Phases of Compiler

To be discussed

- First generation – Machine Languages
- Second generation – Assembly Language
- Third generation - High level language (c, c++)
- Fourth generation – SQL, Postscript
- Fifth generation - Prolog

To be discussed

- Type Checking
- Bound Checking
- Memory Management
- Static and Dynamic Distinction
- Environment and States (N -> L: E, L -> V: S)
- Static Vs Dynamic Binding
- Names, Identifiers and Variables
- Procedure, functions and methods
- Actual Parameter and Formal Parameter
- Aliasing
- Declaration vs definition

Introduction

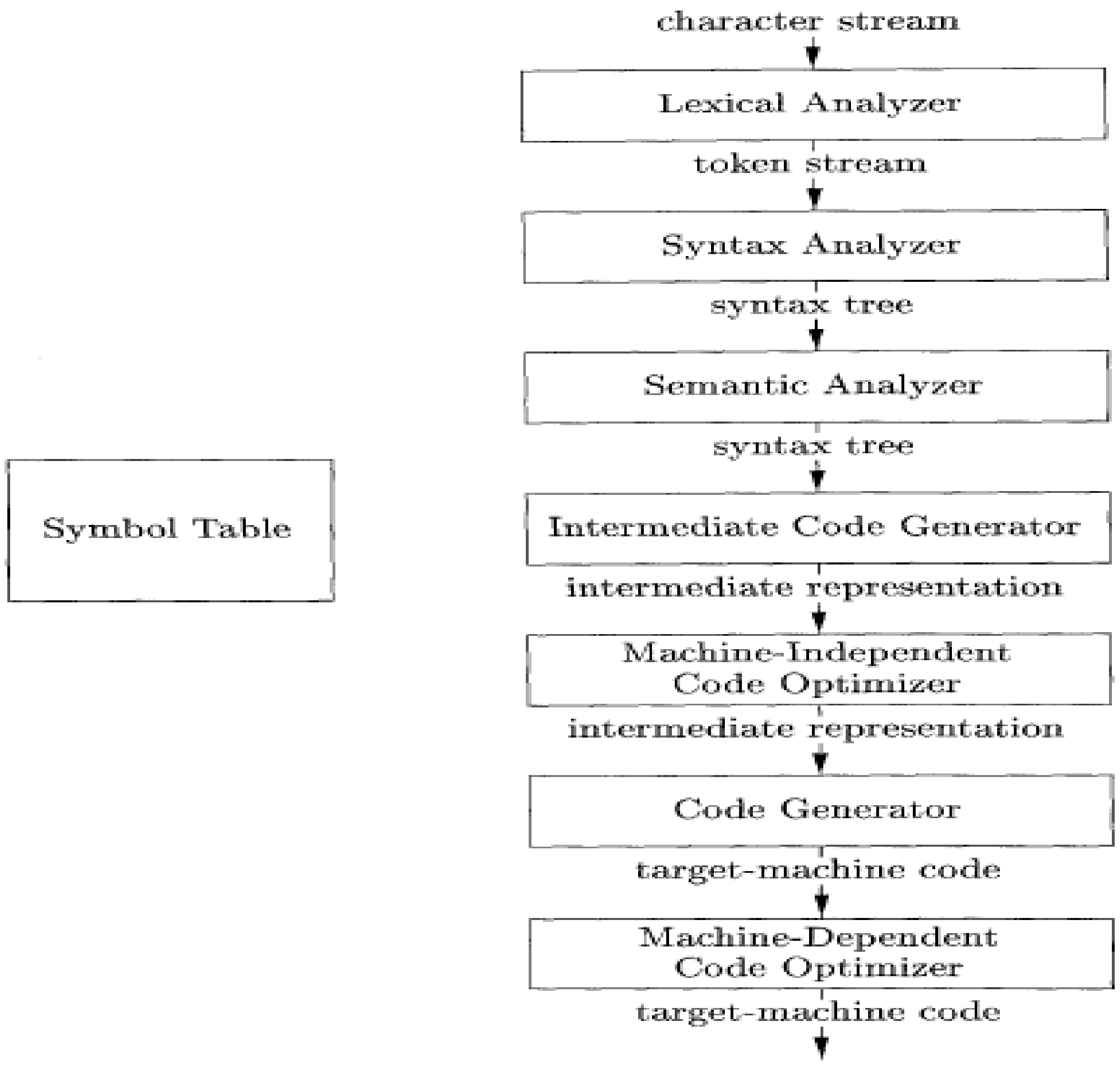
- Two Phases
  - Analysis Phase
  - Synthesis Phase

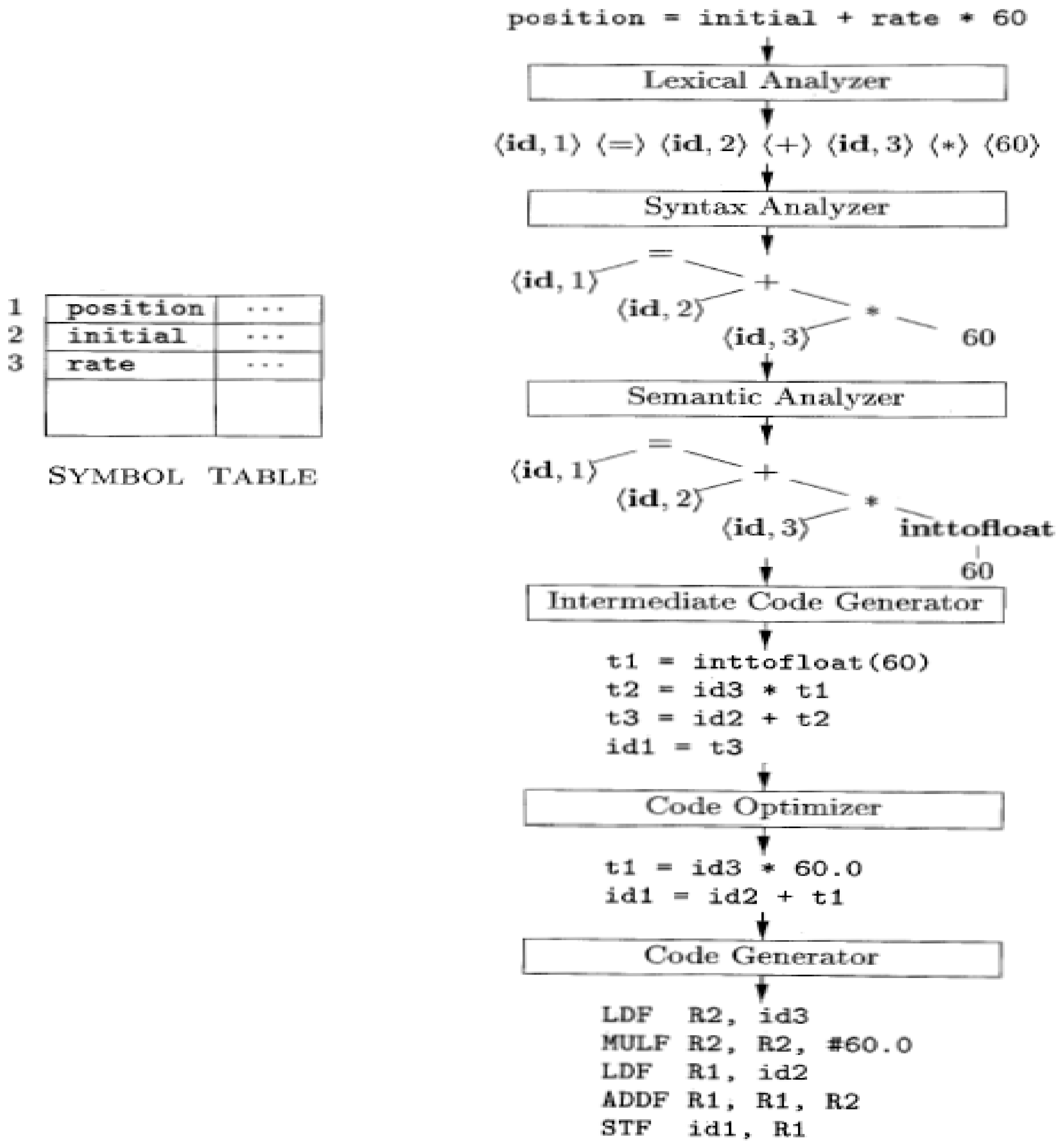
Analysis Phase

- The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them.
- It then uses this structure to create an intermediate representation of the source program.
- If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action.
- The analysis part also collects information about the source program and stores it in a data structure called a **symbol table**, which is passed along with the intermediate representation to the synthesis part.

Synthesis Phase

- The *synthesis part constructs the desired target program from the intermediate* representation and the information in the 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
Symbol Table



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

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

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

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

Lexical Analysis

- The first phase of a compiler is called lexical analysis or scanning.

- The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.

- For each lexeme, the lexical analyzer produces as output a token of the form that it passes on to the subsequent phase, syntax analysis.

LA

Token = (token-name, attribute-value)

- Token name used during syntax analysis second component attribute-value points to an entry in the symbol table for this token.
- In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token.
- Information from the symbol-table entry Is needed for semantic analysis and code generation.

Example

- x = a + c * 30
- position is a lexeme that would be mapped into a token **(id, 1), where id** is an abstract symbol standing for identifier and 1 points to the symbol table entry for position.
- The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

Contd.,

- The assignment symbol = is a lexeme that is mapped into the token (=).
- Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as **assign for** the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.

Contd.,

- initial is a lexeme that is mapped into the token **(id, 2), where 2 points** to the symbol-table entry for initial.
- + is a lexeme that is mapped into the token (+).
- rate is a lexeme that is mapped into the token **(id, 3), where 3 points to** the symbol-table entry for rate .
- * is a lexeme that is mapped into the token (*).
- 60 is a lexeme that is mapped into the token (60).
- Blanks separating the lexemes would be discarded by the lexical analyzer.

Contd.,

(id , l) (=) (id, 2) (+) (id, 3) (*) (60)

- In this representation, the token names =, +, and * are abstract symbols for the assignment, addition, and multiplication operators, respectively.

Syntax Analysis

- The second phase of the compiler is syntax analysis or parsing.

- The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.

- A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.

Contd.

- position = initial + **rate * 60**
- The tree has an interior node labeled * with **(id, 3) as** its left child and the integer **60 as its right child. The node (id, 3) represents** the identifier **rate. The node labeled * makes it explicit that we must first** multiply the value of **rate by 60.**
- This ordering of operations is consistent with the usual conventions of arithmetic which tell us that multiplication has higher precedence than addition, and hence that the multiplication is to be performed before the addition.

Semantic Analysis

- The *semantic analyzer uses the syntax tree and the information in the symbol* table to check the source program for **semantic consistency with the language definition.**

- It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

- An important part of semantic analysis is ***type checking****, where the compiler* checks that **each operator has matching operands**. For example, many programming language definitions require an **array index to be an integer**; the compiler must report an error if a floating-point number is used to index an array.

Promotion

- The language specification may permit some type conversions called ***coercions****.*

- For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers.

- If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

Intermediate Code Generation

- In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms.

- Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

- Reason for converting to ICG: This intermediate representation should have two important properties: **it should be easy to produce and it should be easy to translate into the target machine.**

Three address code

- t1 = i n t t o f l o a t (6 0)
- t2 =id3 * t1
- t3 = id2 + t2
- id1 = t3

- Three operands per instruction and each operand can act like a register.

Three-address instructions

- First, each three-address assignment instruction has at most **one operator on the right side**.
- Thus, these instructions fix the order in which operations are to be done; the multiplication precedes the addition in the source program.
- Second, the **compiler must generate a temporary name** to hold the value computed by a three-address instruction.
- Third, some "three-address instructions" like the first and last in the sequence, above, have **fewer than three operands.**

Code Optimization

- The machine-independent code-optimization phase attempts to improve the intermediate code so that **better target code will result**.

- Usually **better means faster**, but other objectives may be desired, such as shorter code, or **target code that consumes less power.**

- For example, a straightforward algorithm generates the intermediate code, using an instruction for each operator in the tree representation that comes from the semantic analyzer.

Code Optimization

- The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the **inttofloat operation can be eliminated by replacing** the integer 60 by the floating-point number **60.0**.

- Moreover, **t3 is used only** once to transmit its value to id1 so the optimizer can transform into the shorter sequence

Optimized Code

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

- In those that do the most, the so-called "optimizing compilers," a significant amount of time is spent on this phase.

- There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.

Code Generation

- If the **target language is machine code**, registers or memory locations are selected for each of the variables used by the program.
- Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.
- A **crucial aspect of code generation** is the judicious assignment of registers to hold variables.

Code Generation

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

F is floating and first variable is the destination

Symbol-Table Management

- An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name.

- **These attributes may provide information about the storage allocated for a name, its type, its scope (**where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.

Symbol Table Management

- The symbol table is a data structure containing a record for **each variable name, with fields for the attributes of the name.**

- The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

Grouping of Phases into Passes

- The discussion of phases deals with the logical organization of a compiler.
- In an implementation, activities from **several phases may be grouped together into a *pass that reads an input file and writes an output file.***
- *For example,* the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass.
- Code optimization might be an optional pass. Then there could be a back-end pass consisting of code generation for a particular target machine.

Lexical Output

- (keyword "print")
- (delim "(")
- (int 3)
- (punct "+")
- (id "x")
- (punct "*")
- (int 2)
- (delim ")")

```
print (3 + x *2 ) # comment
```

Thank You