Language Engineering

Steve Gregory, room 3.21 (MV)

steve@cs.bris.ac.uk

http://www.cs.bris.ac.uk/Teaching/Resources/COMS22201/

~20 lectures:

M2, F11 — Weeks 1-12

Lab sessions:

M4, M5 — Weeks 3-12

Office hours:

F1

Assignment: Due Nov 6 and Nov 20 (feedback), Dec 18.

Contents of unit (TB1)

1-2: Introduction to compilation

3-4: Languages and grammars

5: Lexical analysis

6-7: Syntax analysis

8-11: Code generation

11-14: Liveness and register allocation

15-19: Code optimization

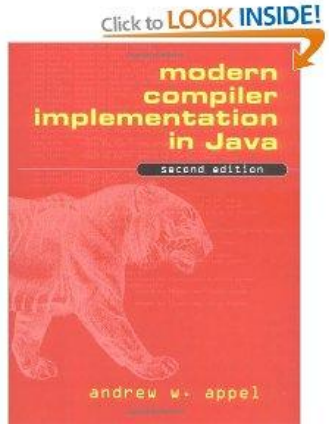
+ Revision lecture (later)

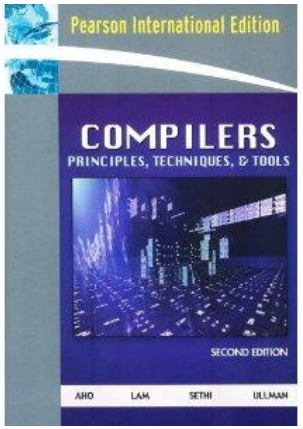
Books

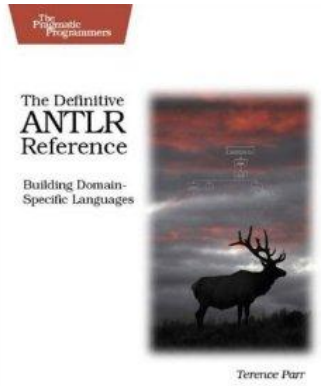
- A. Appel. *Modern Compiler Implementation in Java*. 2nd edition. Cambridge University Press. 2002. ISBN: 052182060X.

- A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd edition. Addison-Wesley. 2007. ISBN: 0321491696.

- T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. 2007. ISBN: 978-0-9787-3925-6







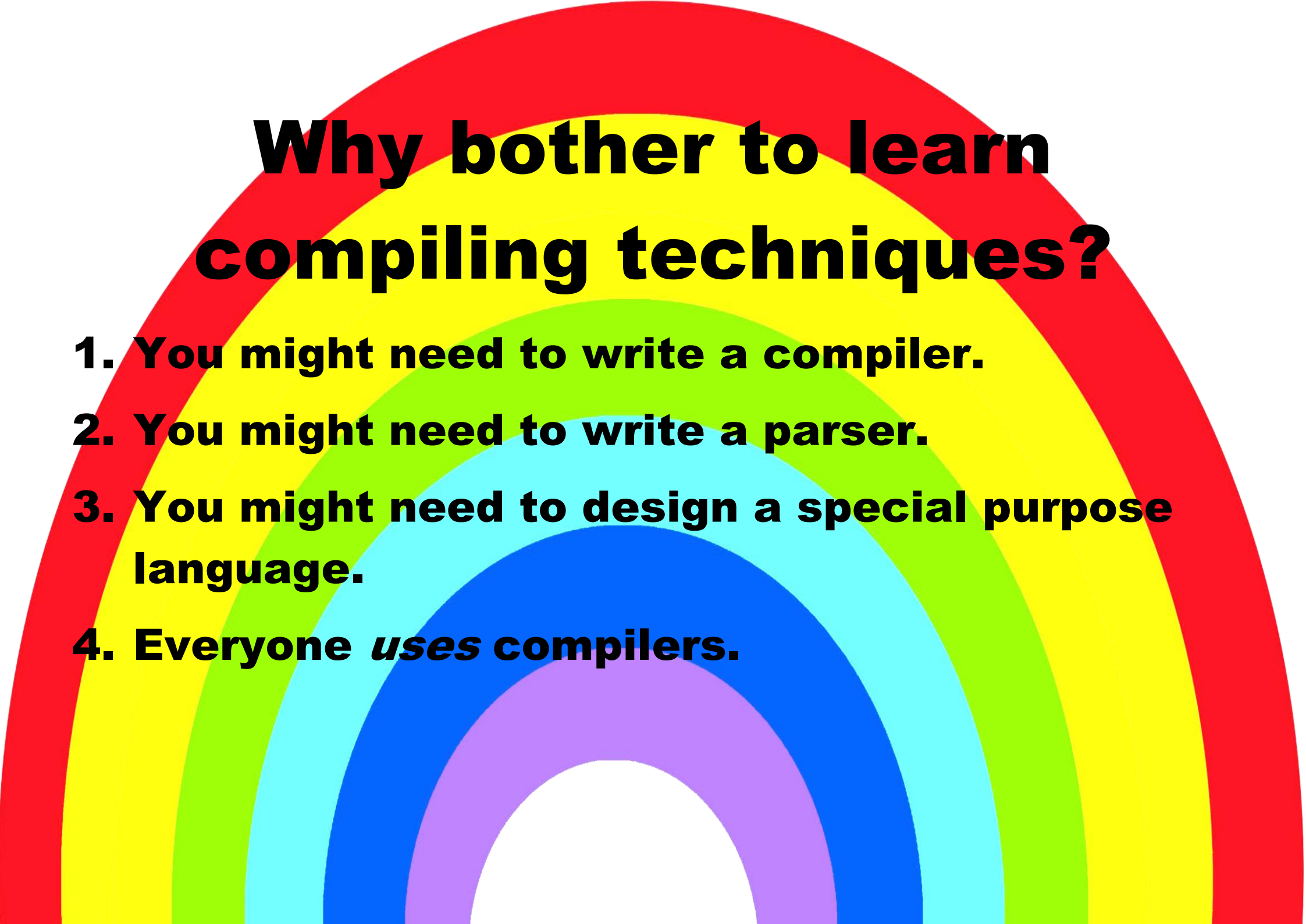
Lectures

Weird features of lectures:

1. **Motivation**: look out for occasional (colourful) slides explaining **why**.

2. **Interaction**: small “class test” in several lectures: you are expected to answer question to test understanding of that lecture’s material.

3.

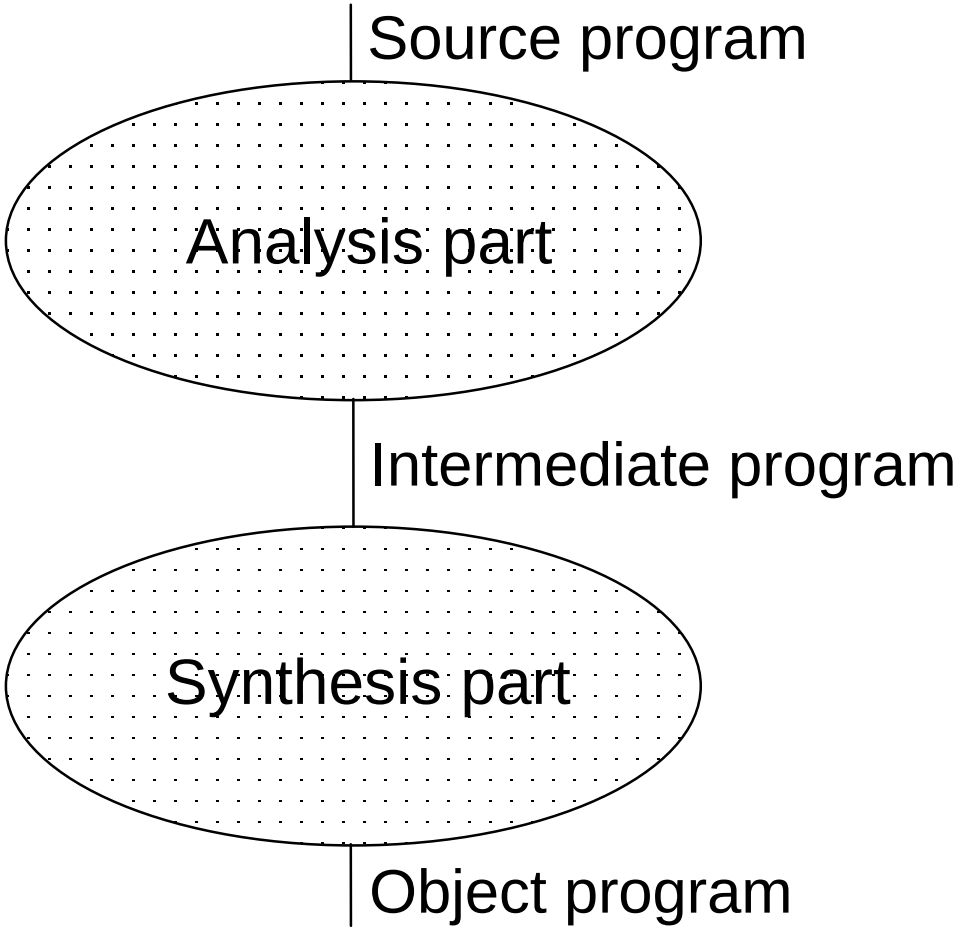


Why bother to learn compiling techniques?

1. You might need to write a compiler.
2. You might need to write a parser.
3. You might need to design a special purpose language.
4. Everyone *uses* compilers.

What is a compiler?

A program that translates program in source language to program in another *target* language.



Examples:

- Traditional compiler:

  programming language → machine code

- Source-source transformation:

  programming language → programming language

- Compiler compiler:

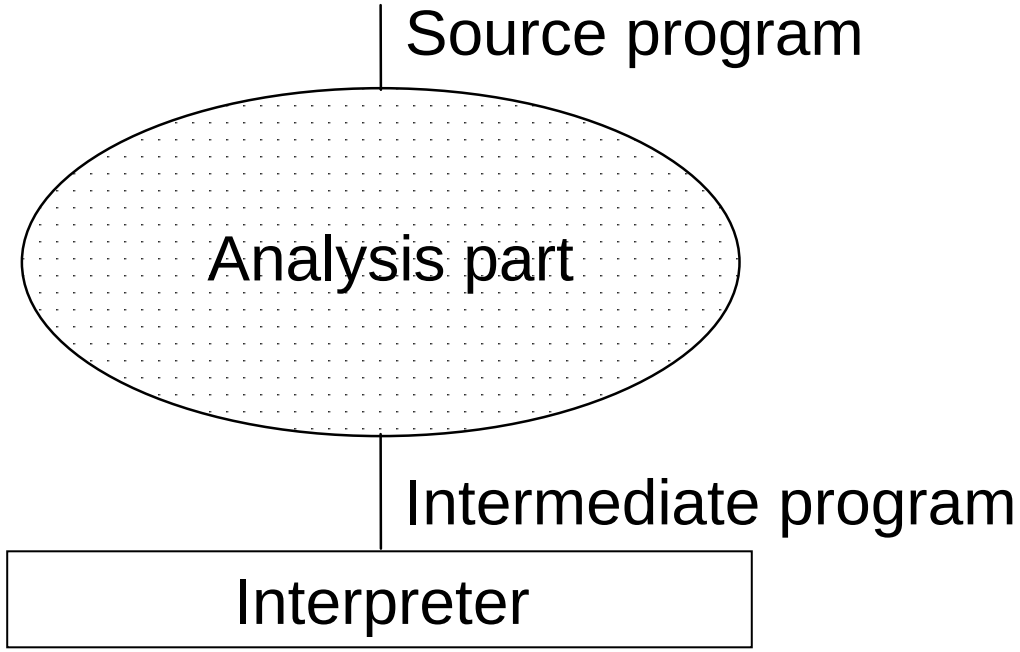
  grammar → programming language

- Word processor, typesetting program:

  text → PostScript

What is an interpreter?

A program that “executes” program in source language.



Interpreted program is usually slower than compiled one.

Examples:

- Traditional interpreter:

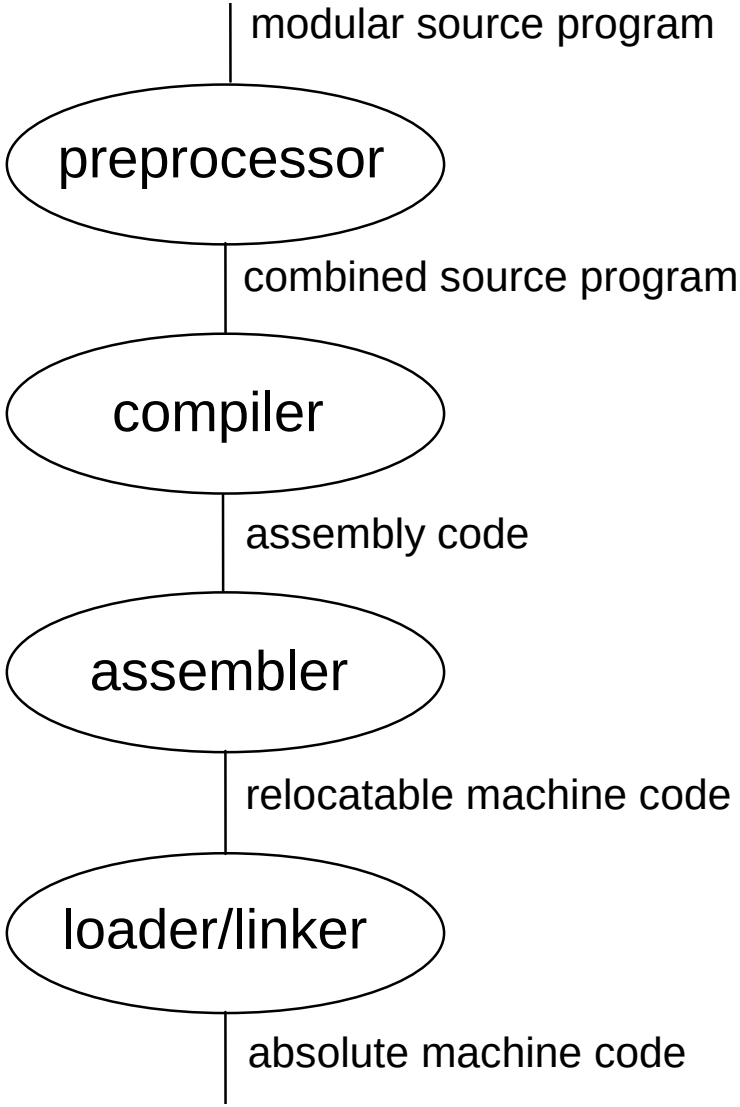
  for programming language

- Command interpreter (shell)

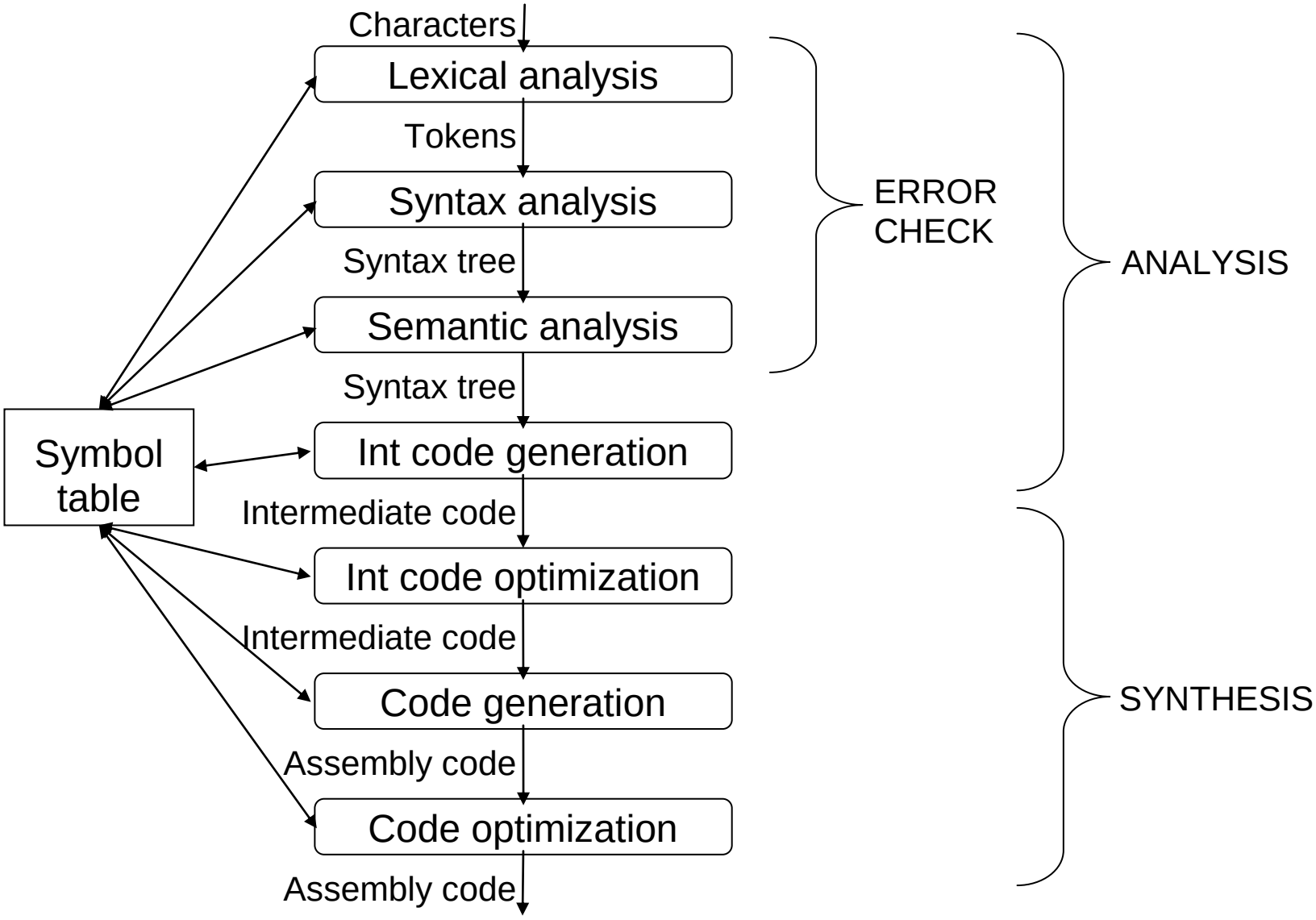
- Scripting languages: Javascript, etc.

- PostScript interpreter

Environment of a compiler



Structure of a compiler



Analysis part of compiler

Analyse and understand the source program:

- **Lexical analysis**

  finds tokens (numbers, keywords, etc.) in source program.

- **Syntax analysis**

  finds useful components (statements, expressions, etc.) in source program.

- **Semantic analysis**

  checks “meaning” of source program.

- **Intermediate code generation**

  translate program into an internal form.

Intermediate code

- Code (machine instructions) that performs the operations specified by the program.
- Code is for some idealized **abstract machine**
- Intermediate code may be interpreted or translated to real machine code

Synthesis part of compiler

Translate (intermediate code of) the source program into a machine code program:

- **Intermediate code optimization**

  Change code into more efficient form with the same result

- **Code generation**

  Translate intermediate code into code for a *real* machine

- **Code optimization**

Abstract machine emulators

Popular implementation technique. E.g.:

- Pascal: P-Code.
- Prolog: WAM.
- Java: JVM.

No code generation phase:

Emulator (interpreter) executes intermediate, abstract machine, code

+ **Portability**: porting the language to a different machine takes less effort

+ **Code size**: intermediate code program is usually more compact than (real) machine code

– **Speed**: executes slower than a machine code program

Symbol table

- Records identifiers and keywords:

  text name, type, scope, address, etc.

- Used by several stages of the compiler

Lexical analysis

Groups characters into tokens. Can be specified by regular grammar.

Example source program (C):

```
count = 0;  /* initialize */
while (ok) count = count+1;
```

1. Divide into chunks:

```
|count| |=| |0|;|  |/* initialize */|
|while| |(|ok|)| |count| |=| |count|+|1|;|
```

2. Discard white space and comments:

```
|count|=|0|;|while|(|ok|)|count|=|count|+|1|;|
```

3. Replace each chunk by a token (*token type*, *lexical value*):

```
(IDENTIFIER,"count"), (EQUAL), (INTNUM,"0"), (SEMICOLON), (WHILE),
(OPENPAREN), (IDENTIFIER,"ok"), (CLOSEPAREN), (IDENTIFIER,"count"),
(EQUAL), (IDENTIFIER,"count"), (PLUS), (INTNUM, "1"), (SEMICOLON)
```

Token type: small integer.

Lexical value:

- Keywords/operators: no value
- Numbers: value is text string or the number
- Identifiers: value is text string or pointer to symbol table

Distinguishing keywords from identifiers:

- keep a table of keywords, *or*
- use symbol table

string	token type	
`"while"`	`WHILE`	`...`
`"count"`	`IDENTIFIER`	`...`
`"ok"`	`IDENTIFIER`	`...`

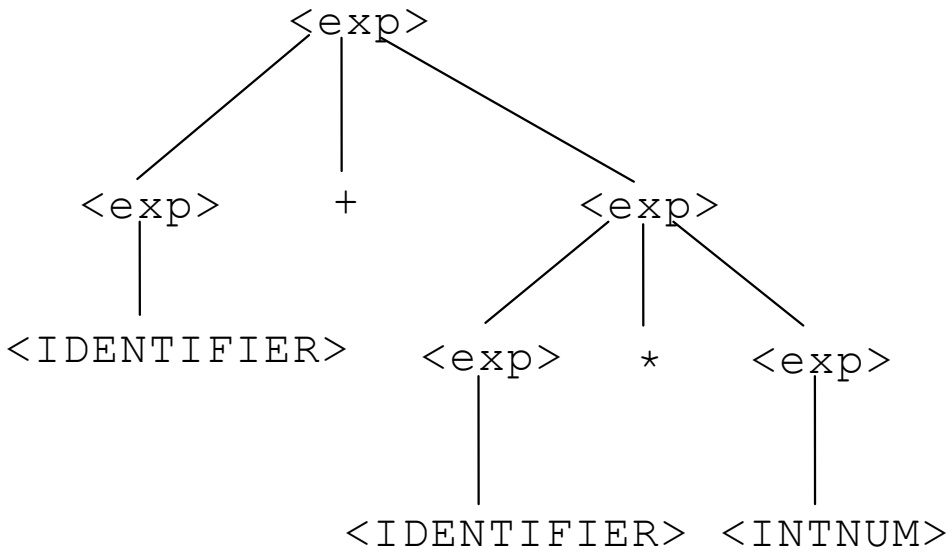
Syntax analysis (parsing)

Check structure of input tokens. Specified by context-free grammar.

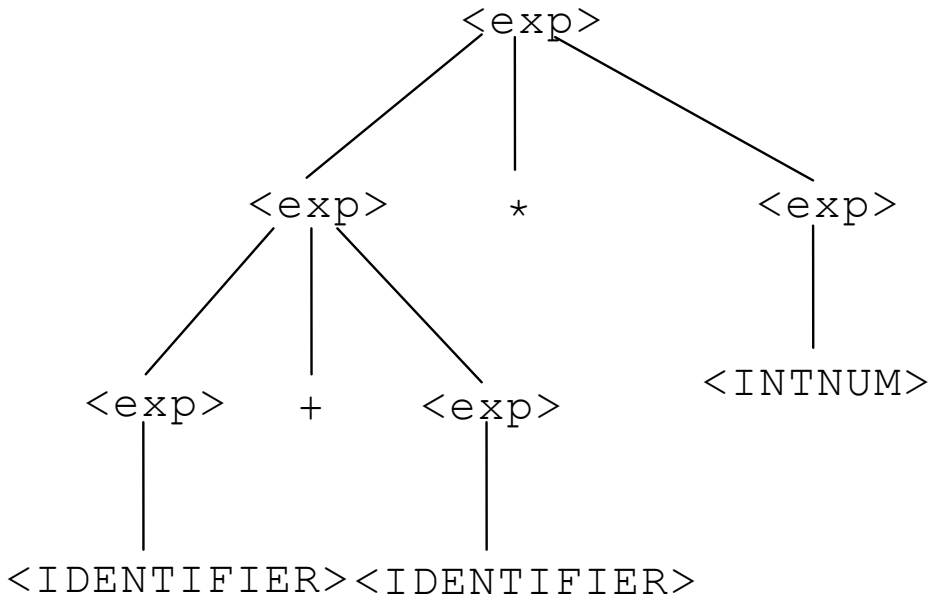
Example: arithmetic expressions:

For "`min+hours*60`"

Parse tree should be:



Not:



Why separate lexical and syntax analysis?

- Simplicity
- Efficiency
- Portability

Semantic analysis

Programming languages cannot be completely specified by context-free grammars.

Semantic analysis performs remaining checking and analysis.

Example: type checking:

- **Declaration**: insert variable's type in symbol table

```
int count;
```

string	token type	attributes
"count"	IDENTIFIER	type=int

- **Use**: check variable's type in symbol table

```
count = count+1;
```

Intermediate code generation

Intermediate code is for an *abstract* machine, not a specific real machine.

E.g.: `time = min+hours*60`

Generating code for a stack machine:

```
address time
value min
value hours
push 60
multiply
add
assign
```

Generating quadruples (3-address code):

```
t1 = hours*60
t2 = min+t1
time = t2
```

Memory management:

Address of a data object depends on its type and scope.

E.g.:

- register
- data segment of memory
- stack in memory

Type and scope of object can be found from symbol table entry.

Address allocated is added to symbol table for future use.

Intermediate code optimization

E.g., remove redundant operations:

```
time = min+hours*60
```

compiles to:

```
t1 = hours*60
time = min+t1
```

E.g., reuse temporary storage locations:

```
time = sec+60*(min+hours*60)
```

compiles to:

```
t1 = hours*60
t2 = min+t1
t1 = 60*t2
time = sec+t1
```

Code generation

Instruction selection:

E.g., compile `x = x+1` using `INC` (increment) instruction.

Register allocation:

- Data should be kept in registers as much as possible.
- Avoid moving data between registers and memory.
- How to allocate data to registers optimally?

Passes

Pass = reading input file and producing output file

One-pass compiler (large memory):

- Read whole source program file into memory and then generate code.

One-pass compiler (small memory):

- Interleave compiler phases.

Two-pass compiler:

- Reads source program and produces temporary file, then reads temporary file and generates code.

Syntax directed translation

One-pass compiler with limited memory must interleave phases.

Syntax analysis:

- demands tokens needed (for a statement) from lexical analyser
- does semantic analysis and code generation for that statement.

Compiler generators

Unix tools for syntax directed translation:

Lex/Flex:

Generates lexical analyser in C from regular grammar

Yacc/Bison:

Generates parser in C from context-free grammar