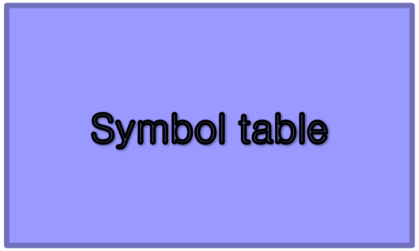
The Structure of a Compiler

Dragon ch. 1.2

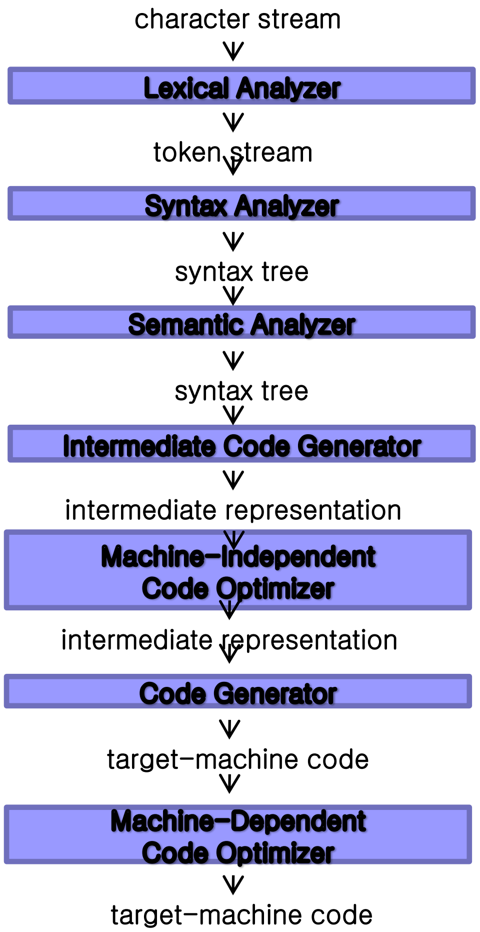
Analysis and Synthesis

- Analysis (front-end)
  - Breaks up the source program into pieces and builds a grammatical structure
  - Creates an intermediate representation
  - Collect information on the program and stores it in a symbol table
- Synthesis (back-end)
  - Constructs target program from intermediate representation and symbol table

Phases of a Compiler



Symbol table



character stream

Lexical Analyzer

token stream

Syntax Analyzer

syntax tree

Semantic Analyzer

syntax tree

Intermediate Code Generator

intermediate representation

Machine-Independent Code Optimizer

intermediate representation

Code Generator

target-machine code

Machine-Dependent Code Optimizer

target-machine code

Let's see how to compile a statement

`position = initial + rate * 60`

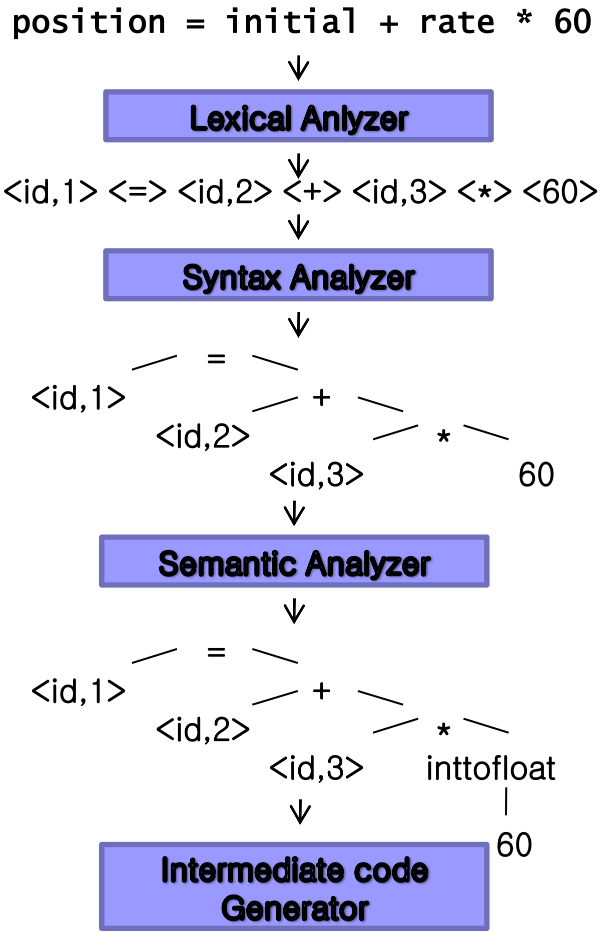
Lexical Analysis

- Reads character stream and groups characters into a meaningful sequence called a lexeme
  - Returns a token, <token-name, attribute-value>
  - Lexemes from **`position = initial + rate * 60`** are
    - **`position`**: <id, entry1>
    - **`=`**: <=>
    - **`initial`**: <id, entry2>
    - **`+`**: <+>
    - **`rate`**: <id, entry3>
    - **`*`**: <*>
    - **`60`**: <60> or <int, entry4>
- Regular expressions and finite automata

Front-end Translation

1	position	…
2	initial	…
3	rate	…

SYMBOL TABLE



`position = initial + rate * 60`

Lexical Anlyzer

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

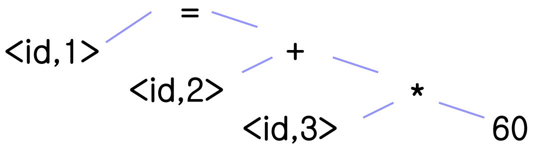
Syntax Analyzer

Semantic Analyzer

Intermediate code Generator

Syntax Analysis

- Builds a parse tree from tokens
  - □ Represents grammatical structure
  - □ Parse tree for `position = initial + rate * 60` are



- Context-free grammar and parsing theory

Semantic Analysis

- Check semantic correctness with language definition using parse tree and symbol table
  - Type checking (type coercion)
  - Check # of actual and formal arguments
  - Check if variables are used after definition
  - Type coercion for `position = initial + rate * 60`
- Type theory

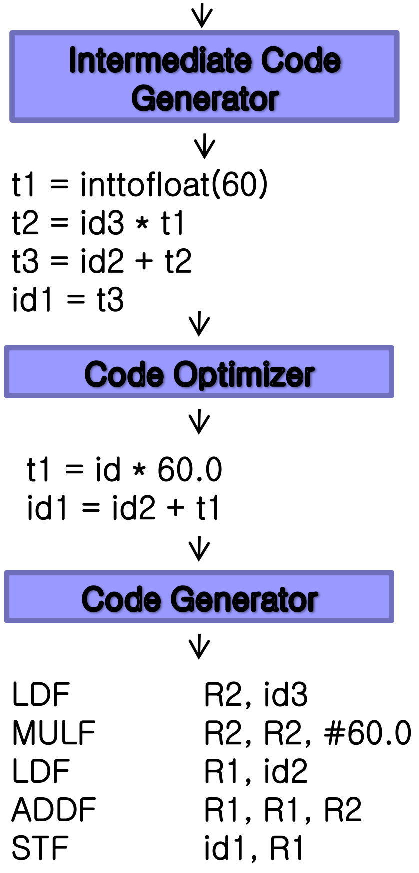
Intermediate Code Generation

- Generate intermediate form which is easy to generate and translate to machine code
  - □ Parse tree is a form of intermediate code
  - □ Other popular intermediate forms
    - Three-address code
    - Stack-machine-based code (U-code, P-code, bytecode)
  - □ For our `position = initial + rate * 60`
    - t1 = inttofloat(60)
    - t2 = id3 * t1
    - t3 = id2 + t2
    - id1 = t3

Back-end Translation

1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE



Intermediate Code Generator

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

Code Optimizer

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

Code Generator

```
LDF          R2, id3
MULF         R2, R2, #60.0
LDF          R1, id2
ADDF         R1, R1, R2
STF          id1, R1
```

Figure 1.7 (2) : Translation of an assignment statement

Intermediate Code Optimization

- Produce better intermediate code
  - Reduce temporaries or compile-time evaluation
  - Ideal for parallelization
  - For our `position = initial + rate * 60`
    - t1 = id3 * 60.0
    - id1 = id2 + t1

Machine Code Generation

- Generate machine code from intermediate form
  - Assign memory locations and registers for variables
    - Local variables (stack) or global variables (global data area)
  - Choose machine instructions
  - For our `position = initial + rate * 60`
    - Intermediate form: t1 = id3 * 60.0, id1 = id2 + t1
    - Machine code
    - `LDF   R2, @id3`
    - `MULF  R2, R2, #60.0`
    - `LDF   R1, @id2`
    - `ADDF  R1, R1, R2`
    - `STF   @id1, R1`

Symbol Table Management

- Collect information for names (variable, function)
  - Storage
  - Type
  - Scope
  - Number and type of arguments, return type
- Use symbol table at various phases