Introduction

- Attributes of a good language

Attributes of a good language

- Clarity, simplicity, and unity - provides both a framework for thinking about algorithms and a means of expressing those algorithms
- Orthogonality -every combination of features is meaningful
- Naturalness for the application - program structure reflects the logical structure of algorithm
- Support for abstraction - program data reflects problem being solved

Attributes of a good language (continued)

- Ease of program verification - verifying that program correctly performs its required function
- Programming environment - external support for the language
- Portability of programs - transportability} of the resulting programs from the computer on which they are developed to other computer systems
- Cost of use - program execution, program translation, program creation, and program maintenance

Program structure

* Syntax
  * What a program looks like
  * BNF (context free grammars) - a useful notation for describing syntax.
* Semantics
  * Execution behavior
  * Static semantics - Semantics determined at compile time:
    * var A: integer; Type and storage for A
    * int B[10]; Type and storage for array B
    * float MyProcC(float x;float y){...}; Function attributes
  * Dynamic semantics - Semantics determined during execution:
    * X = ``ABC'' SNOBOL4 example: X a string
    * X = 1 + 2; X an integer
    * :(X) X an address; Go to label X

Aspects of a program

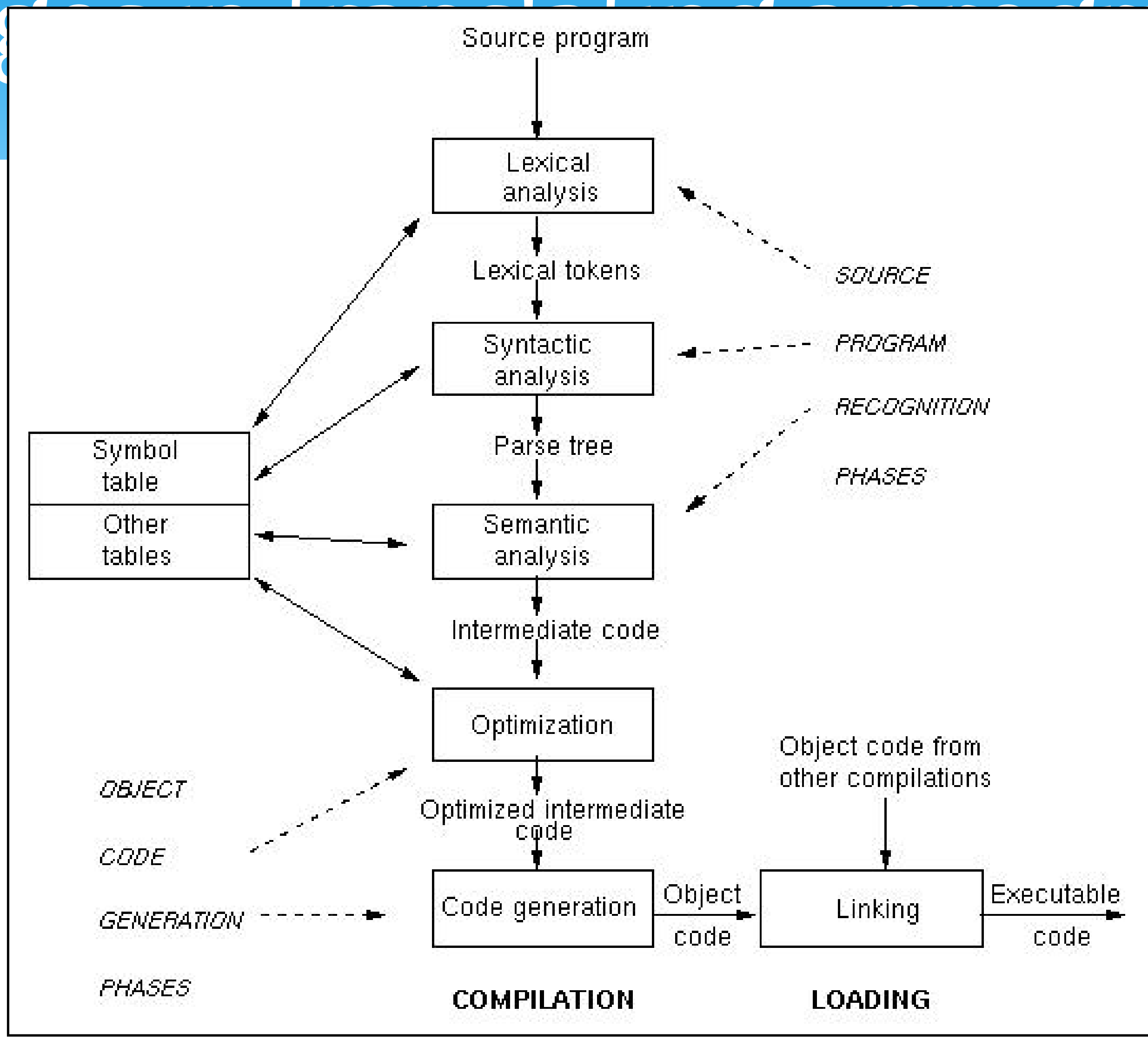
* Declarations - Information for compiler
  * var A: integer;
  * typedef struct { int A; float B } C;

* Control - Changes to state of the machine
  * if (A<B) { ... }
  * while (C>D) { ... }

* Structure often defined by a Backus Naur Form (*BNF*) grammar (First used in description of Algol in 1958. Peter Naur was chair of Algol committee, and John Backus was secretary of committee, who wrote report.)

* We will see later - BNF turns out to be same as context free grammars developed by Noam Chomsky, a linguist)

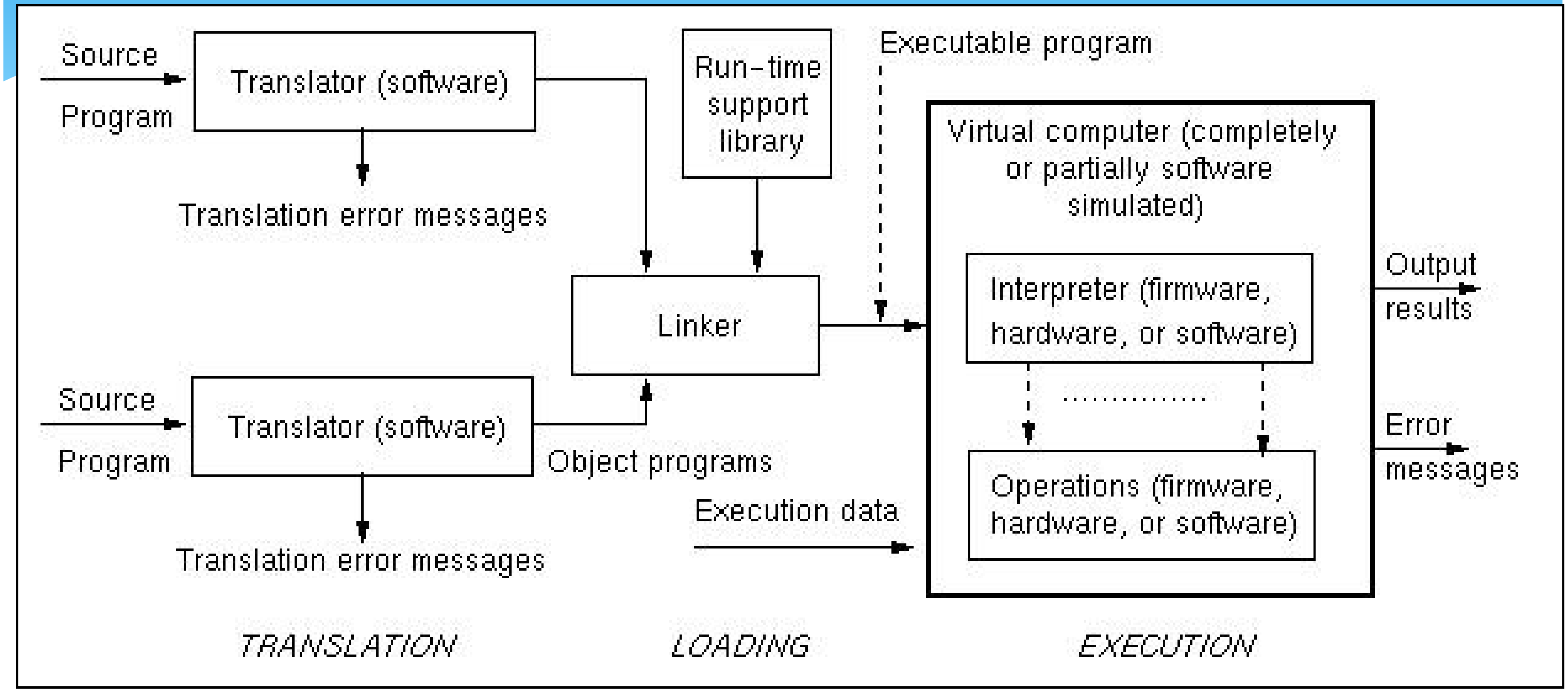
Stages in translating a program



Major stages

* Lexical analysis (Scanner): Breaking a program into primitive components, called tokens (identifiers, numbers, keywords, ...) We will see that regular grammars and finite state automata are formal models of this.
* Syntactic analysis (Parsing): Creating a syntax tree of the program. We will see that context free grammars and pushdown automata are formal models of this.
* Symbol table: Storing information about declared objects (identifiers, procedure names, ...)
* Semantic analysis: Understanding the relationship among the tokens in the program.
* Optimization: Rewriting the syntax tree to create a more efficient program.
* Code generation: Converting the parsed program into an executable form.
* We will briefly look at scanning and parsing. A full treatment of compiling is beyond scope of this course.

Translation environments



BNF grammars

- Nonterminal: A finite set of symbols: <sentence> <subject> <predicate> <verb> <article> <noun>
- Terminal: A finite set of symbols: the, boy, girl, ran, ate, cake
- Start symbol: One of the nonterminals: <sentence>
- Rules (productions): A finite set of replacement rules:

```
<sentence> ::= <subject> <predicate>
<subject>  ::= <article> <noun>
<predicate>::= <verb> <article> <noun>
<verb>     ::= ran | ate
<article>  ::= the
<noun>     ::= boy | girl | cake
```

- Replacement Operator: Replace any nonterminal by a right hand side value using any rule (written ⇒)

Example BNF sentences

* <sentence> ⇒ <subject> <predicate> First rule
* ⇒ <article> <noun> <predicate> Second rule
* ⇒ the <noun> <predicate> Fifth rule
* ... ⇒ the boy ate the cake

* Also from <sentence> you can derive
* ⇒ the cake ate the boy
* Syntax does not imply correct semantics

* Note:
* Rule <A> ::= <B><C>
* This BNF rule also written with equivalent syntax:
* $A \rightarrow BC$

Languages

* Any string derived from the start symbol is a sentential form.

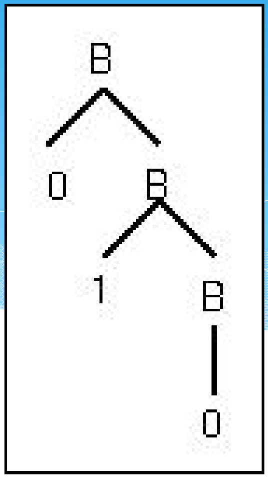
* Sentence: String of terminals derived from start symbol by repeated application of replacement operator

* A language generated by grammar G (written L(G)) is the set of all strings over the terminal alphabet (i.e., sentences) derived from start symbol.

- That is, a language is the set of sentential forms containing only terminal symbols.

Derivations

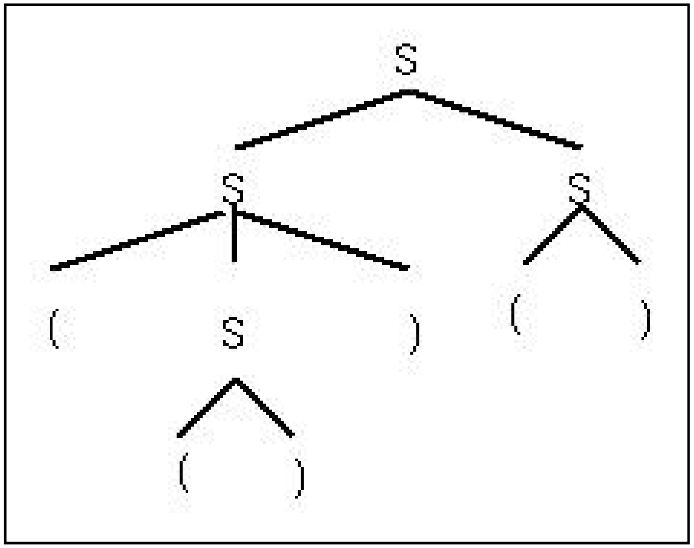
* A derivation is a sequence of sentential forms starting from start symbol.



* Derivation trees:
* Grammar: B → 0B | 1B | 0 | 1
* Derivation: B ⇒ 0B ⇒ 01B ⇒ 010
* From derivation get parse tree

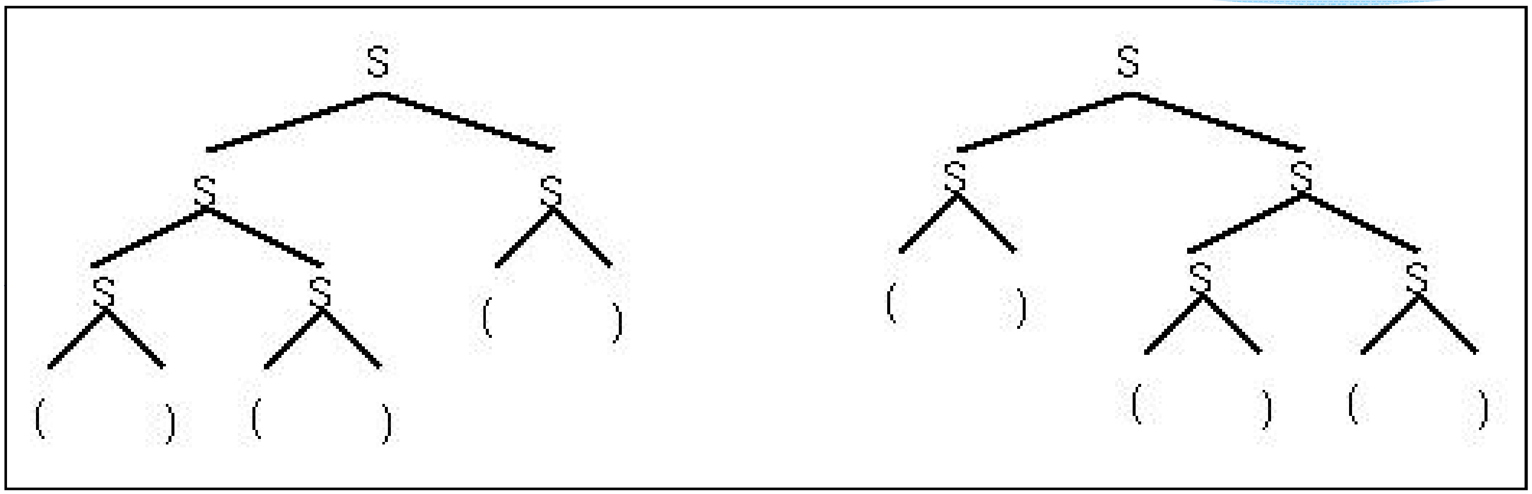
* But derivations may not be unique
* S → SS | (S) | ()

* S ⇒ SS ⇒(S)S ⇒(())S ⇒(())()
* S ⇒ SS ⇒ S() ⇒(S)() ⇒(())()
* Different derivations but get the same parse tree



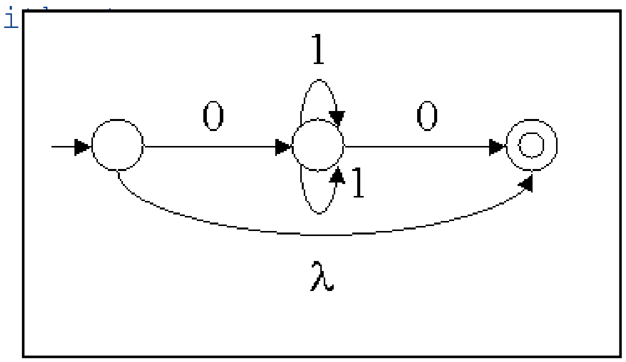
Ambiguity

- But from some grammars you can get 2 different parse trees for the same string: ()()()
- Each corresponds to a unique derivation:
- $S \Rightarrow SS \Rightarrow SSS \Rightarrow ()SS \Rightarrow ()()S \Rightarrow ()()()$



Role of λ

* How to characterize strings of length 0? – Semantically it makes sense to consider such strings.
* 1. In BNF, ε-productions: S → SS | (S) | () | ε
* Can always delete them in grammar. For example:
* X → abYc
* Y → ε
* Delete ε-production and add production wi
* X → abYc
* X → abc
* 2. In fsa - λ moves means that
* in initial state, without input
* you can move to final state.



Syntax can be used to determine some semantics

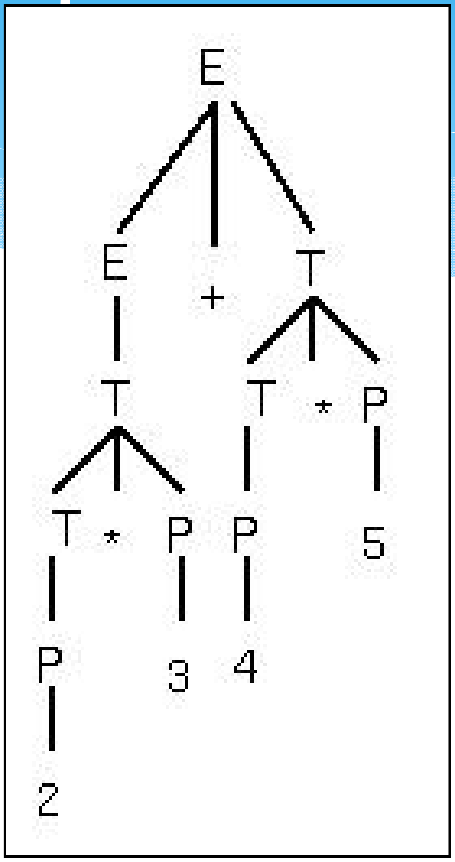
* During Algol era, thought that BNF could be used for semantics of a program:

* What is the value of: 2 * 3 + 4 * 5?
*   (a) 26
*   (b) 70
*   (c) 50

* All are reasonable answers? Why?

Usual grammar for expressions

* E → E + T | T
* T → T * P | P
* P → i | (E)



* "Natural" value of expression
* is 26
  - Multiply 2 * 3 = 6
  - Multiply 4 * 5 = 20
  - Add 6 + 20 = 26

But the "precedence" of operations is only a convention

* Grammar for 70
* E → E * T | T
* T → T + P | P
* P → i | (E)

* Grammar for 50
* E → E + T | E * T | T
* T → i | (E)

All 3 grammars generate exactly the same language, but each has a different semantics (i.e., expression value) for most expressions.

Draw parse tree of expression 2*3+4*5 for each grammar

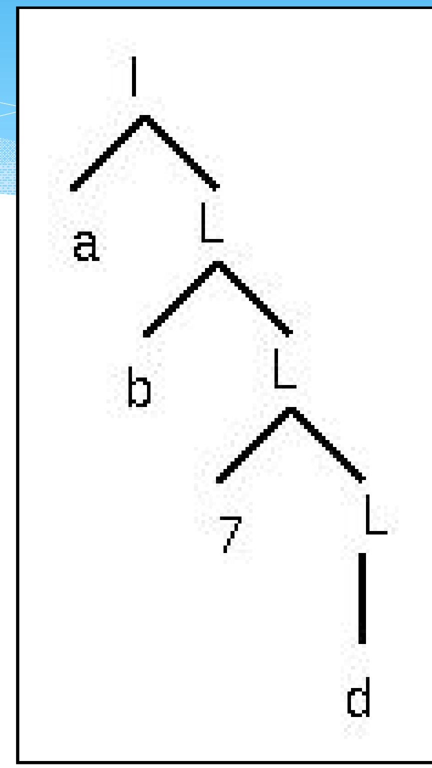
Classes of grammars

* BNF: Backus-Naur Form – Context free – Type 2 – Already described

* Regular grammars: subclass of BNF – Type 3:
* BNF rules are restricted: A → t N | t
* where: N = nonterminal, t = terminal

* Examples:
* Binary numbers: B → 0 B | 1 B | 0 | 1

* Identifiers:
* I → a L | b L | c L |...| z L | a |...| y | z
* L → 1 L | 2 L |...| 9 L | 0 L | 1 |...| 9 | 0 | a L | b L | c L |...| z L | a |...| y | z



ab7d

Other classes of grammars

* The context free and regular grammars are important for programming language design. We study these in detail.

* Other classes have theoretical importance, but not in this course:

* Context sensitive grammar: Type 1 - Rules: $\alpha \to \beta$ where $| \alpha | \leq | \beta |$ [That is, length of $\alpha \leq$ length of β, i.e., all sentential forms are length non-decreasing]

* Unrestricted, recursively enumerable: Type 0 -
* Rules: $\alpha \to \beta$. No restrictions on α and β.