Compilers

-

Course Overview

Dr.-Ing. Ali Diab

Outline

- Introduction
- Goals & Organizational Stuff
- Compilers, Interpreters, …
- Brief Introduction
- Simple One Pass Compiler

Goals & Organizational Stuff

Goals

- Provision of up-to-date knowledge concerning compilers
  - Architecture
  - Algorithms
  - ...
- In-depth understanding of how to solve technical problems in this concern
- Focus on
  - Lexical & Semantic Analysis...
  - Parsing
  - Code Generation
  - ...

Organizational Stuff

- Lecturer contact information
  - Dr.-Ing. Ali Diab
  - email: adiab@albaath-univ.edu.sy (Subject: Compilers)
  - email: dring_alidiab@outlook.de (Subject: Compilers)

- Course prerequisites
  - Basics in programming language
    - High level programming language
    - Assembly language

- Course budget
  - 1 Lecture per week
  - 1 praxis appointment per week

How to Handle the Course

- Cover your knowledge holes
  - Go through basics
- Go through the material provided
  - Will be electronically provided
  - References will be listed for each lecture
    - If possible, electronic copy will be provided
  - Search the Internet for knowledge if required
- Questions catalog will be provided
  - Cover 50 – 60 % of the course
- Exam
  - ?????

Introduction

Programming Languages

- Programming language is a set of instructions and rules used to implement blocks of software to perform certain operation

- Two types of programming languages
  - Low-level languages
    - Machine language: the language the CPU understads and executes
    - Assemply language: the first level of coding of machine language into readable instructions

  - High level languages
    - Written in languages similar to those used in nature
    - Simpler and more readable than machine and assemply languages
    - Need to be compiled and built

Machine Language

- The language the CPU understands
- Set of “1“s and “0“s
- E.g.
  - 10100001 00000000 00000000 (fetch the content of the address “0” and put them in the register AX)
  - 00000101 00000100 00000000 (add 4 to AX)
  - 10100011 00000000 00000000 (save the content of AX in the memory under the address “0”)
- Writing programms in machine language is a tough task

Assemply Language

- The first level of coding of machine language into readable instructions

- Write programms using
  - Commands: MOV, SUB, XCHNG, etc.
  - Register names: AX, BX, CX, etc.
  - Memory addresses: [1000H], [2345H], etc.
  - Data: A DW 2 (define a variable with the name “A“ and the value “2“)

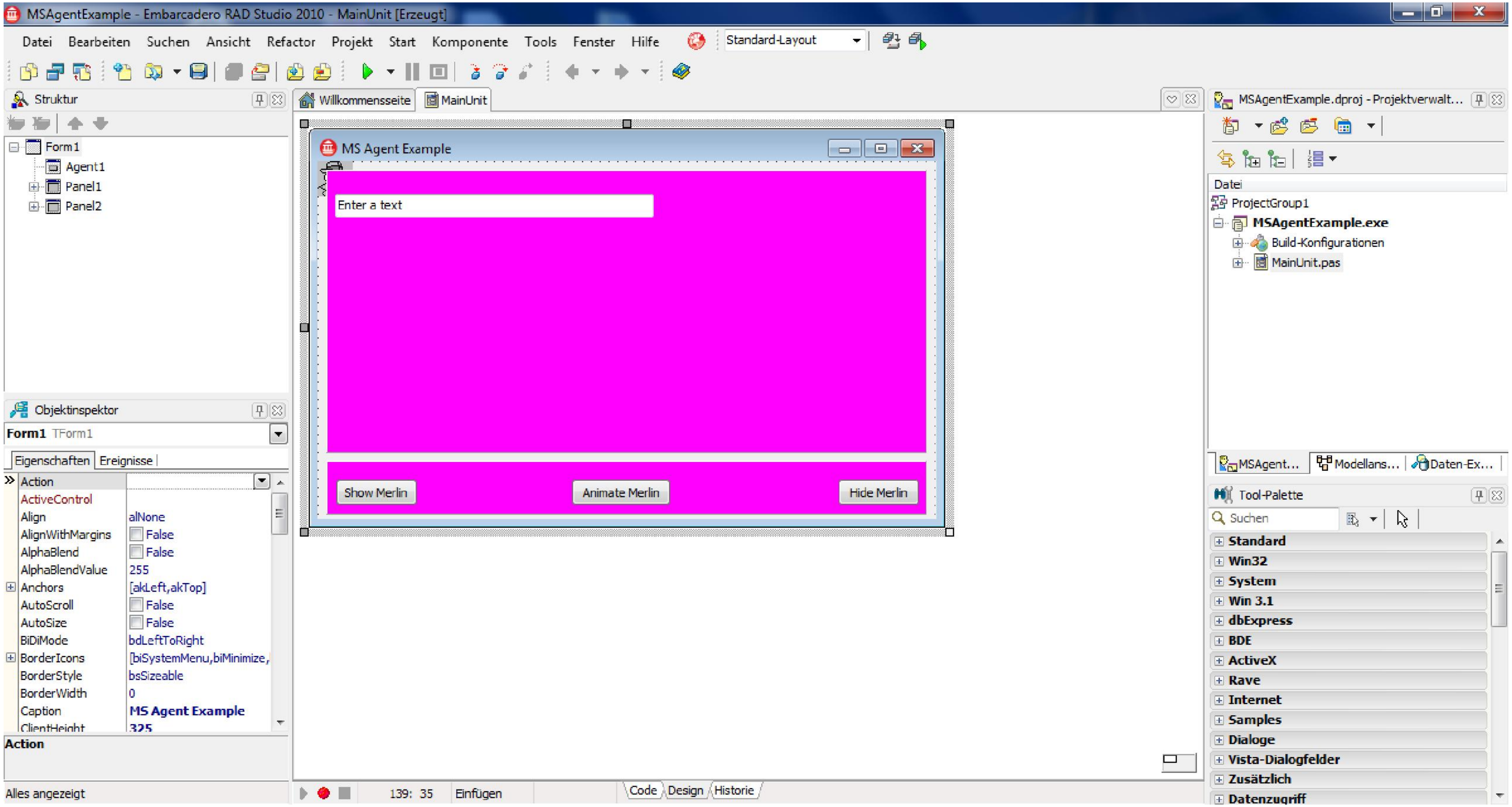
- Programms written using assemply are faster than those written using high-level languages

- Programms written using assemply should be converted into programms written in machine language
  - The converter is called assempler

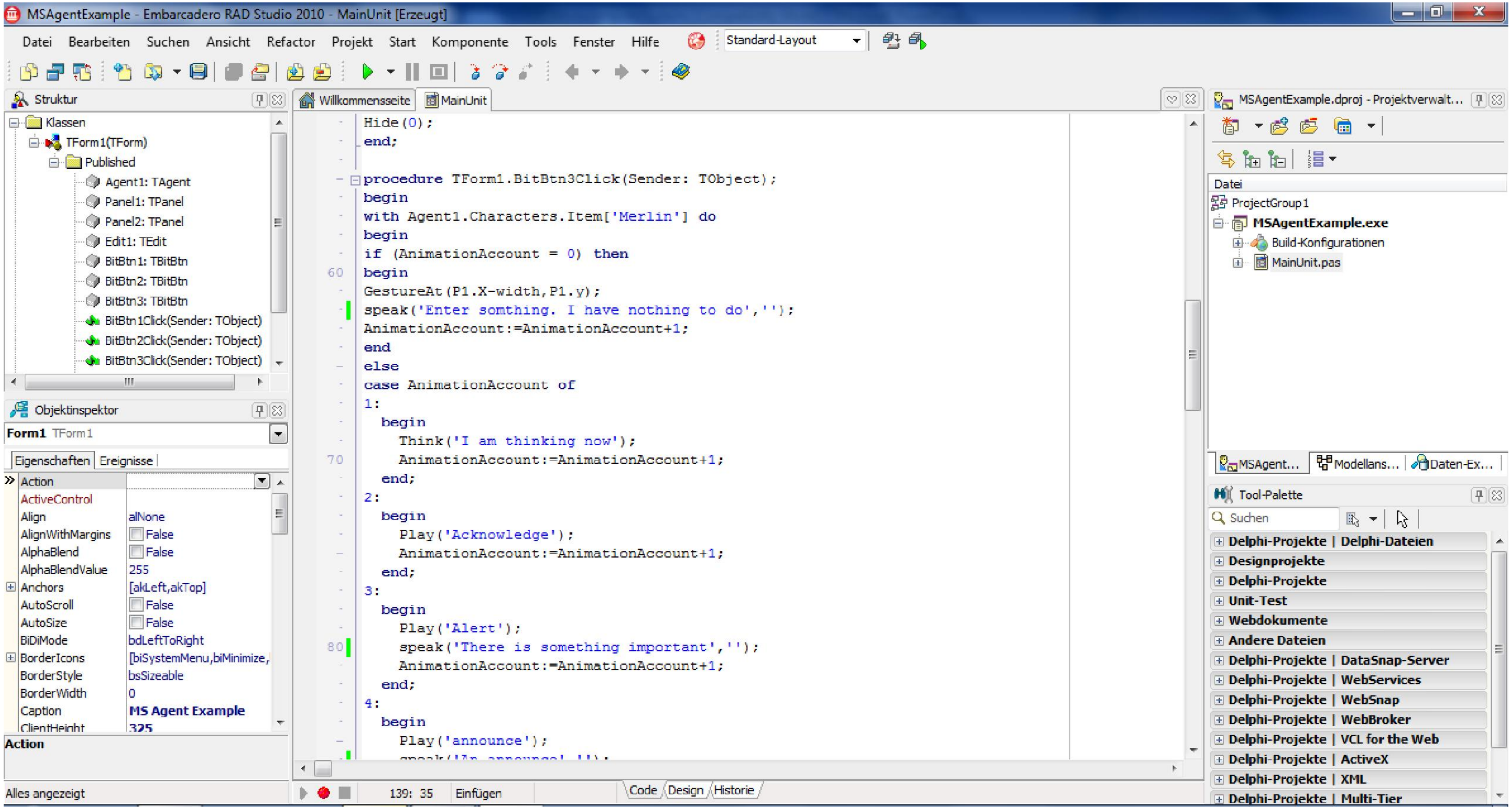
High-Level Languages

- Written in languages similar to those used in nature
- Widely used and acceptable
- E.g.
  - C++, Delphi, Java, C#, PHP, TCL, etc.
- Visual versions are widely used
- Programms written using high-level language should be
  - Compiled to check for syntax errors
  - Built to be converted into programms written assemply language and machine language

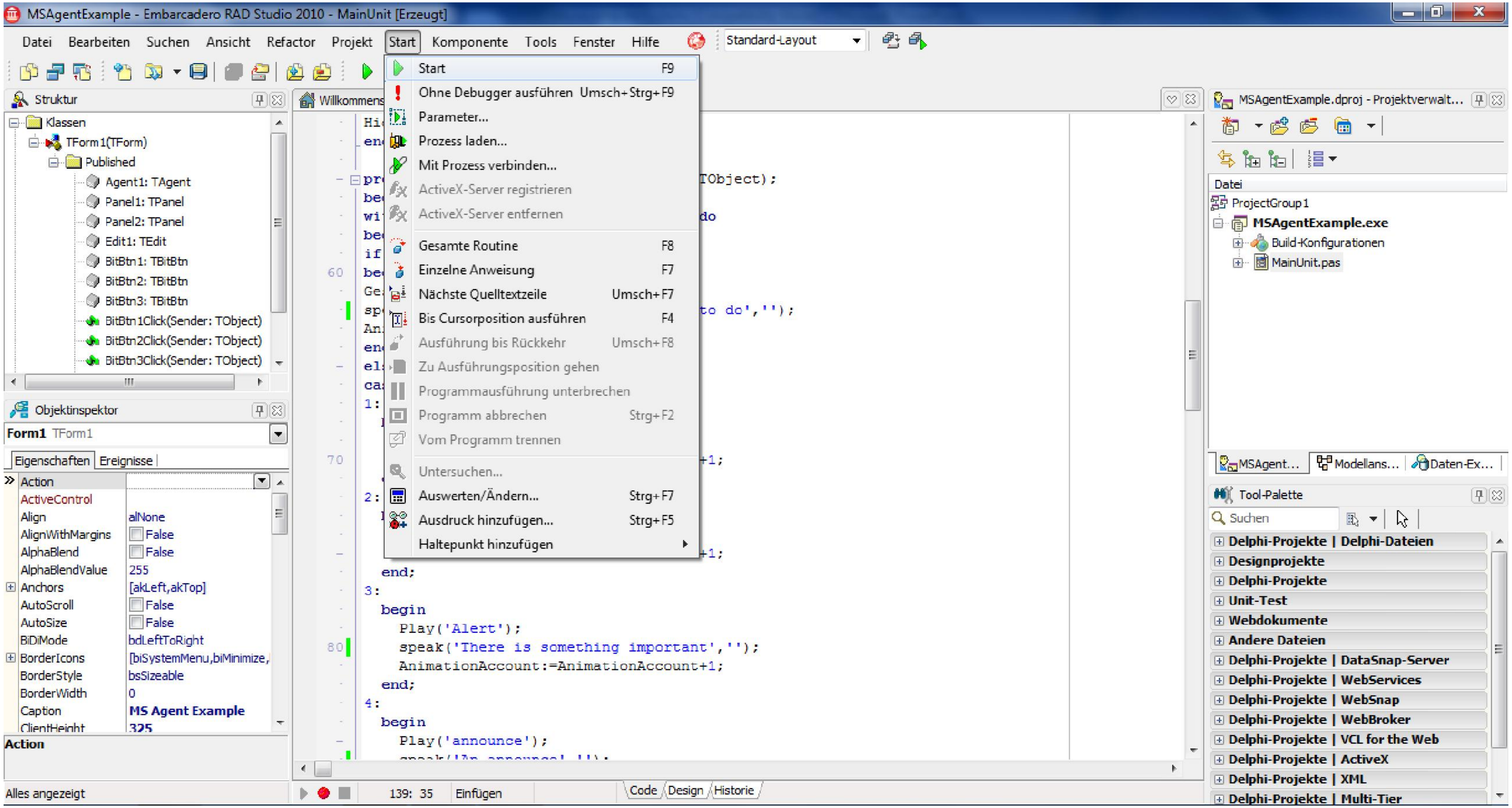
High-Level Languages



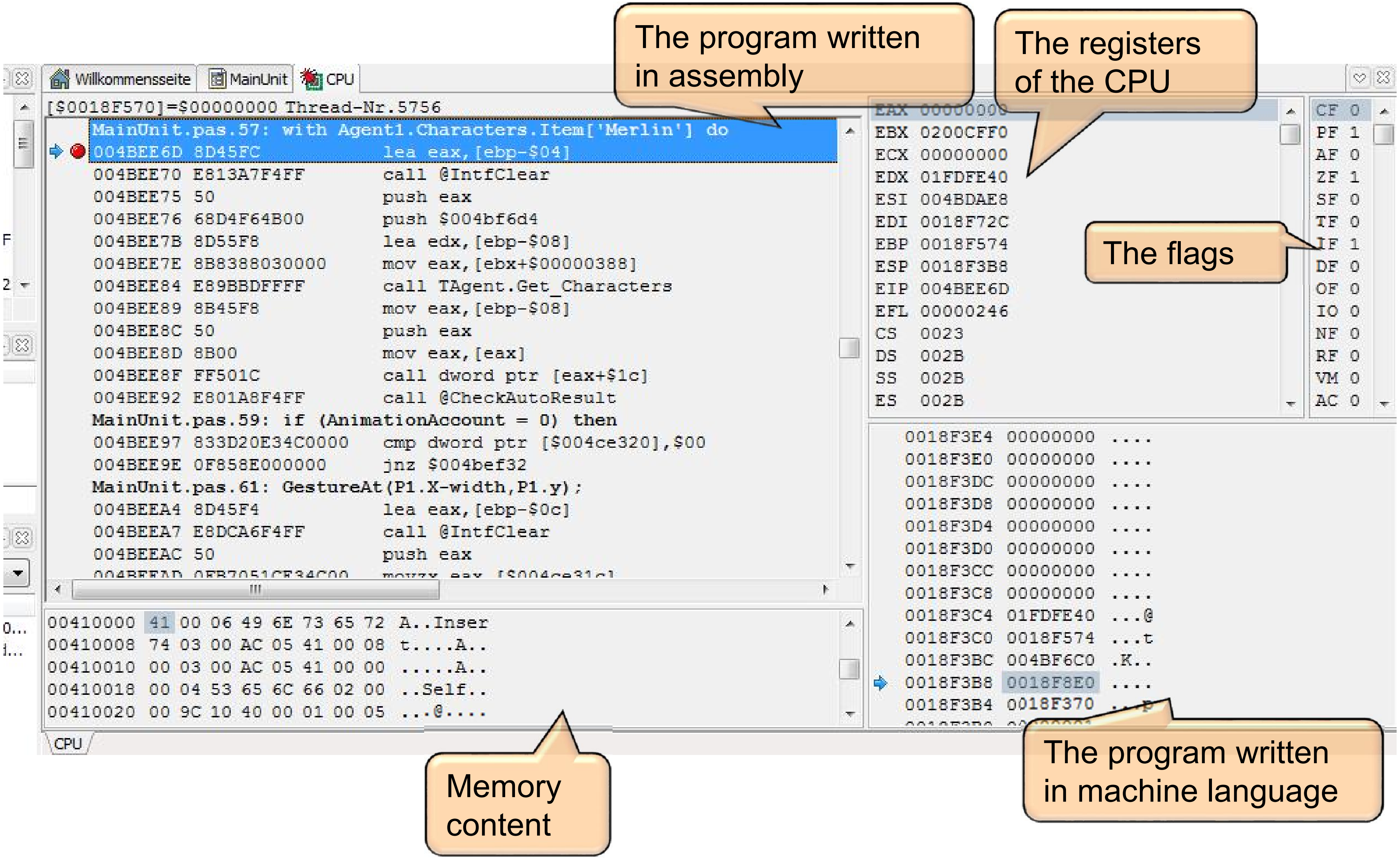
Compile & Build of Programs



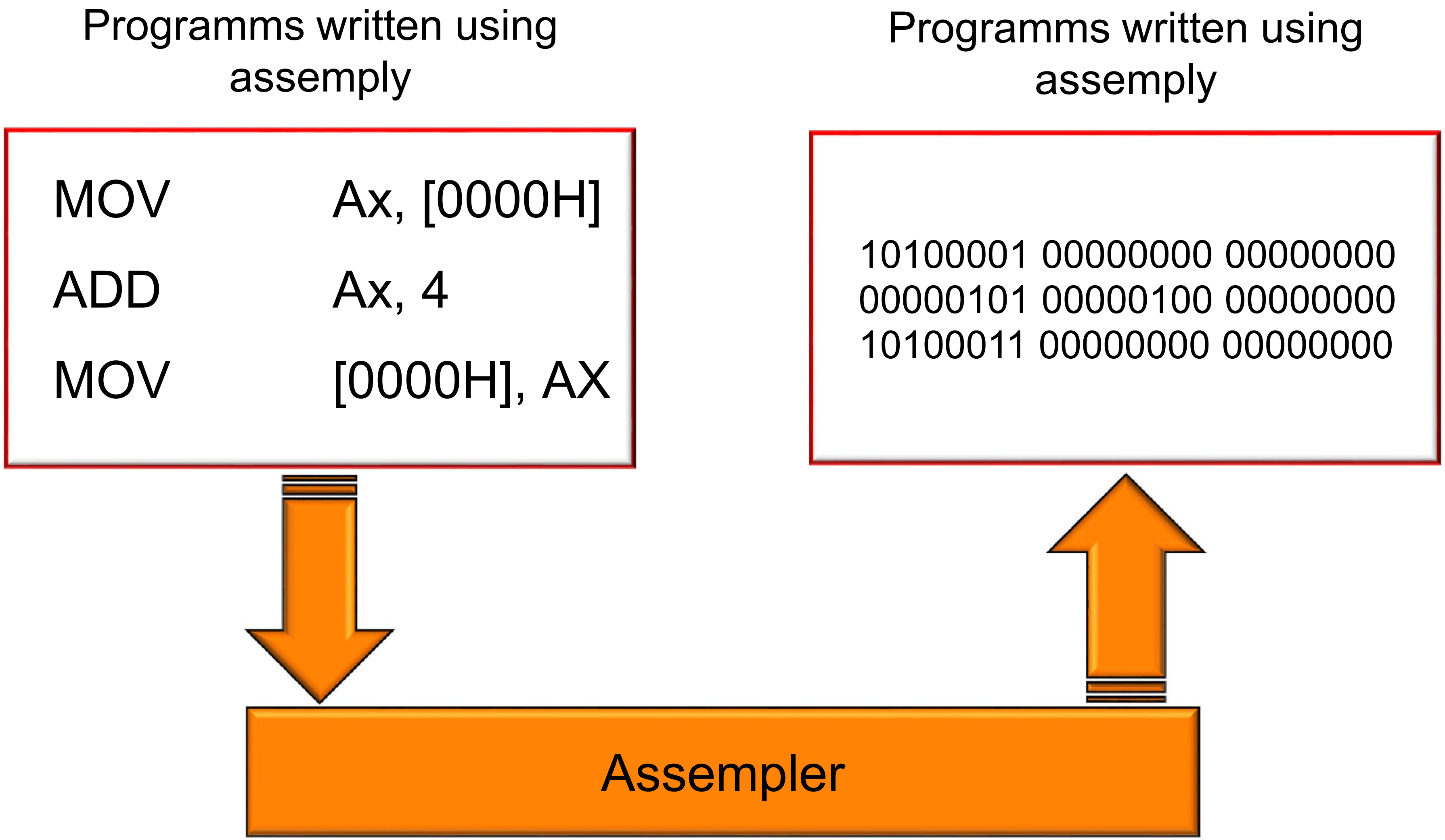
Compile & Build of Programs - Con



Compile & Build of Programs - Con



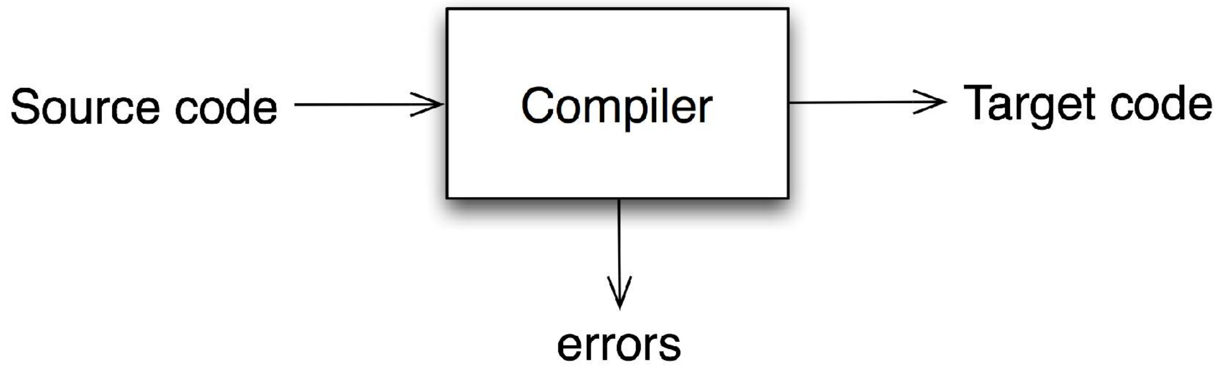
Compile & Build of Programs - Con



Compilers, Interpreters, …

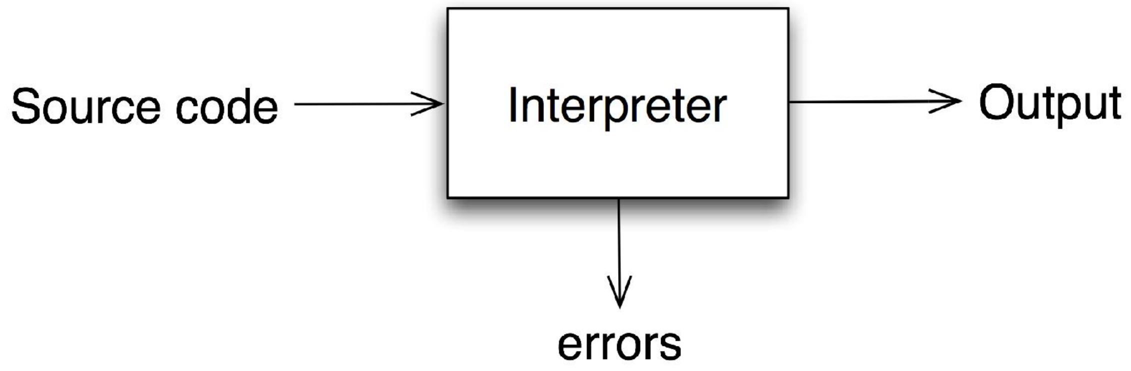
What is a compiler?

- a program that **translates** an **executable** program in one language into an **executable** program in another language

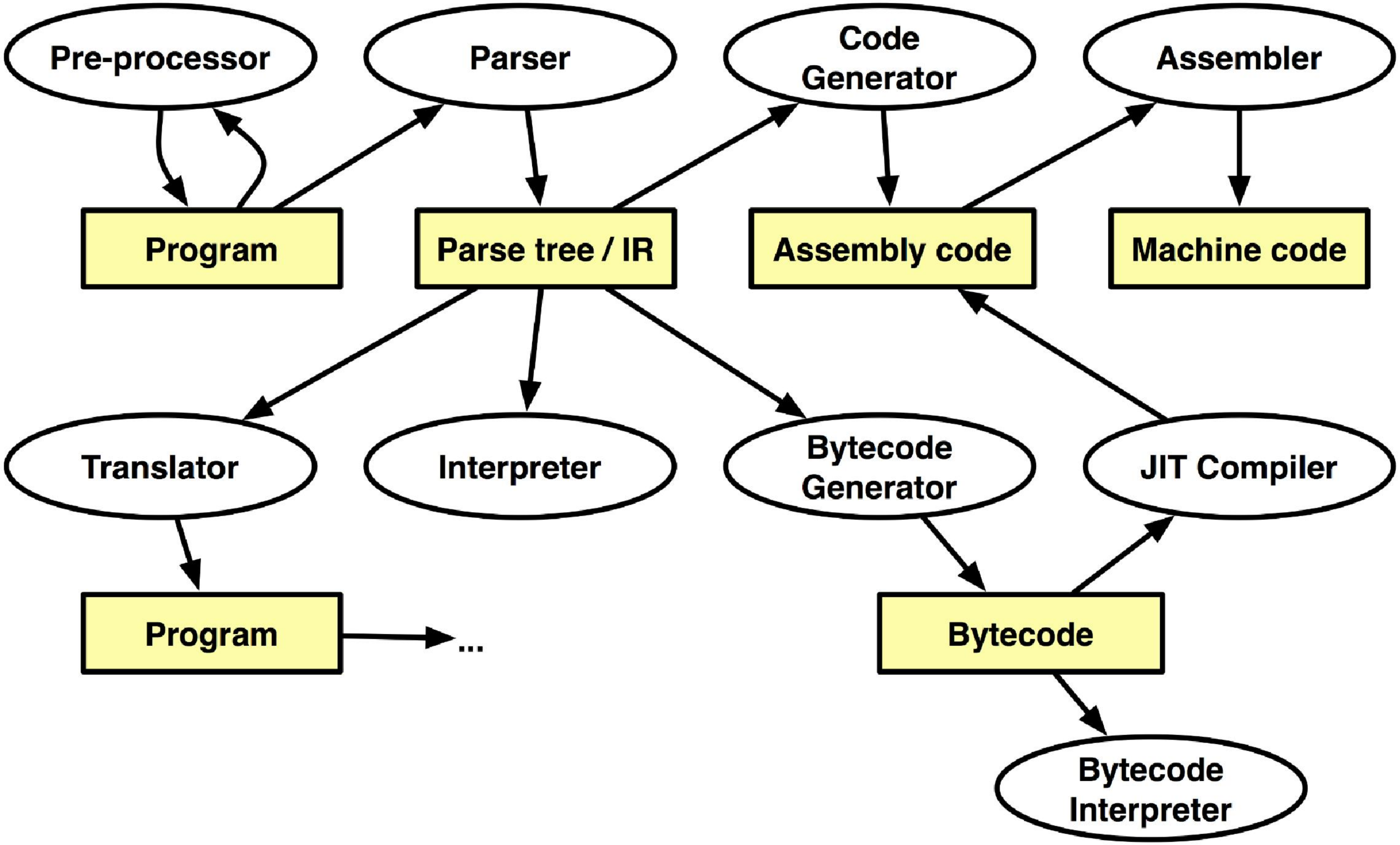


What is an interpreter?

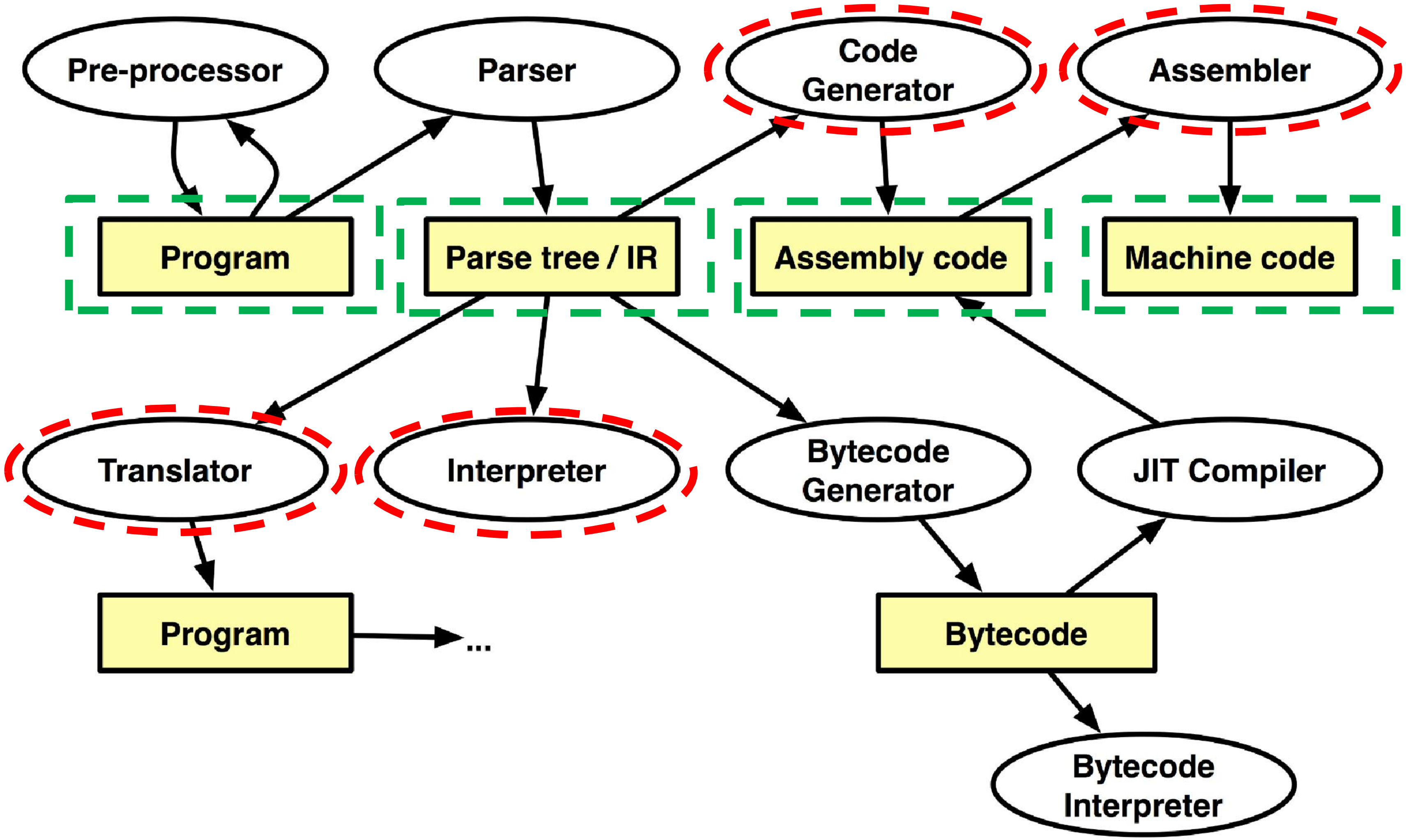
- a program that **reads** an **executable** program and **produces** the results of running that program



Compilers, Interpreters, …



Compilers, Interpreters, …



Brief Introduction

Does Compilers Evolve

- Machines are constantly changing
  - Changes in architecture $\Rightarrow$ changes in compilers
  - New features pose new problems
  - Changing costs lead to different concerns
  - Old solutions need re-engineering

- Innovations in compilers should prompt changes in architecture
  - New languages and features

Why Do We Care?

- Compiler construction is a microcosm of computer science

Artificial intelligence	Greedy algorithms Learning algorithms
Algorithms	Graph algorithms Union-find Dynamic programming
Theory	DFAs for scanning Parser generators Lattice theory for analysis
Systems	Allocation and naming Locality Synchronization
Architecture	Pipeline management Hierarchy management Instruction set use

What Qualities are Important in a Compiler?

- Correct code
- Output runs fast
- Compiler runs fast

- Compile time proportional to program size
- Support for separate compilation
- Good diagnostics for syntax errors

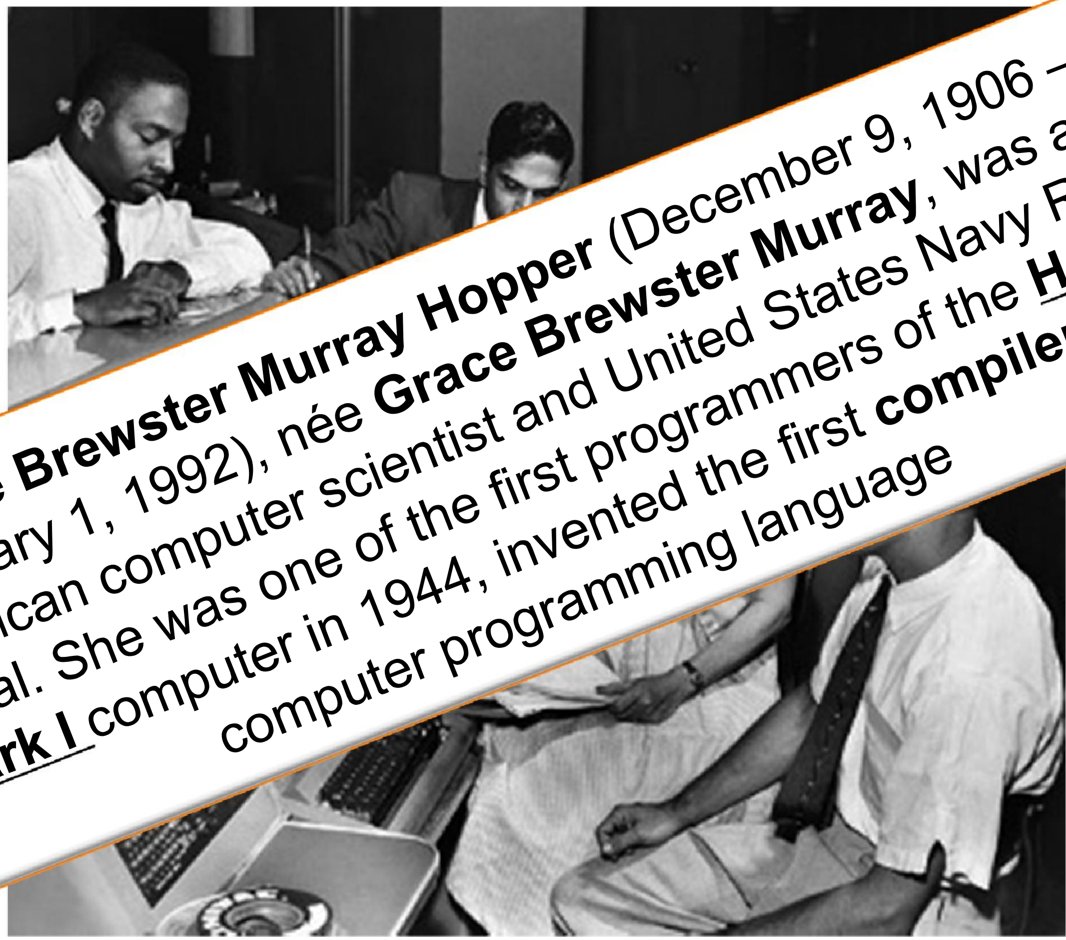
- Works well with the debugger
- Good diagnostics for flow anomalies
- Cross-language calls
- Consistent, predictable optimization

Brief History

- **1952:** First compiler (linker/loader) written by Grace Hopper for **A-0** programming language

Brief History

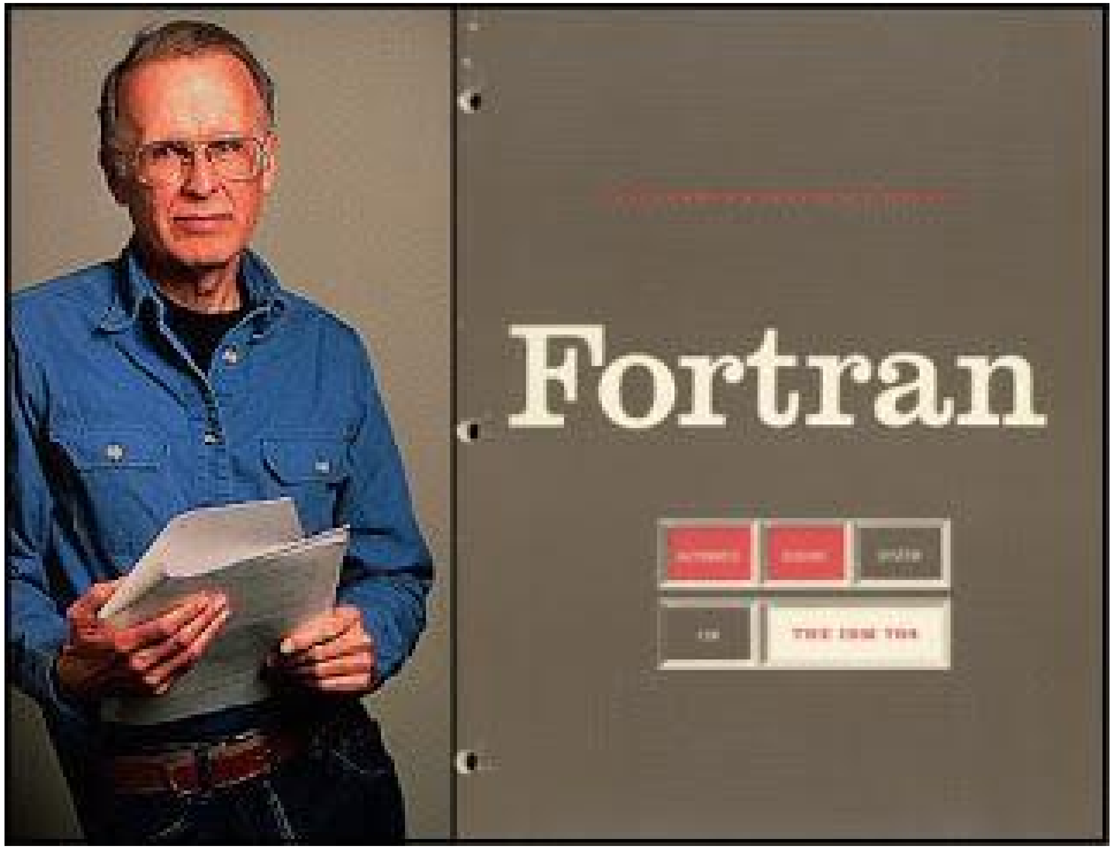
- **1952:** First compiler (linker/loader) written by Grace Hopper for **A-0** programming language



Grace Brewster Murray Hopper (December 9, 1906 – January 1, 1992), née **Grace Brewster Murray**, was an American computer scientist and United States Navy Rear Admiral. She was one of the first programmers of the **Harvard Mark I** computer in 1944, invented the first **compiler** for a computer programming language

Brief History

- **1957:** First complete compiler for **FORTRAN** by John Backus and team



Brief History

- **1952:** First compiler (linker/loader) written by Grace Hopper for **A-0** programming language

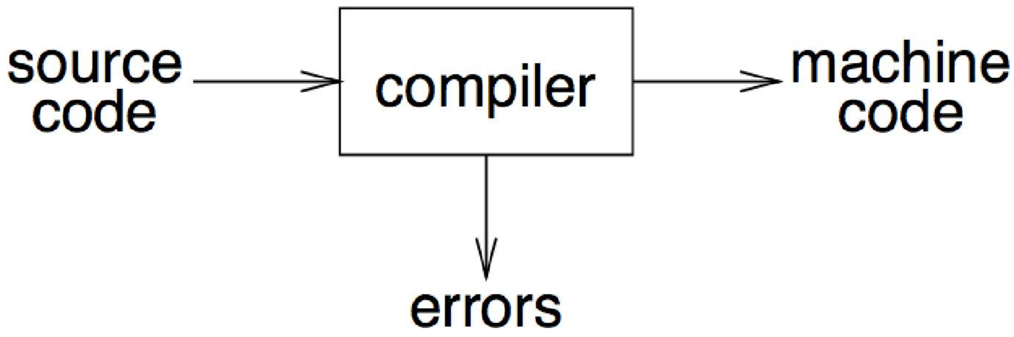
- **1957:** First complete compiler for **FORTRAN** by John Backus and team

- **1960: COBOL** compilers for multiple architectures

- **1962:** First self-hosting compiler for **LISP**

Abstract View

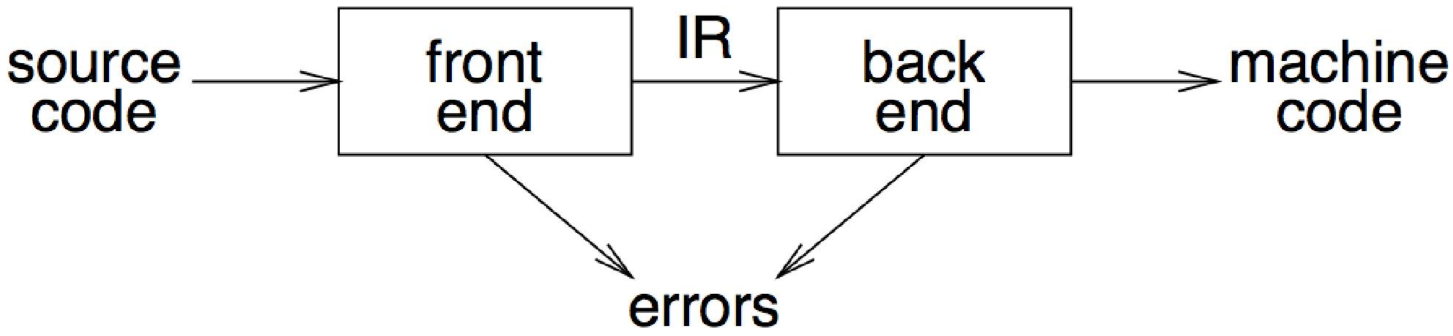
- Recognize legal (and illegal) programs
- Generate correct code
- Manage storage of all variables and code
- Agree on format for object (or assembly) code



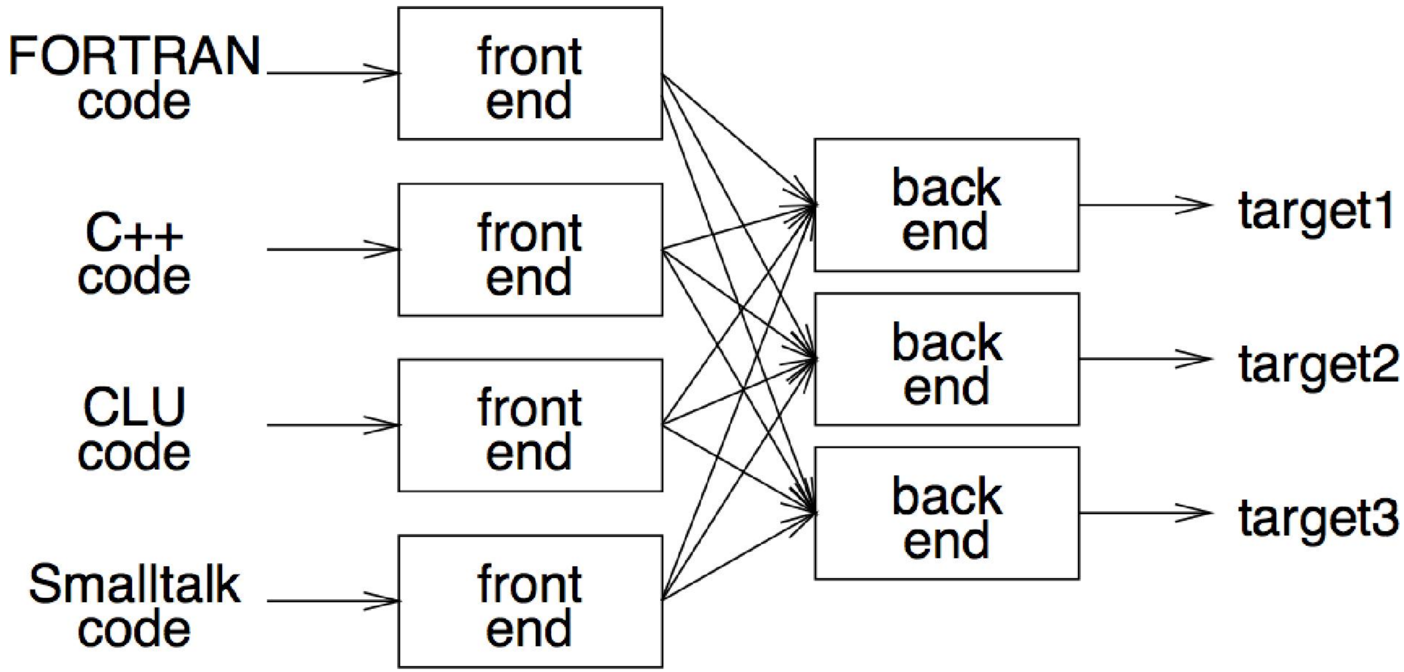
Big step up from assembler — higher level notations

Traditional Two Pass Compiler

- Intermediate Representation (IR)
- Front-end maps legal code into IR
- Back-end maps IR onto target machine
- Simplify retargeting
- Allows multiple front-ends
- Multiple passes $\Rightarrow$ better code



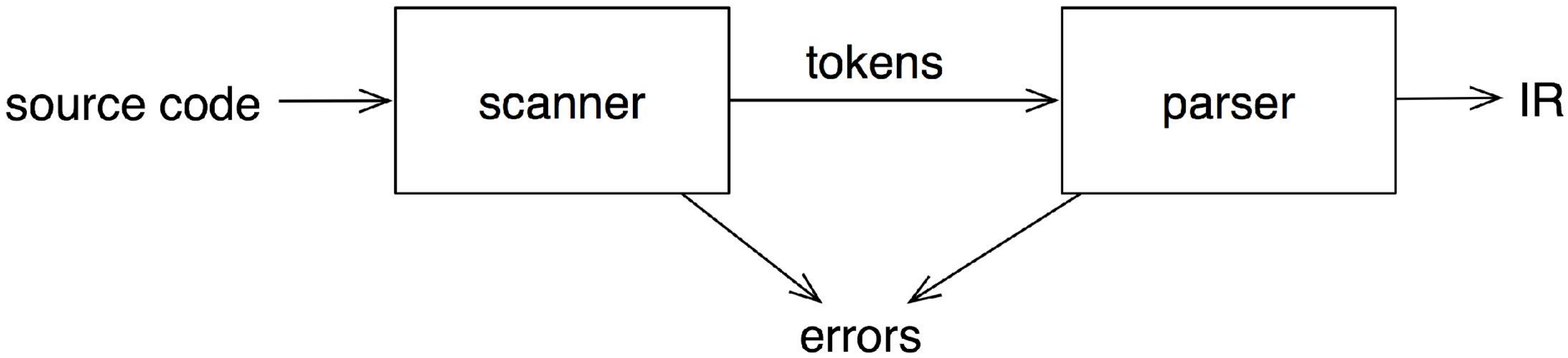
Traditional Two Pass Compiler



Front-end, IR and back-end must encode knowledge needed for all n×m combinations!

Front-end

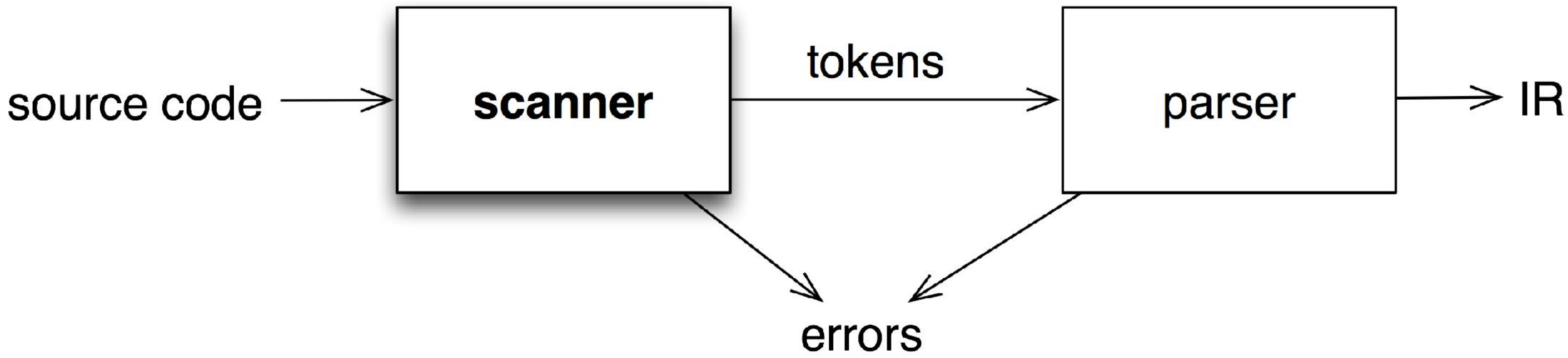
- Recognize legal code
- Report errors
- Produce IR
- Preliminary storage map
- Shape code for the back end



Much of front end construction can be automated

Scanner

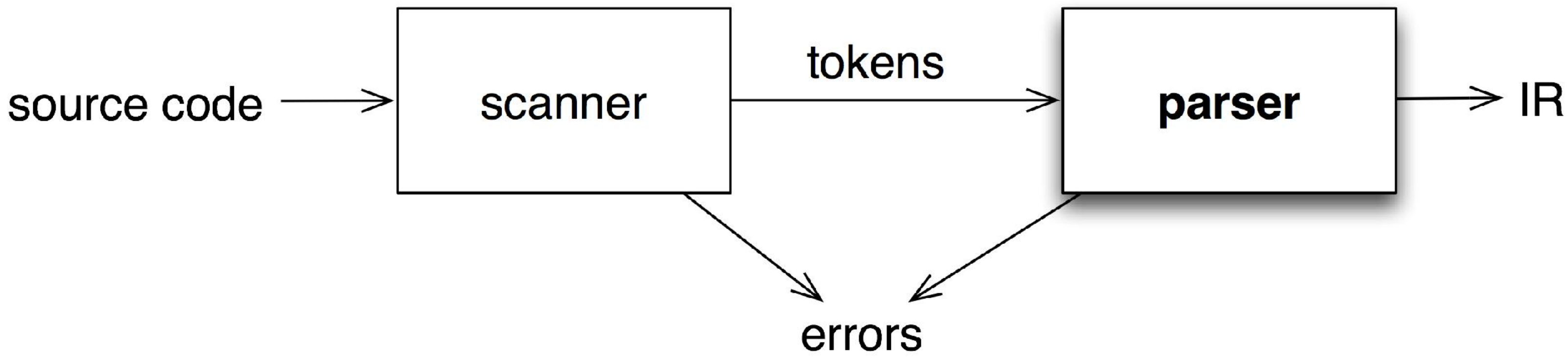
- Map characters to **tokens**
- Character string value for a token is a **lexeme**
- Eliminate white space



X= X + Y → `<id,x> = <id,x> + <id,y>`

Parser

- Recognize context-free syntax
- Guide context-sensitive analysis
- Construct IR(s)
- Produce meaningful error messages
- Attempt error correction



Parser generators mechanize much of the work

Context-Free Grammars

- Context-free syntax is specified with a grammar, usually in Backus-Naur form (BNF)

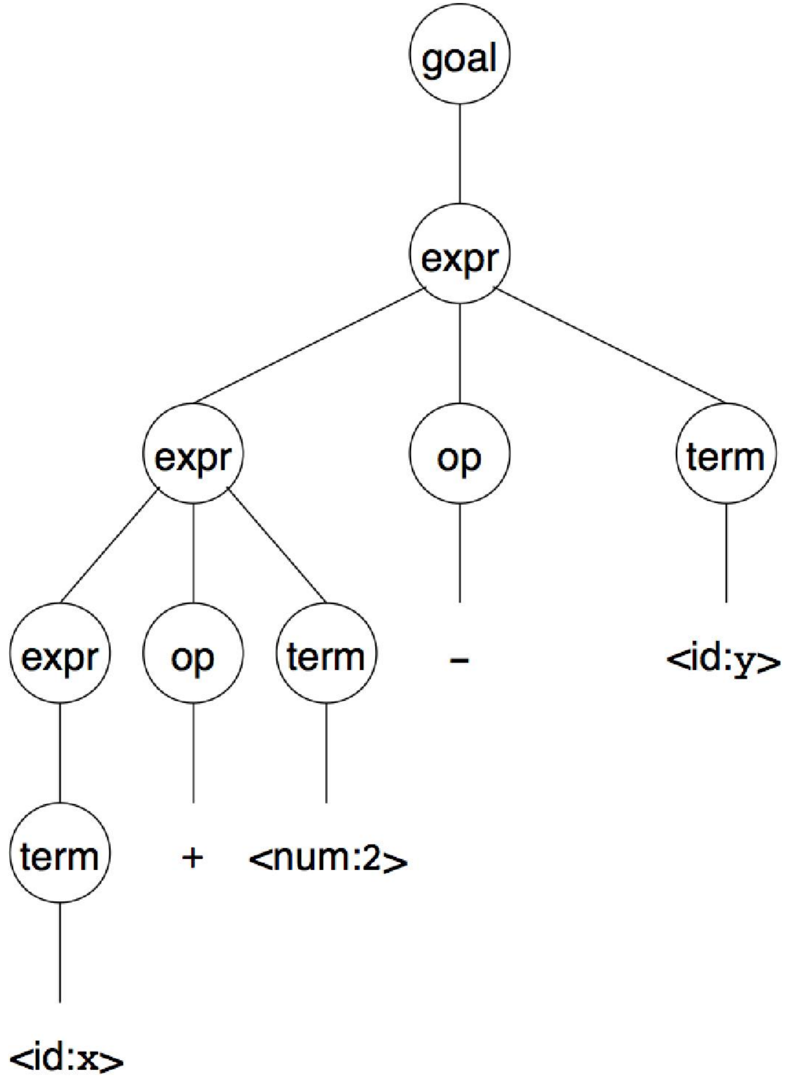
```
1.  <goal>    :=  <expr>
2.  <expr>    :=  <expr> <op> <term>
3.            |   <term>
4.  <term>    :=  number
5.            |   id
6.  <op>  :=  +
7.            |   -
```

A grammar G = (S,N,T,P)
S is the start-symbol
N is a set of non-terminal symbols
T is a set of terminal symbols
P is a set of productions — P: $N \rightarrow (N \cup T)^*$

Parse Trees

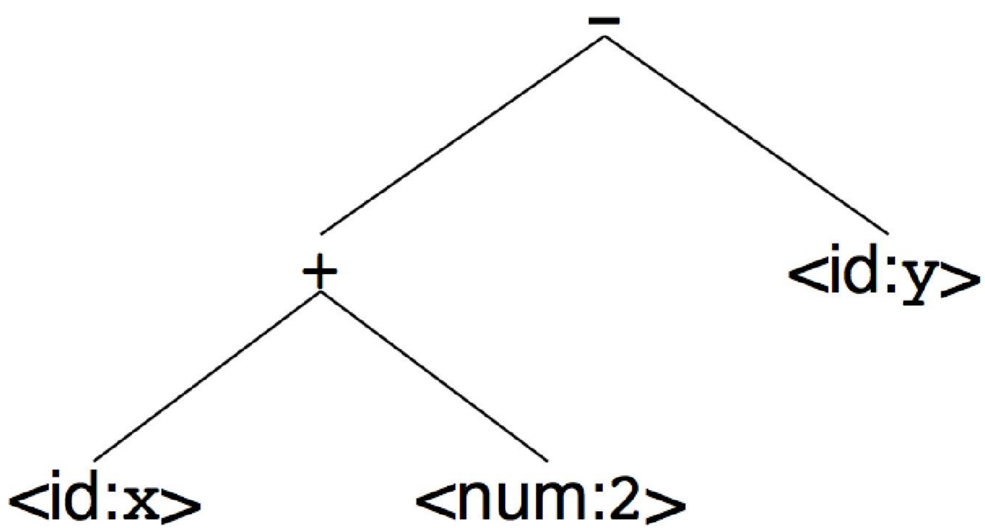
- A parse can be represented by a tree called a *parse* or *syntax* tree

Obviously, this contains a lot of unnecessary information



Abstract Syntax Trees

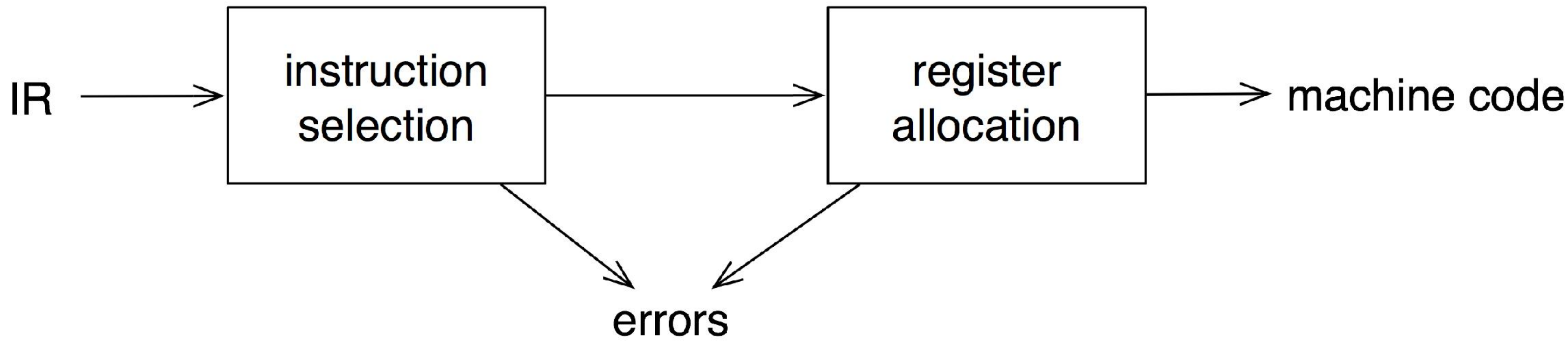
- So, compilers often use an *abstract syntax tree* (AST)



ASTs are often used as an IR

Back-end

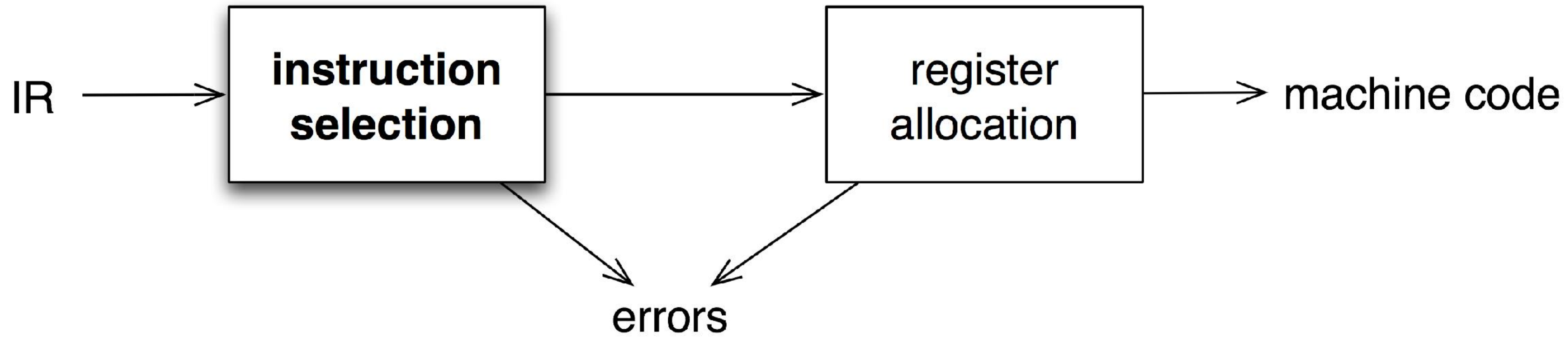
- Translate IR into target machine code
- Choose instructions for each IR operation
- Decide what to keep in registers at each point
- Ensure conformance with system interfaces



Automation has been less successful here

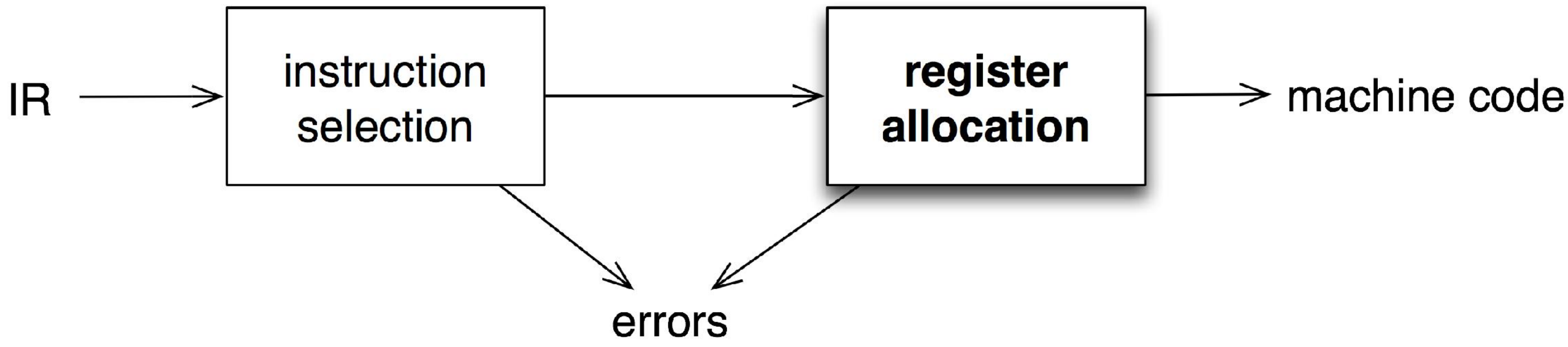
Instruction Selection

- Produce compact, fast code
- Use available addressing modes
- Pattern matching problem
  - Ad hoc techniques
  - Tree pattern matching
  - String pattern matching
  - Dynamic programming



Register Allocation

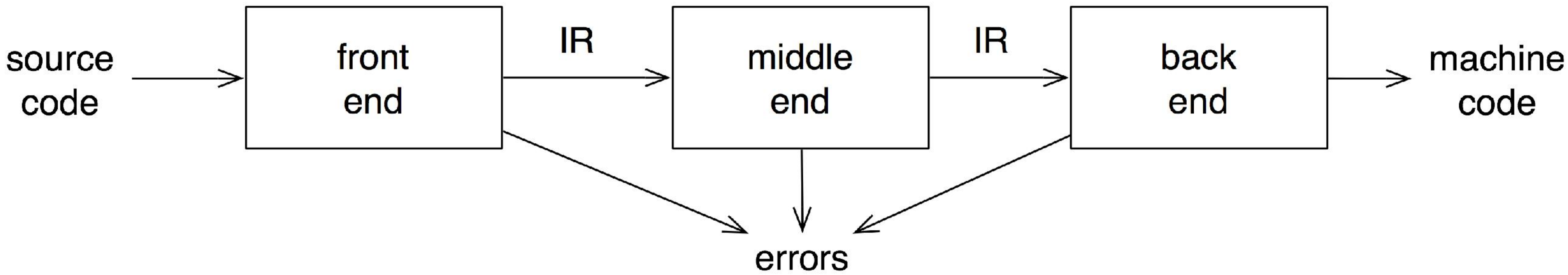
- Have value in a register when used
- Limited resources
- Changes instruction choices
- Can move loads and stores
- Optimal allocation is difficult



Modern allocators often use an analogy to graph coloring

Traditional Three-Pass Compiler

- Analyzes and changes IR
- Goal is to reduce runtime
- Must preserve values



Simple One-Pass Compiler

Introduction

- One-Pass compiler
  - A compiler that passes through the parts of each compilation unit only once
  - Immediately translating each part into its final machine code
  - E.g. Pascal

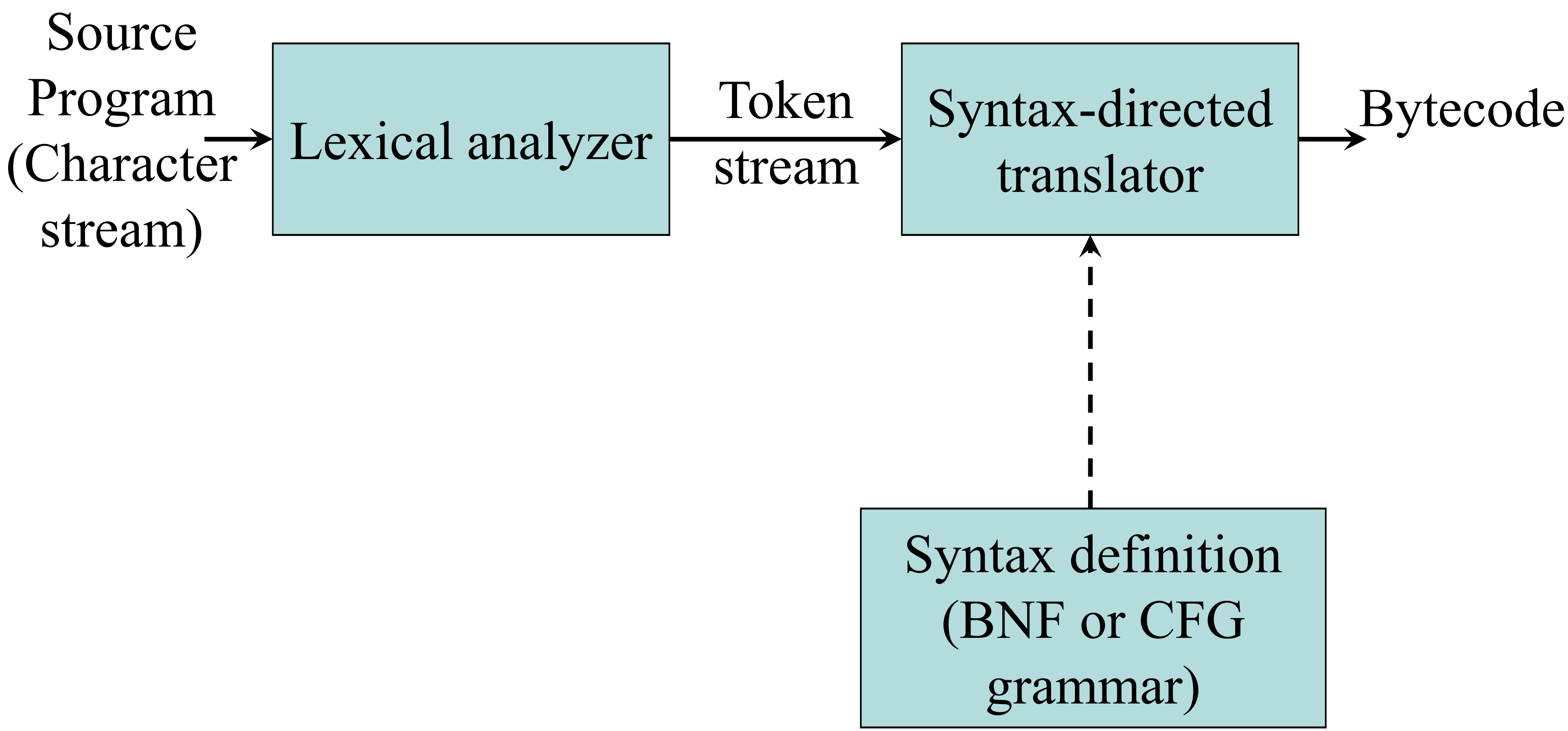
- Multi-Pass compiler
  - Converts the program into one or more intermediate representations in steps between source code and machine code
  - Reprocesses the entire compilation unit in each sequential pass

Introduction

- Advantages
  - One-pass compilers are
    - Smaller and
    - Faster than multi-pass compilers

- Disadvantages
  - One-pass compilers are unable to generate as efficient programs as multi-pass compilers due to the limited scope of available information

One-Pass Compiler Structure



One-Pass Compiler Structure

- Building our compiler involves
  - Defining the *syntax* of a programming language
  - Develop a source code parser
  - Implementing *syntax directed translation* to generate intermediate code
  - Generating required B*ytecode*
  - Optimize the generated Bytecode

- Language Definition
  - Appearance of programming language
    - Vocabulary : Regular expression
    - Syntax : Backus-Naur Form (BNF) or Context Free Form (CFG)

Backus-Naur Form (BNF)

- BNF (Backus Normal Form or Backus–Naur Form)
  - Notation techniques for context-free grammars, often used to describe the **syntax** of **languages** used in computing, such as computer programming languages, document formats, etc.

- A BNF specification is a set of derivation rules, written as
  - **<symbol> ::= __expression__**

- Example
  - As an example, consider this possible BNF for a **U.S. postal address**
    - <postal-address> ::= <name-part> <street-address> <zip-part>

<postal-address> ::= <name-part> <street-address> <zip-part>

Backus-Naur Form (BNF)

- BNF (Backus Normal Form or Backus–Naur Form)
  - Notation techniques for context-free grammars, often used to describe the **syntax** of **languages** used in computing, such as

```
 <postal-address> ::= <name-part> <street-address> <zip-part>

      <name-part> ::= <personal-part> <last-name> <opt-suffix-part> <EOL>
                    | <personal-part> <name-part>

  <personal-part> ::= <initial> "." | <first-name>

 <street-address> ::= <house-num> <street-name> <opt-apt-num> <EOL>

       <zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>

<opt-suffix-part> ::= "Sr." | "Jr." | <roman-numeral> | ""
    <opt-apt-num> ::= <apt-num> | ""
```

Grammars for Syntax Definition

- A **Context-free Grammar** (**CFG**) is utilized to describe the syntactic structure of a language
- A CFG Is characterized by
  - A set of tokens or terminal symbols
  - A set of non-terminals
  - A set of production rules, each rule has the form
    - $NT \rightarrow \{T, NT\}^*$
  - A non-terminal designated as the start symbol

  - Terminal symbols : bold face string **if**, **num**, **id**
  - Nonterminal symbol, grammar symbol: **italicized names**, **list**, **digit**, **A**, **B**

Grammars for Syntax Definition

- ***list* $\rightarrow$ *list* + *digit***
- ***list* $\rightarrow$ *list* - *digit***
- ***list* $\rightarrow$ *digit***
- ***digit* $\rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

- (the "|" means OR) → So we could have written
  - *list* $\rightarrow$ *list* + *digit* | *list* - *digit* | *digit*)

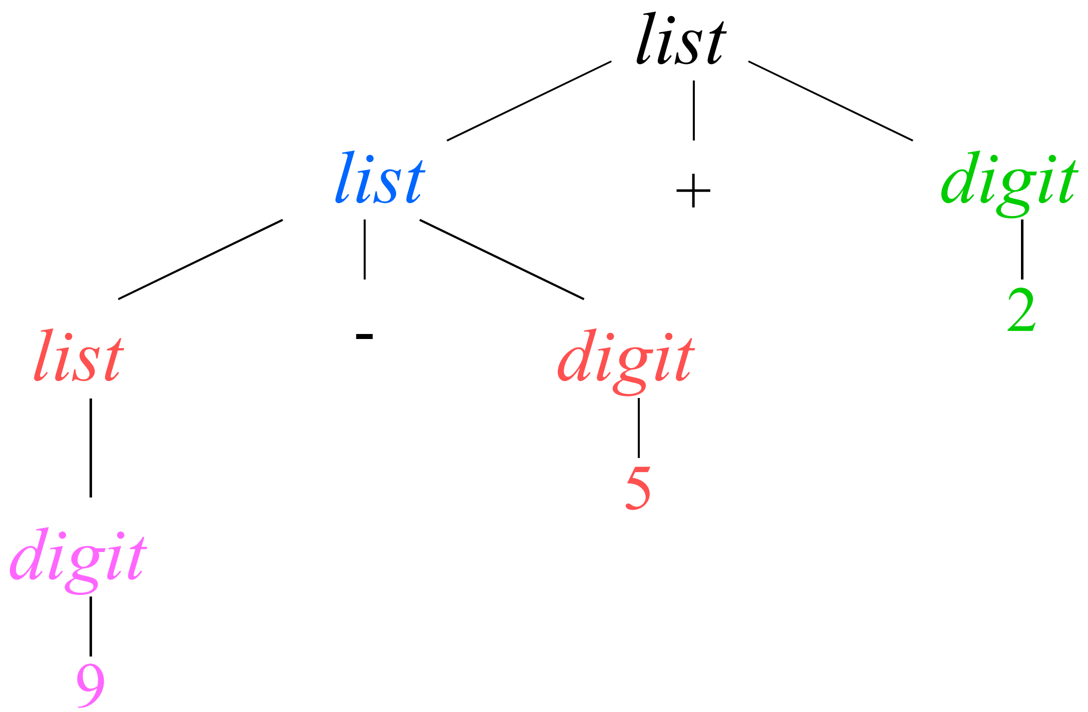
Grammars for Syntax Definition

- Using the CFG defined on the previous slide, we can derive the string:
  9 - 5 + 2   as follows

list → *list* + *digit*	P1 : *list* → *list* + *digit*
→ *list* - *digit* + *digit*	P2 : *list* → *list* - *digit*
→ *digit* - *digit* + *digit*	P3 : *list* → *digit*
→ 9 - *digit* + *digit*	P4 : *digit* → 9
→ 9 - 5 + *digit*	P4 : *digit* → 5
→ 9 - 5 + 2	P4 : *digit* → 2

Grammars for Syntax Definition

- This derivation could also be represented via a Parse Tree (parents on left, children on right)



Grammars for Syntax Definition

- A More Complex Grammar

$$block \rightarrow \underline{\text{begin}}\ \ opt\_stmts\ \underline{\text{end}}$$

$$opt\_stmts \rightarrow\ stmt\_list \mid\ \in$$

$$stmt\_list \rightarrow\ stmt\_list\ ;\ stmt \mid stmt$$

- What is this grammar for ?
- What does "∈" represent ?
- What kind of production rule is this ?

Syntax Definition

- To specify the syntax of a language : CFG and BNF

- An alphabet of a language is a set of symbols
  - Examples
    - {0,1} for a binary number
      - System(language)={0,1,100,101,...}

    - {a,b,c} for language={a,b,c, ac,abcc..}
    - {if,(,),else ...} for a if statements={if(a==1)goto10, if--}

- A string over an alphabet
  - is a sequence of zero or more symbols from the alphabet
  - Examples
    - 0,1,10,00,11,111,0202 ... strings for a alphabet {0,1}
  - Null string is a string which does not have any symbol of alphabet

Syntax Definition

- Language
  - Is a subset of all the strings over a given alphabet
  - Alphabets Ai Languages Li for Ai
  - A0={0,1} L0={0,1,100,101,...}
  - A1={a,b,c} L1={a,b,c, ac, abcc..}
  - A2={all of C tokens} L2= {all sentences of C program}

- Grammar G=(N,T,P,S)
  - N : a set of nonterminal symbols
  - T : a set of terminal symbols, tokens
  - P : a set of production rules
  - S : a start symbol, $S \in N$

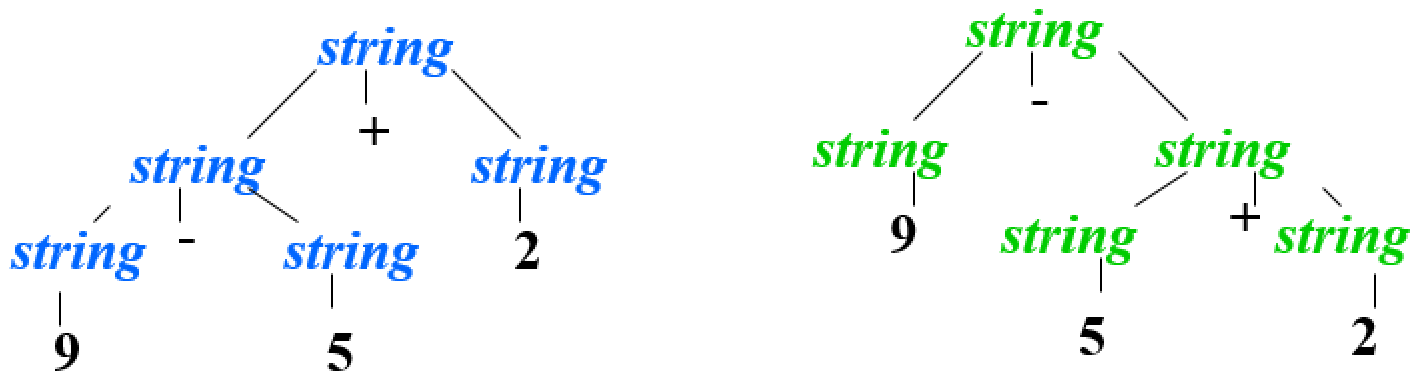
Syntax Definition

- Grammar G for a language L={9-5+2, 3-1, ...}
  - G=(N,T,P,S)
    - N={list,digit}
    - T={0,1,2,3,4,5,6,7,8,9,-,+}
      - P : list -> list + digit
      - list -> list - digit
      - list -> digit
      - digit -> 0|1|2|3|4|5|6|7|8|9
    - S=list

Parse Tree

- More formally, a parse tree for a CFG has the following Properties:
  - Root Is labeled with the Start Symbol
  - Leaf node is a token or ∈
  - Interior node (now leaf) is a Non-Terminal
  - If A → x1x2…xn, Then A Is an Interior; x1x2…xn Are Children of A and may be Non-Terminals or Tokens

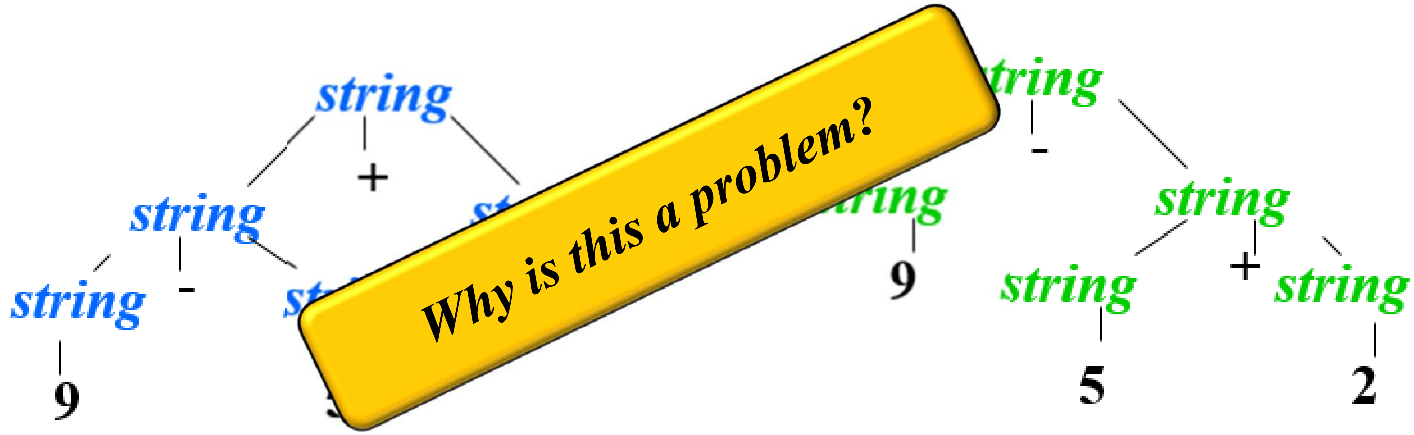
- Two derivations (Parse Trees) for the same token string



Parse Tree

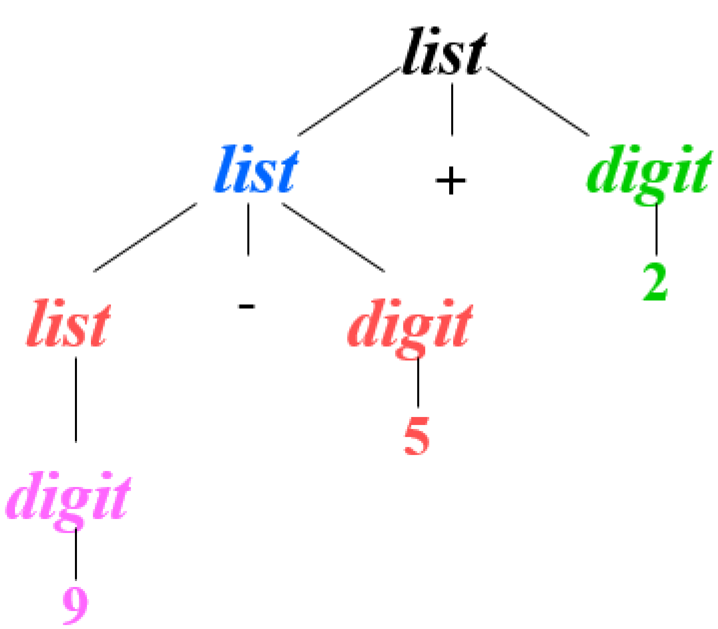
- More formally, a parse tree for a CFG has the following Properties:
  - Root Is labeled with the Start Symbol
  - Leaf node is a token or ∊
  - Interior node (now leaf) is a Non-Terminal
  - If A → x1x2…xn, Then A Is an Interior; x1x2…xn Are Children of A and May Be Non-Terminals or Tokens

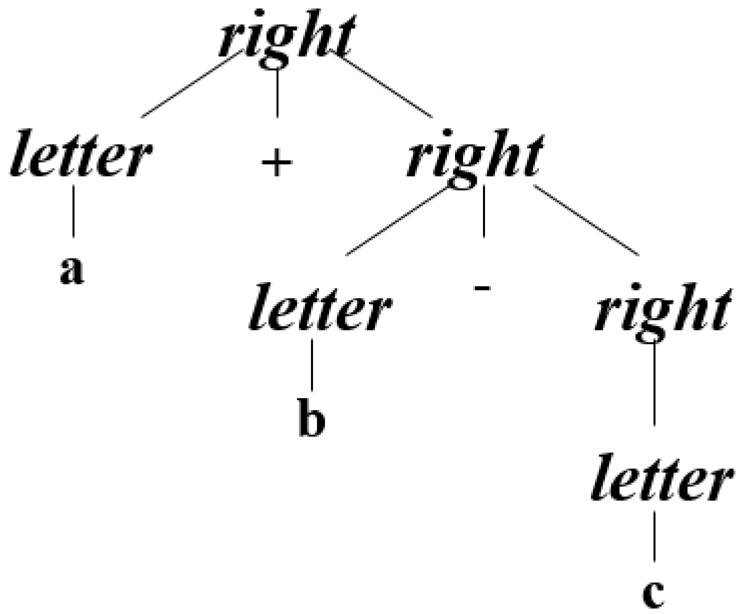
- Two derivations (Parse Trees) for the same token string



Parse Tree

- Other representation





$$right \rightarrow letter = right \mid letter$$

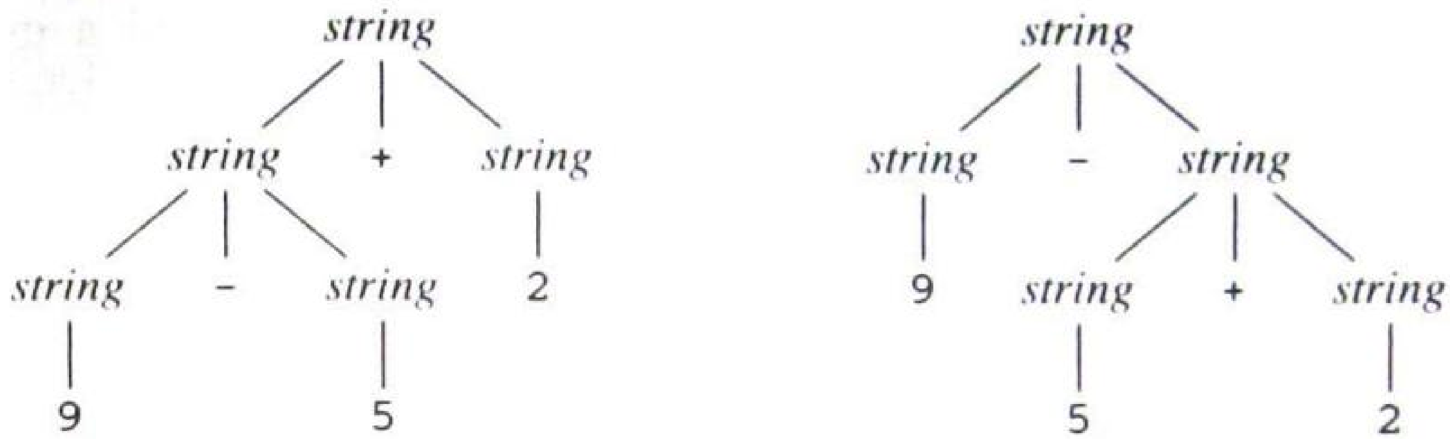
$$letter \rightarrow \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \mid \ldots \mid \mathbf{z}$$

Ambiguity

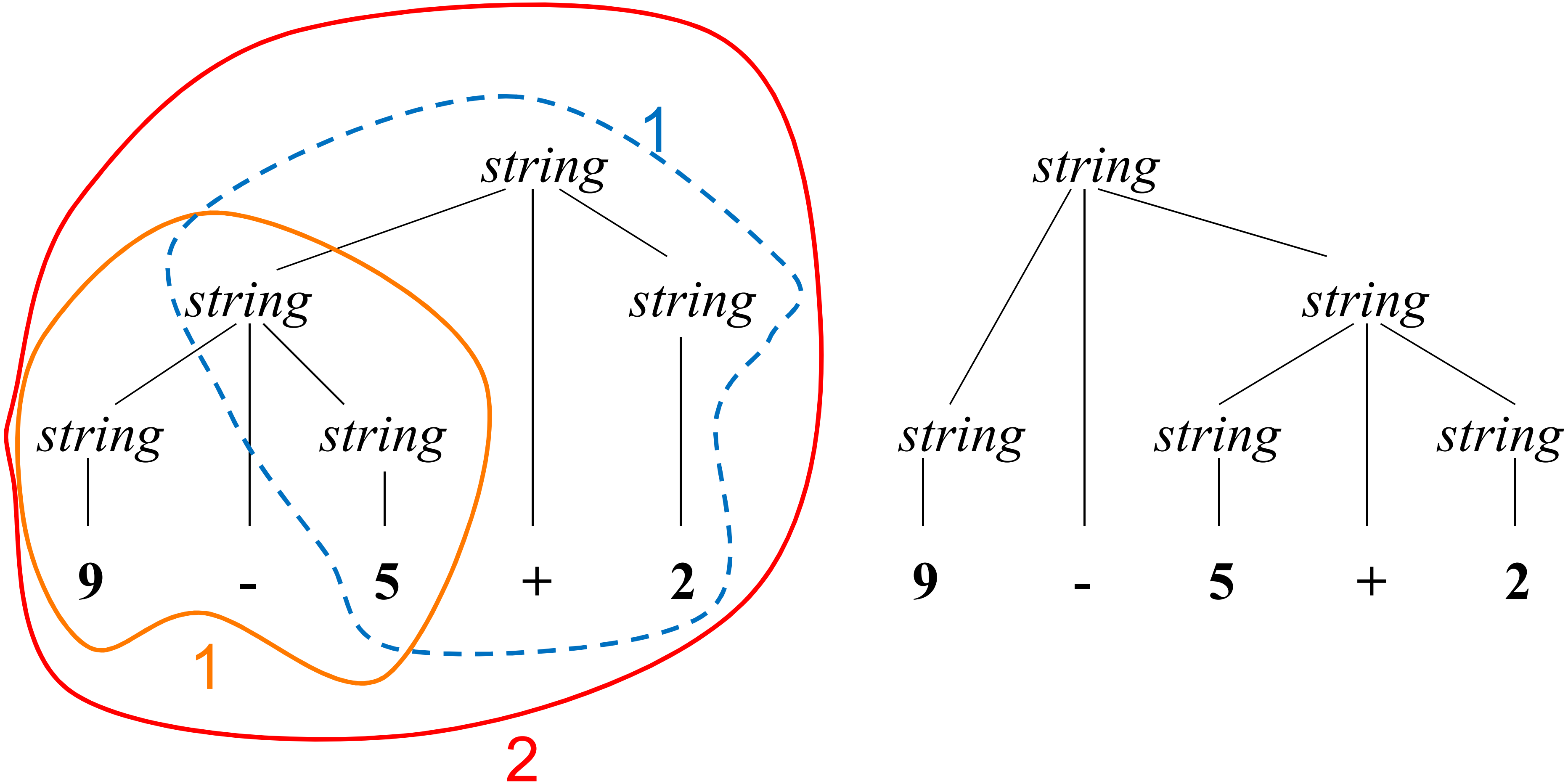
- A grammar is said to be ambiguous if the grammar has more than one parse tree for a given string of tokens

- Suppose a grammar G that can not distinguish between lists and digits as in the following example
  - G : string →string + string | string - string |0|1|2|3|4|5|6|7|8|9

- Assume 9-5+2



Ambiguity



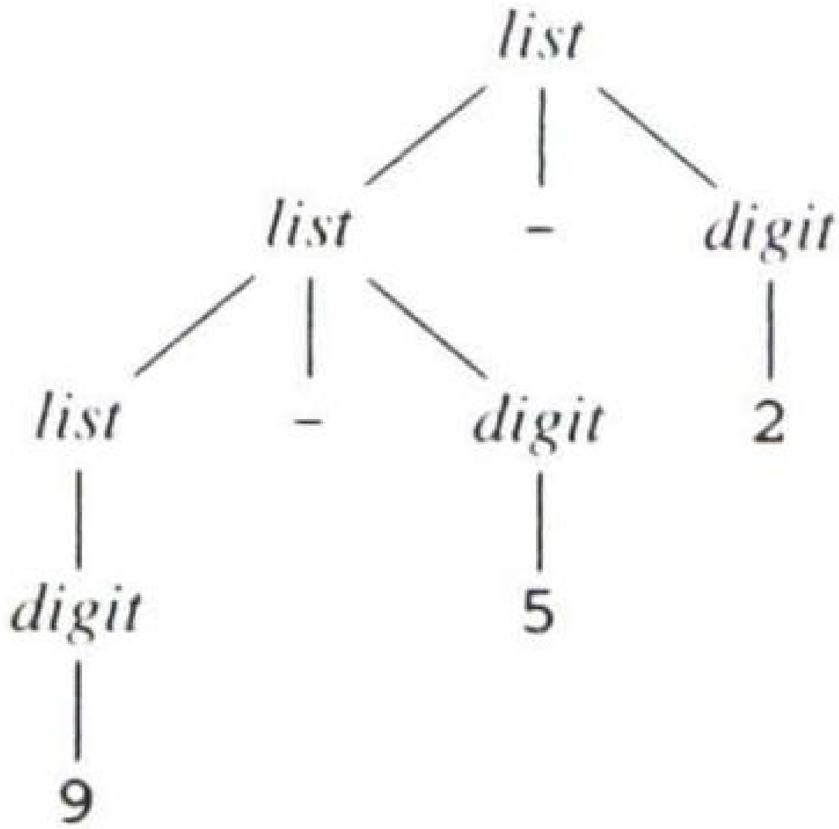
1
string
string
string
string
string
9
-
5
+
2
1
2
string
string
string
string
string
9
-
5
+
2

Associativity of Operator

- An operator is said to be left associative if an operand with operators on both sides of it is taken by the operator to its left
  - *Left-associative* operators have *left-recursive* productions
- Left Associative Grammar
  - list →list + digit | list - digit
  - digit →0|1|…|9
  - String **a+b+c** has the same meaning as **(a+b)+c**
- Right Associative Grammar
  - right →letter = right | letter
  - letter →a|b|…|z
  - String **a=b=c** has the same meaning as **a=(b=c)**

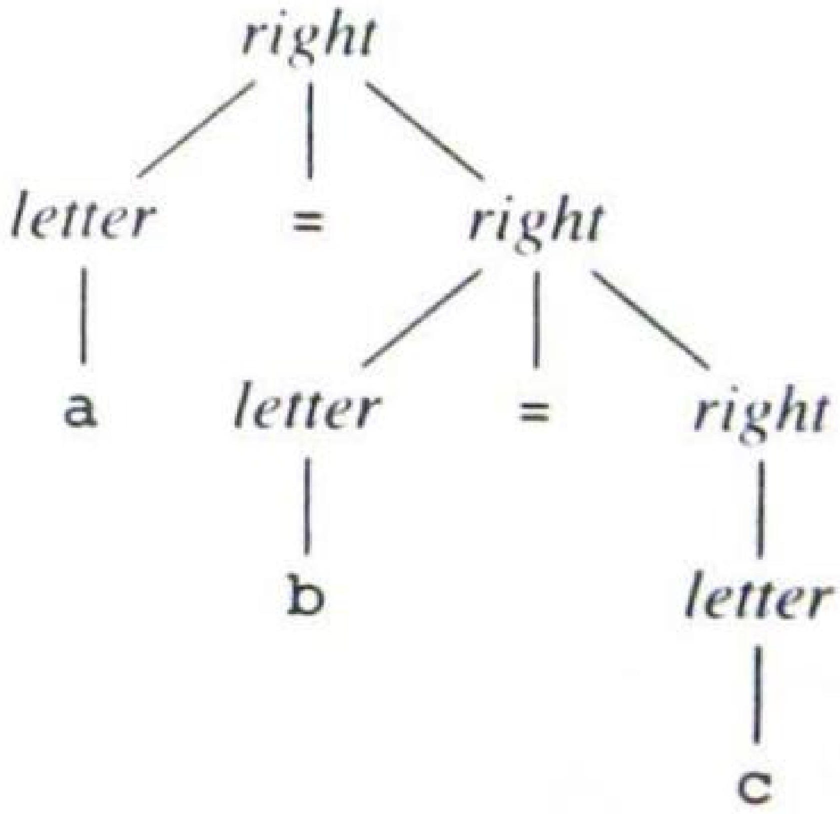
Associativity of Operator

- Left associative grammer



Associativity of Operator

- Right associative grammer



Precedence of Operators

- We say that an operator (*) has a higher precedence than other operator (+), if the operator (*) takes operands before other operator (+) does

- Left associative operators : + , - , * , /
- Right associative operators : = , **

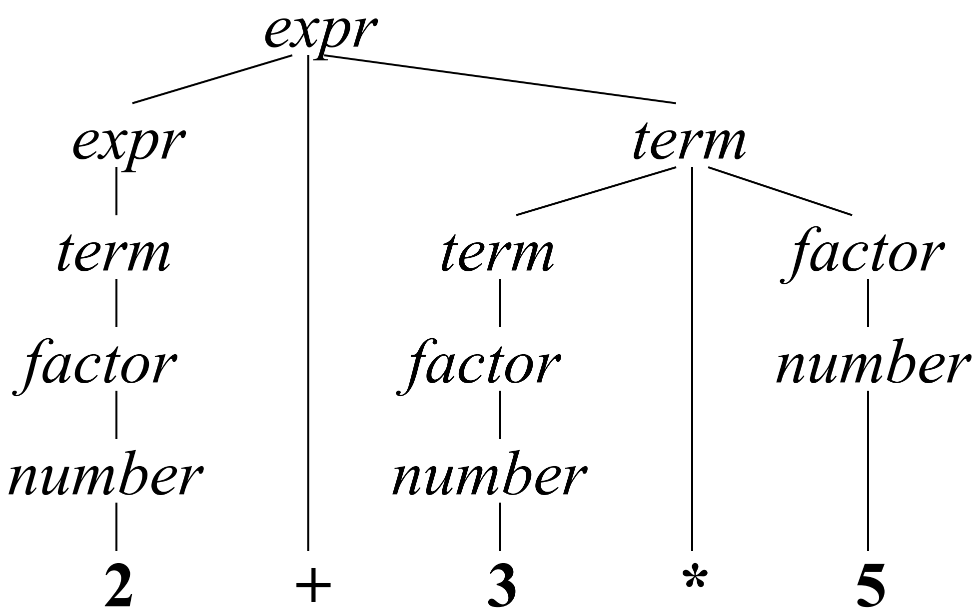
- Syntax of full expressions

Operator	Associative	Precedence
+ , -	Left	1 Low
* , /	Left	2 High

Precedence of Operators

- Syntax
  - expr →expr + term | expr - term | term
  - term →term * factor | term / factor | factor
  - factor →digit | (expr)
  - digit →0 | 1 | … | 9

- String **2+3*5** has the same meaning as **2+(3*5)**



Precedence of Operators

- Syntax of statements
  - stmt →id = expr ;
    - | if (expr) stmt ;
    - | if (expr) stmt else stmt ;
    - | while (expr) stmt ;
  - expr →expr + term | expr - term | term
  - term →term * factor | term / factor | factor
  - factor →digit | (expr)
  - digit →0 | 1 | … | 9

Syntax-Directed Translation (SDT)

1. Use a CF grammar to specify the syntactic structure of the language

2. Associate a set of *attributes* with the terminals and non-terminals of the grammar

3. Associate with each production a set of *semantic rules* to compute values of attributes

4. A parse tree is traversed and semantic rules applied
    1. After the tree traversal(s) are completed, the attribute values on the non-terminals contain the translated form of the input

Example Attribute Grammar

- Production
  - *expr* → *expr₁* **+** *term*
    expr → *expr₁* **-** *term*
    expr → *term*
    term → **0**
    term → **1**
    …
    term → **9**

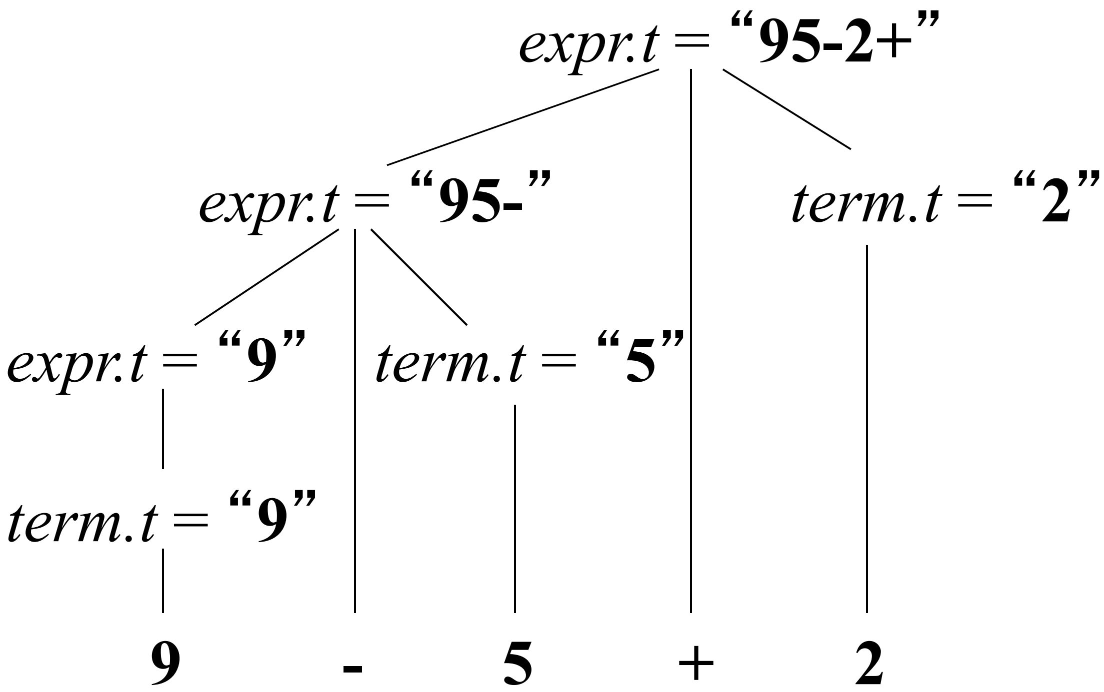
- Semantic rule
  - *expr.t* := *expr₁.t* // *term.t* // "+"
    expr.t := *expr₁.t* // *term.t* // "-"
    expr.t := *term.t*
    term.t := "0"
    term.t := "1"
    …
    term.t := "9"

Synthesized & Inherited Attributes

- **Synthesized** attribute
  - if its value at a parse-tree node is determined from the attribute values at the children of the node

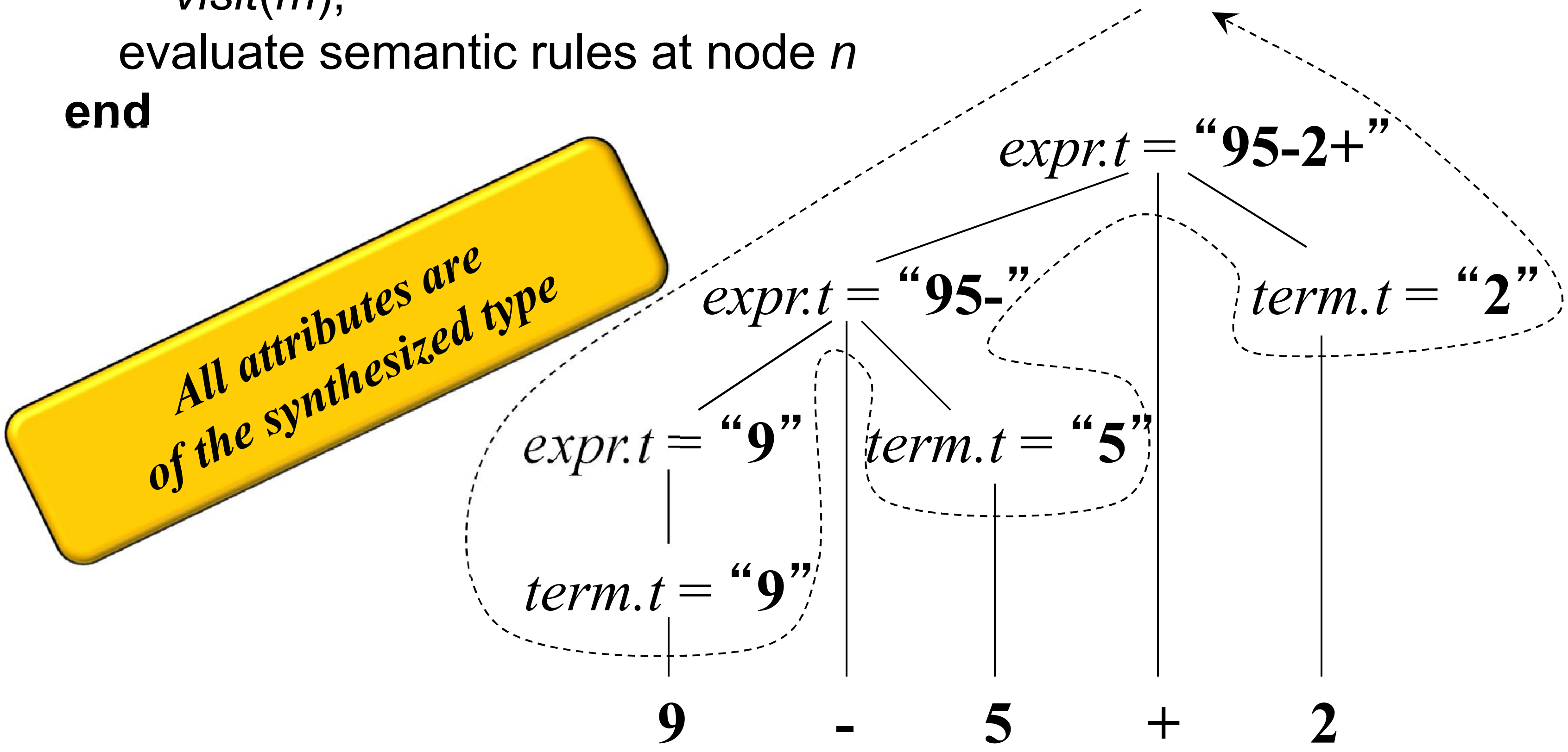
- **Inherited** attribute
  - if its value at a parse-tree node is determined by the parent (by enforcing the parent's semantic rules)

Example Annotated Parse Tree



Depth-First Traversals

- **procedure** *visit*(*n* : *node*);
  begin
      for each child *m* of *n*, from left to right **do**
          visit(*m*);
      evaluate semantic rules at node *n*
  end



Parsing

- Main rule
  - If token string x ∈ L(G), then
    - parse tree
    - else error message

- Top-Down parsing
  - At node n labeled with nonterminal A, select one of the productions whose left part is A and construct children of node n with the symbols on the right side of that production
  - Find the next node at which a sub-tree is to be constructed
  - E.g.
    - G: type →simple
    - |↑id
    - |array [simple] of type
    - simple →integer
    - |char
    - |num dotdot num

Parsing (Top-Down Parsing While Scanning the Input From Left to Right)

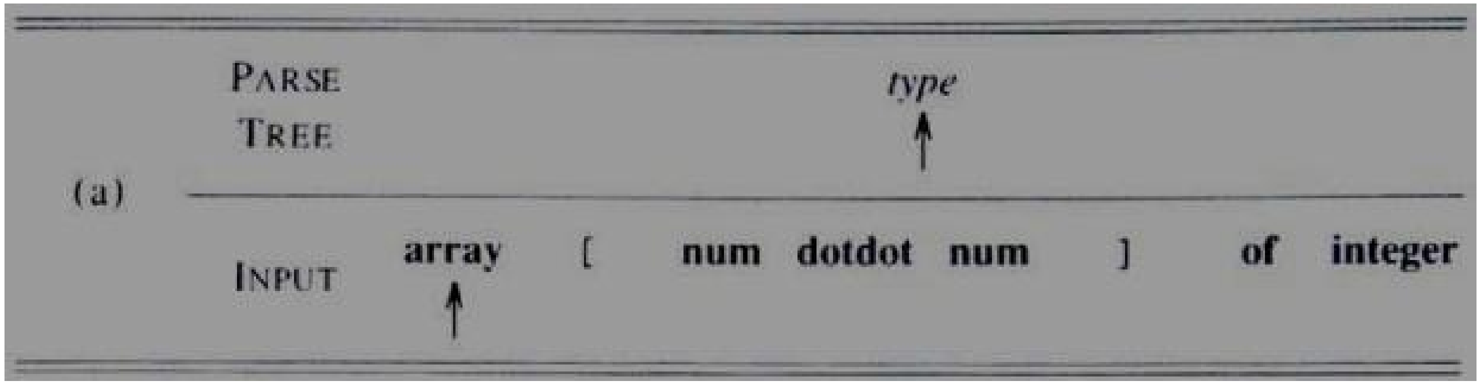
Grammar G :

type → *simple*
| ↑ id
| array [*simple*] of type

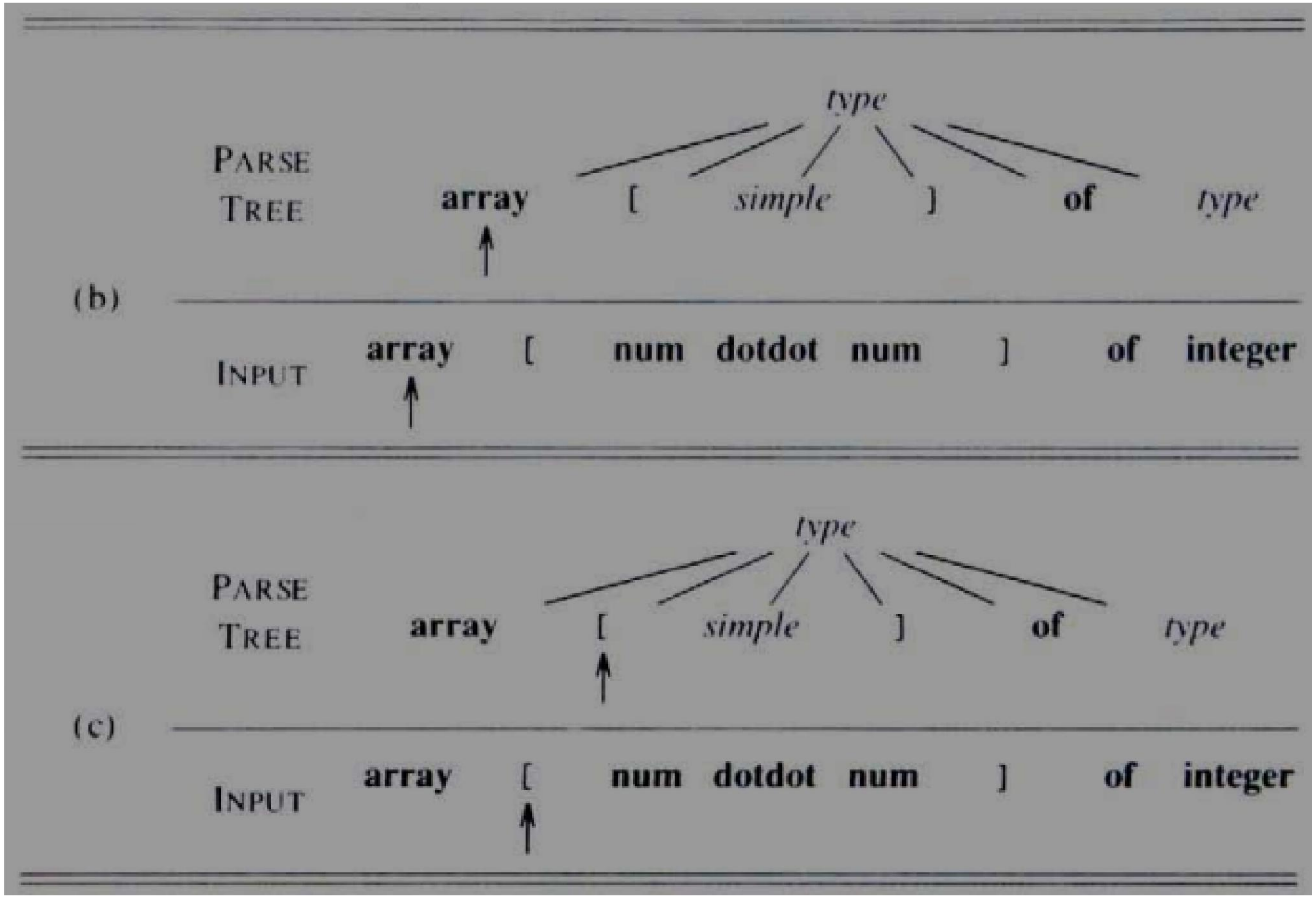
simple → integer
| char
| num dotdot num

Input string

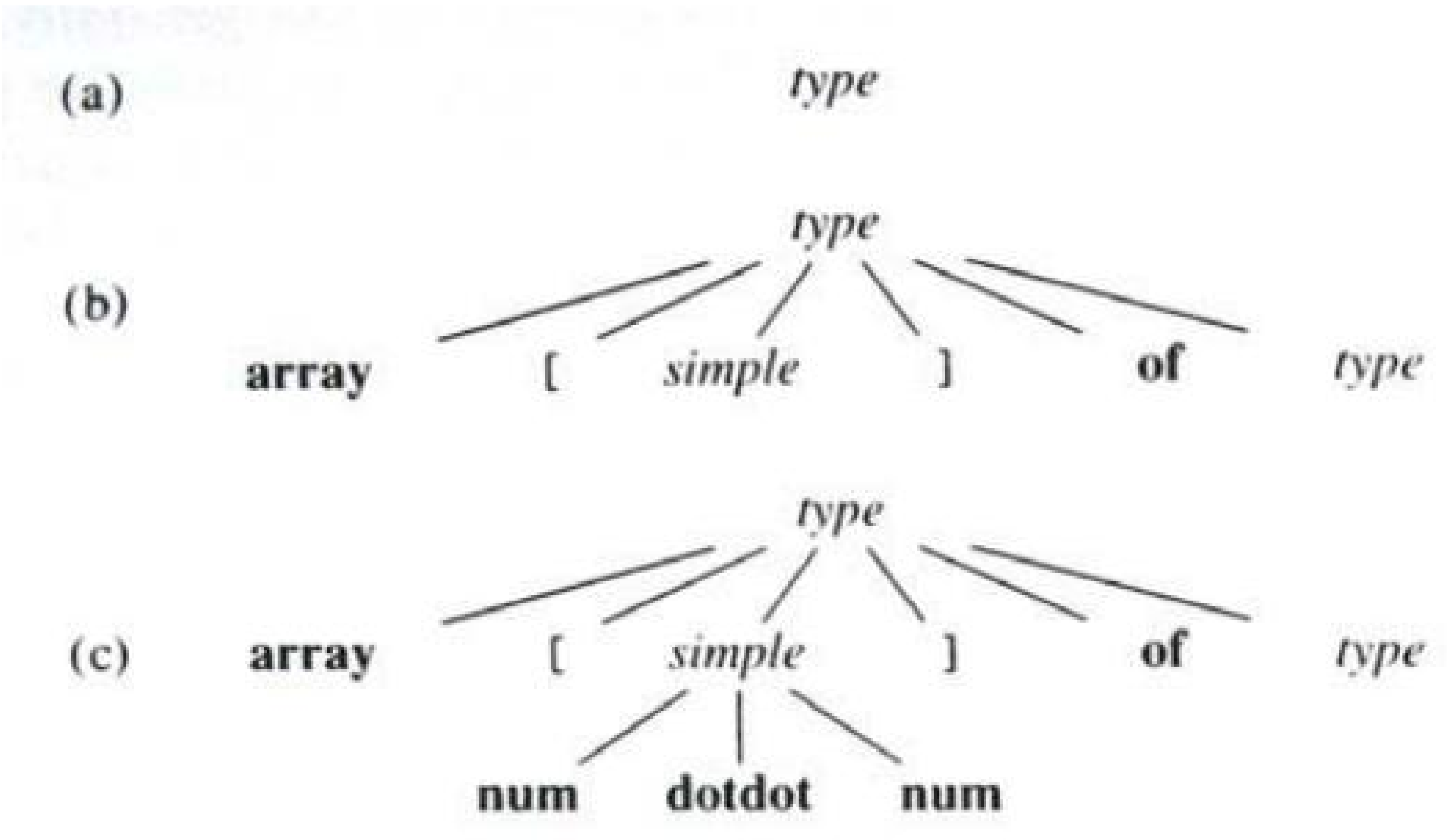
arrray [num dotdot num] of integer



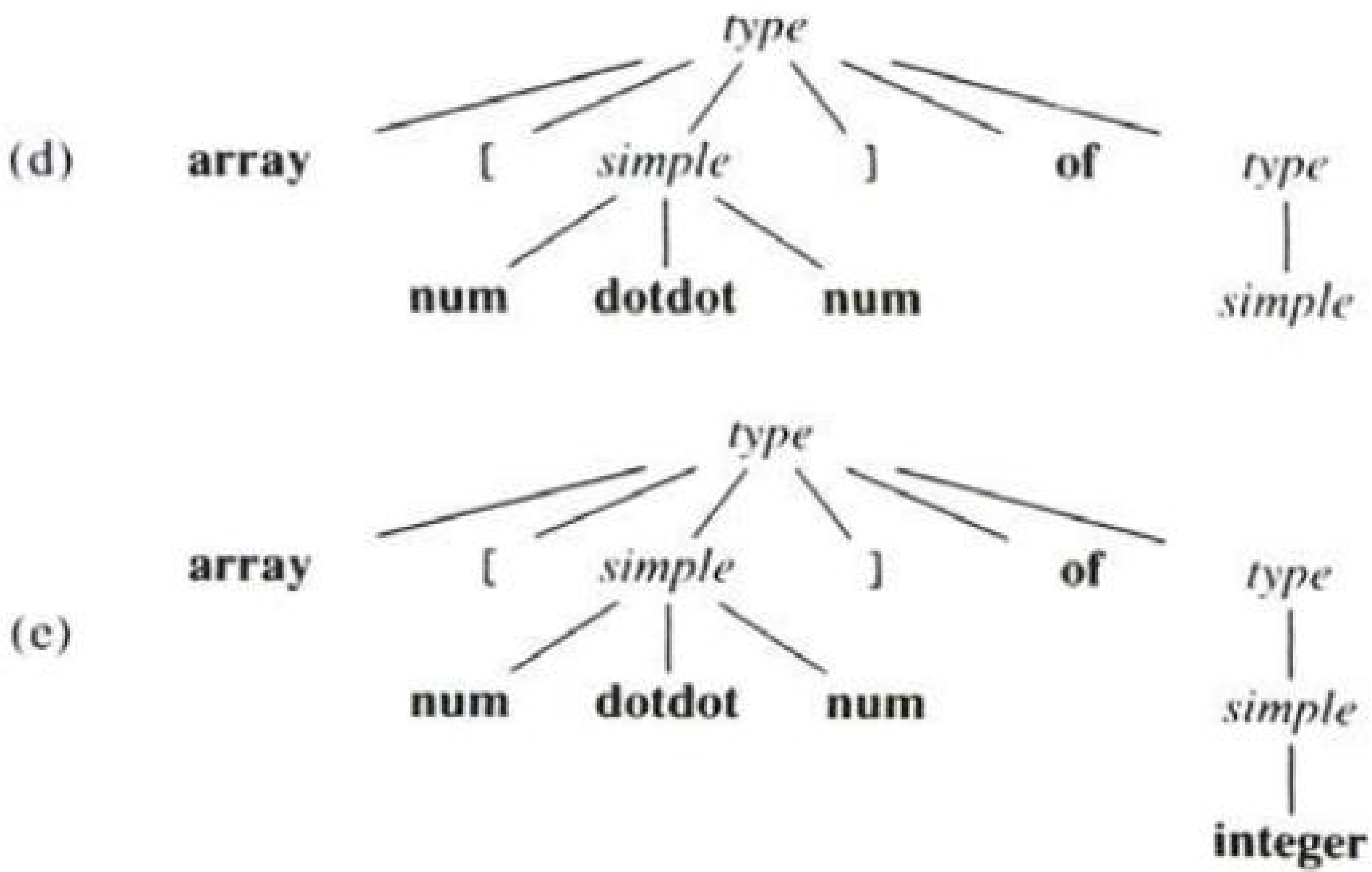
Parsing (Top-Down Parsing While Scanning the Input From Left to Right)



Parsing (Top-Down Parsing While Scanning the Input From Left to Right)

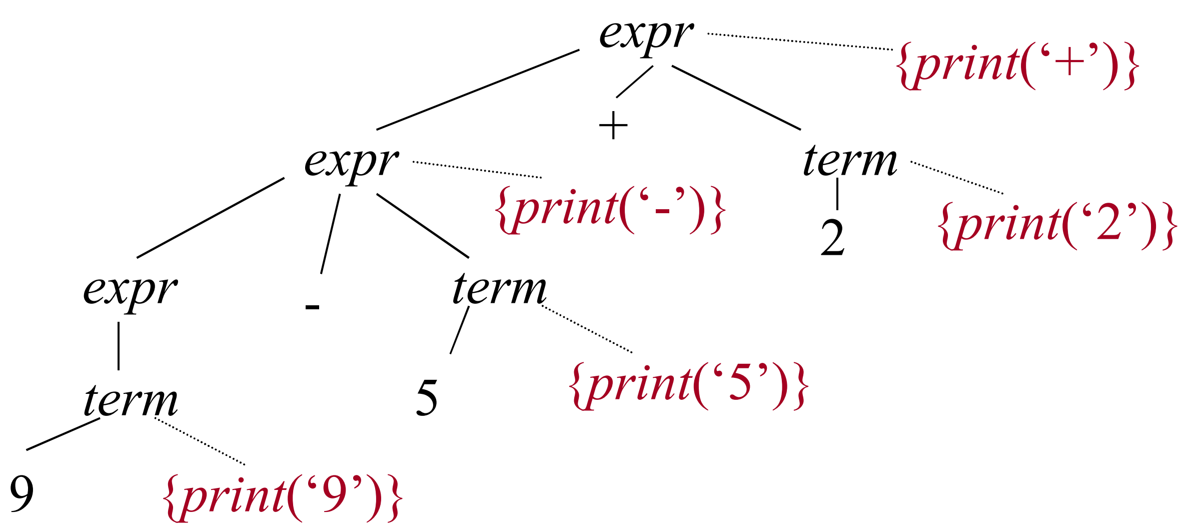


Parsing (Top-Down Parsing While Scanning the Input From Left to Right)



Comparing Grammars with Left Recursion

- Notice the location of semantic actions in the tree
- What is order of processing?



Comparing Grammars without Left Recursion

- Now, notice the location of semantic actions in tree for revised grammar

- What is the order of processing in this case?