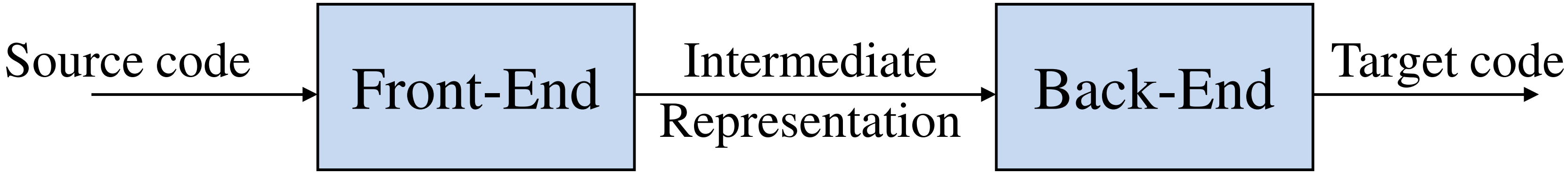
Lecture 2

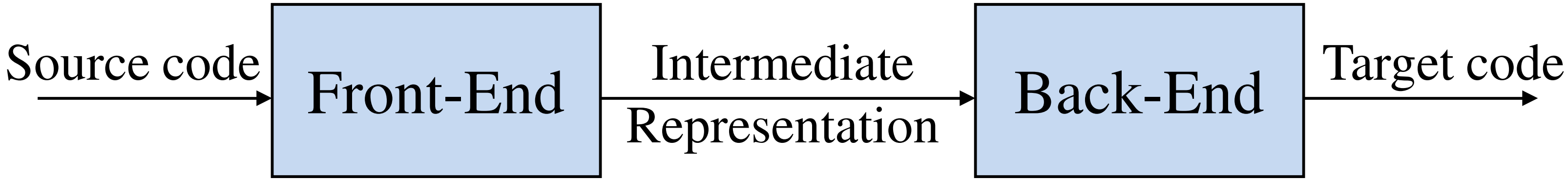
General Structure of a Compiler

Conceptual Structure



- **Front-end** performs the **analysis** of the source language:
  - Breaks up the source code into pieces and imposes a grammatical structure.
  - Using this, creates a generic Intermediate Representation (IR) of the source code.
  - Checks for syntax / semantics and provides informative messages back to user in case of errors.
  - Builds a symbol table to collect and store information about source program.

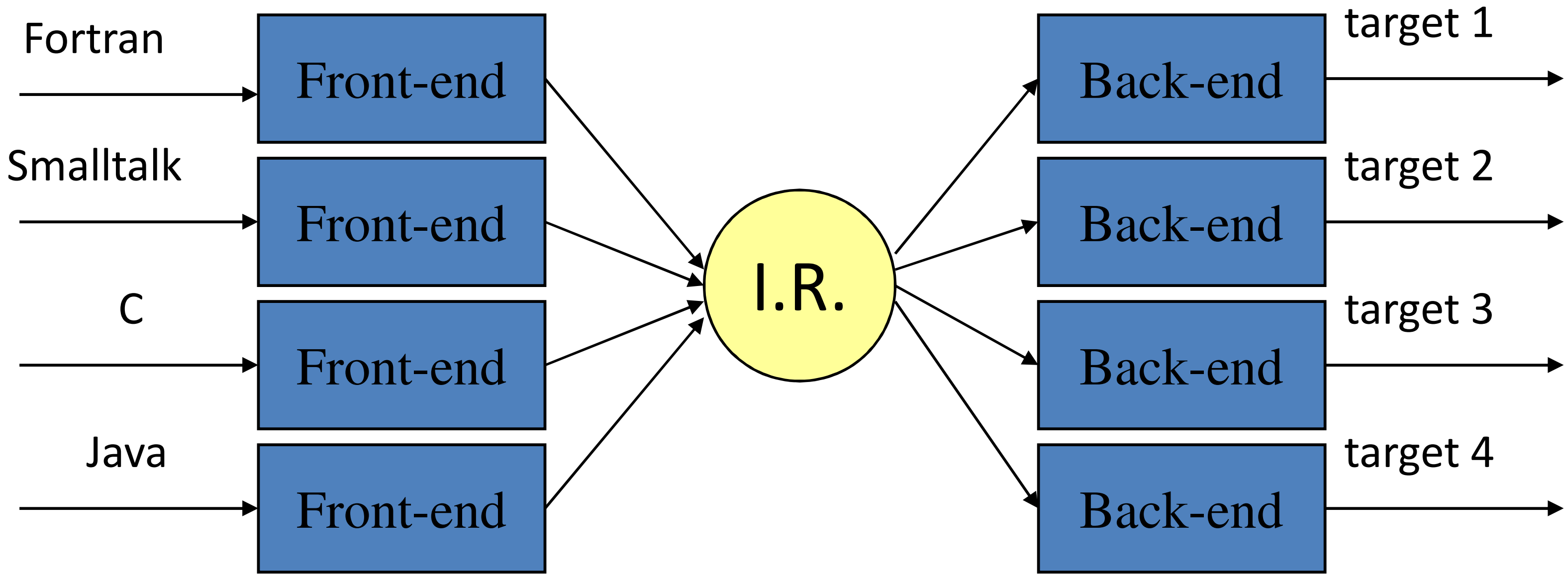
Conceptual Structure



- **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.

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!

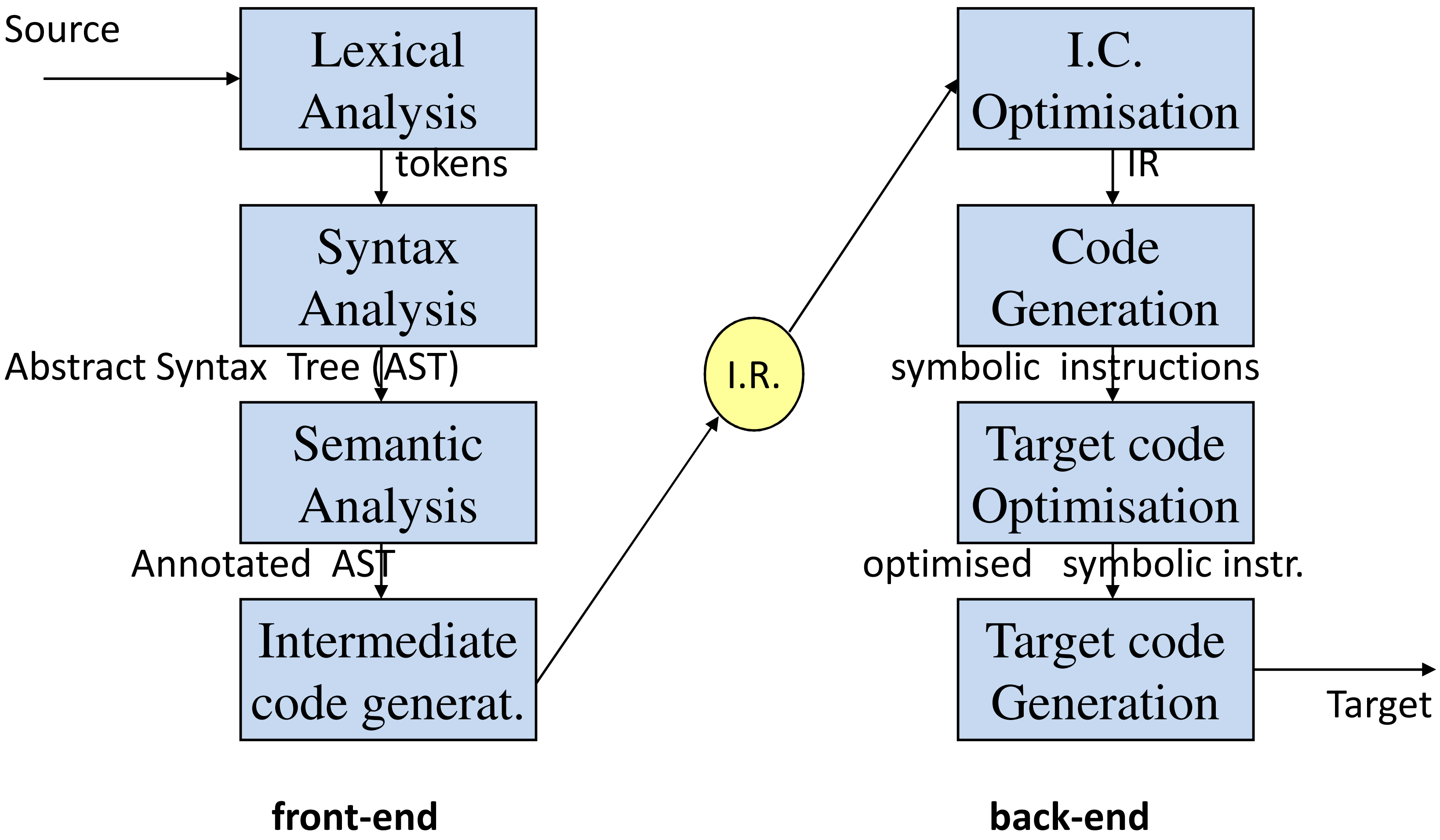


- All language specific knowledge must be encoded in the front-end

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

But: in practice, this strict separation is not trivial.

General Structure of a Compiler



Lexical Analysis (Scanning)

- Reads characters in the source program and groups them into meaningful sequences called *lexemes*.

- Produces as output a *token of the form <Token-class, Attribute> and* passes it to the next phase *syntax analysis*

- *Token-class: The symbol for this token to be used during syntax Analysis*

- *Attribute: Points to symbol table entry for this token*

  e.g.: The tokens for: *fin = ini + rate * 60* *are:*

  <id, 1> <=> <id, 2> <+> <id, 3> <*> <const, 60>

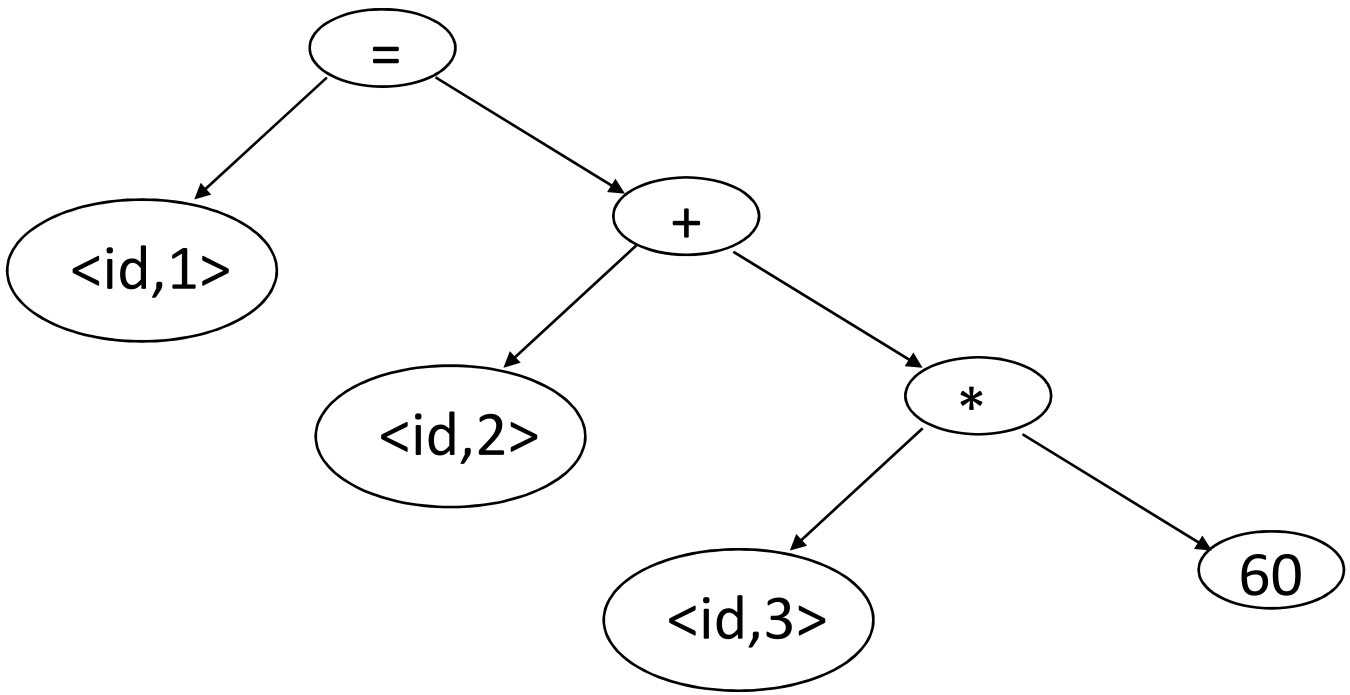
Symbol Table		
1	pos	…
2	ini	…
3	rate	…
…	…	…

Syntax 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 to flag syntax error in case of any mismatch.
- The IR is usually represented as *syntax trees*
  - Interior nodes represent *operation*
  - Leaves depict the *arguments of the operation*

Syntax Analysis (Parsing)

- Parse tree for: *fin* = *ini* + *rate* * *60*
- Tokens: <id, 1> <=> <id, 2> <+> <id, 3> <*> <const, 60>



Semantic Analysis (context handling)

- Collects context (semantic) information from syntax tree and symbol table and checks for semantic consistency with language definition.

- Annotates nodes of the tree with the results.

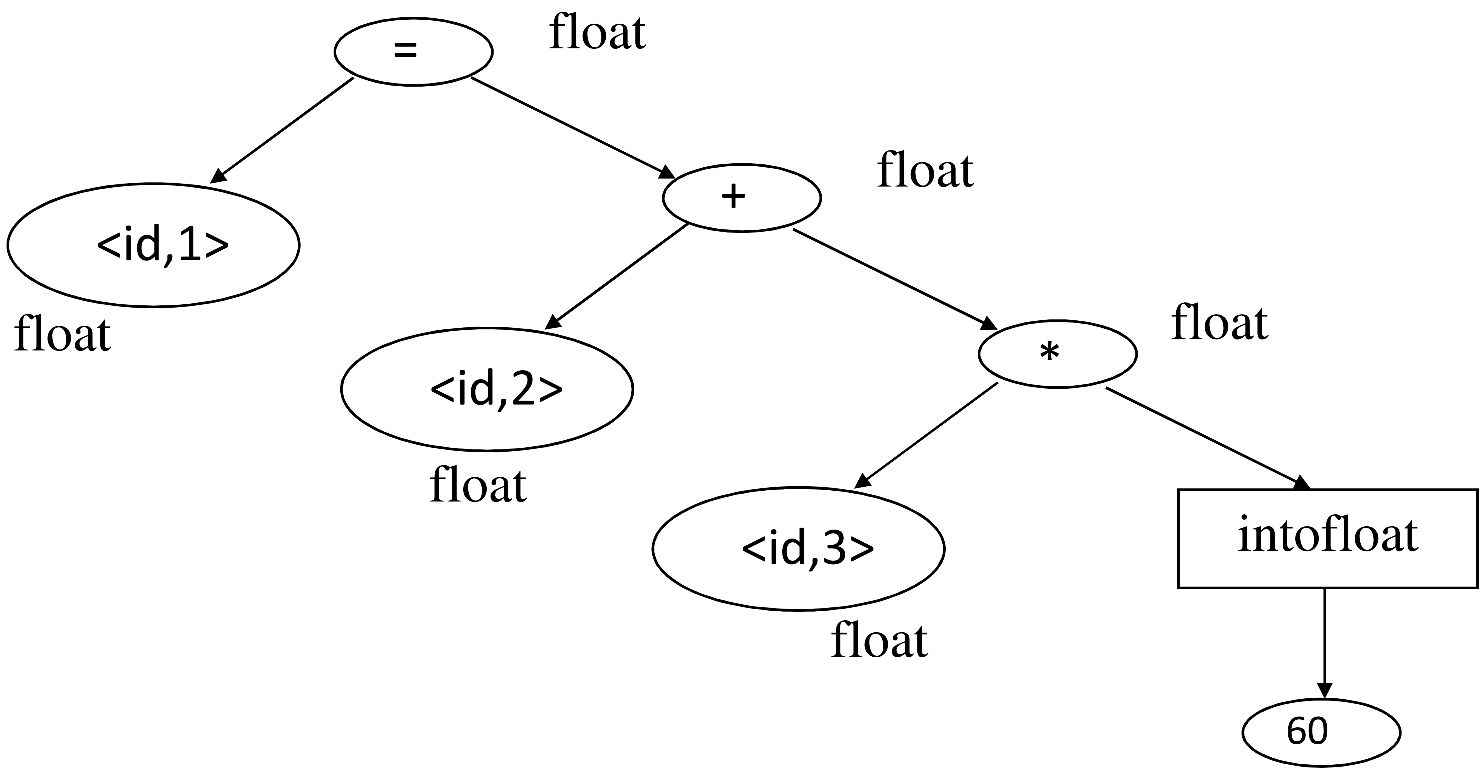
- Semantic errors:
  - Type mismatches, incompatible operands, function called with improper arguments, undeclared variable, etc.

  - e.g: int ary[10], x; x = ary * 20;

- *Type checkers in the semantic analyzer may also do automatic type* conversions if permitted by the language specification (*Coercions).*

Semantic Analysis (context handling)

- Annotated Syntax Tree (AST) for the expression fin = ini + rate * 60
- Tokens: <id, 1> <=> <id, 2> <+> <id, 3> <*> <const, 60>



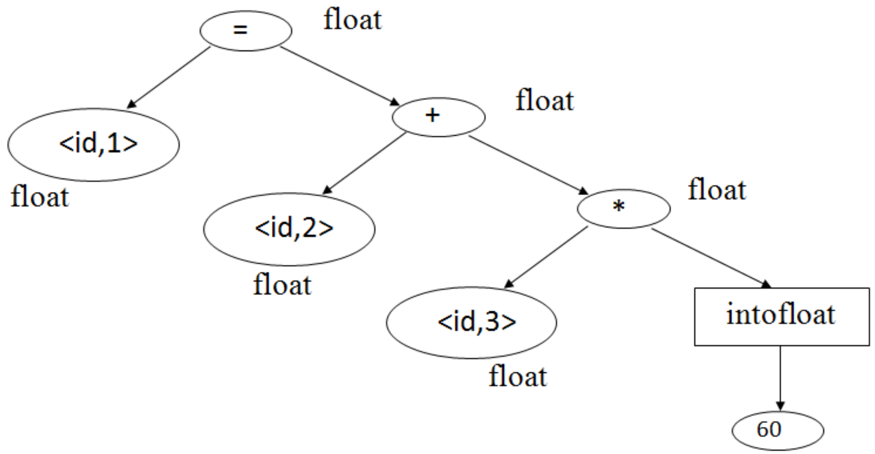
Intermediate code generation

- 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):

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```



Code Optimisation

- Generates streamlined code still in intermediate representation.

- A range of optimization techniques may be applied. For example,
  - Removing unused variables
  - Suppressing generation of unreachable code segments
  - Constant Folding
  - Common Sub-expression Elimination
  - Loop optimization (Removing unmodified statements in a loop)... etc.

- Example:

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

→

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Generation

- Map 3-address code into assembly code of the target architecture
  - *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).
  - *Instruction scheduling: take advantage of multiple functional units.*
- Example — A possible translation using two registers:

```
t1 = id3 * 60.0
id1 = id2 + t1
```

→

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

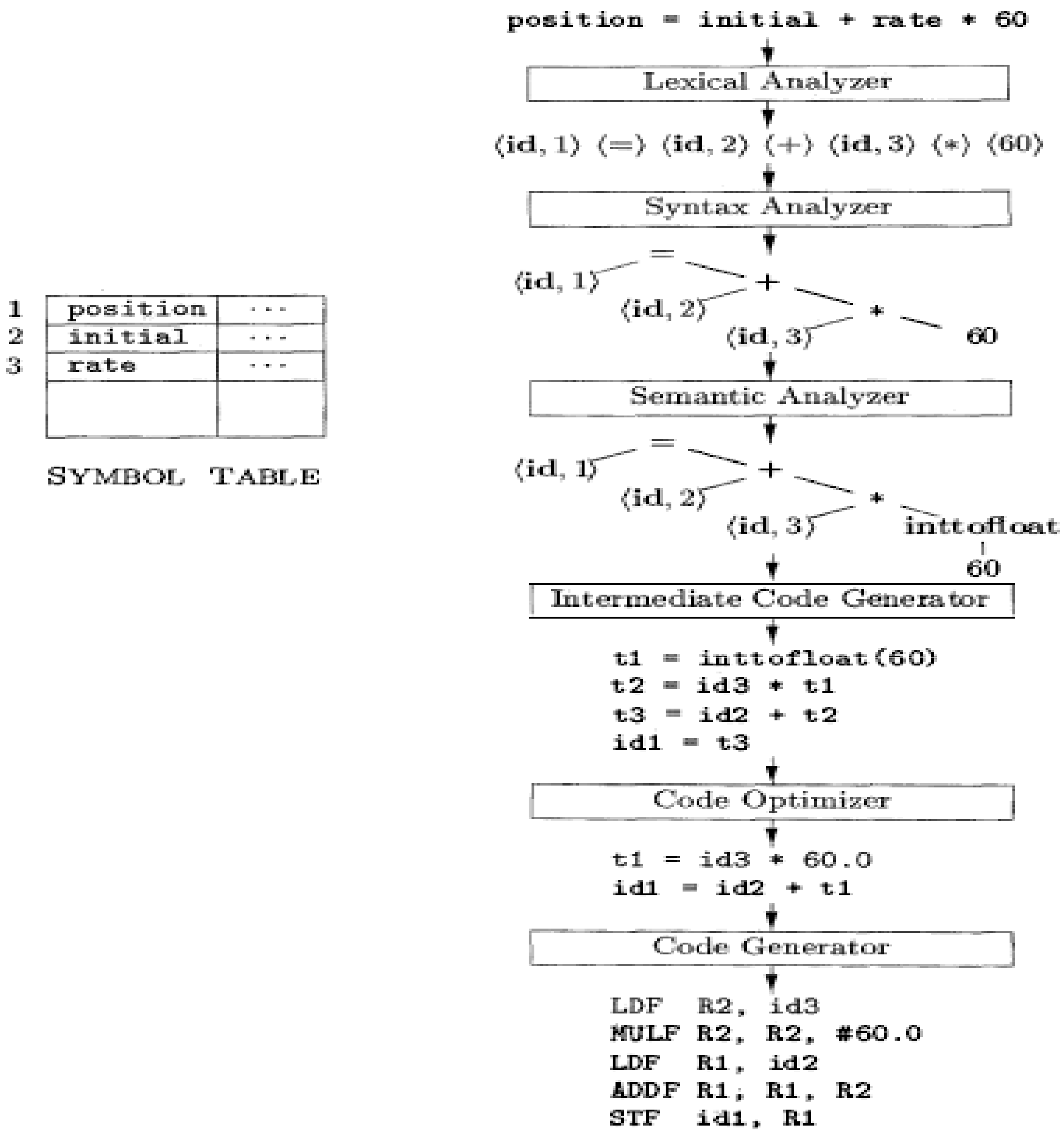


Figure 1.7: Translation of an assignment statement

Thanks