl r10, r9, r1
shl r11, r10, r1
shl r12, r11, r1
shl r13, r12, r1
shl r14, r13, r1
shl r15, r14, r1
shl r16, r15, r1
shl r17, r16, r1
// now sum them spending registers to save adds
add r20, r1, r2
add r21, r3, r4
add r22, r5, r6
add r23, r7, r8
add r24, r9, r10
add r25, r11, r12
add r26, r13, r14
add r27, r15, r16
add r30, r20, r21
add r31, r22, r23
add r32, r24, r25
add r33, r26, r27
add r34, r30, r31
add r35, r32, r33
add r36, r35, r34
add r37, r36, r17
// wow! Now store the result
store r37,@a
// sum i=1 to 16 (2^i) is 2^17 -1!
// that was a really useless calculation…
add r40, r5, r1
shl r41, r1, r40
sub r42, r41, r1
store r41, @b
```

More complex scenarios

- Basic blocks (BB) rarely exist in isolation:

  BB1: … store r17, @a is followed by

  BB2: load r12, @a …

  – Could replace load with a move; needs control-flow graph.
- Blocks with multiple (control-flow) predecessors:

  BB1: … store r4,@x and BB2: … store r7,@x followed by

  BB3: load r1, @x

  – What if BB1 has x in a register but BB2 not? (BB3 follows)
- Multiple basic blocks increase complexity:
  – How to compute the "farthest" in Best's algorithm?

Global Register Allocation

- Taking a global approach:
  - Abandon the distinction between local and global.
  - Generalised frequency counts: weigh uses and defs and BBs; apply to each BB; try to remove loads and stores between adjacent BBs (Fortran H; IBM 360,370)
- Graph colouring paradigm:
  - Build an interference graph:
  - (try to) construct a k-colouring
    - Minimal colouring is NP-complete
    - Spill placement becomes a critical issue
  - Map colours onto physical registers.

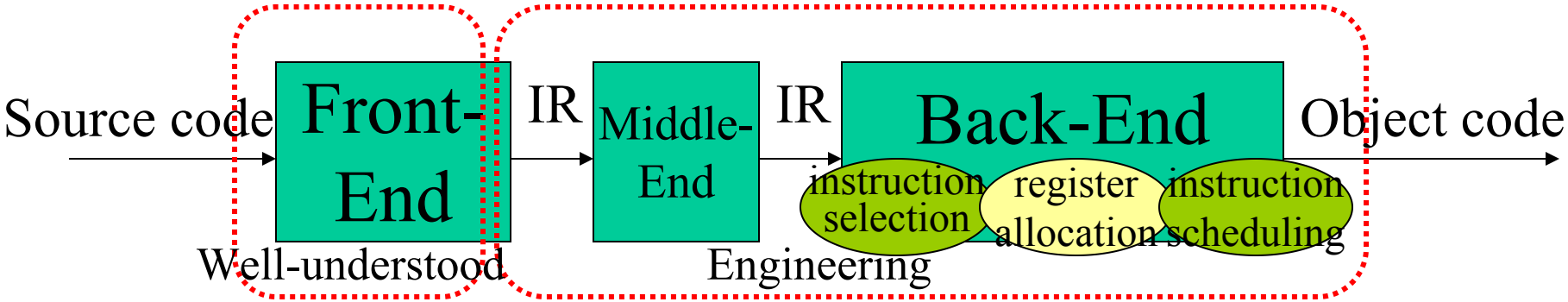
Conclusion

- Register allocation in real cases is NP-complete.
- Best's algorithm, which has been reinvented repeatedly, performs well for local register allocation.

- Reading:
  - Aho2, pp. 553-556; Aho1, pp.541-546 (too condensed)
  - Cooper, Sections 13.1-13.4.1.

- Next time: Register allocation via graph colouring.

Lecture 17: Register Allocation via Graph Colouring



- Last Lecture: (Local) Register Allocation.
- Global Register Allocation:
  - Go beyond basic blocks.
  - The task (again!):
    - Produce correct *k* register code.
    - Minimise number of loads and stores (spill code) and their space.
    - The allocator must be efficient (e.g., no backtracking)
- Today: Register Allocation via Graph Coloring:
    - 1st part: within a basic block. 2nd part: globally.

Register Allocation via graph colouring

The idea:

- live ranges that do not interfere can share the same register.

The algorithm:

1. Construct live ranges
2. Build ***interference graph***
3. (try to) construct a ***k-colouring*** of the graph:
   - If unsuccessful, choose values to spill (on the basis of some cost estimates in each case) and repeat from start (spill placement becomes a critical issue).
4. Map colours onto physical registers

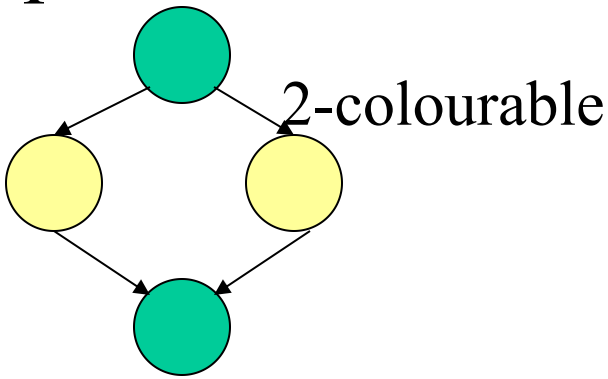
(can be generalised for global register allocation, by constructing global live ranges and building interference graph for procedure)

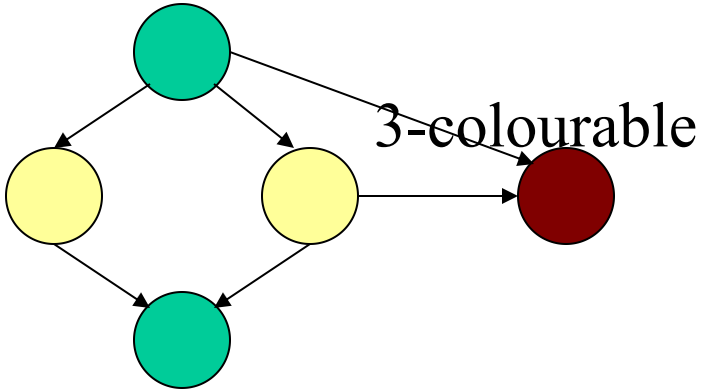
Graph Colouring - Background

- The problem:

  A graph is said to be *k*-colourable iff the nodes can be labelled with integers *1*…*k* so that no edge connects nodes with the same label.

- Examples:





A colouring that uses *k* colours is termed a *k*-colouring and *k* is the graph's *chromatic number*.

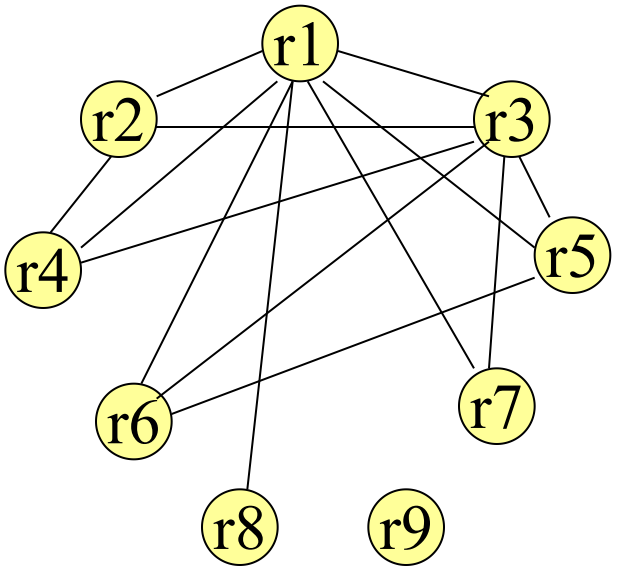
(Famous problem: the map-colouring problem)

Build the interference graph

- What is an interference?
  - Two values interfere if there exists an operation where both are simultaneously live; if they interfere they cannot occupy the same register.
- The interference graph:
  - Nodes: represent values (or live ranges).
  - Edges: represent individual interferences.

Example (using the basic block from last lecture):

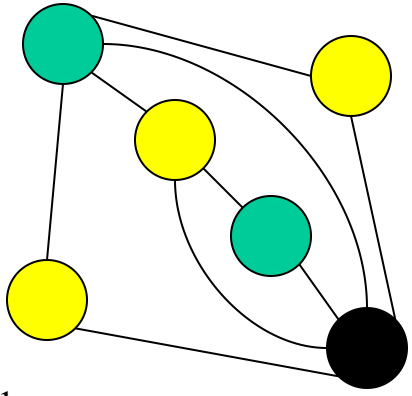
r1	[1,9]	*	*	*	*	*	*	*	*	*
r2	[2,5]		*	*	*	*				
r3	[3,8]			*	*	*	*	*	*	
r4	[4,5]				*	*				
r5	[5,7]					*	*	*		
r6	[6,7]						*	*		
r7	[7,8]							*	*	
r8	[8,9]								*	*
r9	[9,9]									*
# of live values		1	2	3	4	4	4	4	3	2

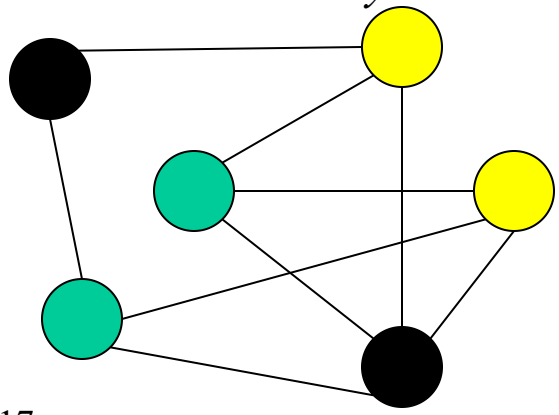


Construct a k-colouring

- **Top-down colouring:**
  - Rank the live ranges (that is, the nodes):
    - Possible ways of ranking: number of neighbours (in decreasing order), spill cost (starting with nodes that is more important to have in registers)
  - Follow the ranking to assign colours:
    - (for each node, pick the first colour that is not used by the node's neighbours)
  - If a live range cannot be coloured: spill (store after definition, load before each use) or split the live range.

(Observation: every node with a number of neighbours less than k will always receive a colour!)



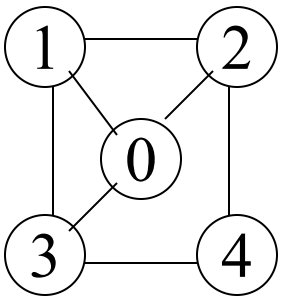


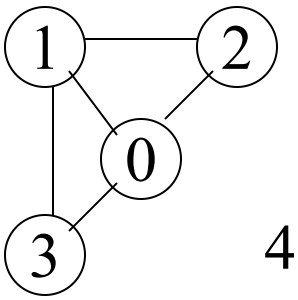
Construct a k-colouring (cont.)

- **Bottom-up colouring**: (Chaitin's algorithm)
  - 1. Simplify the graph:
    - Pick any node, such that the number of neighbouring nodes is less than k (degree of the node), and put it on the stack.
    - Remove that node and all edges incident to it
  - 2. If the graph is non-empty (i.e., all nodes have k or more neighbours), then:
    - Use some heuristic to spill a live range; remove corresponding node; if this causes some neighbours to have fewer than k nodes goto step 1, otherwise repeat step 2.
  - 3. Successively pop nodes off the stack and colour them using the first colour not used by some neighbour.
    - If a node cannot be coloured, leave it uncoloured (will have to spill).

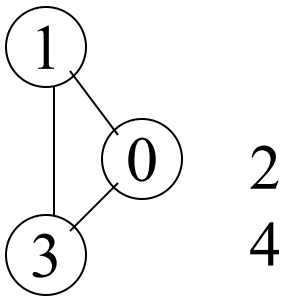
(Observation: A graph having a node n with degree < k is k-colourable iff the graph with node n removed is k-colourable)

Bottom-up colouring: example (k=3)

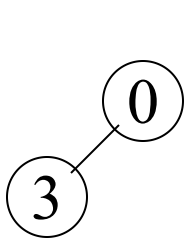




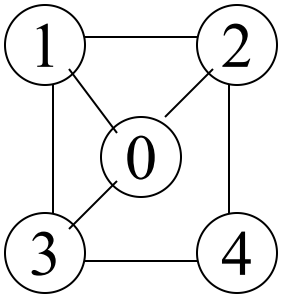
4



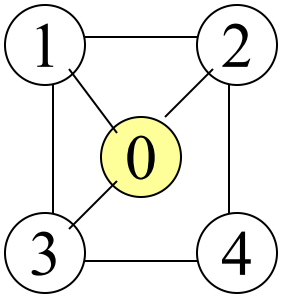
2
4



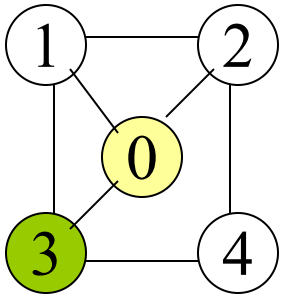
1
2
4



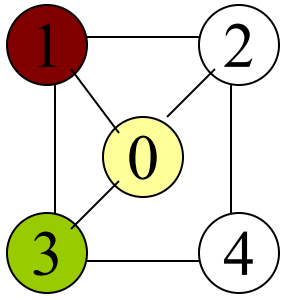
3
1
2
4



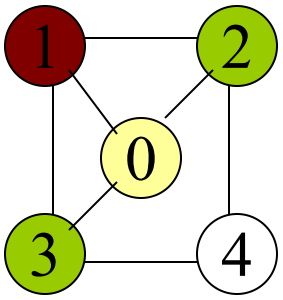
3
1
2
4



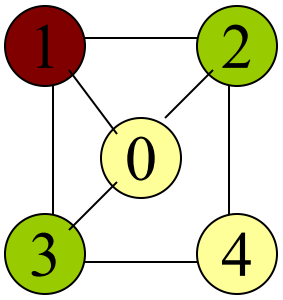
1
2
4



2
4

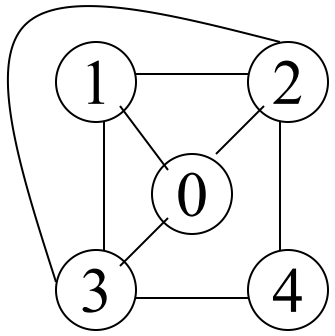


4



What about

?



Global Register Allocation via graph colouring

The idea:

- live ranges that do not interfere can share the same register.

The algorithm:

1. Construct **global** live ranges
2. Build ***interference graph*** for **procedure**
3. (try to) construct a ***k-colouring*** of the graph:
   - If unsuccessful, choose values to spill (on the basis of some cost estimates in each case) and repeat from start (spill placement becomes a critical issue).
4. Map colours onto physical registers

(we need to deal with 1 and 2)

Global live ranges

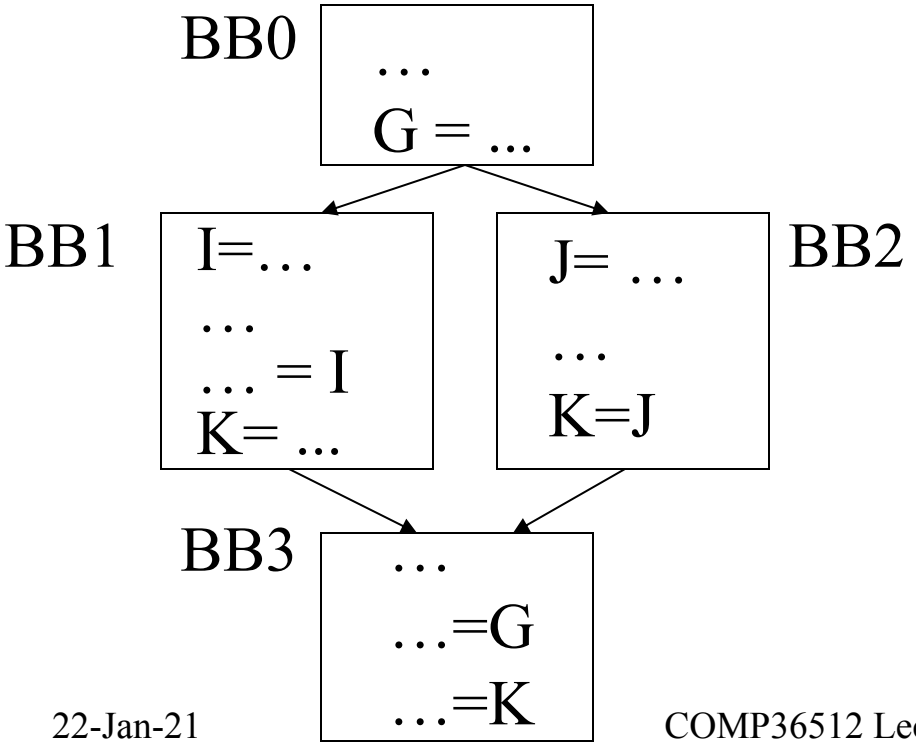
- At some point p in a procedure, a value v is live if it has been defined along a path from the procedure entry's to p and a path exists from p to a use of v, along which v is not redefined.

  (Hmm, paths imply that we need the Control Flow Graph!)

- To discover global live ranges, the compiler must discover the set of entries that are live on entry to each basic block, as well as those that are live on exit from each basic block. Each basic block, b, is annotated with:
  - LIVEIN(b): a value is in LIVEIN if it is defined along some path through the control-flow graph that leads to b and it is either used directly in b or it is in LIVEOUT(b).
  - LIVEOUT(b): a value is in LIVEOUT if it is used along some path leaving b before being redefined, and it is either defined in b or is in LIVEIN(b).

Construct LIVEIN, LIVEOUT

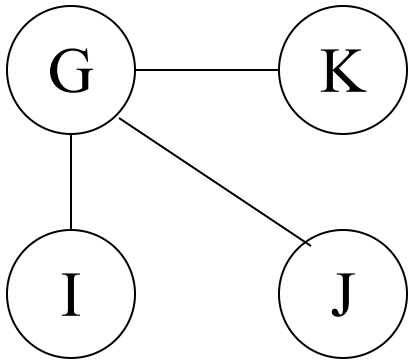
- If basic block has no successors, LIVEOUT(b) = ∅
- For all other basic blocks: LIVEOUT(b)=∪ LIVEIN(s) for all immediate successors, s, of b in the control flow graph.
- A value is in LIVEIN(b):
  - if it is used before it is defined (if it is defined) in basic block b; or
  - it is not used, nor defined, but it is in LIVEOUT(b)



LIVEOUT(BB3) ={}
LIVEIN(BB3)={G,K}
LIVEOUT(BB2)={G,K}
LIVEIN(BB2)={G}
LIVEOUT(BB1)={G,K}
LIVEIN(BB1)={G}
LIVEOUT(BB0}={G}
LIVEIN(BB0)={}

Build the interference graph

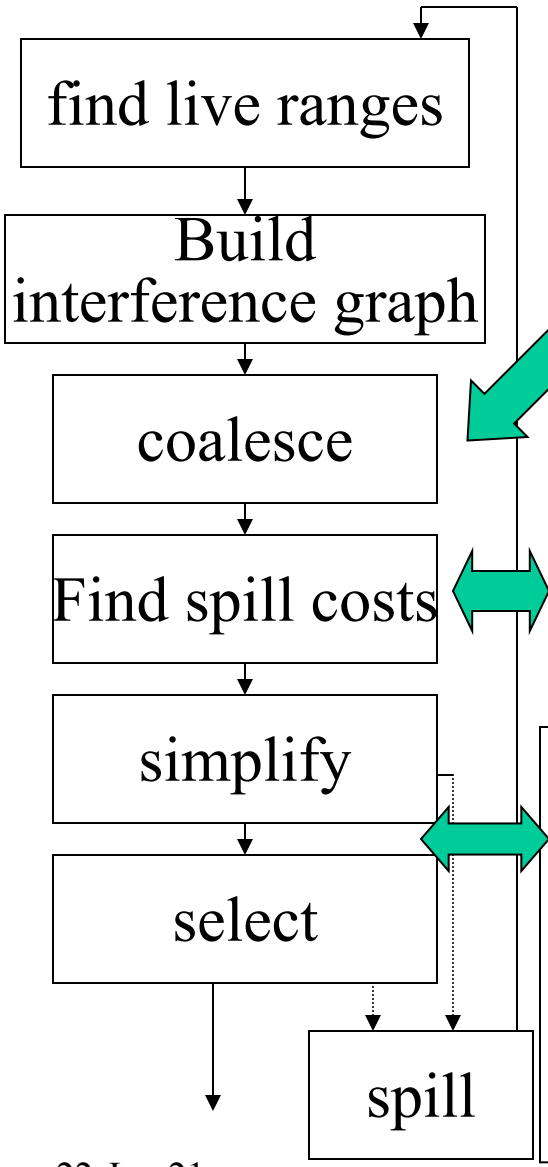
for each basic block b
    LIVENOW(b)←LIVEOUT(b)
    for each operation in b (in reverse order) of the type op lr1,lr2,lr3
        for each live_range in LIVENOW(b) except lr2 and lr3
            add an edge between live_range and lr1
        remove lr1 from LIVENOW(b)
        add lr2 and lr3 to LIVENOW(b)



Estimate spill costs

- Components of cost for a spill:
  - Address computation:
    - Minimise by keeping spilled values in activation record.
    - Perform load/store with (pointer+offset) instruction.
  - Memory operation:
    - Unavoidable (compiler hopes that spill locations stay in the cache).
  - Estimated execution frequencies:
    - Static analysis to estimate execution counts for basic blocks: spill in outer loops not in inner loops.

Chaitin-Briggs Register Allocators



Coalesce: Consider the operation mov r1, r2. If r1, r2 don't interfere, the operation can be eliminated and all references to r1 can be rewritten to use r2.

Chaitin's heuristic: use max(spill_cost/degree). degree is the number of neighbours and spill cost is (#refs)*10^d–(# LD+STORE)*2*10^d, where d is the loop nesting depth.

Simplify will push onto the stack nodes with less than k neighbours. If at some point only nodes with ≥ k neighbours exist, Chaitin stops and spills. Briggs will continue, and will do the spilling, during the coloring phase (when pop-ping from the stack) if no color can be assigned.

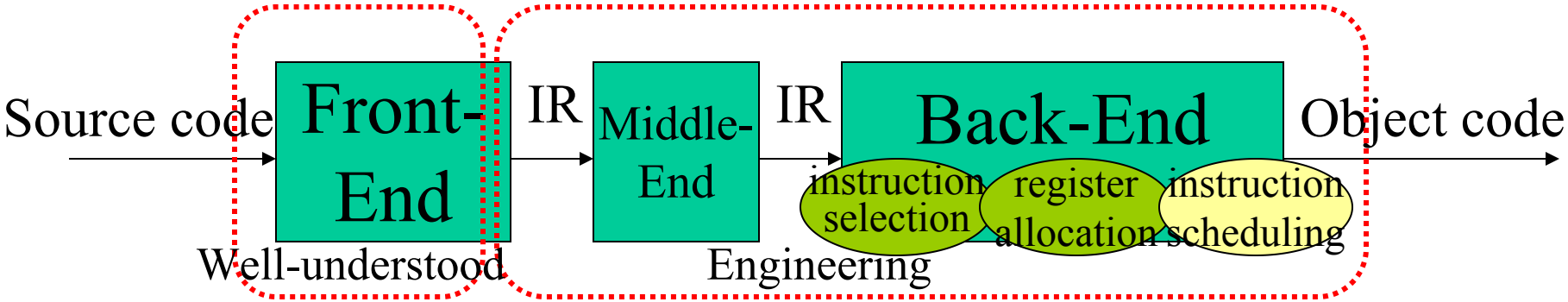
Exercise: Perform Register Allocation using Graph Colouring

1. mov r1,1
2. shl r2, r1, r1
3. shl r3, r2, r1
4. add r20, r1, r2
5. shl r4, r3, r1
6. shl r5, r4, r1
7. add r21, r3, r4
8. add r30, r20, r21
9. shl r6, r5, r1
10. add r22, r5, r6
11. shl r7, r6, r1
12. shl r8, r7, r1
13. add r23, r7, r8
14. add r31, r22,r23
15. shl r9, r8, r1
16. shl r10, r9, r1
17. shl r11, r10, r1
18. add r24, r9, r10
19. add r35, r24, r31
20. store r35

Conclusion

- Register allocation is an active research field. Main problem: minimise spill costs.
- Register allocation through graph colouring is popular because colouring captures critical aspects of the global problem.
- Size of interference graph may be quite large.
- Huge amount of work in the literature and an active research field:
  - Different solutions tend to be sensitive to small decisions.
  - Variations address compile-time speed/quality of the resulting code.
- For those interested:
  - G. Chaitin *et al*. "Register Allocation via Coloring". Computer Languages 6(1), Jan. 1981.
  - P. Briggs *et al*. "Improvements to Graph Coloring Register Allocation". ACM Transactions on Programming Languages and Systems, 16(3), May 1994 (and PLDI 1989).
  - Traub *et al*. "Quality and Speed in Linear-scan Register Allocation". ACM SIGPLAN'98 PLDI, 1998.
- Reading: Aho2, pp.556-557; Aho1, pp 545-546; Hunter, pp.202-204 (all of these are too condensed); Cooper, Chapter 13.
- Next time: Instruction Scheduling

Lecture 18: Instruction Scheduling



- Instruction Scheduling
  - The problem:
    - Given a code fragment for some target machine and the latencies for each individual operation, reorder operations to minimise execution time.
    - Recall: modern processors may have multiple functional units.
  - The task:
    - Produce correct code; minimise wasted cycles; avoid spilling registers; operate efficiently.

Background

- Many operations have delay latencies for execution.
  - E.g., load, store: <delay> CPU cycles (depends on the processor)
    - Issue load, result appears <delay> cycles later.
    - Execution continues unless result is referenced.
    - Premature reference causes hardware to stall.
- Modern machines can issue several operations per cycle.
- Execution time is order-dependent (has been since the 60s)
- Overview of a solution:
  - Move loads back at least <delay> slots from where they are needed, but this increases register pressure (i.e., more registers may be needed) – recall the example in lecture 14 (slide 7).
    Ideally, we want to minimise both hardware stalls and added register pressure.

Motivating Example

- Two variants to compute (a+b)+c:

(9 cycles):

```
load r1, @a
load r2, @b
add r1,r1,r2
load r3, @c
add r1,r1,r3
```

(6 cycles):

```
load r1, @a
     load r2, @b
load r3, @c
     add r1,r1,r2
add r1,r1,r3
```

Hardware will stall until result is available!

(assume that the latency of a load is 3 cycles; all other instructions have a latency of 1 cycle)

(NB: costs due to the memory hierarchy are not part of the picture)

Instruction Scheduling for a basic block
The big picture

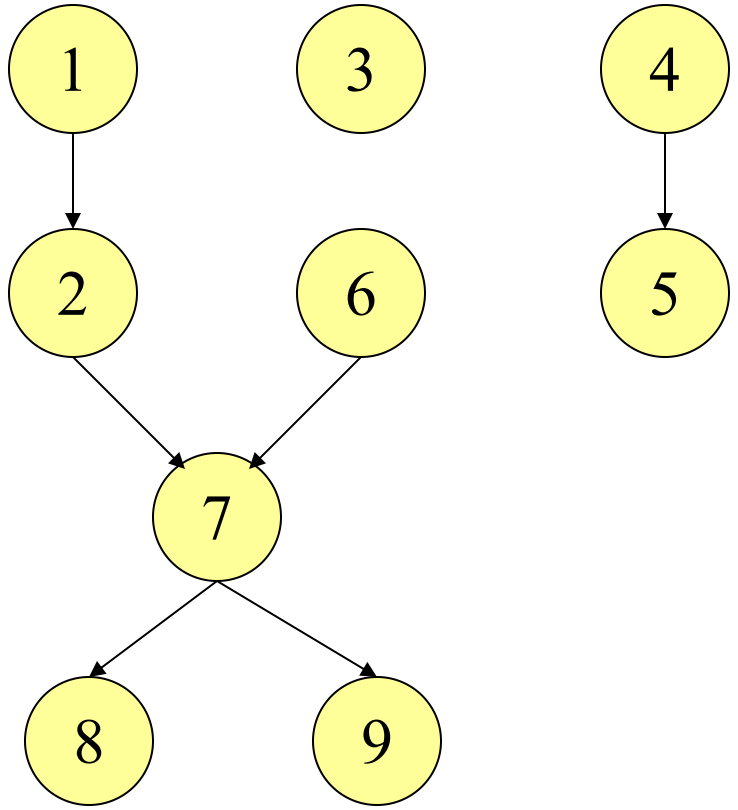
1. Build a **precedence** (data dependence) **graph**.
2. Compute a **priority function** for the nodes of the graph.
3. Use **list scheduling** to construct a schedule, one cycle at a time:
    1. Use a queue of operations that are ready
    2. At each cycle:
        1. Choose a ready operation and schedule it
        2. Update the ready queue.

A greedy heuristic; open to variations.

(greedy heuristic: An algorithmic technique in which an optimisation problem is solved by finding locally optimal solutions)

Build a precedence graph

1. load r1, @x
2. load r2, [r1+4]
3. and r3, r3, 0x00FF
4. mult r6, r6, 100
5. store r6
6. div r5, r5, 100
7. add r4, r2, r5
8. mult r5, r2, r4
9. store r4

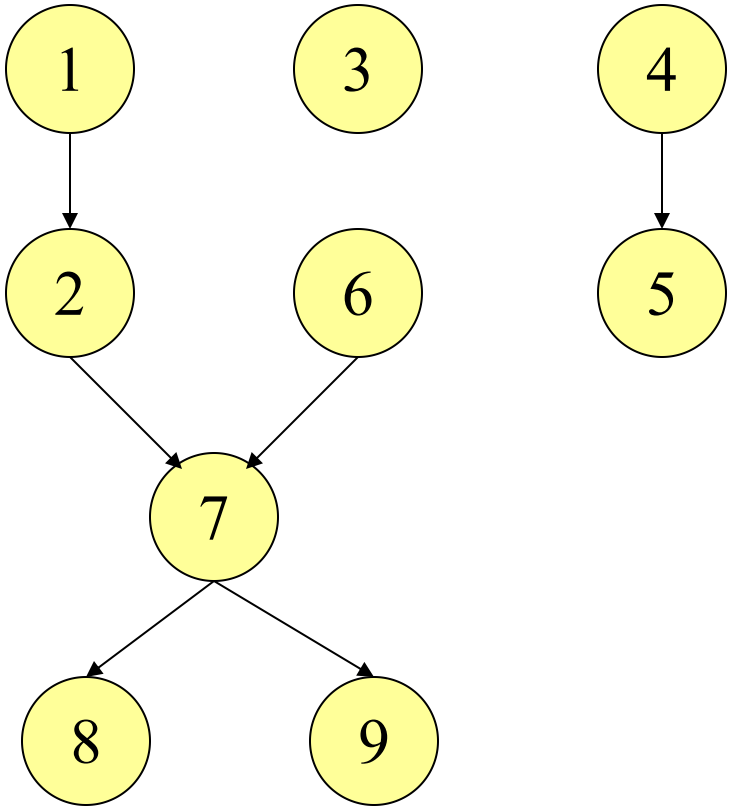


Compute a priority function

- Assign to each node a weight equal to the longest delay latency (total) to reach a leaf in the graph from this node (include latency of current node).
- $weight_i = latency_i + max(weight_{\text{all successor nodes}})$

(Assume: div 4 cycles, and mult 3 cycles; 1 cycle for the rest):

node	weight
1	6
2	5
3	1
4	4
5	1
6	8
7	4
8	3
9	1



(Local) List scheduling

```
Cycle=1; Ready=set of available operations; Active={}
While (Ready ∪ Active != {})
    if (Ready!={}) then
        remove an op from Ready (based on the weight)
        schedule(op)=cycle
        Active=Active ∪ op
    cycle=cycle+1
    for each op in Active
        if (schedule(op)+delay(op)<=cycle) then
            remove op from Active
            for each immediate successor s of op
                if (all operand of s are available) then
                    Ready=Ready ∪ s
```

Find the schedule

cycle	Instructions ready	Schedule	Instructions active
1	6, 1, 4, 3	div r5,r5,100	6
2	1, 4, 3	load r1,@x	6, 1
3	2, 4, 3	load r2, [r1+4]	6, 2
4	4, 3	mult r6,r6,100	6, 4
5	7, 3	add r4,r2,r5	4, 7
6	8, 3, 9	mult r5,r2,r4	4, 8
7	3, 9, 5	and r3,r3,0x00ff	8, 3
8	9, 5	store r4	8, 9
9	5	store r6	5

More list scheduling

- Two distinct classes of list scheduling:
  - Forward list scheduling: start with all available operations; work forward in time (Ready: all operands available)
  - Backward list scheduling: start with leaves; work backward in time (Ready: latency covers uses)
  - Folk wisdom is to try both and keep the best result.
- Variations on computing priority function:
  - Maximum path length containing node (decreases register usage).
  - Prioritise critical path.
  - Number of immediate successors or total number of descendants.
  - Increment weight if node contains a last use (shortens live ranges)
  - Do not add latency to node's weight.
  - Maximum delay latency from first available node.

Multiple functional units

- Modern architectures can run operations in parallel.
- List scheduling needs to be modified so that it can schedule as many operations per cycle as functional units (assuming that there are instructions available)

  (3rd line in the algorithm of slide 7 will have to be modified to:
  while (Ready!={} && there_are_free_functional_units)
- Back to the previous example (assume two functional units that can issue any instruction) [3rd column shows the ready set]

6. div r5, r5, 100	1. load r1, @x	{6, 1, 4, 3}
2. load r2, [r1+4]	4. mult r6, r6, 100	{2, 4, 3}
3. and r3, r3, 0x00FF	nop	{3}
nop	nop	{}
7. add r4, r2, r5	5. store r6	{7, 5}
8. mult r5, r2, r4	9. store r4	{8, 9}

Exercise

Assume two functional units: one for ALU operations only and another for memory operations only. A load and a mult have a latency of 2 cycles, all other instructions have a latency of 1 cycle.

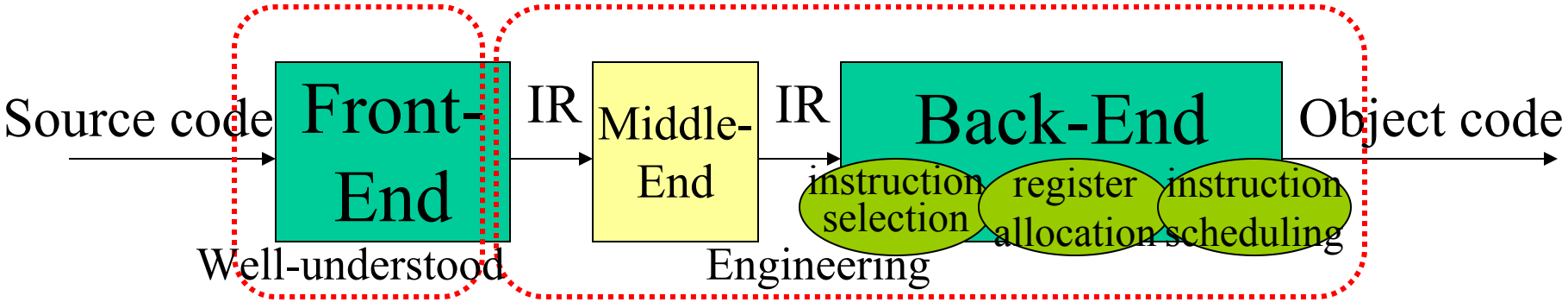
1. load r1, @x
2. load r2, @y
3. add r2,r2,42
4. load r3, @z
5. shl r4,r1,4
6. store r4
7. mult r5,r2,r3
8. add r6,r5,r4
9. store r5

ALU	MEM
nop	2
nop	4
3	1
7	nop
5	nop
8	9
nop	6

Further Issues and Conclusion

- Going beyond basic blocks:
  - Identify high-frequency path and schedule as if a single block.
- Modulo scheduling:
  - Schedule multiple iterations together (and run several iterations concurrently - i.e., overlap successive iterations).
- Register allocation with instruction scheduling:
  - The former before the latter restricts the choices for scheduling.
  - If the latter before the former and register allocation has to spill registers, the whole (carefully done) schedule changes!
- Besides performance, we may want to minimise power consumption, size of the code, ...
- Most of the active compiler research revolves around these areas!
- Reading: Cooper, Chapter 12; Next time: Code Optimisation.

Lecture 19: Code Optimisation



- Code Optimisation: (could be a course on its own!)
  - Goal: improve program performance within some constraints.
    - (may also reduce size of the code, power consumption, etc…)
  - Issues:
    - Legality: must preserve the meaning of the program.
      - Externally observable meaning may be sufficient/may need flexibility.
    - Benefit: must improve performance on average or common cases.
      - Predicting program performance is often non-trivial.
    - Compile-time cost justified: list of possible optimisations is huge.
      - Interprocedural optimisations (O4).

Optimising Transformations

- Finding an appropriate sequence of transformations is a major challenge: modern optimisers are structured as a series of passes:
  - optimisation 1 is followed by optimisation 2; optimisation 2 is followed by optimisation 3, and so on…
- Transformations may improve program at:
  - Source level (algorithm specifics)
  - IR (machine-independent transformations)
  - target code (machine-dependent transformations)
- Some typical transformations:
  - Discover and propagate some constant value.
  - Remove unreachable/redundant computations.
  - Encode a computation in some particularly efficient form.

Classification

- By Scope:
  - Local: within a single basic block.
  - Peephole: on a window of instructions (usually local)
  - Loop-level: on one or more loops or loop nests.
  - Global: for an entire procedure
  - Interprocedural: across multiple procedures or whole program.
- By machine information used:
  - Machine-independent versus machine-dependent.
- By effect on program structure:
  - Algebraic transformations (e.g., x+0, x*1, 3*z*4, …)
  - Reordering transformations (change the order of 2 computations)
    - Loop transformations: loop-level reordering transformations.

Some transformations…

- Common subexpression elimination:
  - An expression, say x+y, is redundant iff along every path from the procedure's entry it has been evaluated and its constituent subexpressions (x, y) have not been redefined.
- Copy propagation:
  - After a 'copy' statement, x=y, try to use y as far as possible.
- Constant propagation:
  - Replace variables that have constant values with these values.
- Constant folding:
  - Deduce that a value is constant, and use the constant instead.
- Dead-code elimination:
  - A value is computed but never used; or, there is code in a branch never taken (may result after constant folding).
- Reduction in strength:
  - Replace x/4.0 with x*0.25

Examples

Before optimisation

After optimisation

// Common subexpression elimination

```
A[I,I*2+10]=B[I,I*2+10]+5
```

```
tmp=I*2+10
A[I,tmp]=B[I,tmp]+5
```

// Copy propagation

```
t=I*4
s=t
a[s]=a[s]+4
```

```
t=I*4
s=t
a[t]=a[t]+4
```

// Constant propagation

```
N=64
c=2
for (I=0;I<N;I++)
   a[I]=a[I]+c
```

```
N=64
c=2
for (I=0;I<64;I++)
   a[I]=a[I]+2
```

// Constant folding

```
tmp=5*3+8-12/2
```

```
tmp=17
```

// Dead–code elimination

```
if (3>7) then { … }
```

```
// removed (some of the
// above optimisations may
// create `useless' code…)
```

// Reduction in strength

```
x*2+x*1024
```

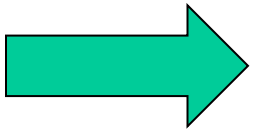
```
x+x+(x<<10)
```

Loop Transformations

- **Loop-invariant code-motion:**
  - Detect statements inside a loop whose operands are constant or have all their definitions outside the loop - move out of the loop.
- **Loop interchange:**
  - Interchange the order of two loops in a loop nest (needs to check legality): useful to achieve unit stride access.
- **Strip mining:**
  - May improve cache usage when combined with loop interchange.

Example: Applying stip mining + loop interchange (**loop tiling**)

```
DOALL J=1,N
  DO K=1,N
    DOALL I=1,N
      A(I,J)=A(I,J)+B(I,K)*C(K,J)
    ENDDO
  ENDDO
ENDDO
```



```
DOALL JJ=1,N,SJ
  DOALL II=1,N,SI
    DO J=JJ,MIN(JJ+SJ-1,N)
      DO K=1,N
        DO I=II,MIN(II+SI-1,N)
          A(I,J)=A(I,J)+B(I,K)*C(K,J)
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Loop Transformations – Loop Unrolling

- Change: **`for(i=0;i<n;i++)`** to **`for(i=0;i<n-s+1;i+=s)`** and replicate the loop body **`s`** times (changing also **`i`** as needed to **`i+1`**, **`i+2`**, etc…). Will need an 'epilogue' if **`s`** does not divide **`n`**.
- Creates larger basic blocks and facilitates instruction scheduling.

Example:

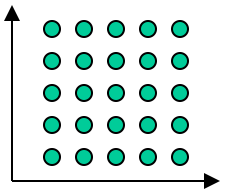
```
for (i=0; i<n; i++)  {
    a[i]=b[i]*c[i]; }
```

After loop unrolling it becomes:

```
for (i=0; i<n-s+1; i+=s) {
    a[i]=b[i]*c[i];
    a[i+1]=b[i+1]*c[i+1];
     … (the loop body repeated s times, from i to i+s-1)
}
/* epilogue */
for (j=i; j<n; j++) {
    a[j]=b[j]*c[j]; }
```

Is loop interchange legal? A more complex problem

- A model to represent loops: the polytope model.
  - Set of inequalities: $1 \leq i \leq 5$; $1 \leq j \leq 5$.
  - (Geometrical equivalence is possible)



- Locating data dependences between two statement instances in two different iterations of a loop (loop-carried dependence) is a complex problem.
  - What if the loop body contains a[i,j]=a[i-1,j-1]?
- A dependence vector is defined by the distance of two iterations that cause a dependence: for instance, [1,1] above.
- Complex analysis allows a formal framework to be developed.
- A loop nest of two loops can be interchanged when it has a dependence vector where both elements have the same sign.

Conclusion

- Program optimisation is a major research issue with several challenges: find an appropriate sequence of transformations (feedback-based, iterative compilation are amongst the ideas currently pursued); apply optimisations interprocedurally.
- Analysis for some transformations may be very expensive.
- Lots of work/research in several contexts...
- Reading: Aho2, Ch.9 (skim through Ch.11); Aho1 pp.585-602; Cooper, Ch.8 (for those interested further in the topic: Bacon et al, "Compiler Transformations for …", ACM Computing Surveys 26(4), 1994; M.Wolfe, High-Performance Compilers for Parallel Computing, Addison-Wesley; R.Allen & K.Kennedy, Optimizing Compilers for Modern Architectures, Morgan Kaufmann, 2002)