

Languages and Compilers

The Structure of a Compiler

Allan Milne and Adam Sampson
School of Design and Informatics
Abertay University

Syntax and Semantics.

- A programming language must be defined in terms of both its *syntax* and *semantics*.
- *Syntax* defines the format of the programs, statements and structures.
 - What a program looks like.
- *Semantics* defines the meaning of language constructs.
 - *What a program does.*

Semantics.

- Overall, we're usually translating from a **source program to low-level instructions**
- The compiler reads a program, and checks consistency of meaning
 - e.g. declaring identifiers, type consistency, operator usage,....
- The compiler generates artifacts according to the semantics of the language constructs.

Compiler Input.

- The primary input to a compiler is the *source program*.
- Formally, this is a “sentence” in the language supported by the compiler.
- The compiler reads the program as a stream of individual characters.

Compiler Output.

- A traditional compiler generates machine instructions or intermediate code (e.g. JVM instructions)
- A compiler might also...
 - generate some data structure, file or other artifact (e.g. Doxygen); or
 - directly perform some actions corresponding to the semantics (e.g. the bc calculator).

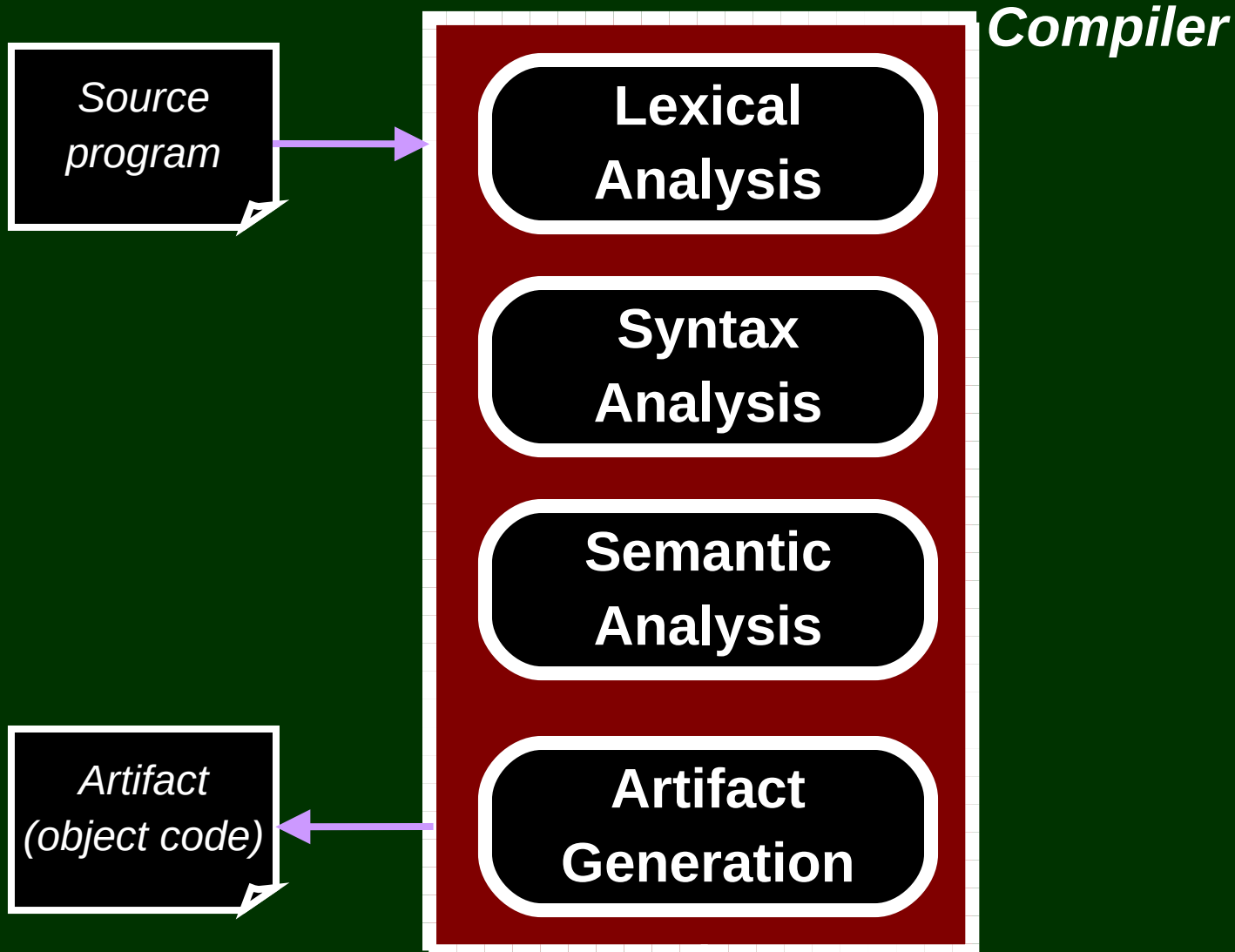
Compiler Components.

- Processing Phases.
- Interactions Via Data Structures.

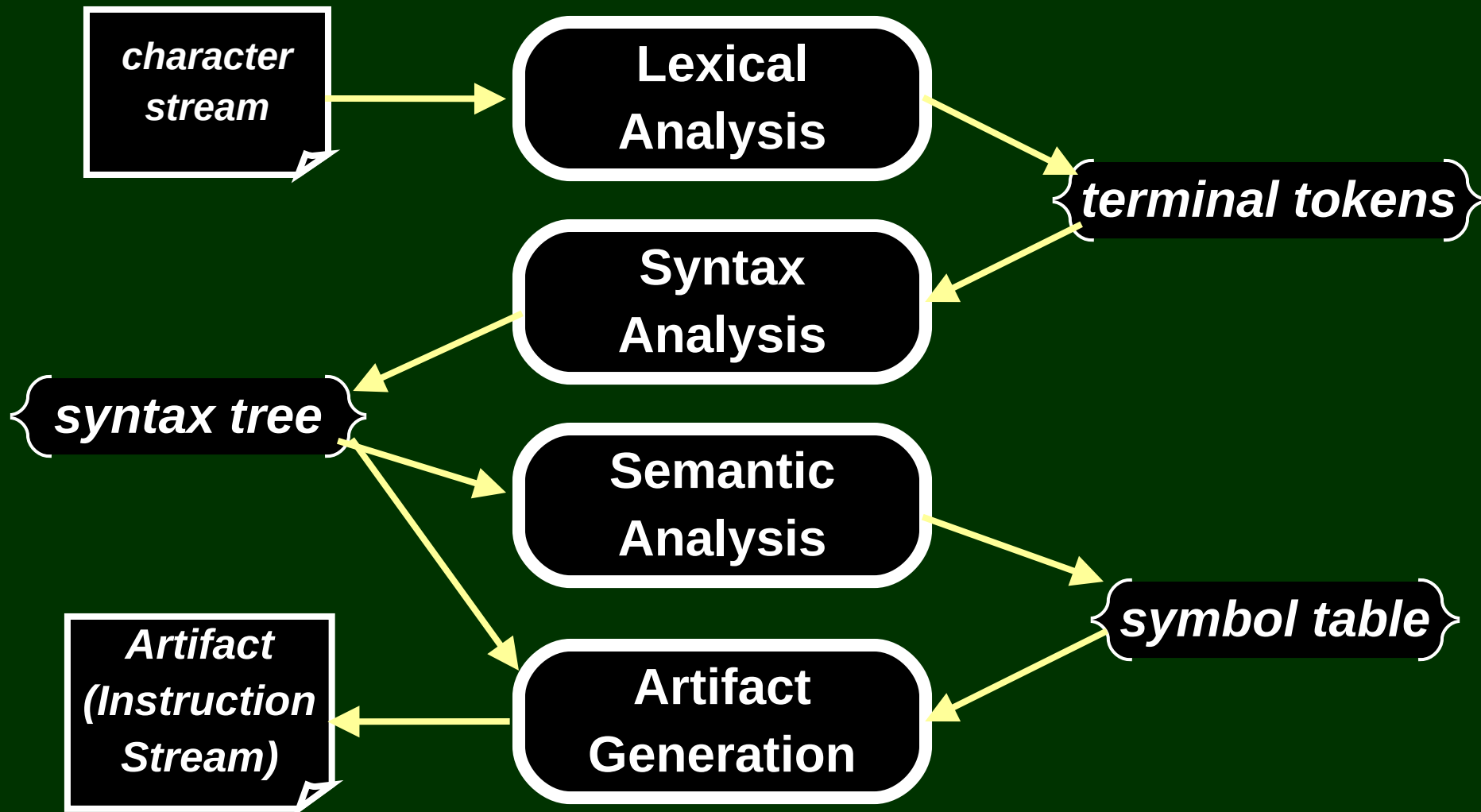
Example Walkthrough.

We'll come back to all of these
in more detail later on in the module...

Compiler Phases.



Component Interactions.



By Example.

- The syntax of our programming language is defined by a formal **grammar** – e.g.

```
... ..  
<If> ::= if <Bool> then <Stat> else <Stat> ;  
<Bool> ::= <Expr> <RelOp> <Expr> ;  
<RelOp> ::= "<" | "<=" | "=" | ">" | ">=" ;  
... ..
```

- Let's suppose we're compiling this bit of code:

```
... ..  
if a>5 then x=a else x=5  
... ..
```

Lexical Analysis (Scanning).

- Performed by the *lexical analyser* or *scanner*.
- The source program is read in as a stream of characters and output as a stream of *tokens*.
- Tokens represent the “words” in the language (formally, the **terminal symbols**)
- So in our example the output is

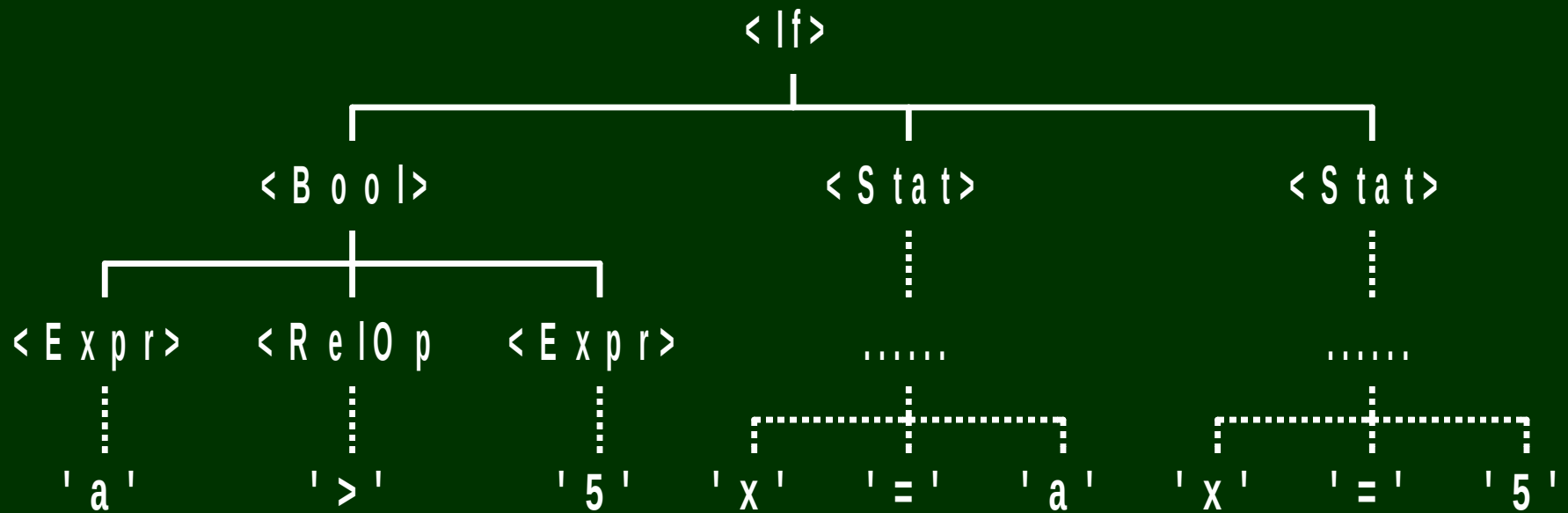
```
... ..  
'if' 'a' '>' '5' 'then' 'x'  
'=' 'a' 'else' 'x' '=' '5'  
... ..
```

Syntax Analysis (Parsing).

- Performed by the *syntax analyser* or *parser*.
- The parser recognises “sentences” in the language, based on the language definition.
- It builds an internal tree representation of the program's syntactic structure – a **parse tree** or **abstract syntax tree (AST)**

AST

`if a>5 then x=a else x=5`



Semantic analysis.

- Puts the “sentences” into context, and checks the overall meaning is consistent
 - e.g. declaration of identifiers, type checking – if I've declared a variable as an integer on one line, I shouldn't use it as a different type later
- Manages a **symbol table** to track information about **names** in the program, e.g.

<i>Identifier</i>	<i>Type</i>	<i>Address</i>	<i>Other ...</i>
a	Integer	124
x	Integer	128


Artifact Generation.

- Processes information from the syntax tree & symbol table to create appropriate artifacts.
- The generated output depends on
 - the semantics of the language, and
 - the target execution platform.
- e.g. the assignment $x := y$ might turn into `MOV r2, r3` – the symbol table says which registers x and y are stored in

Compiler Structure.

- Overview.
- Multi-Pass.
- Single-Pass.

Multi-Pass Structure.

- **Phases** run sequentially.
 - Scanner reads in entire program and produces entire stream of tokens.
 - Parser reads in all the tokens and produces the syntax tree.
 - Semantic analyser traverses the entire tree and produces the symbol table.
 - Artifact generator traverses syntax tree and accesses symbol table to produce artifacts for the entire program.

Multi-Pass Pros and cons.

- Each phase produces a data structure which the next phase reads
 - Can actually be separate programs, storing data in files... (e.g. GCC works this way)
- No limits on language complexity or the order in which information is available
- Lexical, syntax and semantic analysers can all interact with the syntax tree and symbol table data structures

Nanopass Structure

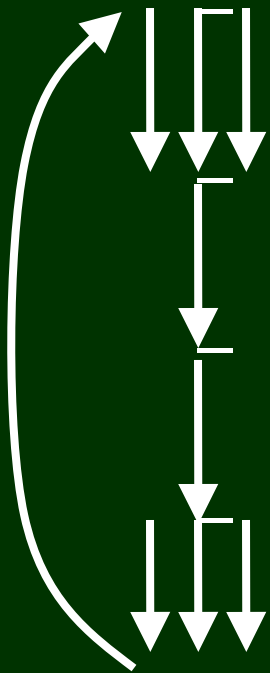
- Like multi-pass but more so – potentially **hundreds** of tiny passes, each of which makes a simple transformation
- e.g. in a C-like language, nanopasses might include:
 - “turn for loops into while loops”
 - “turn while loops into if+goto”
 - “turn gotos into JMP instructions”

Nanopass Pros and Cons

- **Pro:** Tiny passes are easier to get right – and much easier to write unit tests for
- **Pro:** Easy to slot in new passes to extend the compiler – e.g. new optimisations, new language features
- **Con:** Parse tree needs to be able to represent intermediate states of the program – and must be efficient to search/transform for good performance

Single-Pass Structure.

- Phases run in an interleaved manner.



Scanner reads in characters that make up a single token.

Parser takes tokens to build a single syntactic element.

Semantic analyser processes syntactic element and updates the symbol table.

Artifact generator processes meaningful group of syntactic elements and accesses symbol table to produce artifacts for that element.

Single-Pass Pros and cons.

- The compiler only needs a symbol table – no AST necessary!
- Simple implementation (recursive descent)
- Can't handle all languages – has to make sense when encountered in order (e.g. you can't use a variable that's not been defined yet – we'll see more limitations later)
- Does syntactic and semantic processing at the same time

Passes in Practice

- What do **real** compilers do?
- Simple calculator-like tools are single-pass
- Traditional “big” compilers – e.g. GCC, Visual Studio – are multi-pass with a few passes
- Cutting-edge compilers – e.g. LLVM/Clang – are somewhere between multipass and nanopass – there's a move towards reusable components

Other Compiler Functions.

- Error Handling.
- Compiler Control.
- Executable Code Linking & Loading.

Error Handling.

- The compiler must handle syntactic and semantic errors.
- **Error detection**: find invalid syntactic structures or misuse of semantics.
- **Error reporting**: report to the user the type, position and cause of the error.
- **Error recovery**: allow the compilation process to continue (or not!) after an error.

Compiler Control.

- The compiler may be controlled externally or internally in terms of aspects of its operation.
 - E.g. analysis, generation and/or reporting functions.
- External control is often through the use of *flags* when the compiler is initiated.
- Internal control is often through *compiler directives* embedded in the source program.
 - Note these are not part of the language (dubious... discuss!)

Executable Code Linking and Loading.

- Before generated code can be executed
 - the linker must link it with separately compiled and/or library code components, and then
 - the loader must load it into the execution platform memory, and execution must be initiated at an appropriate point.
- The compiler must adhere to any code format standards expected by the linker and loader. (Modern linkers can do complex work too – e.g. *whole-program optimisation*).

Summary.

So Now You Can ...

- ... identify the processing phases and data structures of a compiler.
- ... explain the role of each phase and the contents of each data structure.
- ... describe multi-pass, nanopass and single-pass compiler structures.
- ... identify and describe other compiler functions.