Outline

Chapter 1

- PL**
  - 编译与解释的区别和联系
- History
- Contents*
  - 编译各阶段的内容

Chapter 2

- Example
  - Motivation, Example
- Formal Lang.*
  - Natural Lang., Definitions, Operations
- Description
  - Grammar

Chap. 1 Introduction

Chaokun Wang

IISE@Tsinghua

1.4 Programming Languages

- imperative programming languages
  - Fortran, C
- objective-oriented programming languages
  - SmallTalk, Objective-C, C++, Java
- functional programming languages
  - Lisp, Scheme
- logical programming languages
  - Prolog
- other languages
  - SQL

How are languages implemented?

- Various strategies depend on how much pre-processing is done before a program can be run, and how CPU-specific the program is.
- Interpreters run a program “as is” with little or no pre-processing, but no changes need to be made to run on a different platform.
- Compilers take time to do extensive pre-processing, but will run a program generally 2- to 20- times faster.

Language implementations cont'd

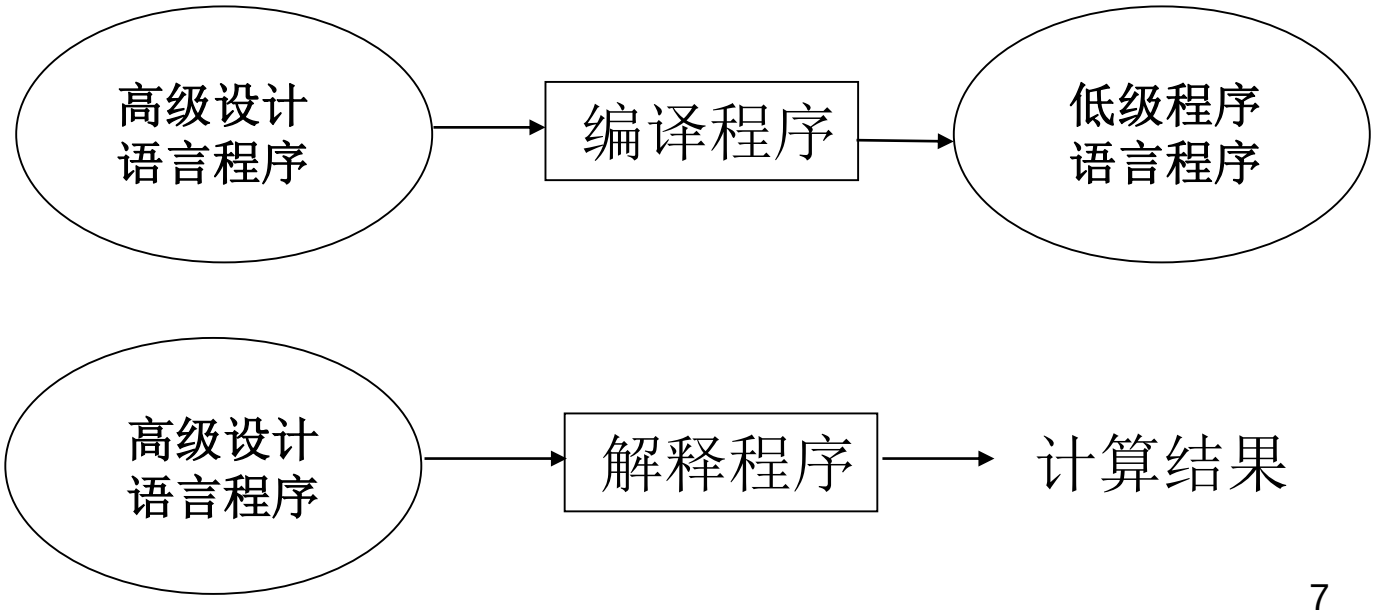
- Some newer languages use a combination of compiler and interpreter to get many of the benefits of each.
- Examples are Java and Microsoft's .NET, which compile into a virtual assembly language (while being optimized), which can then be interpreted on any computer.
- Other languages (such as Basic or Lisp) have both compilers and interpreters written for them.
- Recently, "Just-in-Time" compilers are becoming more common - compile code only when its used!

Buddies of Compilation

- 反编译、汇编、反汇编、翻译、…
- 高级语言之间的转换工具 由于计算机硬件的不断更新换代，更新更好的程序设计语言的推出为提高计算机的使用效率提供了良好条件，然而一些已有的非常成熟的软件如何在新机器新语言情况下使用呢?
- 为了减少重新编制程序所耗费的人力和时间，就要解决如何把一种高级语言转换成另一种高级语言，乃至汇编语言转换成高级语言的问题。这种转换工作要对被转换的语言进行词法和语法分析，只不过生成的目标语言是另一种高级语言而已。这与实现一个完整的编译程序相比工作量要少些。
  - 比如：
  - **Scheme to C 的翻译器chicken**
  - **C、PASCAL、FORTRAN到Ada的翻译器**
  - **COBOL 到Java 的翻译器**

Compilation and Interpretation

- 程序设计语言的实现主要的途径有两个：通过编译程序和解释程序
- 有些语言基本通过解释程序：**Python**
- 有些环境同时提供编译程序和解释系统 **Lisp**



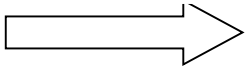
Compiler and Interpreter

Source program：… …

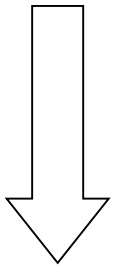
```
b := 2      ;
a :=  b+2  ;
write   a   ;
… …
```

Compiler

生成代码



```
Int 2
St b
Ld b
add 2
St a
```



Interpreter

直接将4的值输出（显示）

（直接对源程序中的语句进行分析，执行其隐含的操作。）

编译程序和解释程序

编译程序是一个语言处理程序,

把一个高级语言程序翻译成某个机器的汇编或二进制代码程序, 这个二进制代码程序在机器上运行以生成结果.

通过编译程序使得我们可以先准备好一个在该机器上运行的程序,然后这个程序便会以机器的速度运行.

但是在不把整个程序全部都翻译结束之后,这个程序是不能开始运行,也不能产生任何结果的.

编译和运行是两个独立分开的阶段.

但在一个交互环境中,不需要将这两个阶段分隔开,编译就不如解释的方法更方便.

解释程序不需要在运行前先把源程序翻译成目标代码,也可以让我们实现在某台机器上运行程序并生成结果.

解释程序

- 是这样一个程序,它接受某个语言的程序并立即运行这个源程序.
- 它的工作模式是一条条地获取、分析并执行源程序语句。一旦第一条语句分析结束，源程序便开始运行并且生成结果。它特别适合程序员交互方式的工作情况，即希望在获取下一条语句之前了解每条语句的执行结果,允许执行时修改程序.
- 著名的解释程序有
  - **Basic语言解释程序**
  - **Lisp语言解释程序**
  - **UNIX命令语言解释程序(shell)**
  - 数据库查询语言**SQL** 解释程序
  - **bytecode解释程序**

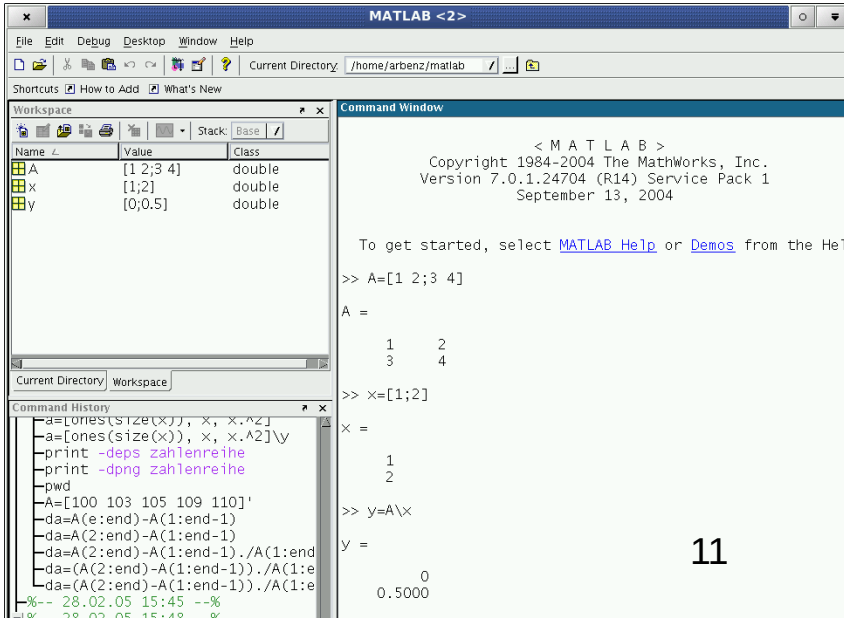
高级语言解释系统(interpreter)

- 功能　让计算机执行高级语言（basic,lisp,prolog)
- 与编译程序的不同　1）不生成目标代码
  2）能支持交互环境
  （同增量式编译系统）

源 程 序 ⟹

初始数据 ⟹

解释程序 ⟹ 计算结果



1.5 History of compiler development

1953 IBM develops the 701 EDPM (Electronic Data Processing Machine), the first general purpose computer, built as a "defense calculator" in the Korean War

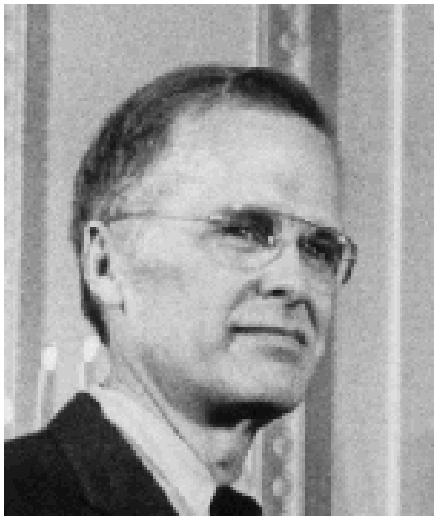
No high-level languages were available, so all programming was done in assembly

History of compilers (cont'd)

As expensive as these early computers were, most of the money companies spent was for software development, due to the complexities of assembly.

In 1953, John Backus came up with the idea of "speed coding", and developed the first interpreter. Unfortunately, this was 10-20 times slower than programs written in assembly.

He was sure he could do better.



John Backus

History of compilers (cont'd)

In 1954, Backus and his team released a research paper titled "Preliminary Report, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN."

The initial release of FORTRAN I was in 1956, totaling 25,000 lines of assembly code. Compiled programs ran almost as fast as handwritten assembly!

Projects that had taken two weeks to write now took only 2 hours. By 1958 more than half of all software was written in FORTRAN.

How to make a compiler

- 自然语言翻译...

The A-S Model

The Analysis-Synthesis Model of Compilation

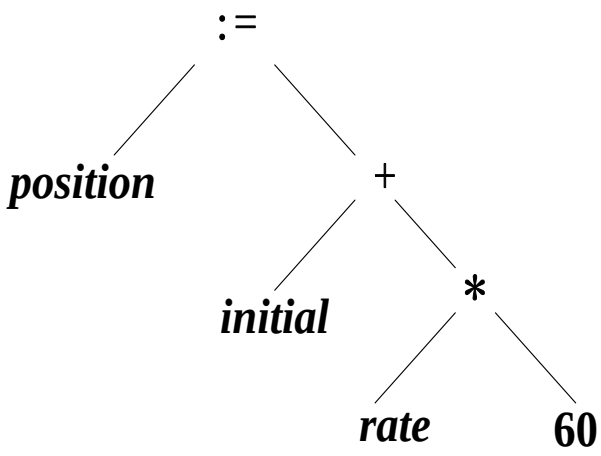
- analysis
  - breaks up the source program into constituent pieces
  - creates an intermediate representation of the source program
- synthesis
  - constructs the desired target program from the intermediate representation.
  - requires the most specialized techniques

During analysis

- Data → Symbol Table

Syntax Tree

- **Operations → A hierarchical structure**
  - the operations implied by the source program are determined.
  - the operations are recorded
- **Syntax tree**
  - a special kind of tree
  - stores the operations implied by the source program
  - each node represents an operation
  - the children of a node represent the arguments of the operation.
- **A syntax tree for the assignment statement position:= initial + rate *60**



Symbol-Table Management

- An essential function of a compiler
  - is to record the identifiers used in the source program
  - collect information about various attributes of each identifier. These attributes may provide information about
    - the storage allocated for an identifier,
    - its type,
    - its scope (作用域, where in the program it is valid),
    - in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (e.g., by reference), and the type returned, if any.

Symbol-Table

- A *symbol table* (符号表) is a data structure
  - containing a record for each identifier,
  - with fields for the attributes of the identifier.
- The data structure allows us
  - to find the record for each identifier quickly and
  - to store or retrieve data from that record quickly.

- When an identifier in the source program is detected by the lexical analyzer,
  - the identifier is entered into the symbol table.
- However, the attributes of an identifier cannot normally be determined during lexical analysis.
- For example, in a Pascal declaration like

  var position, initial, rate : real ;

  the type real is not known when position, initial, and rate are seen by the lexical analyzer.

Error Detection and Reporting

- Each phase can encounter errors.
- However, after detecting an error,
  - a phase must somehow deal with that error,
  - so that compilation can proceed,
  - allowing further errors in the source program to be detected.
  - A compiler that stops when it finds the first error is not as helpful as it could be.
- The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler.
- The lexical phase can detect errors where the characters remaining in the input do not form any token of the language.

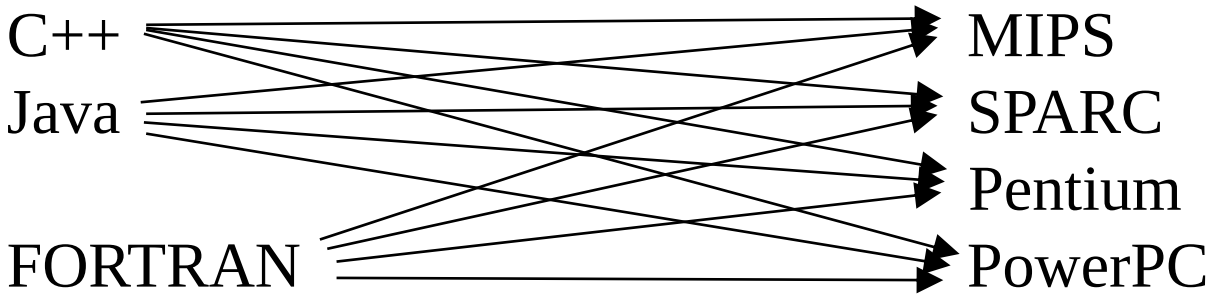
- Errors where the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase.
- During semantic analysis the compiler
  - tries to detect constructs that have the right syntactic structure but no meaning to be operation involved,
  - e.g., if we try to add two identifiers, one of which is the name of an array, and the other the name of a procedure.

Challenges

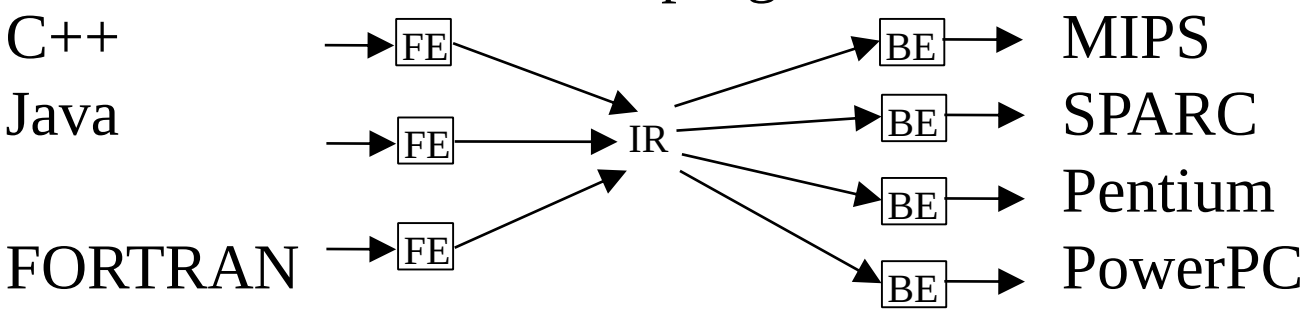
- **Many variations:**
  - **many programming languages (eg, FORTRAN, C++, Java)**
  - **many programming paradigms (eg, object-oriented, functional, logic)**
  - **many computer architectures (eg, MIPS, SPARC, Intel, alpha)**
  - **many operating systems (eg, Linux, Solaris, Windows)**

Addressing Portability

- **Suppose you want to write compilers from `m` source languages to `n` computer platforms. A naïve solution requires `n*m` programs:**

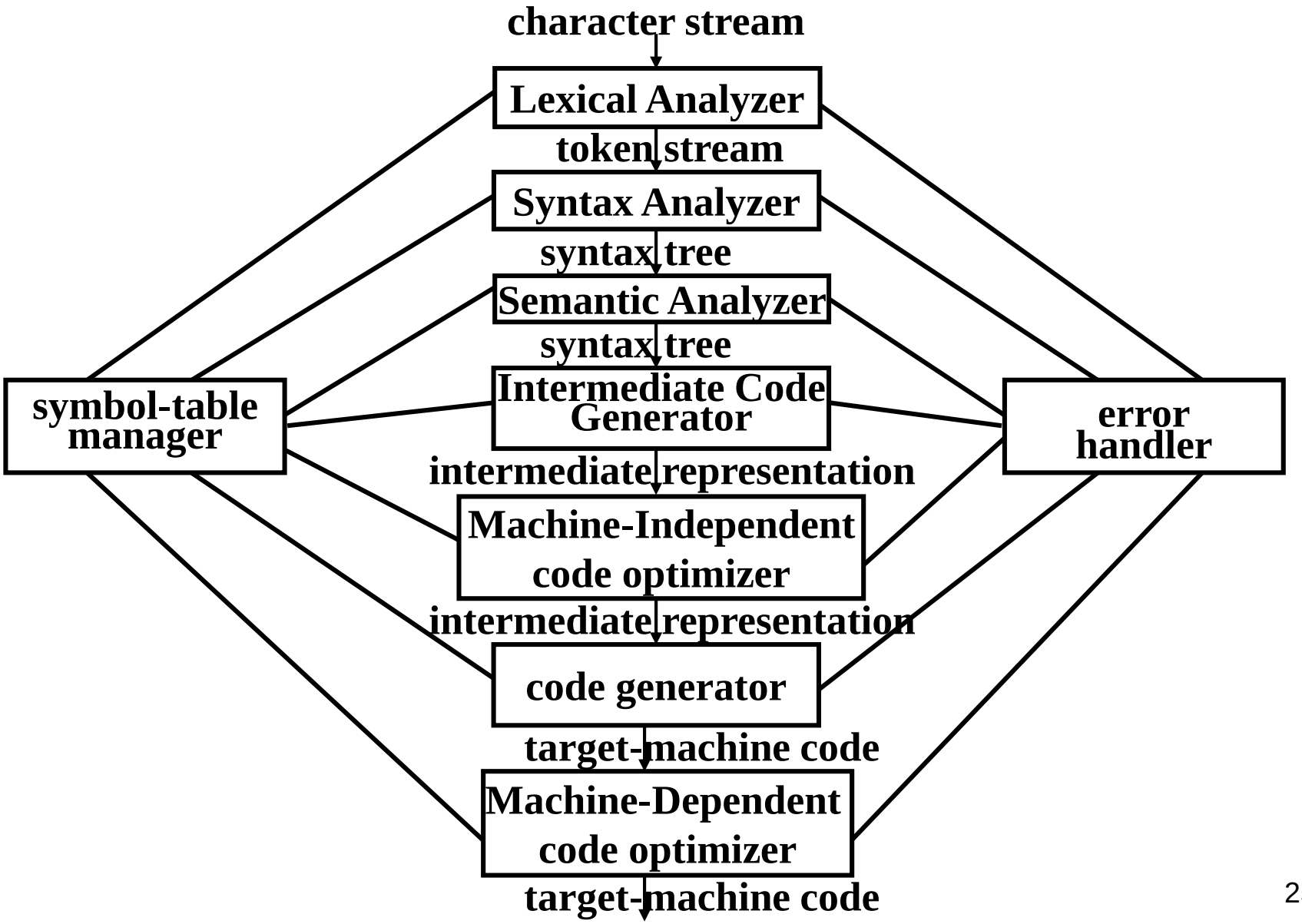


- but we can do it with `n+m` programs:



  - IR: Intermediate Representation
  - FE: Front-End
  - BE: Back-End

The Phases of a Compiler



Front and Back Ends

- Often, the phases are collected into a front end and a back end.
- The front end consists of those phases or parts of phases, that depend primarily on the source language and are largely **independent** of the target machine.
- These normally include
  - lexical and syntactic analysis,
  - the creation of the symbol table,
  - semantic analysis, and
  - the generation of intermediate code.
  - the error handling that goes along with each of these phases.
- A certain amount of code optimization can be done by the front end as well.

- The back end includes
  - those portions of the compiler that **depend** on the target machine,
  - and generally, these portions do not depend on the source language, just the intermediate languages.
  - aspects of the code optimization phase
  - code generation,
  - the necessary error handling and symbol-table operations.
- It has become fairly routine to take the front end of a compiler and redo its associated back end to produce a compiler for the same source language on a different machine.
- If the back end is designed carefully, it may not even be necessary to redesign too much of the back end.

1.6 Contents of this Part

- 编译程序概述
- 编译程序是现代计算机系统的基本组成部分之一
- 编译程序一般由
  - 词法分析程序
  - 语法分析程序
  - 语义分析程序
  - 中间代码生成程序
  - 目标代码生成程序
  - 代码优化程序
  - 符号表管理程序
  - 错误处理程序

  等成分构成。

After 44100593(2) you should

- 本部分重点讲述
  - 编译器的设计原理和常用实现技术
- 通过这部分课程学习和项目实践
  - **Theoretical framework**
    - 清楚地理解一个编译器是如何工作的
    - 几个阶段
  - **Compiler methodology**
    - 遇到任何一种程序设计语言，知道如何实现这个语言的多数机制
    - 基本语句
  - **Key algorithms**
    - 掌握编译原理中的关键算法
  - **Hands-on experience.**
    - 具有一定的使用编译构造工具开发编译器的经验
    - **ANTLR, Lex/Yacc**
  - **build a complete compiler…**
    - 能够将所学技术和算法灵活应用于类似的软件的设计和实现中
    - **SQL**

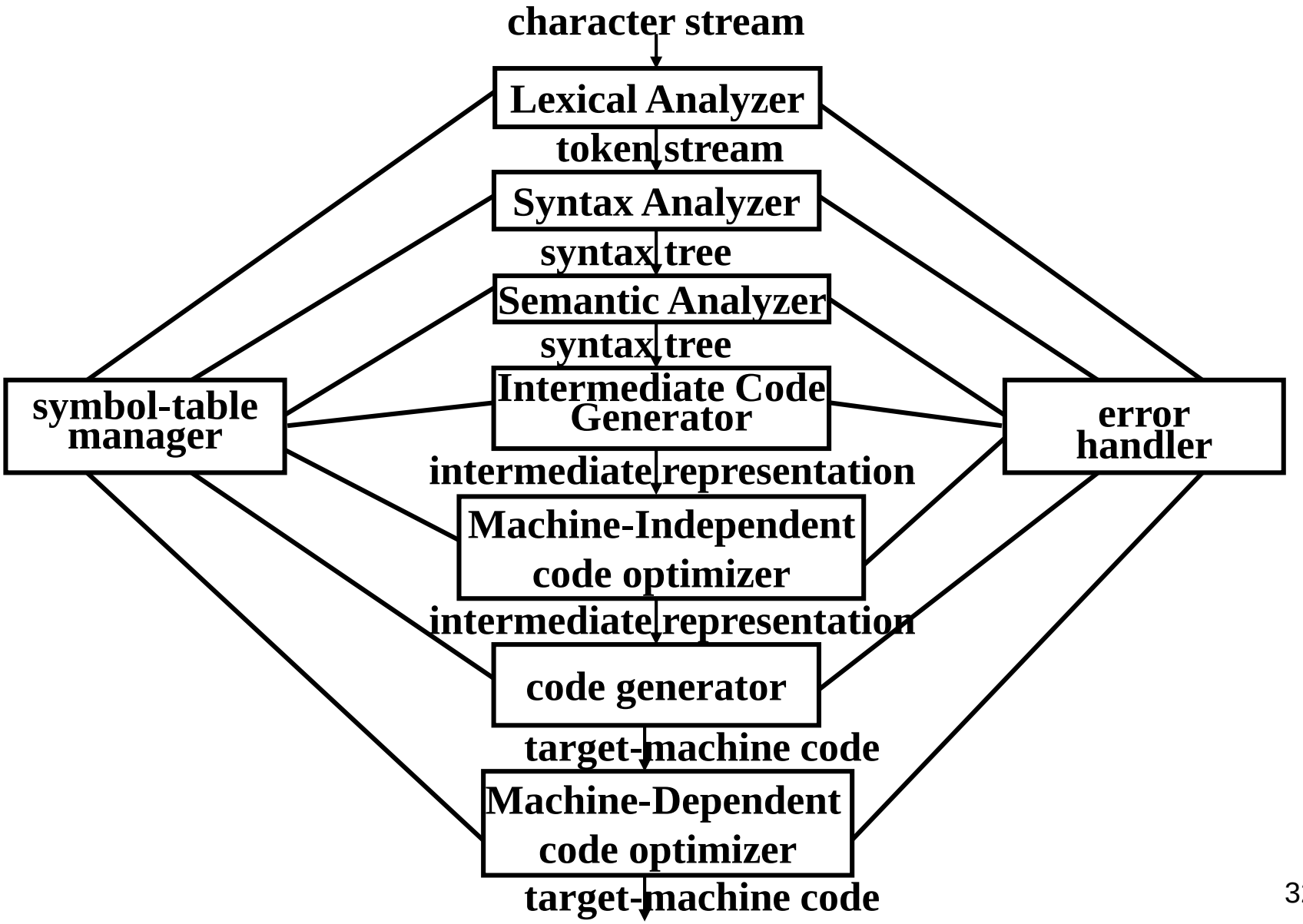
Chap. 2　编译示例及形式语言简顾

王朝坤
chaokun@tsinghua.edu.cn

Outline

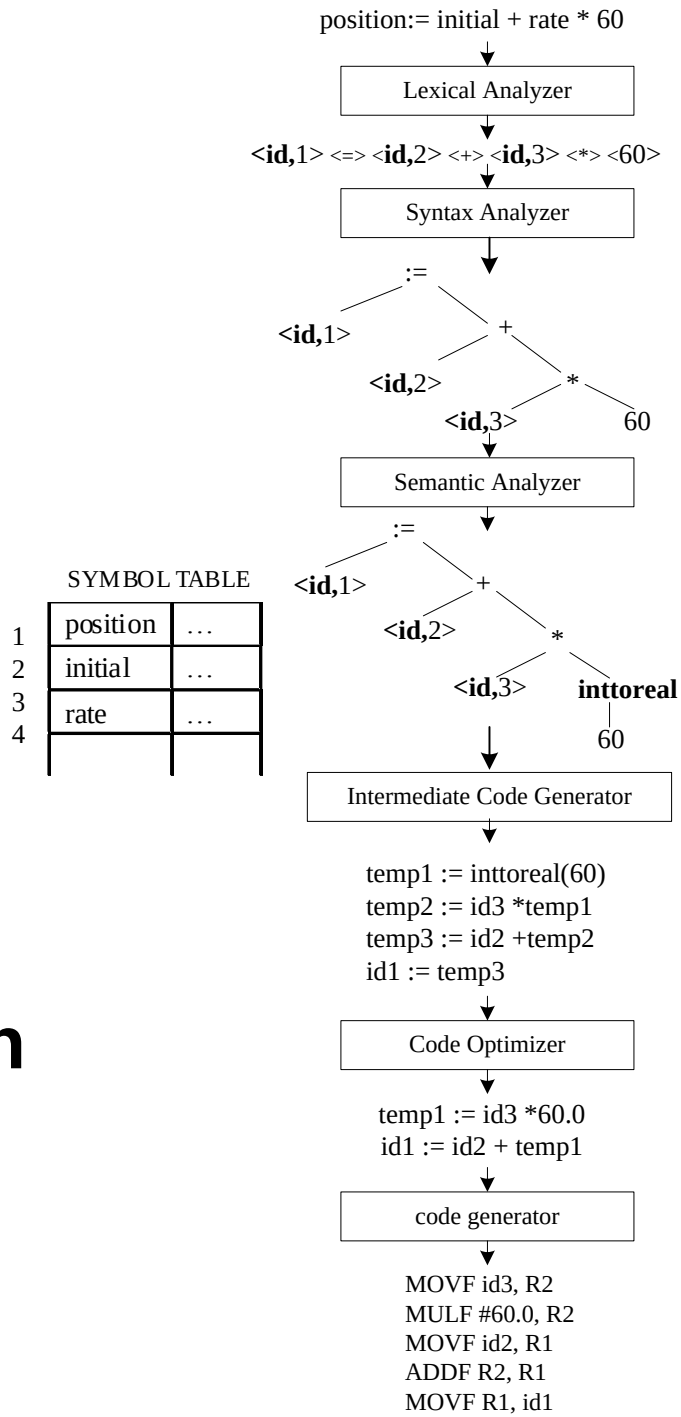
- **Example**
  - Motivation
  - Example
- **Formal Lang.***
  - Natural Lang.
  - Definitions
  - Operations
- **Description**
  - Grammar
- **CFG***

1. Example — Motivation



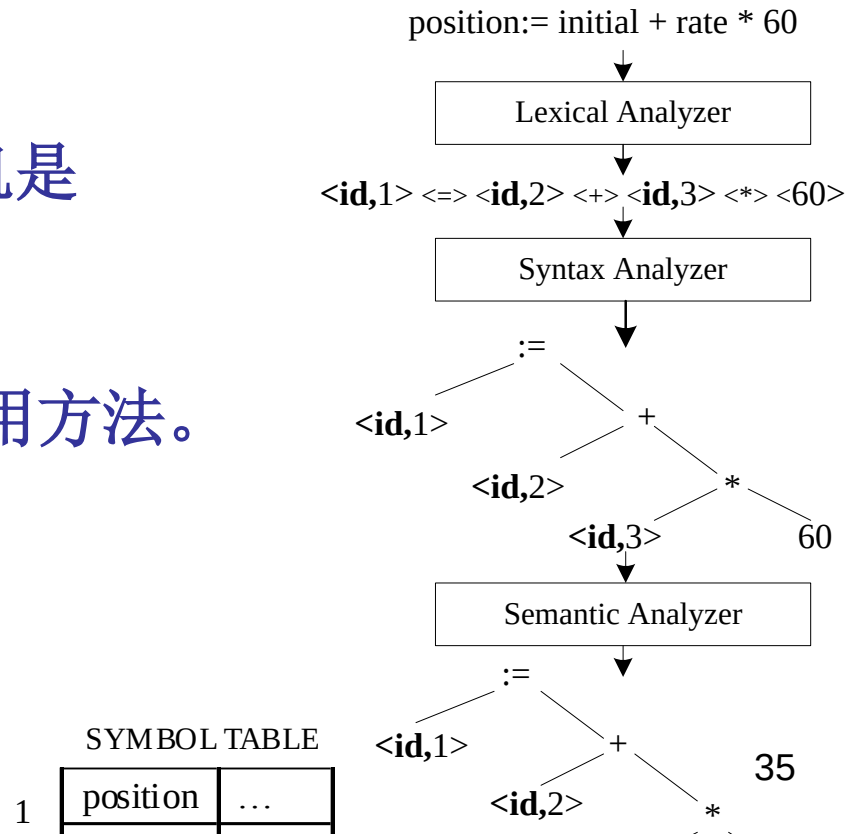
Translation of an assignment statement

position:= initial + rate * 60

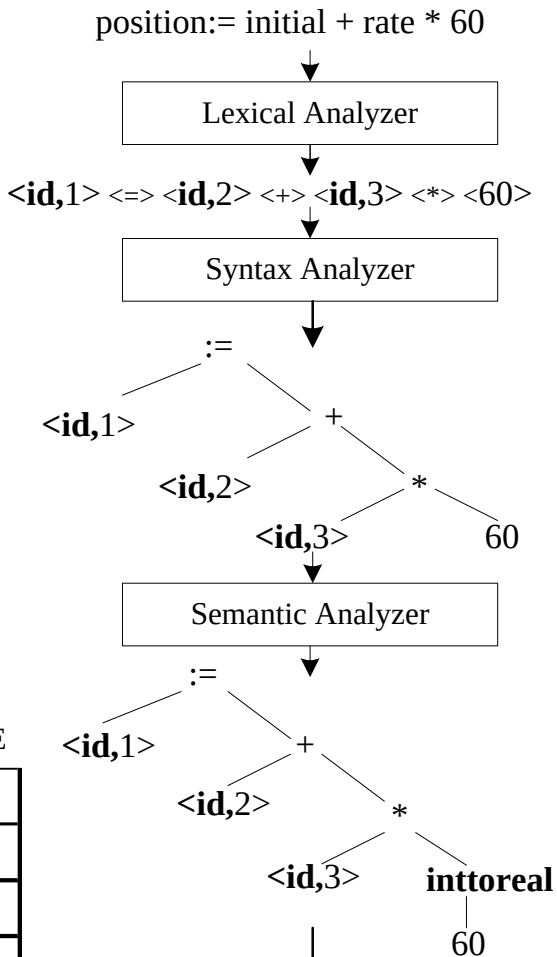


- 高级语言的认识
- 要学习和构造编译程序，理解和定义程序设计语言是必不可少的。
- 每个程序设计语言都有一定的规则用于规定合适程序的语法结构，也需要有对一个程序的含义的精确描述。
- 上下文无关文法CFG给出程序设计语言的精确的、易于理解的语法说明。
- 尚没有公认的形式系统描述程序含义。
- 流行的描述语义规则的方法—属性文法。

- 词法分析程序的自动构造
- 词法分析程序是编译程序的一个构成部分，它的主要任务是
  - 扫描源程序
  - 按构词规则识别单词
  - 报告发现的词法错误
- 正则表达式和有穷状态自动机是
  - 单词的描述工具和识别机制
  - 词法分析程序的自动构造原理
- 学习Lex/ANTLR等工具的使用方法。



- 语法分析程序的构造
- 自顶向下的语法分析
- 可以看作是为一个输入串寻找一个最左推导的过程,
- 等价于从根开始，按前序生成结点，为输入串构造分析树的过程。
- 讨论一种有效的无回溯的自顶向下分析程序，这种分析程序称为预测分析程序。
- 介绍对于一个文法类：LL（1）文法。
- 如何自动构造预测分析程序。

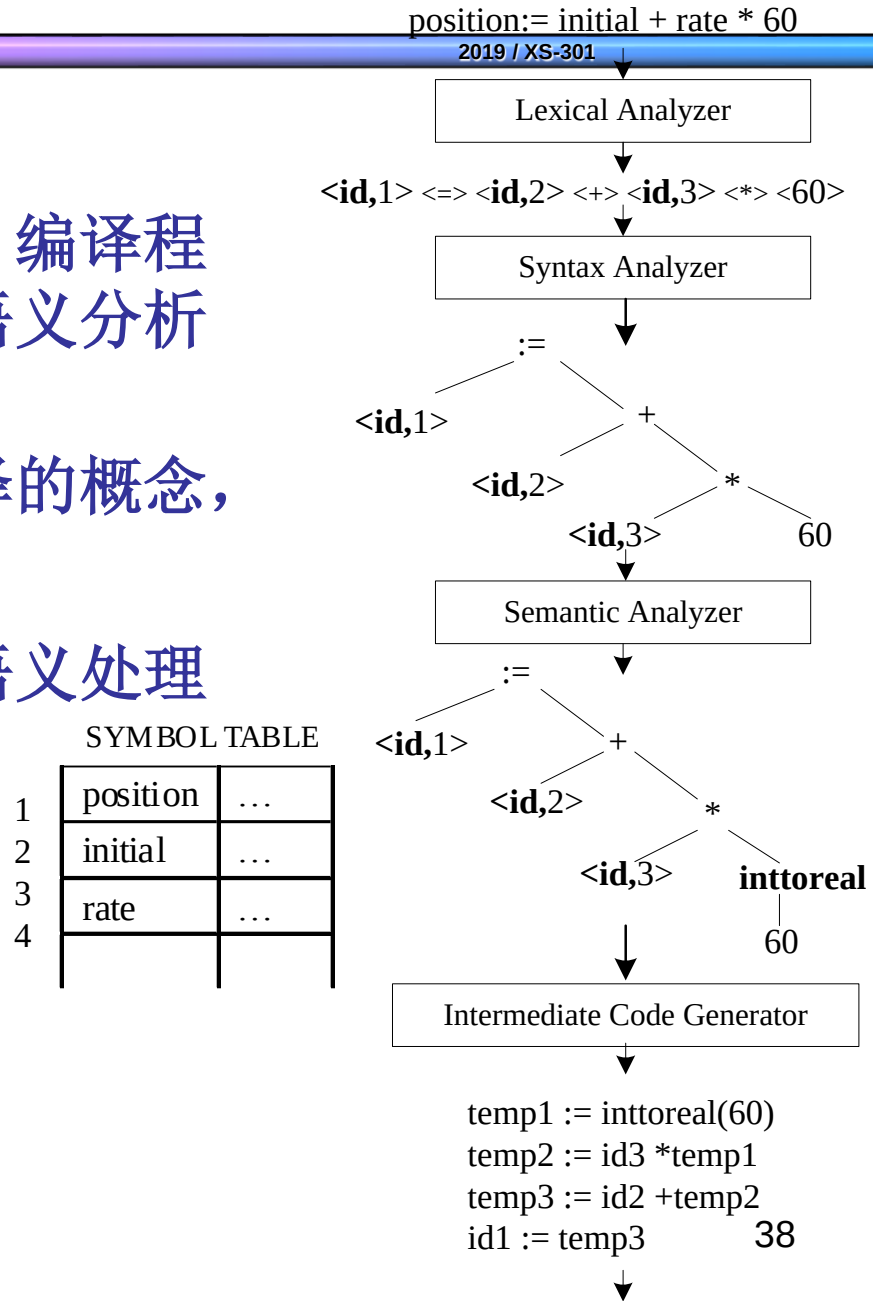


SYMBOL TABLE

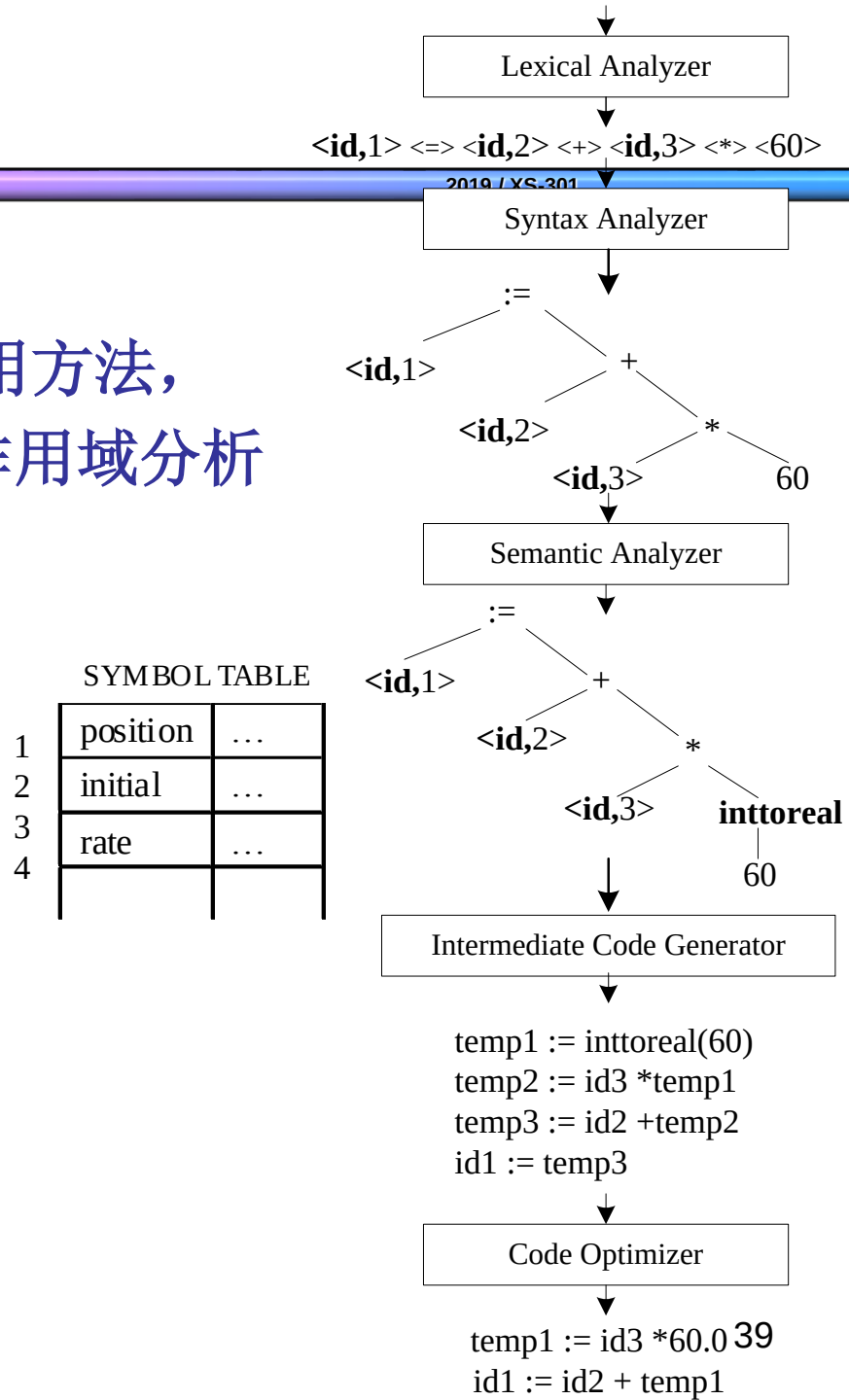
1	position	…
2	initial	…
3	rate	…
4		

- 自底向上（自下而上）语法分析方法
- 也称移进-归约分析法
- 基本思想是对输入符号串自左向右进行扫描，并将输入符逐个移入一个后进先出栈中，边移进边分析，一旦栈顶符号串形成可归约串，就用相应非终结符代替可归约串，这称为一步归约，重复这一过程，直到归约到栈中只剩文法的开始符号时，则为分析成功，并确认输入串是文法的句子。
- 介绍**LR**分析法，分析过程中归约的是当前句型的句柄，称为规范归约。
- 重点讲解**LR**类（**SLR**（**1**）、**LALR**（**1**）、**LR**（**1**））文法的分析表的构造原理。

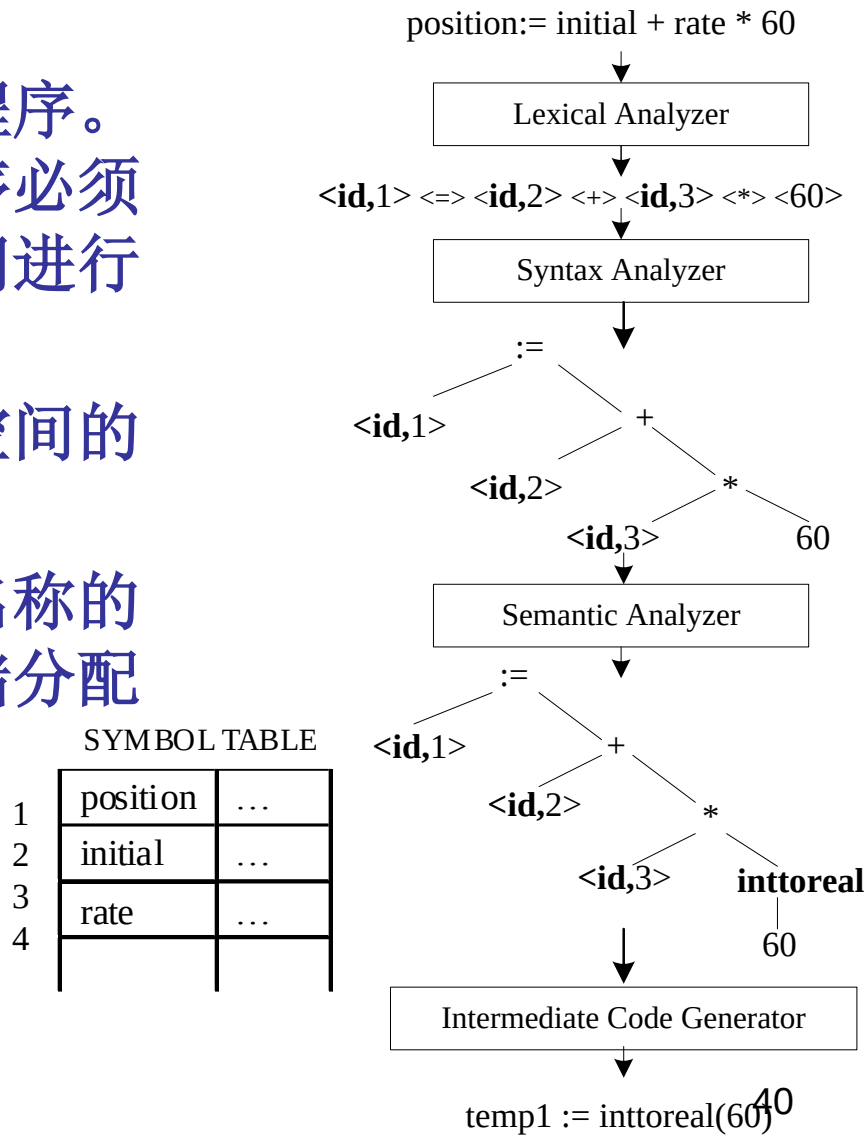
- 语义分析和中间代码生成
- 在词法分析和语法分析之后，编译程序下一个逻辑阶段的任务是语义分析和生成中间代码。
- 引入属性文法和语法制导翻译的概念，
- 介绍中间代码的形式，
- 针对一些语法成分讨论相应语义处理工作的描述。



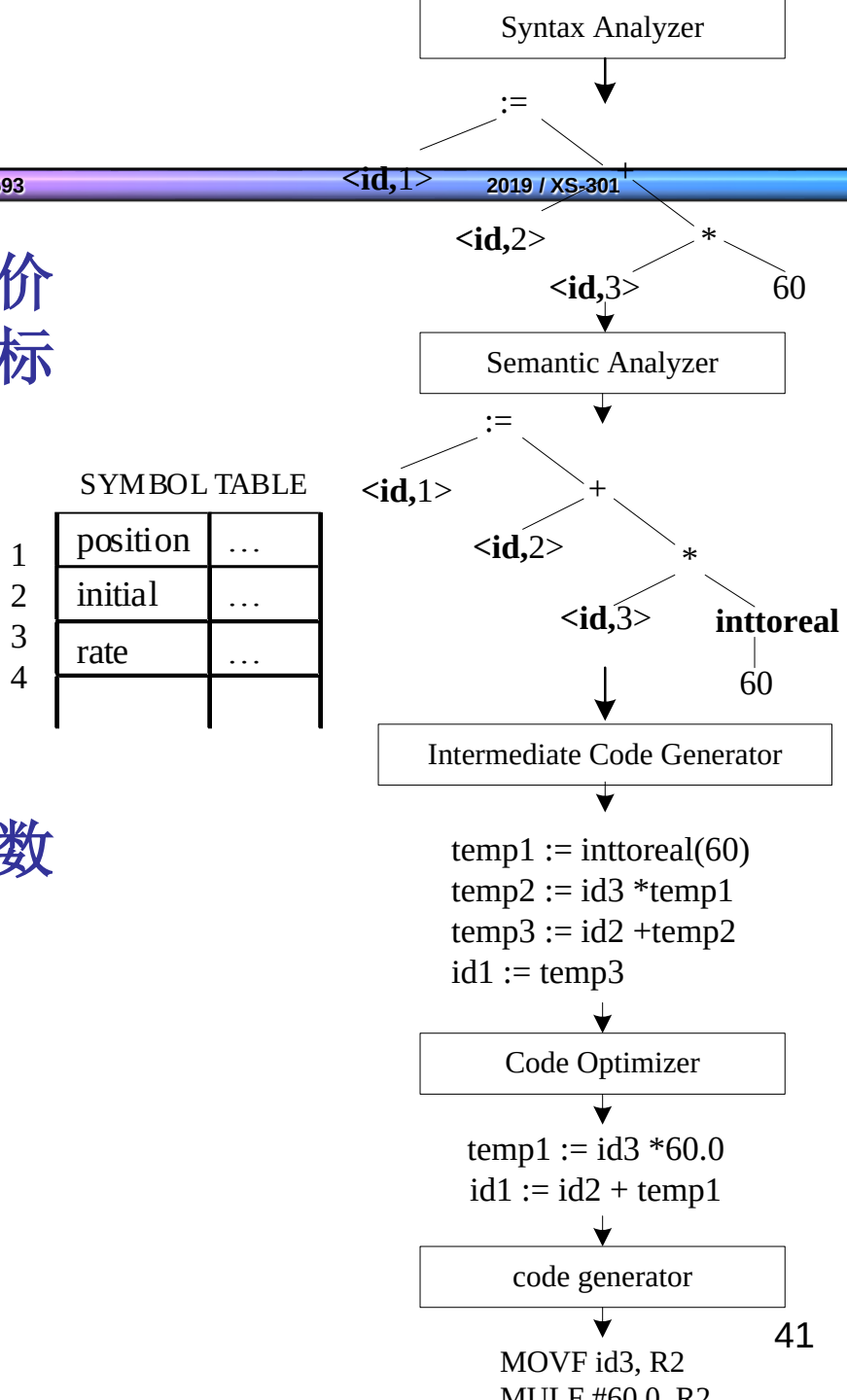
- 符号表
- 介绍符号表的一般组织和使用方法，
- 讨论分程序结构语言的名字作用域分析
- 讨论符号表设计方案

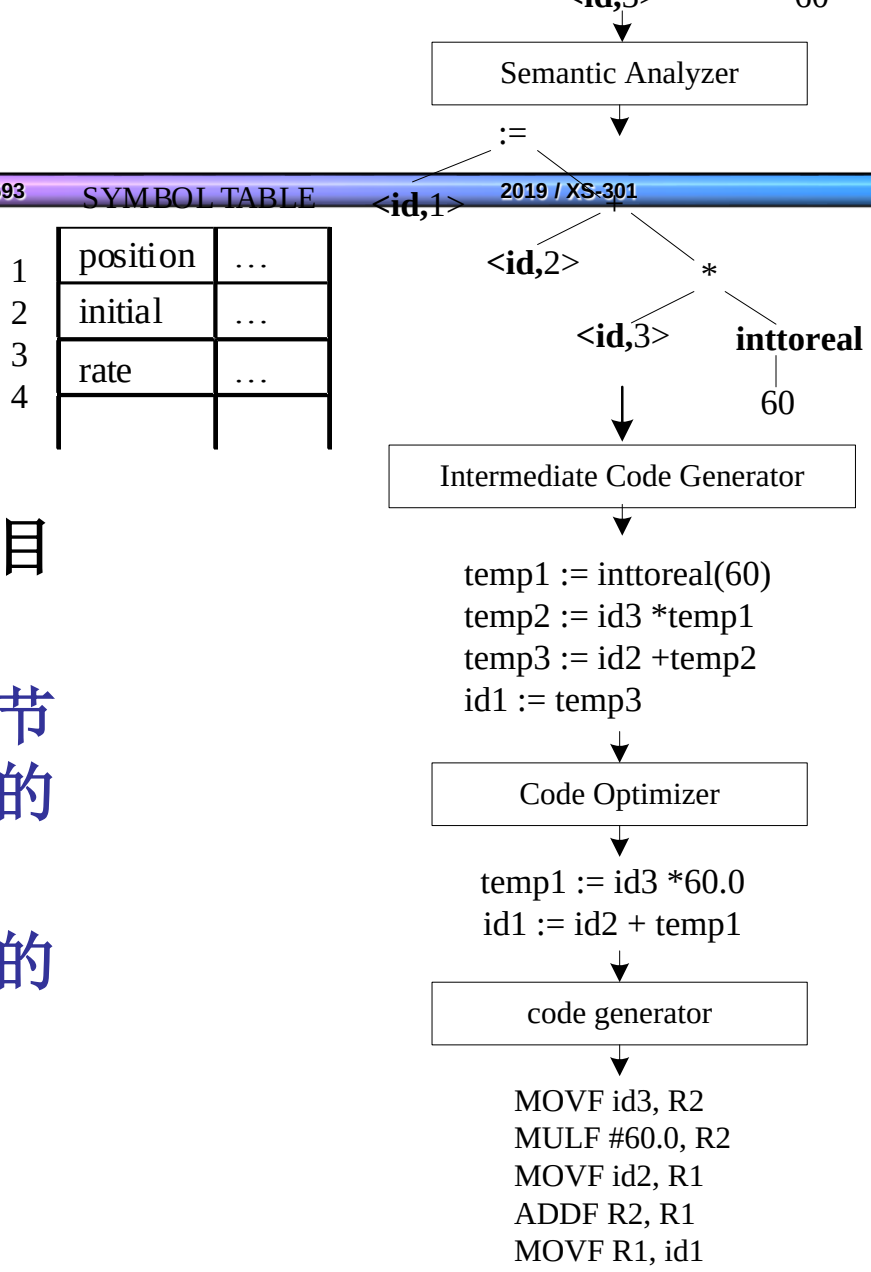


- 运行时的存储组织和管理
- 编译的最终目标是生成目标程序。在目标代码生成前，编译程序必须对目标程序运行时的数据空间进行组织和安排.
- 介绍目标程序运行时的数据空间的存储分配策略，
- 说明程序设计语言本身关于名称的作用域和生存期的规则与存储分配策略的关系，
- 重点讨论栈式动态存储方案.



- 代码优化是对代码作一些等价变换，以使得最后生成的目标代码更为高效。
  - 机器无关的优化
  - 机器相关的优化
- 介绍优化技术
- 优化分类
- 优化工作的基础—控制流和数据流分析问题

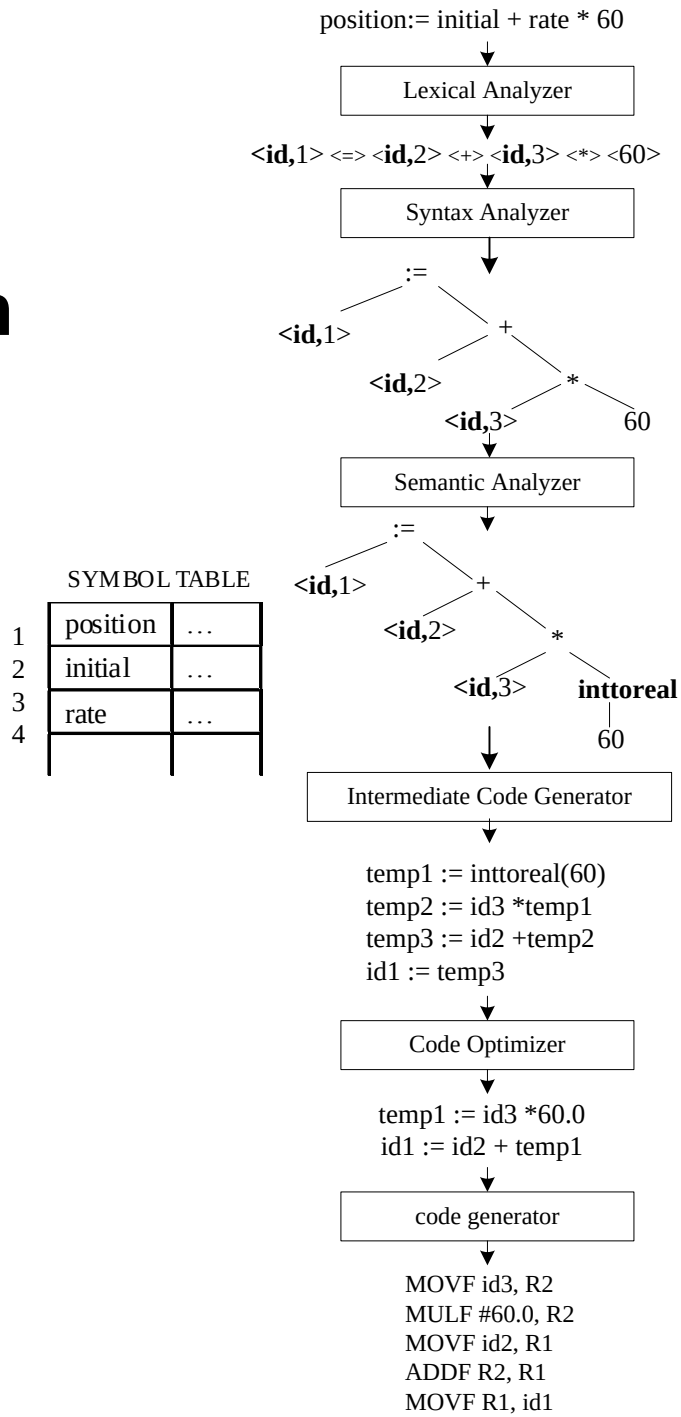




- 编译的最后一个逻辑阶段是目标代码生成。
- 目标代码生成程序的设计细节要考虑目标语言和操作系统的特点。
- 讨论目标代码生成程序设计的一般问题
  - 指令选择
  - 寄存器分配
  - 计算顺序选择

Translation of an assignment statement

- ✓ CFG
- ✓ RE
- ✓ Automata
- ✓ Graph
- ✓ …



2. Formal Languages

- **Artificial language**
  - **Programming Languages**
- **Formal languages**
- **Formal language theory**

Terminology

- An *alphabet* Σ (字母表, *character class* 字符类)
  - is a finite set of symbols (符号)
  - letters, characters
  - {0, 1}, ASCII, EBCDIC
- A *string s* over Σ (串, *sentence* 句子, *word* 字)
  - is a finite sequence of symbols from Σ
  - |s| denotes the length of string *s*
  - ε denotes the empty string, thus |ε| = 0
- A *language L* (语言)
  - is simply any set of strings over a fixed alphabet.
- camera
  - prefix of *s*, suffix of *s*, substring of *s*
  - proper prefix, suffix or substring of *s*
  - subsequence of *s* // car?

3. 如何描述一种语言

- 穷举法
- 语言中的句子数量可能是无穷的
  - 有穷表示方法
- 具有无穷句子的语言的有穷表示
  - 生成方式
    - 文法：严格定义的规则
  - 识别方式
    - 自动机：过程
      - 有限次计算后会停止并回答“是”
      - 停止并回答“不是”
      - 永远继续下去

文法即用生成方式描述语言

- 语言中的每个句子可以用严格定义的规则来构造

文法

- 文法G定义为四元组(V_N，V_T，P，S)其中
  - V_N：非终结符号(或语法实体，或变量)集；
  - V_T：终结符号集；
  - P： 规则的集合；
  - S： 称作识别符号或开始符号的一个非终结符，它至少要在一条产生式中作为左部出现。
- 注意：
  - V_N、V_T和P是非空有穷集。
  - $V_N \cap V_T = \phi$
  - 用V表示$V_N \cup V_T$，称为文法G的字母表或字汇表
  - 规则（重写规则、产生式或生成式）
    - 是形如 $\alpha \rightarrow \beta$ 或 $\alpha ::= \beta$ 的(α，β)有序对
    - $\alpha \in V^+$
    - $\beta \in V^*$
    - α 称为规则的左部， β 称作规则的右部。

直接推导（⇒）

直接推导

给定文法G(V_N, V_T, P, S)，若有v,w满足：

- v=γαδ,
- w=γβδ,
- α→β∈G,
- γ∈V*
- δ∈V*

则称v直接推导到w,记作 v ⇒ w

也称w直接归约到v

例：G： **S→0S1， S→01**

S ⇒ 0S1

0S1 ⇒ 00S11

00S11 ⇒ 000S111

000S111 ⇒ 00001111

推导

- 推导
  - 若存在$v = w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n = w, (n>0)$
  - 则记为$v \Rightarrow^+ w$，称作v推导出w，或w归约到v

  - 若有$v=w$ 或 $v \Rightarrow^+ w$
  - 则记为$v \Rightarrow^* w$

例：G： $S \rightarrow 0S1$， $S \rightarrow 01$

$0S1 \Rightarrow 00S11$

$00S11 \Rightarrow 000S111$

$000S111 \Rightarrow 00001111$

$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 00001111$

$S \Rightarrow^+ 00001111$

$S \Rightarrow^* S$

$00S11 \Rightarrow^* 00S11$

句型和句子

- 句型

  给定文法G，若S =>* x，则称x是文法G的句型。

- 句子

  给定文法G，若S =>* x，且x∈V_T*，则称x是文法G的句子。

例：G： S→0S1， S→01

S ⇒0S1 ⇒00S11 ⇒000S111 ⇒00001111

G的句型？

— S, 0S1 , 00S11, 000S111, 00001111

G的句子？

— 00001111, 01

L(G)

由文法G(V_N, V_T, P, S)生成的语言记为L(G)

— 是文法G的一切句子的集合

— L(G)={x|S =>* x，x ∈V_T*}

例：G： S→0S1， S→01

L(G)={0^n1^n|n≥1}

L(G) (cont'd)

例：文法G[S]：

（1）S→aSBE
（2）S→aBE
（3）EB→BE
（4）aB→ab
（5）bB→bb
（6）bE→be
（7）eE→ee

$L(G)=\{ a^n b^n e^n \mid n \geqslant 1 \}$

(1) G生成的每个串都在L(G)中
(2) L(G)中的每个串确实能被G生成

文法的等价

- 文法G_1和G_2是等价的
  - 若$L(G_1)=L(G_2)$

- 文法$G_1[A]$：A→DB 与$G_2[S]$：S→0S1 是否等价？

  A→DE　　S→01

  E→AB

  D→0

  B→1

Conclusions

1. Compilation and Interpretation (Chap.1)

2. The Phases of a Compiler (Chap.1)

3. Example (Chap.2)

4. Formal Lang. (Chap.2)

推荐教学资料

- 龙书 §1.2, §1.5
- §2 A Simple Syntax-Directed Translator
- 《ACM 图灵奖
  ——计算机发展史的缩影》

Thank you!