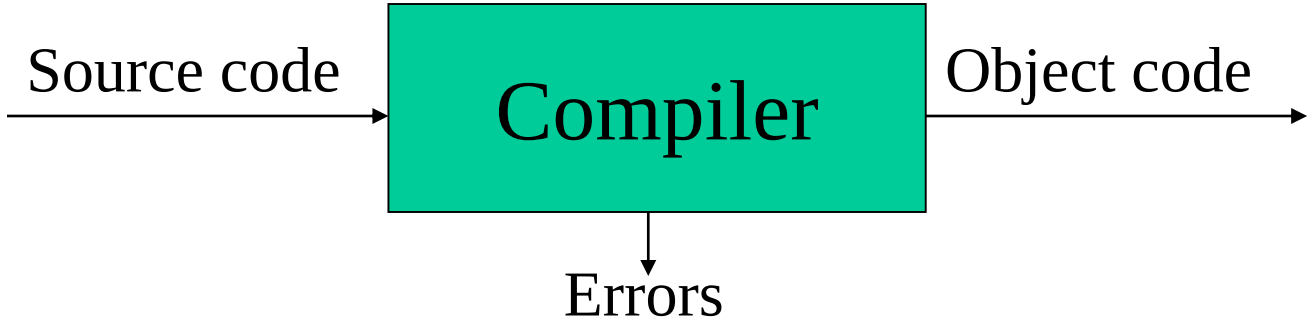
Overview of Compilers



(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.

Recap- What are Compilers?

- Translates from one representation of the program to another
- Typically from **high level source code** to **low level machine code or object code**
- **Source code** is normally optimized for human readability
  - **Expressive**: matches our notion of languages (and application)
- **Machine code** is optimized for hardware
  - Redundancy is reduced
  - Information about the intent is lost

Goals of translation

- **Correctness**
  - A very important issue !!
  - To prove correctness is tedious
  - Correctness has an implication on the development cost.
- **Performance**
  - Good compile time performance
  - Good performance for the generated code
- **Maintainable code**

How to translate?

- **Direct translation is difficult**. Why?

- Source code and machine code mismatch in **level of abstraction**
  - Variables *vs* Memory locations/registers
  - Functions *vs* jump/return
  - Parameter passing
  - structs

- Some languages are farther from machine code than others
  - For example, languages supporting Object Oriented Paradigm

How to translate?

- Translate in steps.
  - Each step handles a reasonably simple, **logical**, and **well defined task**

- Design a **series of program representations**

- **Intermediate representations** should be amenable to program manipulation of various kinds (*type checking*, *optimization*, *code generation* etc).

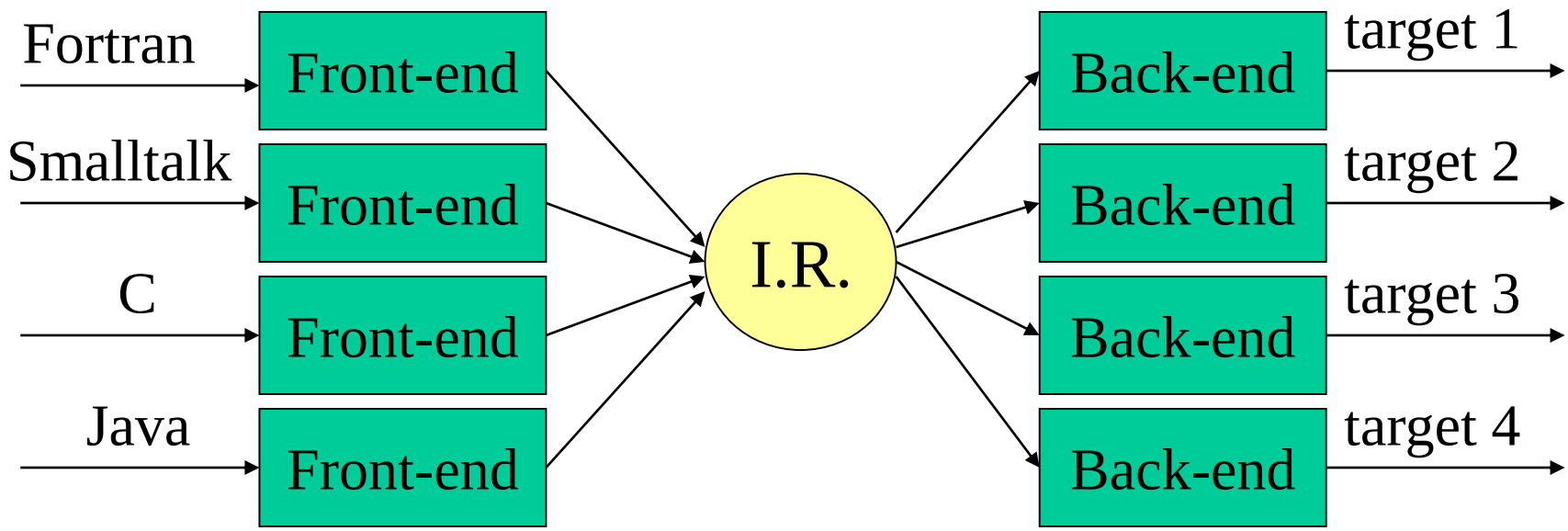
- As the translation proceeds, representations become ***more machine specific*** and ***less language specific***

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.

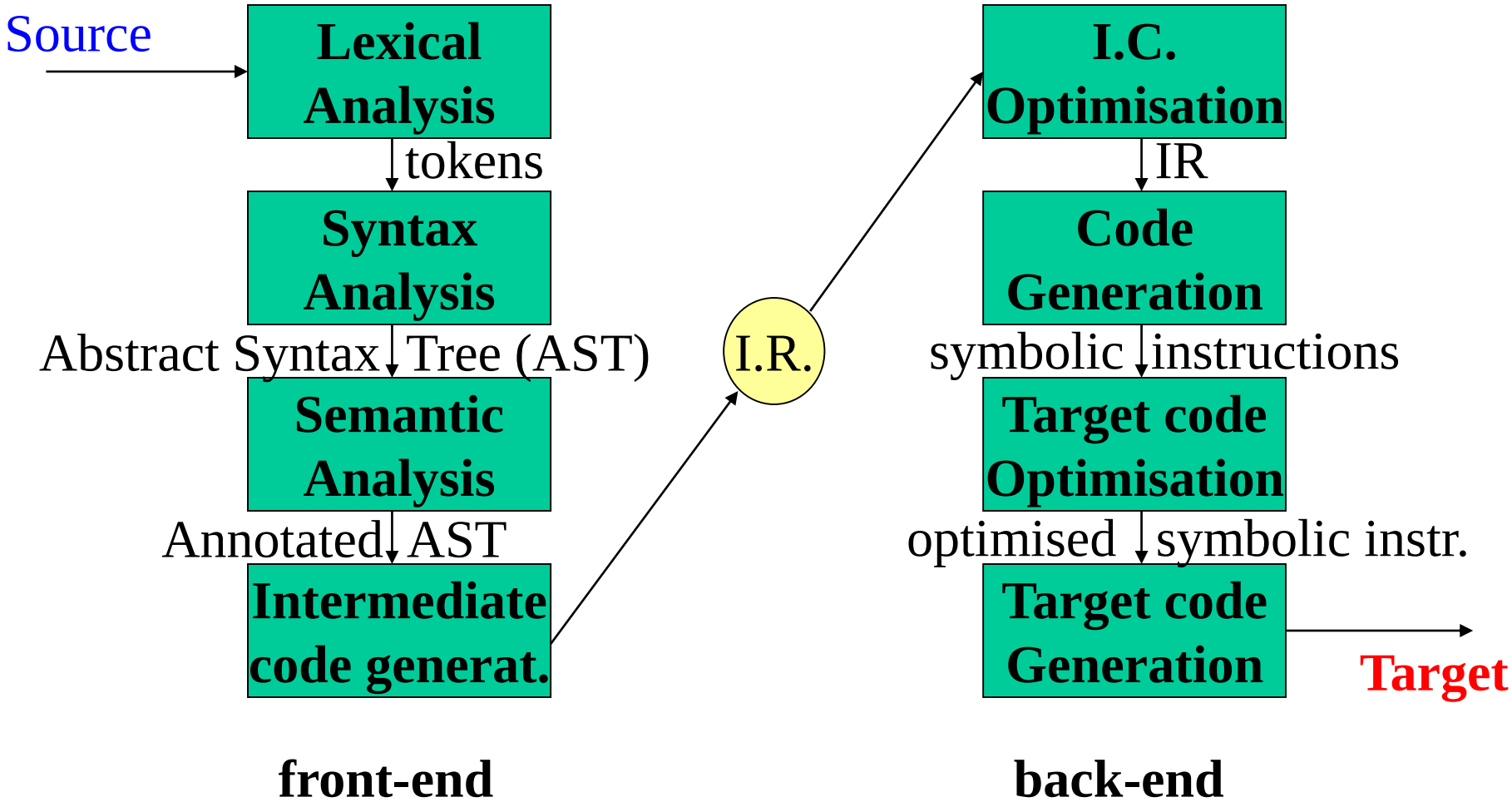
- **Back-end** does the target language **synthesis**:
  - Chooses instructions to implement each IR operation.
  - Translates IR into target code.

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

General Structure of a compiler



The first few steps

- The first few steps can be understood by analogies to how humans comprehend a natural language

- The first step is recognizing/knowing alphabets of a language. For example
  - English text consists of lower and upper case alphabets, digits, punctuations and white spaces

- The next step to understand the sentence is recognizing words
  - How to recognize English words?
  - Words found in standard dictionaries
  - Dictionaries are updated regularly

Recognizing words in a programming language

- How to recognize words in a programming language?
  - a dictionary (of **keywords** etc.)
  - rules for constructing words (**identifiers,** numbers etc.)

- This is called lexical analysis

- Recognizing words is not completely trivial.

For example:

w hat ist his se nte nce?

Lexical Analysis: Challenges

- We must know what the **word separators** are
- The language must **define rules** for breaking a sentence into a sequence of words.
- Normally white spaces and punctuations are word separators in languages.
- In programming languages a **character from a different class** may also be treated as word separator.

Lexical Analysis (Scanning)

- **Reads characters** in the source program and **groups them into words** (basic unit of syntax)

- The lexical analyzer breaks a sentence into a sequence of words (or tokens):
  - if a == b then a = 1 ; else a = 2 ;
  - Sequence of words (total 14 words)

    if a == b then a = 1 ; else a = 2 ;

- Lexical analysis eliminates white space, etc…

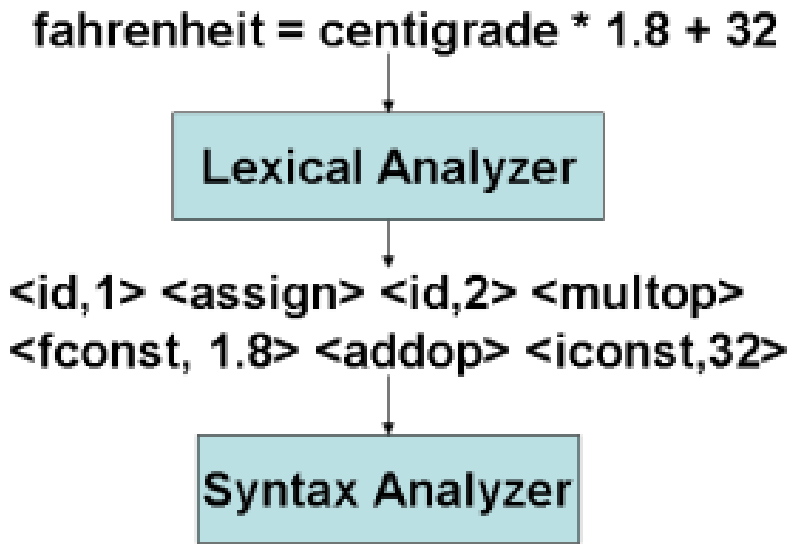
Lexical Analysis (Scanning)

- 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`**>

Lexical Analysis



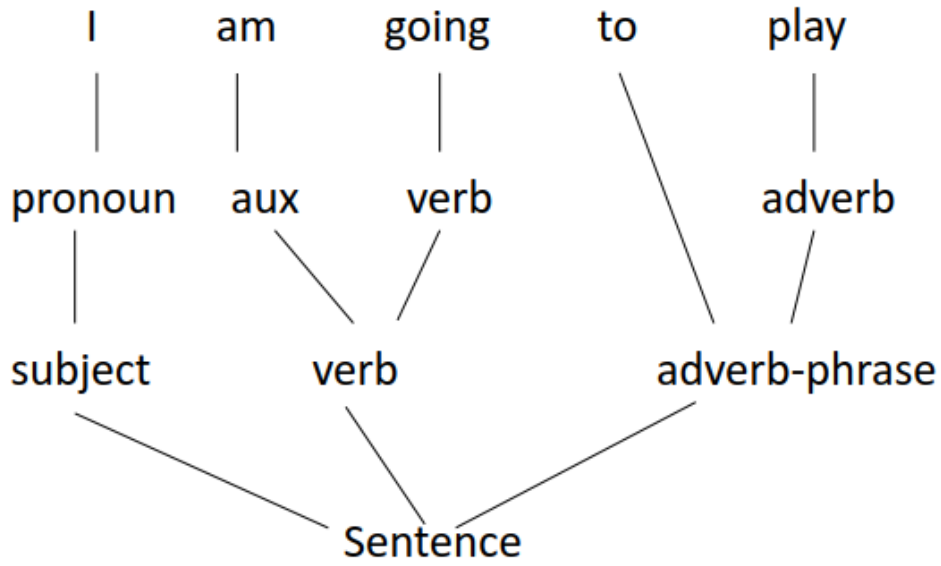
Lexical Analysis

- LA can be generated automatically from regular expression specifications

  – **LEX** and **Flex** are two such tools

The next step

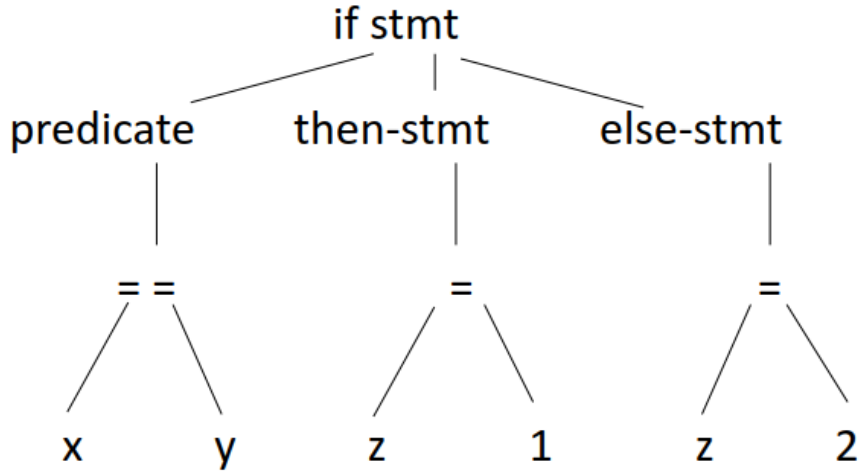
- Once the words are understood, the next step is to understand the structure of the sentence
- The process is known as syntax checking or parsing



Parsing

- Parsing a program is exactly the same process as shown in previous slide.
- Consider an expression

  if x == y then z = 1 else z = 2



Parsing (Syntactic Analysis)

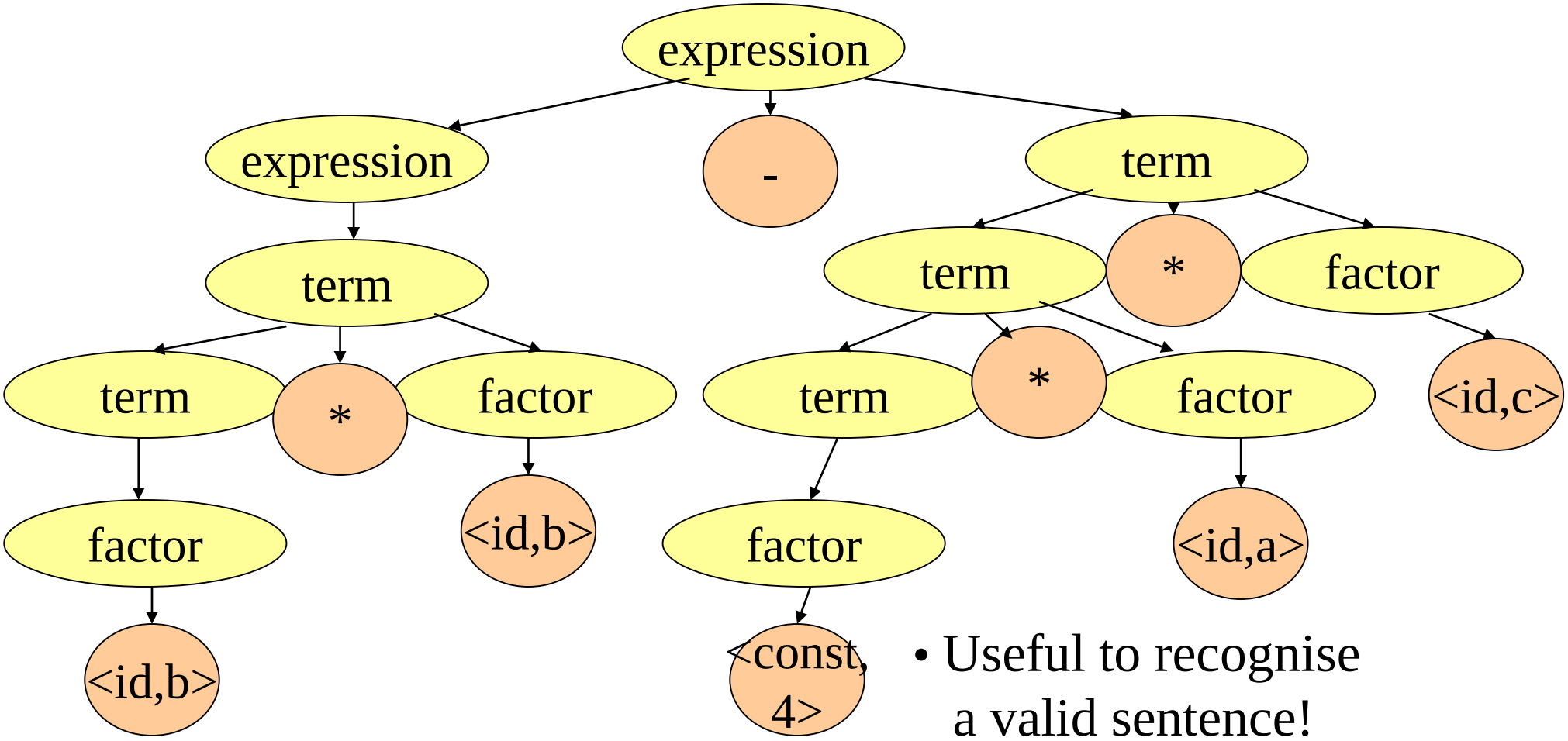
- Imposes a hierarchical structure on the token stream.
  - This hierarchical structure is usually expressed by recursive rules.
  - Context-free grammars formalise these recursive rules and guide syntax analysis.
- Syntax analyzers (parsers) **can be generated automatically from several variants of context-free grammar specifications**
  - E.g.: **ANTLR** (for LL(1)), **YACC** and **Bison** (for LALR(1)) are such tools
- Example:

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

(this grammar defines simple algebraic expressions)

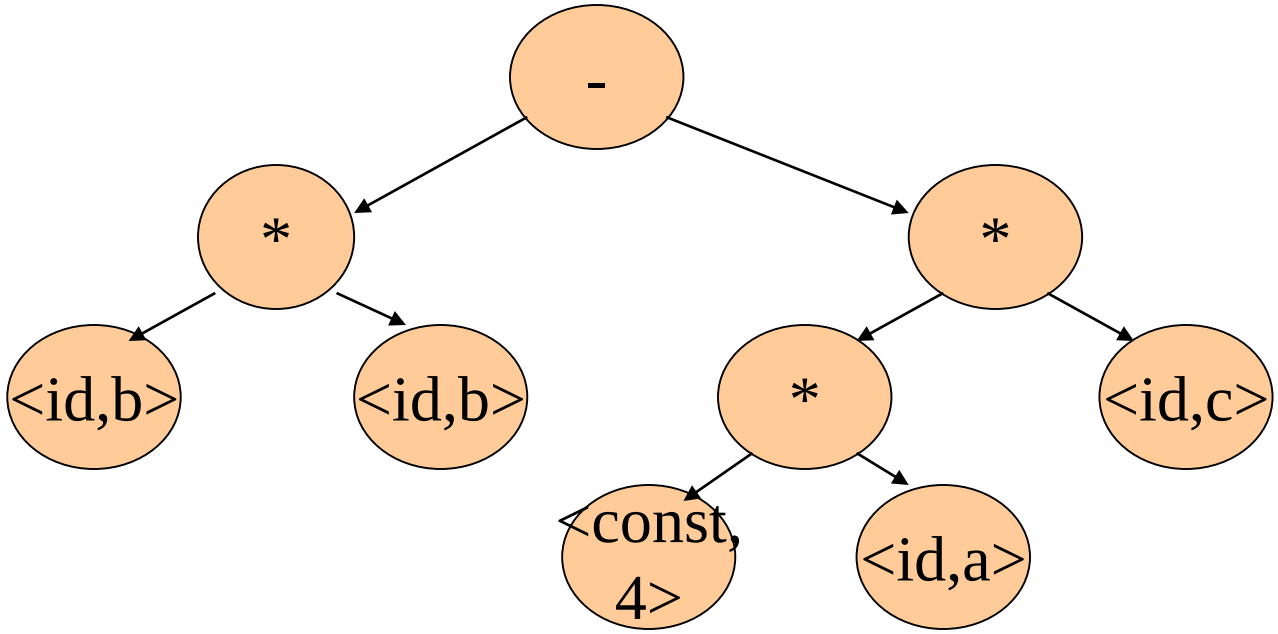
*b*b-4*a*c (valid expression ?)*

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



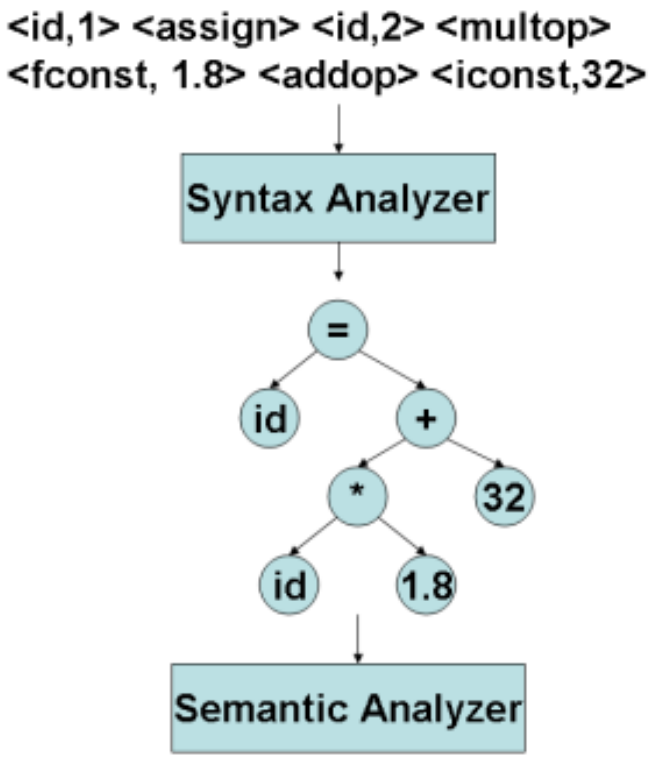
- Useful to recognise a valid sentence!

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)
- **AST**s are one form of **IR**

Parsing or Syntax Analysis



Understanding the meaning

- Once the sentence structure is understood we try to understand the meaning of the sentence (**semantic analysis**)
- A challenging task
- Example:

  Prateek said Nitin left his assignment at home

  • What does his refer to? Prateek or Nitin?

Understanding the meaning

- Worse case

  Amit said Amit left his assignment at home

- Even worse

  Amit said Amit left Amit’s assignment at home

- How many Amits are there? Which one left the assignment? Whose assignment got left?

Semantic Analysis

- Too hard for compilers. They do not have capabilities similar to human understanding
- However, compilers do perform analysis to understand the meaning and catch inconsistencies
- Programming languages define **strict rules to avoid such ambiguities**

```
{ int Amit = 3;
    { int Amit = 4;
      cout << Amit;
    }
}
```

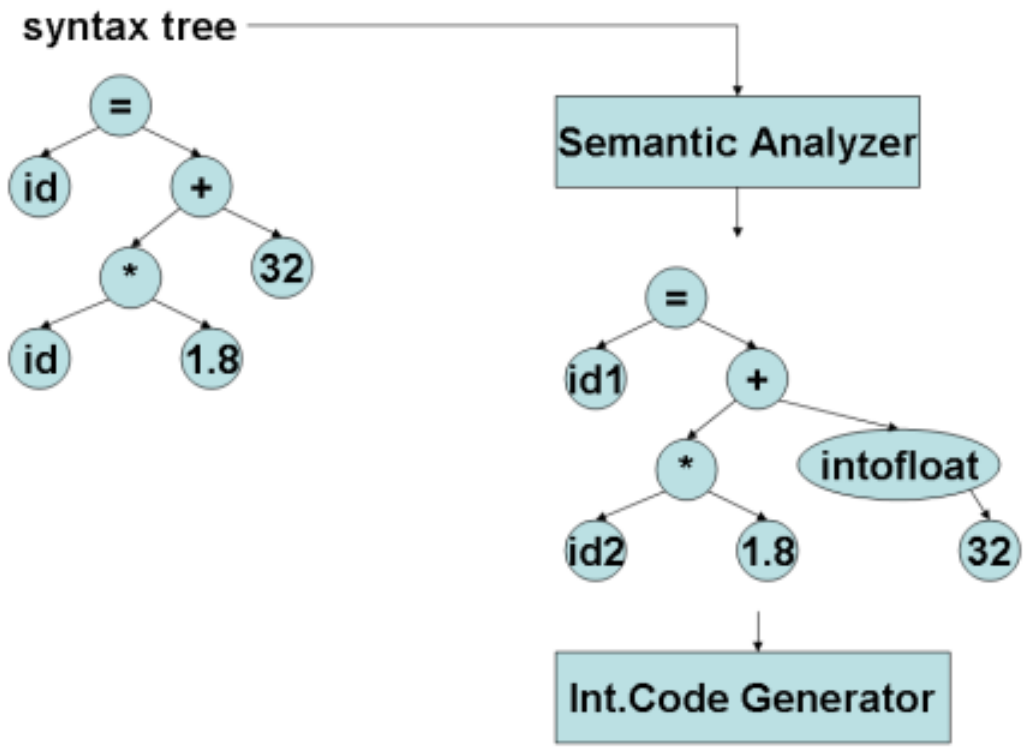
More on Semantic analysis

- Compilers perform many other checks besides variable bindings
- Type checking
- *Amit left her work at home*
- There is a **type mismatch** between her and Amit. Presumably Amit is a male. And they are not the same person.

Semantic Analysis

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

Semantic Analysis



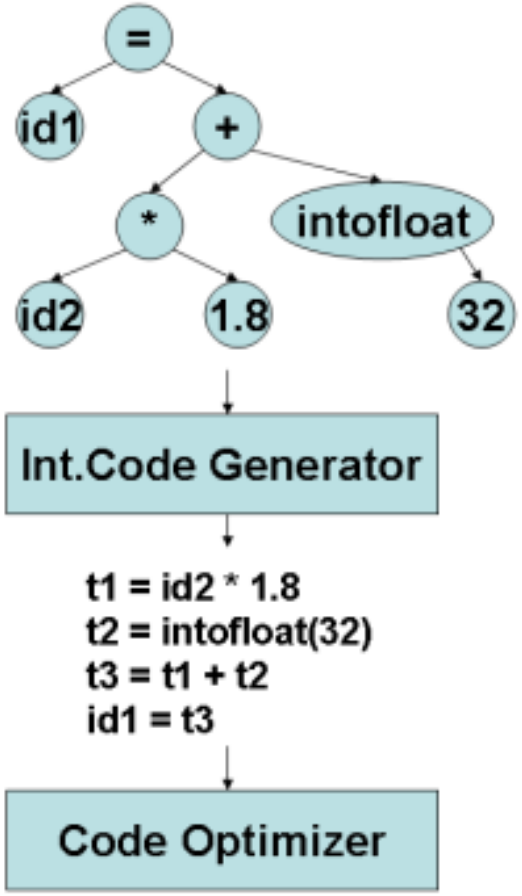
Intermediate code generation

- While generating machine code directly is possible, it entails **two problems**
  - With **m** languages and **n** target machines, we need to write **m×n** compilers
  - The **code optimizer** which is one of the **largest and very-difficult-to-write components** of any compiler **cannot be reused**.

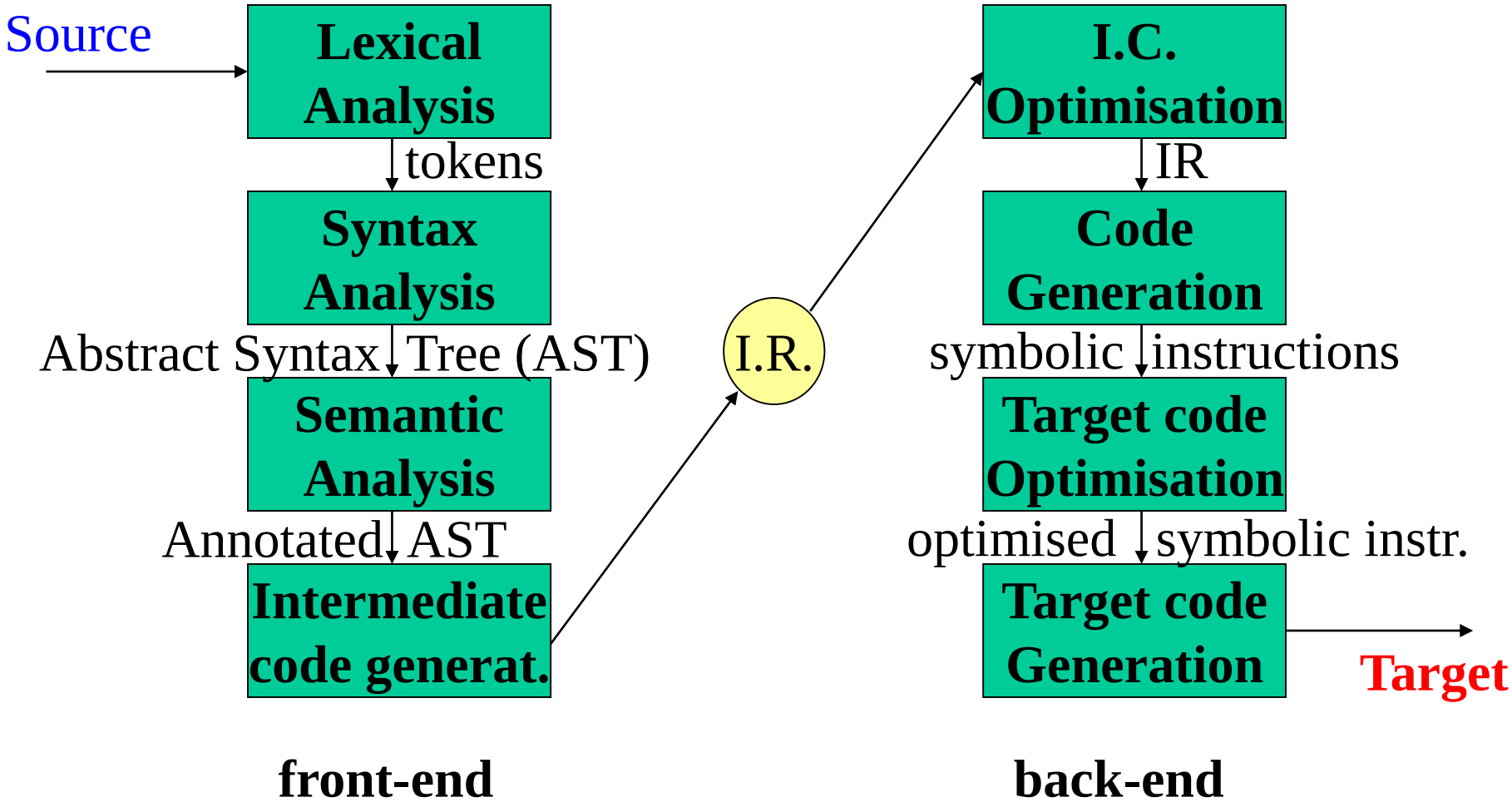
- By converting source code to an intermediate code, a **machine-independent code optimizer** may be written

- Intermediate code must be easy to produce and easy to translate to machine code
  - A sort of **universal** assembly language
  - Should not contain any **machine-specific** parameters (registers, addresses, etc.)

Intermediate Code Generation



General Structure of a compiler



Different Types of Intermediate Code

- ***Quadruples***, ***triples***, ***indirect triples***, ***abstract syntax trees*** are the classical forms used for machine-independent optimizations and machine code generation

- ***Program Dependence Graph (PDG)*** is useful in automatic parallelization, instruction scheduling, and software pipelining

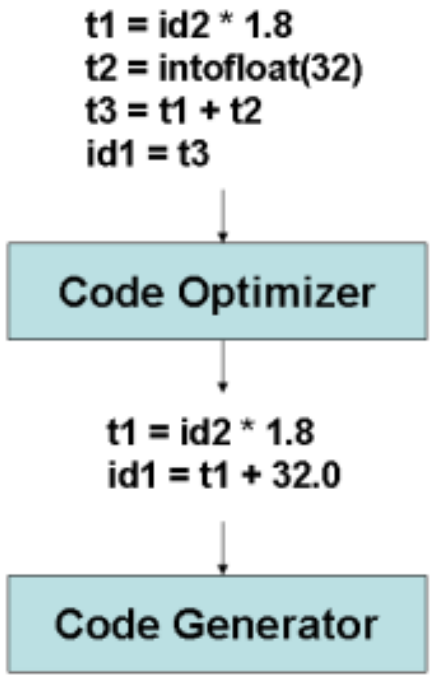
Machine Independent Code Optimization

- Intermediate code generation process **introduces many inefficiencies**
  - Extra copies of variables, using variables instead of constants, repeated evaluation of expressions, etc.
- **Code optimization removes such inefficiencies** and improves code
- It changes the structure of programs, sometimes of beyond recognition
  - eliminates some programmer-defined variables, in-lines functions, unrolls loops, etc.

Code Optimization

- Some common optimizations
  - Common sub-expression elimination
  - Copy propagation
  - Dead code elimination
  - Code motion
  - Constant folding
- **Example**: **x = 15 * 3** is transformed to **x = 45**
- Data-flow analysis used to gather required information.

Code Optimization



Code Generation

- Converts intermediate code to machine code

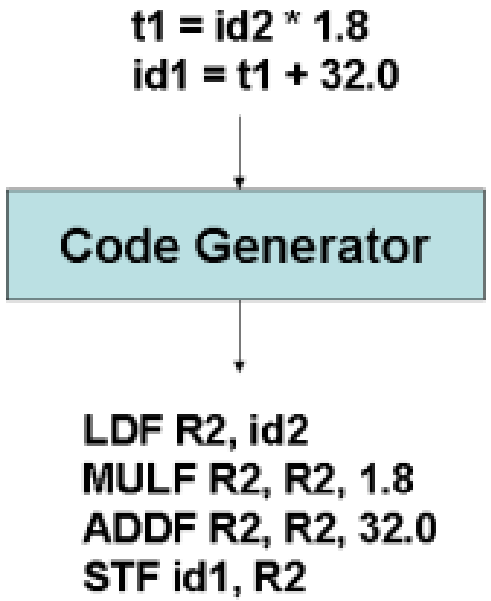
- **Each intermediate code instruction may result in many machine instructions or vice-versa**

- Must handle all aspects of machine architecture
  - Registers, pipelining, cache, multiple function units, etc.

- Generating efficient code
  - Tree pattern matching-based strategies are among the best

- **Storage allocation decisions** are made here
  - ***Register allocation and assignment*** are the most important problems

Code Generation



Code Generation

- Map **identifiers** to **locations** (memory/storage allocation)
- Explicate variable accesses (change **identifier reference** to **relocatable/absolute address**)
- Map source **operators** to **opcodes** or a sequence of opcodes
- Convert **conditionals** and **iterations** to a **test/jump** or **compare** instructions
- ……

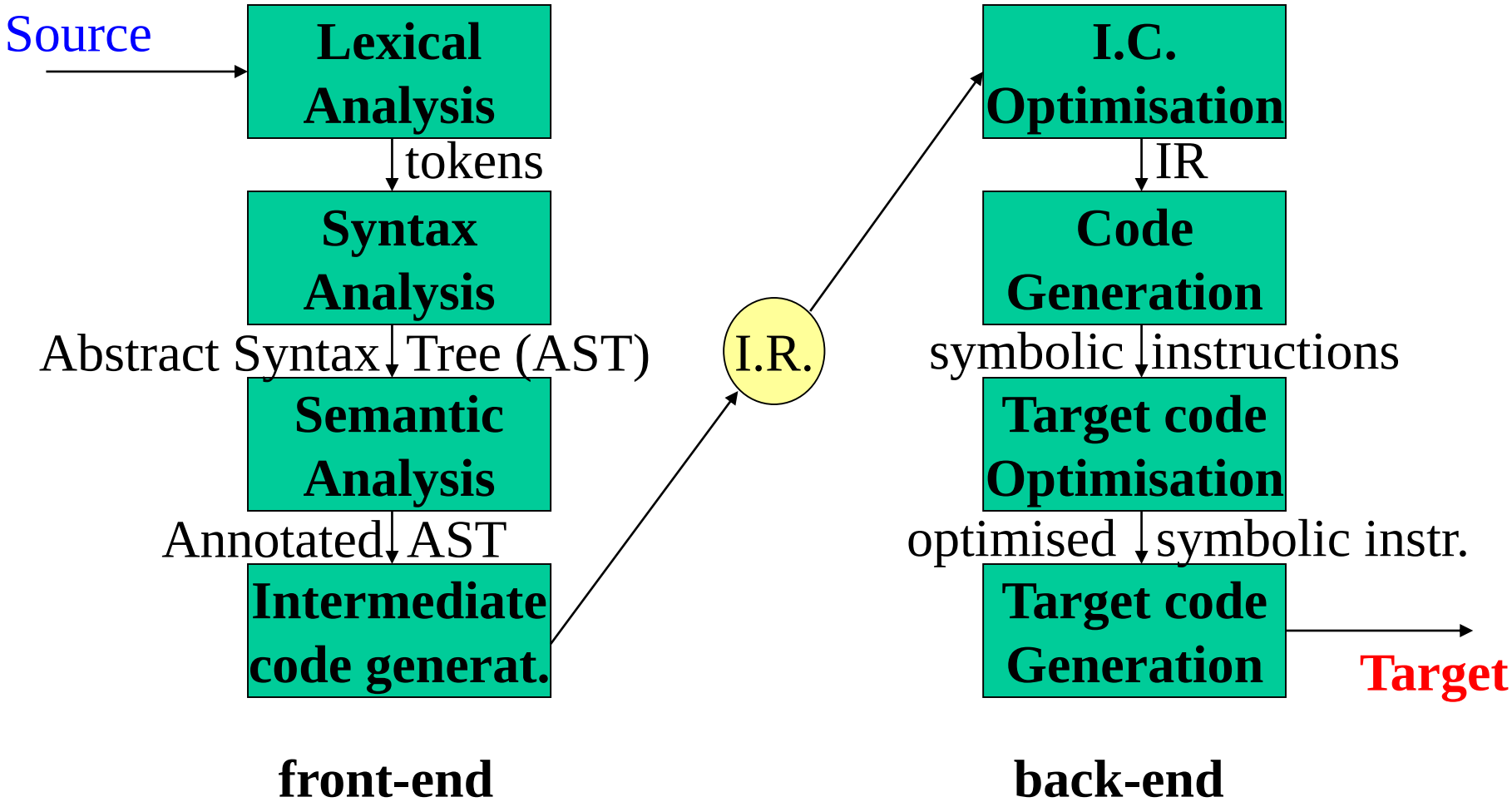
Post translation optimizations

- Machine dependent optimizations

- **Peephole optimizations**
  - Analyze sequence of instructions in a small window (peephole) and using preset patterns, replace them with a more efficient sequence
  - Redundant instruction elimination

    e.g., replace the sequence [LD A,R1][ST R1,A] **by** [LDA,R1]
  - Use machine idioms (use INC instead of LD and ADD)
- Instruction scheduling (reordering) to eliminate pipeline interlocks and to increase parallelism
- Trace scheduling to increase the size of basic blocks and increase parallelism
- Software pipelining to increase parallelism in loops

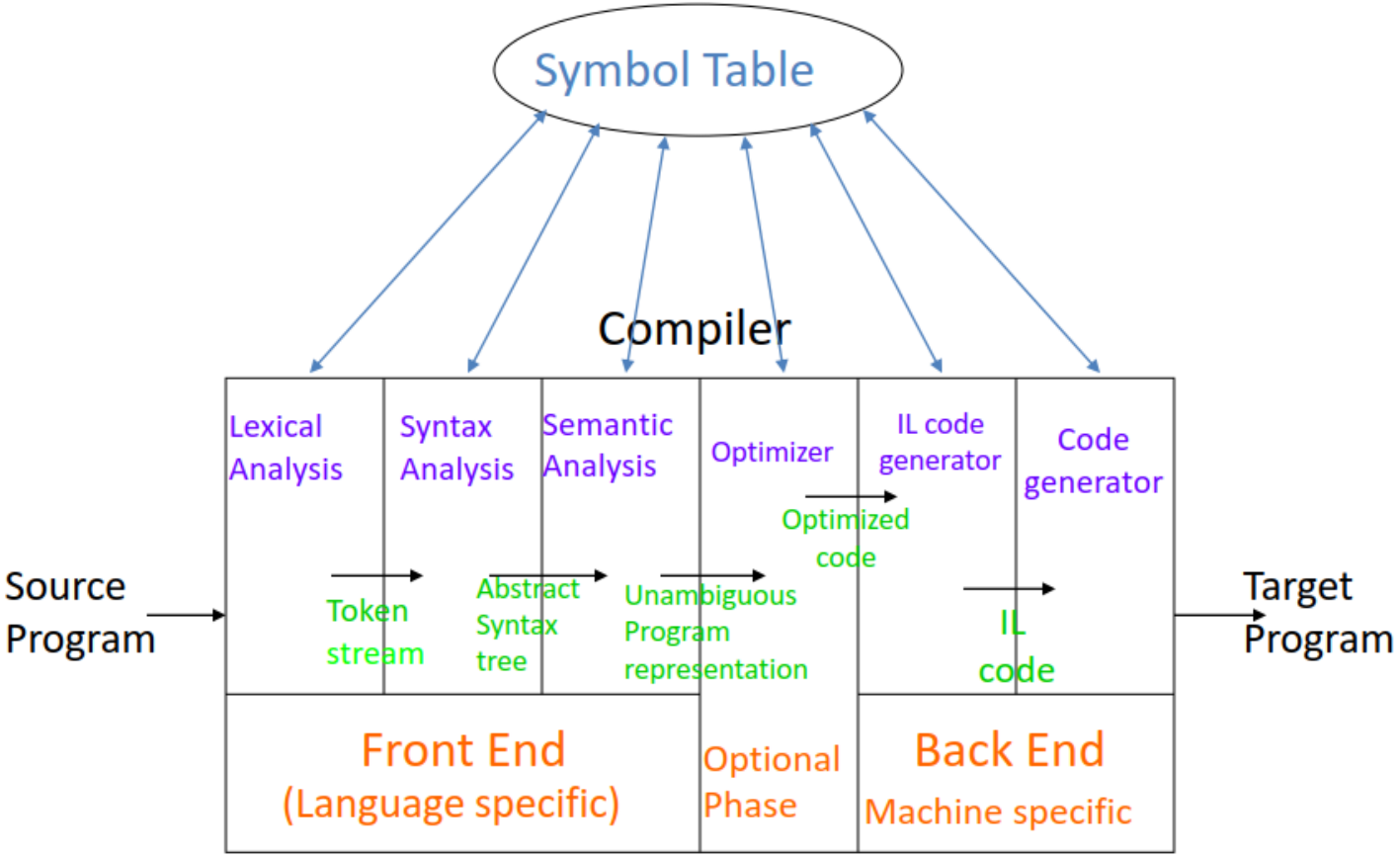
General Structure of a compiler



Something missing?

- Information required about the program variables during compilation
  - **Type of variable**: integer, float, array, function etc.
  - Amount of storage required
  - Address in the memory
  - Scope information
- Location to store this information
  - A **central repository** and every phase refers to the repository whenever information is required
  - Use a data structure called symbol table

Compiler Structure



Advantages of the model…

- Compiler is re-targetable
- **Source** and **machine** independent code optimization is possible.
- Optimization phase can be inserted after the front and back end phases have been developed and deployed

Finally...

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

- Scanner generators: Lex/flex
- Parser generators: Yacc/bison

Summary: the structure of a typical compiler was described.

Next lecture: Introduction to lexical analysis.

Reading: Sections 1.2, 1.3