Compilers I

Chapter 1: Introduction

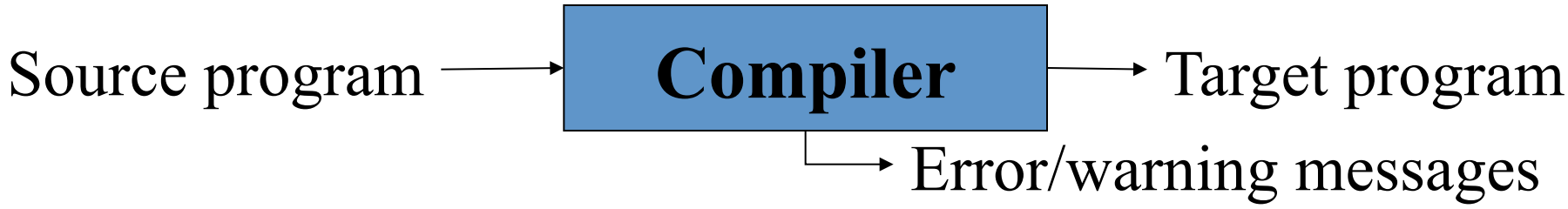
- Lecturers:
  - Paul Kelly (phjk@doc.ic.ac.uk)
    - Office: room 304, William Penney Building
  - Naranker Dulay (nd@doc.ic.ac.uk)
    - Office: room 562
- Materials:
  - CATE
  - Textbook
  - Course web pages (http://www.doc.ic.ac.uk/~phjk/Compilers)
  - Piazza (http://piazza.com/imperial.ac.uk/fall2015/221)

Acknowledgements: Chris Hankin

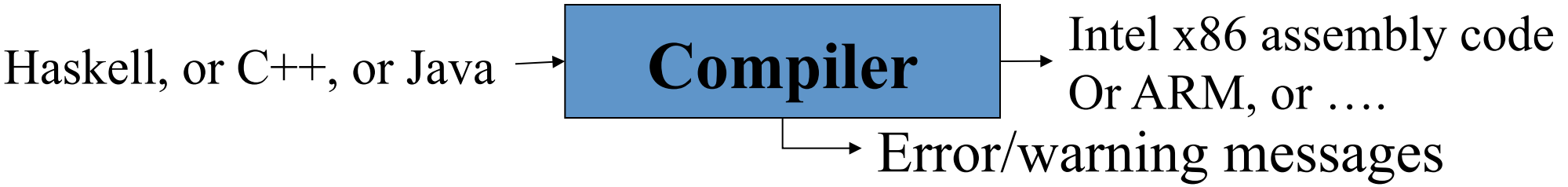
- This course is about a particular class of programs called *language processors*, of which the best example is a compiler.

What is a compiler?

- A program which processes programs, written in some programming language.
- A program which writes programs (in some language).
- A compiler translates programs written in one language into "equivalent" programs in another language.

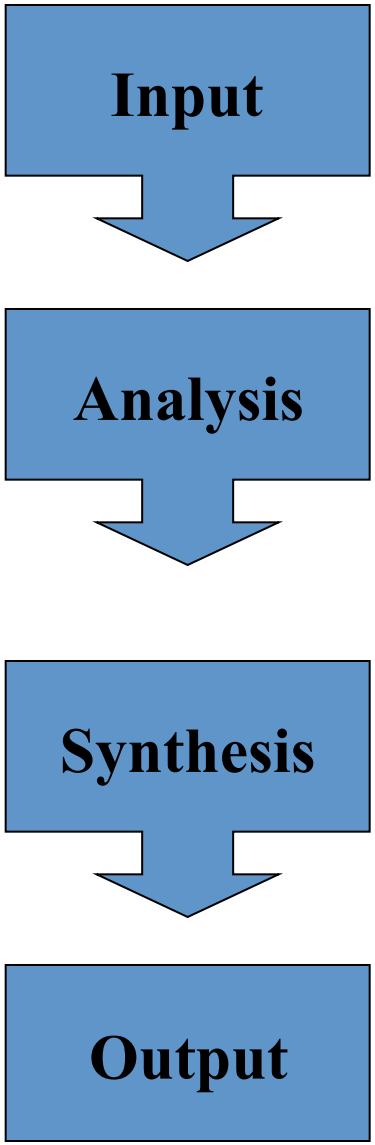


For example:



- Translates from one language into another
- **Or**: Output a low-level program which behaves as specified by the input, higher-level program.
- **That is**: Mediate between higher-level human concepts, and the word-by-word data manipulation which the machine performs.

Basic compiler structure



In some language

Construct an internal representation of the source language structure, and hence its meaning

Use this internal representation to construct target language version

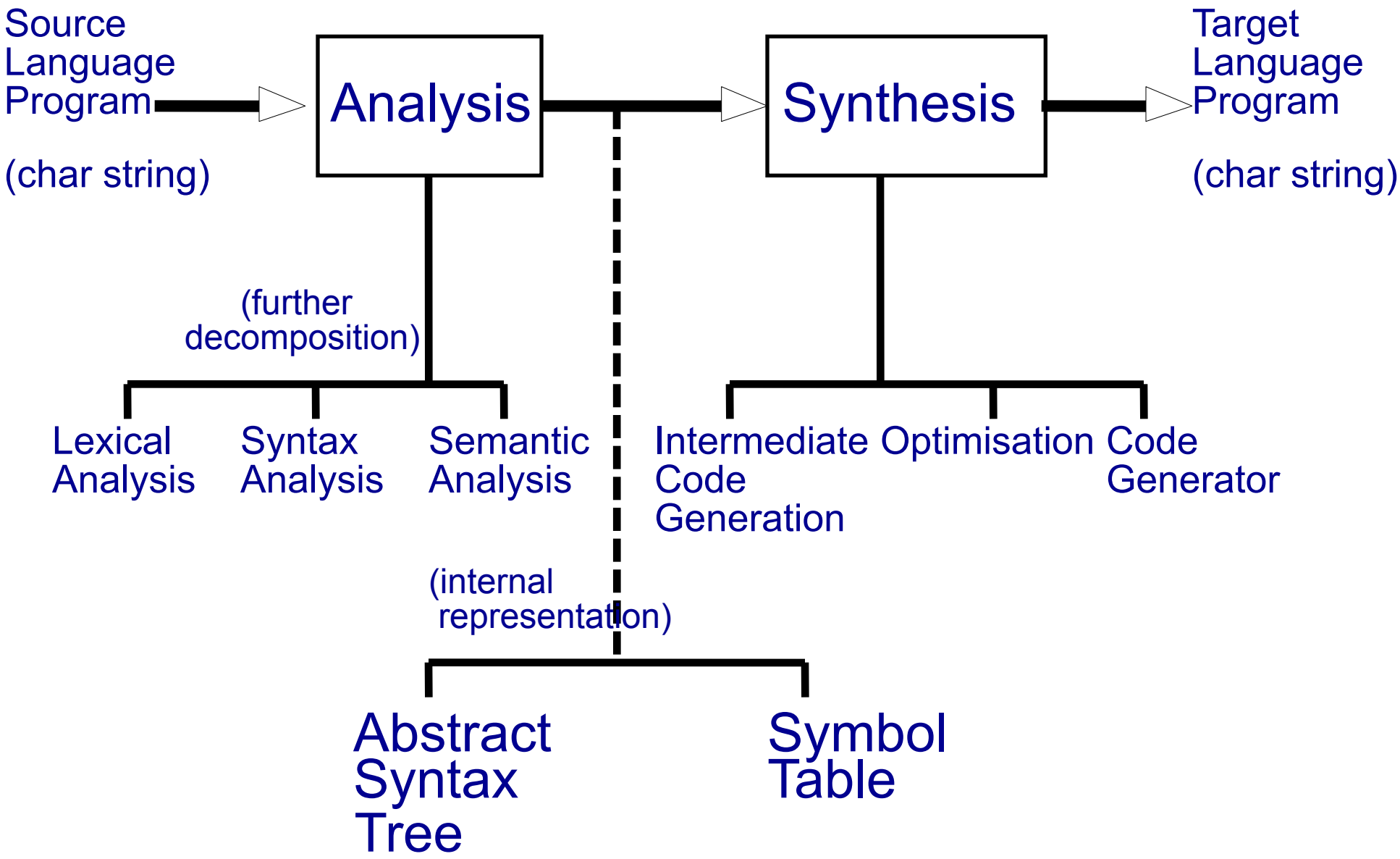
In the target language

Eg C, Java, C#

Usually start by building a tree representation, but may build graph, eg to represent control flow

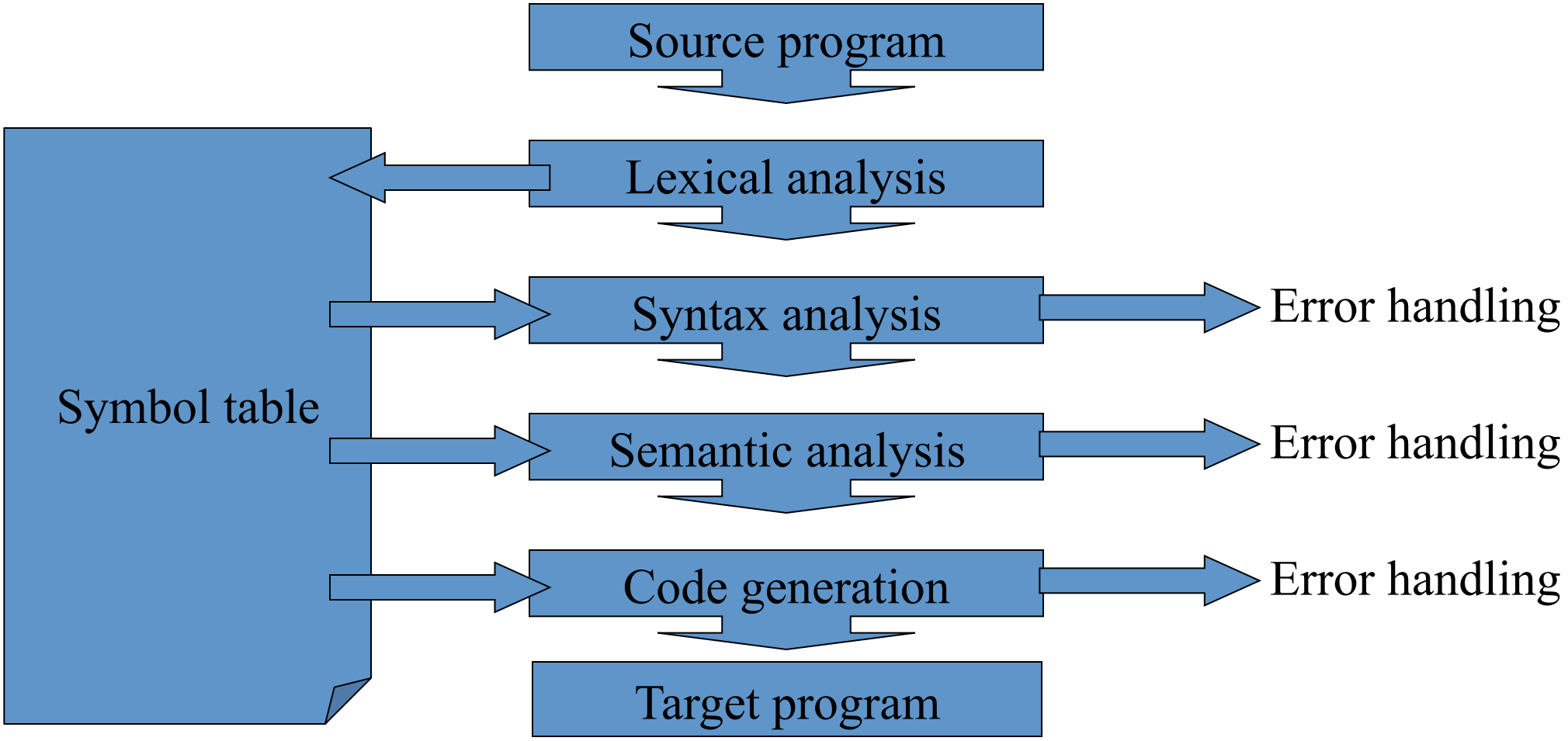
Analyse and transform the internal representation, then traverse it to produce output

Eg in Intel x86 assembler



Compiler structure in more detail

The phases of a compiler



Information from declarations is gathered in the symbol table, and is used to check how each variable is used, to reserve memory for it, and to generate code to access it

Phases of a compiler - example

- *Input file* "`test.c`":

```
int A;
int B;
test_fun()
{
   A = B+123;
}
```

- Command: "`gcc -S -O test.c`"

  (the flag "`-S`" tells the compiler to produce assembly code, "`-O`" turns optimisation on).

- *Output file* "`test.s`":

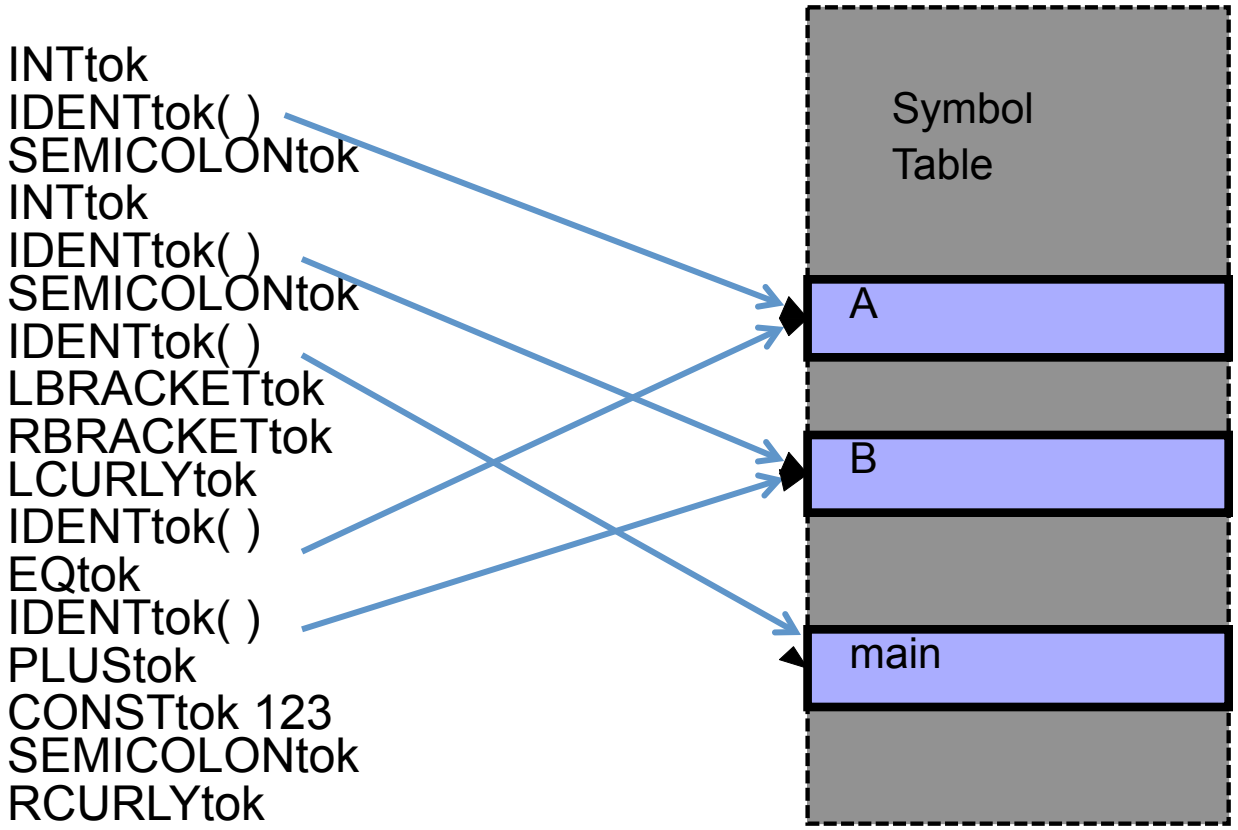
```
        .comm   _A, 4
        .comm   _B, 4
_test_fun:
        pushl %ebp
        movl %esp,%ebp
        movl _B,%eax
        addl $123,%eax
        movl %eax,_A
        movl %ebp,%esp
        popl %ebp
        ret
```

Introduction to lexical analysis

- INPUT: sequence of characters

```
int sp A ; nl int sp B ; nl main ( ) nl { nl A sp = sp B + 123 ; nl } nl
```

- OUTPUT: sequence of tokens:



User identifiers like `A`, `B` and main are all represented by the same lexical token (**`IDENTtok`**), which includes a pointer to a symbol table record giving the actual name.

Introduction to Syntax Analysis (also known as "parsing")

- Programming languages have grammatical structure specified by grammatical rules in a notation such as BNF (Backus-Naur Form)
- Example:

```
stat →   'print' expression |
         'while' expression 'do' stat |
         expression '=' expression
```

- The function of syntax analysis is to extract the grammatical structure— to work out how the BNF rules must have been applied to yield the input program.
- The output of the syntax analyser is a data structure representing the program structure: the **Abstract Syntax Tree** (AST).

Returning to our C example:

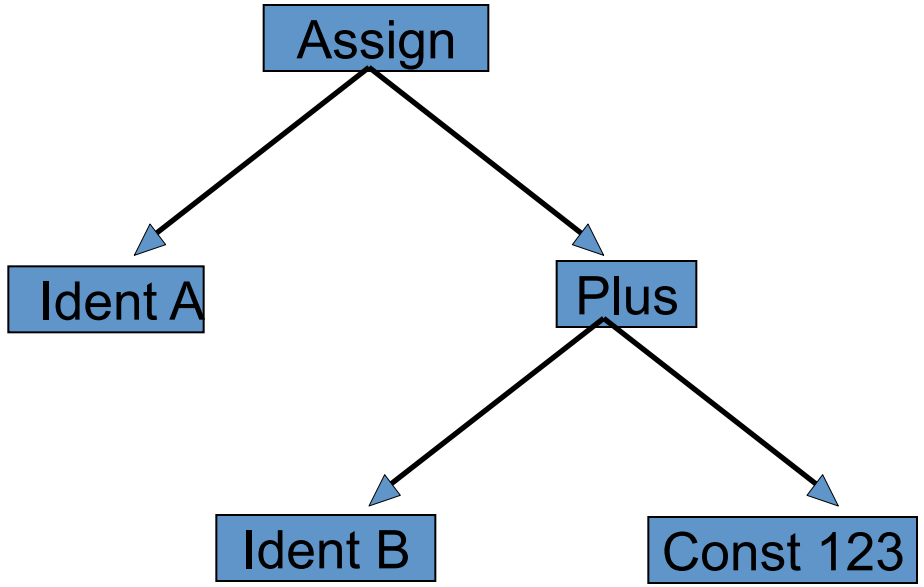
- Input characters:

  “.......A = B+123.......”

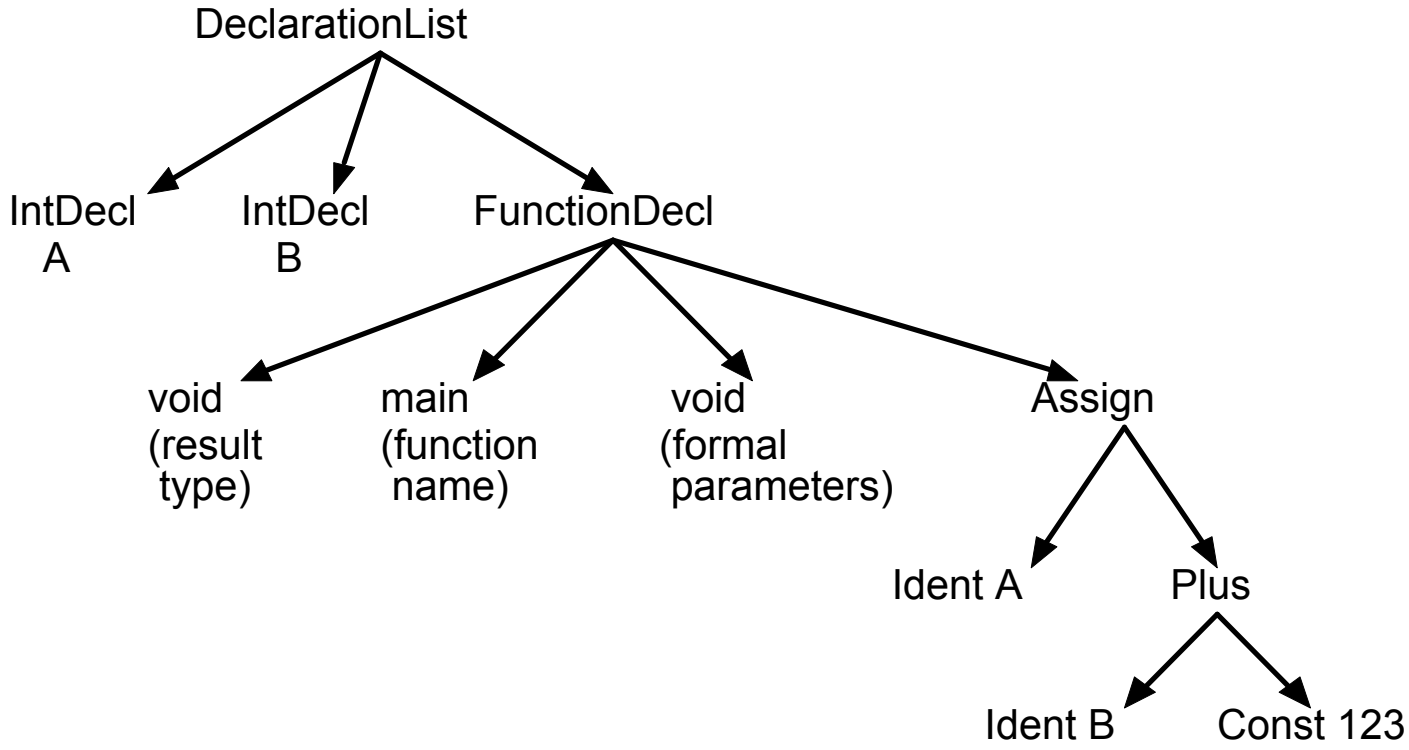
- Lexical tokens:

  [IDENTtok A, EQtok, IDENTtok B, PLUStok, CONSTtok 123]

- Abstract syntax tree:



AST for whole C example:



- The AST is implemented as a tree of linked objects
- The compiler writer must design the AST data structure carefully so that it is easy to build (during syntax analysis), and easy to use (e.g. during code generation).

You try: experimenting with real compilers

- Create a file "file.c" containing a simple C function
- Under Linux, type the command:

  gcc –O –S file.c
- This tells Gnu C compiler 'gcc' to produce optimised translation of the C program and leave result in "file.s"
- Examine "file.s"
- You might try

  gcc –O –S –fomit-frame-pointer file.c

  (This simplifies the output slightly)
- On Windows using Microsoft Visual Studio, try

  cl /Fa /Ox /c test.c

  (The output is written to "test.asm")

Compilers are just one kind of language processor:

- Really useful software tools are useful because they are programmable
- If it's programmable it must have some kind of programming language
- Programming languages are often "domain-specific" – designed for a particular application area
- Some languages are not for programming
- Domain-specific and embedded languages, eg:
  - Word processors, Spreadsheets (eg Word, Excel + Visual Basic)
  - Web server scripting and active web content (eg PHP, ASP, JSP)
  - Linux /etc/fstab, XFree86 config, Makefile, expect
  - XML (may be used to provide a common language for many of examples above)
  - Scripting languages (eg Linux bash shell, Perl, Python, Ruby)
  - LaTeX
  - The FEniCS Project's Unified Form Language (http://fenicsproject.org/)

Textbook - philosophy

- There are many textbooks on compilers, some good
- The purpose of lecture course is to give you enough understanding of the area to be able to use a serious textbook effectively
- Textbook should be worth more to you *after* the course than during it!
- Choose an authoritative book that goes substantially beyond this course

Recommended textbooks

- ***Compilers: Principles, Techniques, and Tools (second edition, 2006)*** by Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Monica Lam.
  - The new edition of the definitive book by pioneers in the subject. Often called the **Dragon book** because of the picture on the front. Compiler engineers regularly refer to "standard Dragon book stuff".
- ***Modern Compiler Implementation in Java (second edition, 2005)*** by Andrew Appel
  - Useful source for specific advice on how to build your compiler, including simple and more sophisticated techniques

The most recommended textbook

- ***Engineering a compiler***, by Keith Cooper and Linda Torczon, Morgan Kaufmann/Elsevier (2nd edition, 2011). About £44.
  - "In my opinion, this book is the best compiler engineering guide ever I read." (review, Amazon.com)
  - "This book has a good introduction guiding the beginning compiler student into understanding basic concepts and gradually revealing the more intimidating stuff, but the authors took great care not to scare the beginners away and instead offers great indepth explanations into how concepts and implementation merge. Its an overall good book!" (review, Amazon.com)
  - "*Three things stand out. First, all the algorithms are consistent with the latest research. Second, the explanations are exceptionally clear, especially compared to other recent books. Third, there's always enough extra context presented so that you understand the choices you have to make, and understand how those choices fit with the structure of your whole compiler*". (review, Jeff Kenton, comp.compilers 3/12/03)
  - "If you are a beginner "do not buy this book"" (review, Amazon.com)

How to enjoy, learn from and pass this course:

- Textbook
- “Linking and loading” lab exercise
- Make handwritten notes during lectures
- Tutorial exercises are used to introduce new examinable material
- Tutorials are designed to reinforce and integrate lecture material; it’s designed to help you pass the exam
- Go look at the past papers - now
- Use the tutorials to get feedback on your solutions
- You are assumed to have studied the past exam papers
- Substantial lab exercise should bring it all together
- Ask questions!

Chapter 1 – the main points again

- Credit will be given in exam and lab marking for evidence that you have used a textbook to expand on the material in the notes.
- The sooner you get the book and read the first couple of chapters the more worthwhile it will be.
- *Some of the later chapters offer far more detail than is needed for this course.*

- See also:
  - Sayler Foundation: http://www.saylor.org/courses/cs304/
  - Coursera: https://www.coursera.org/course/compilers