CHAIR FOR COMPILER CONSTRUCTION

cfaed CENTER FOR ADVANCING ELECTRONICS DRESDEN

Compiler Construction

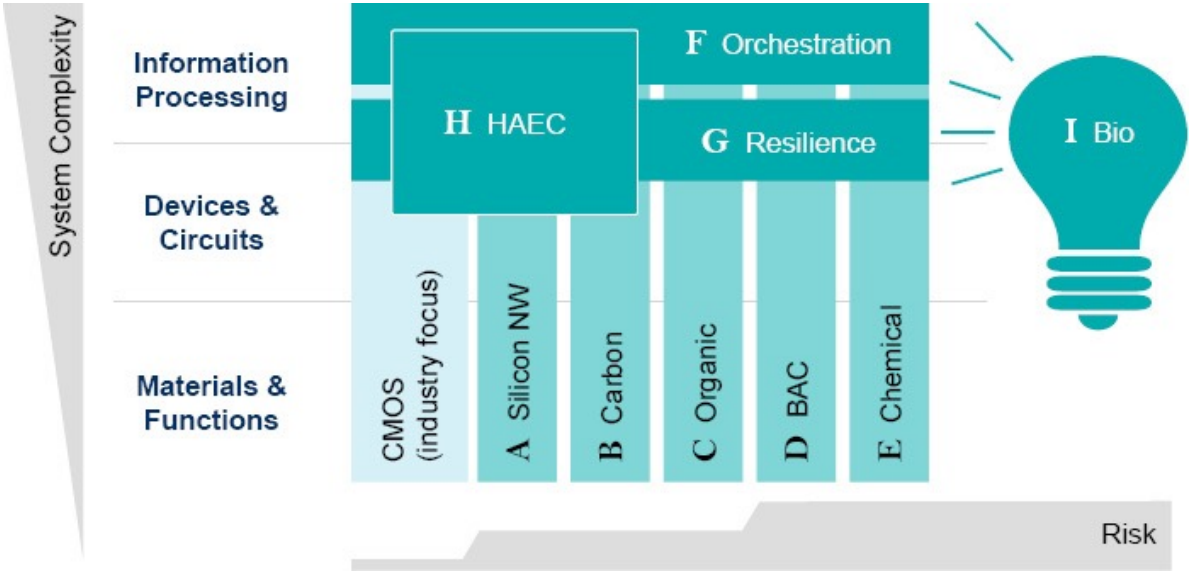
Prof. Dr.-Ing. Jeronimo Castrillon
TU Dresden – Cfaed

WS 2022

Chair for Compiler Construction
Helmholtzstrasse 18
jeronimo.castrillon@tu-dresden.de

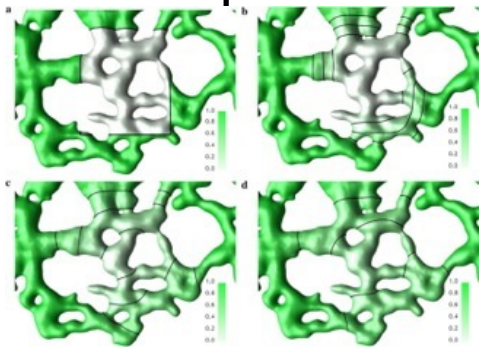
About the Chair for Compiler Construction (CCC)

- Cfaed: Center for Advancing Electronics Dresden
- CCC: Programming methods
  - Parallel heterogeneous systems
  - Post-CMOS systems
  - Domain-specific languages and optimization
  - Goals: Performance, energy efficiency, resilience

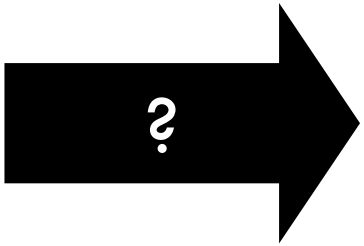


Domain-specific languages (DSLs)

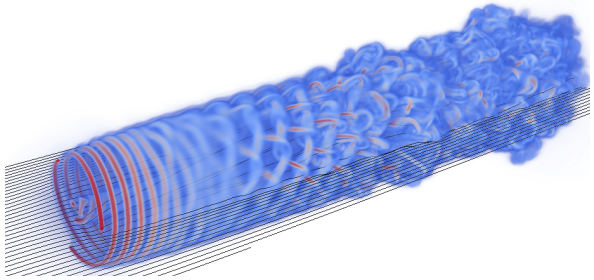
- Raise the level of abstraction for particular application domains to enable more powerful optimizations for large parallel systems



Sbalzarini, Ivo F. et al. 2006. „Simulations of (An)Isotropic Diffusion on Curved Biological Surfaces“.



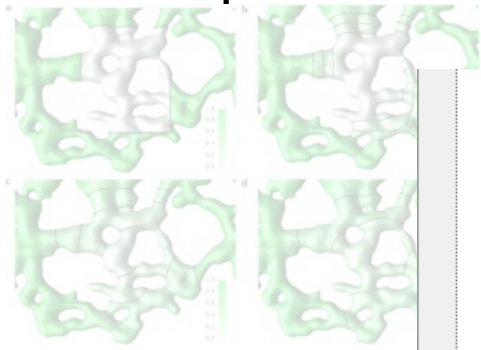
Bull HPC-Cluster „Taurus“, ZIH, TU Dresden



Chatelain et al. 2011. „Vortex particle-mesh methods with immersed lifting lines applied to the Large Eddy Simulation of wind turbine wakes“

Domain-specific languages (2)

- Raise the level of abstraction for particular application domain to enable more powerful optimizations for large parallel systems



Sbalzarini, Ivo F. et al. 2006. „Simulations o
(An)Isotropic Diffusion on Curved Biological

Chatelain et al. 2011. „Vortex particle-mesh methods with immersed lifting lines applied to the Large Eddy Simulation of wind turbine wakes“

```
external real Dv = 1.0;      "diffusion constant of V"

phase initialize
  << ... >>
end   initialize

phase solve
  field <real , 1> U;
  field <real , 1> V;
  integer t;

  δU/δt = Du * ∇² U - U * V²  + F * (1 - U);
  δV/δt = Dv * ∇² V + U * V²  - (F + kRate) * V;
end   solve

phase finalize
  << ... >>
end   finalize
```

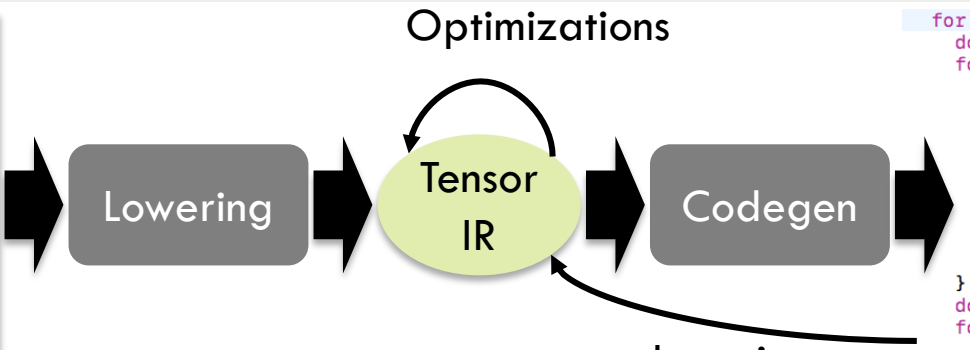
Contact: Nesrine Khouzami
nesrine.khouzami@tu-dresden.de

Bull HPC-Cluster „Taurus“, ZIH,
TU Dresden

Tensor languages

```
source =
type matrix    : [mp np]          &
type tensorIN  : [np np np ne]    &
type tensorOUT : [mp mp mp me]    &
                                  &
var input A    : matrix           &
var input u    : tensorIN         &
var input output v  : tensorOUT   &
var input alpha : []              &
var input beta  : []              &
                                  &
v = alpha * (A # A # A # u .
    [[5 8] [3 7] [1 6]]) + beta * v
```

Fortran embedding



Linkable C code

```
for (unsigned i0 = 0; i0 < 1000; i0++) {
  double t6[18];
  for (unsigned i3 = 0; i3 < 3; i3++) {
    for (unsigned i2 = 0; i2 < 3; i2++) {
      for (unsigned i1 = 0; i1 < 2; i1++)
        t6[(i1 + 2*(i2 + 3*(i3)))] = 0.0;
      for (unsigned i4_contr = 0; i4_contr < 3; i4_contr++) {
        t6[(i1 + 2*(i2 + 3*(i3)))] += A[(i1 + 2*(i4_contr))]
            * u[(i2 + 3*(i3 + 3*(i4_contr + 3*(i0))))];
      }
    }
  }
}
double t7[12];
for (unsigned i7 = 0; i7 < 3; i7++) {
  for (unsigned i6 = 0; i6 < 2; i6++) {
    for (unsigned i5 = 0; i5 < 2; i5++) {
      t7[(i5 + 2*(i6 + 2*(i7)))] = 0.0;
      for (unsigned i8_contr = 0; i8_contr < 3; i8_contr++) {
        t7[(i5 + 2*(i6 + 2*(i7)))] += A[(i5 + 2*(i8_contr))]
            * t6[(i6 + 2*(i7 + 3*(i8_contr)))];
      }
    }
  }
}
double t8[1];
double t9[1];
for (unsigned i11 = 0; i11 < 2; i11++) {
  for (unsigned i10 = 0; i10 < 2; i10++) {
    for (unsigned i9 = 0; i9 < 2; i9++) {
      t9[0] = 0.0;
      for (unsigned i12_contr = 0; i12_contr < 3; i12_contr++
          ) {
        t9[0] += A[(i9 + 2*(i12_contr))] * t7[(i10 + 2*(i11 +
            2*(i12_contr)))];
      }
      t8[0] = alpha[0] * t9[0];
      double t10[1];
      t10[0] = beta[0] * v[(i9 + 2*(i10 + 2*(i11 + 2*(i0))))]
          ;
      v[(i9 + 2*(i10 + 2*(i11 + 2*(i0))))] = t8[0] + t10[0];
    }
  }
}
```

```
auto A = Matrix("A", m, n);
auto u = Tensor<3>("u", n, n, n);

auto Interpolation = (A*A*A)(u);

const CFDlang::Program *Prog =
  Emitter.lowerExpr(Interpolation, "v");
```

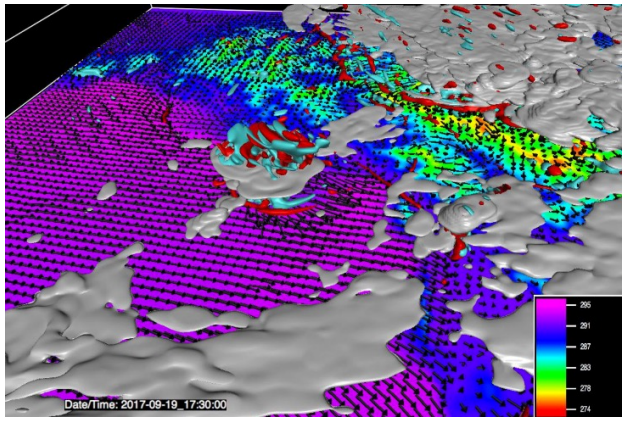
C++ embedding (with templates)

- ❑ Language/IR design and implementation
- ❑ C++/Haskell
- ❑ Tensorflow
- ❑ Parallelization

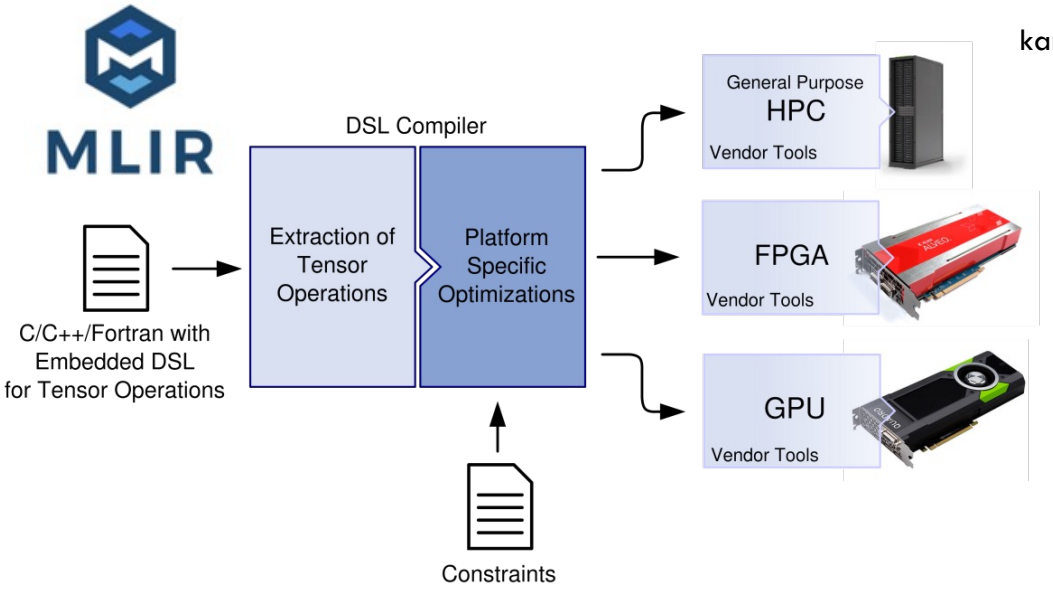
Weather modeling: DSLs and synthesis flows

- Stencil and Tensor Operations in Weather Modelling
- Optimizations for computations in WRF (CPU, GPU, FPGA)
- Integration of stencils on top of MLIR dialects

Contact: Karl Friebel
karl.friebel@tu-dresden.de



Source: CIMA Foundation

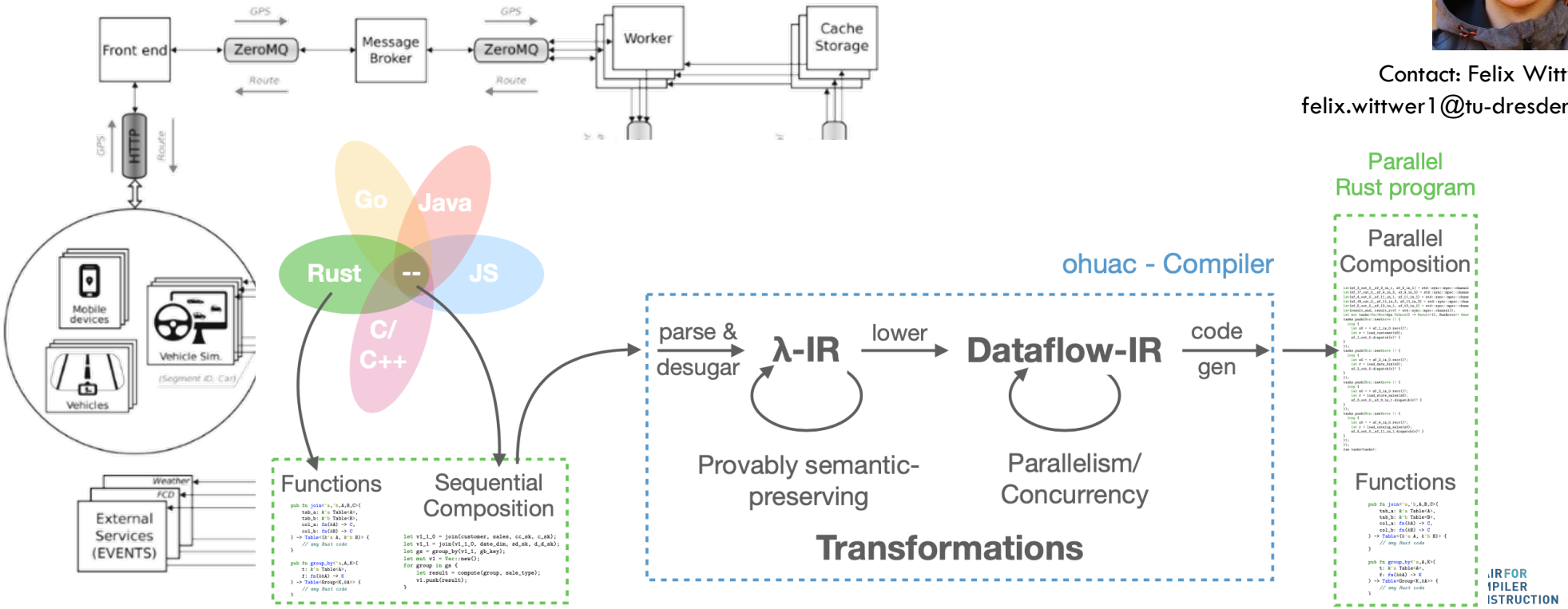


EVEREST

Traffic routing: Coordination + Determinism

- Dataflow functional abstraction
- Implicit parallelism: compiler controlled coordination

Contact: Felix Wittwer
felix.wittwer1@tu-dresden.de



A Toolflow for Programming Cyber-physical Systems (1)

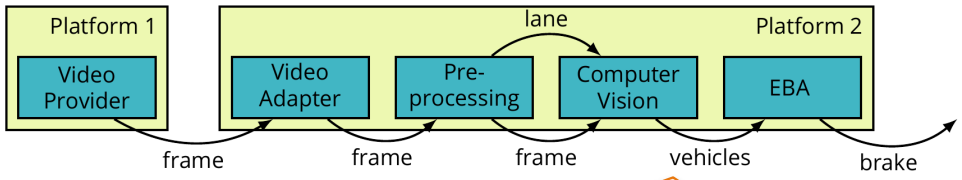
Context

- "A cyber-physical system is a computer without a keyboard that can kill you."
- How can we build confidence in the safety of cyber-physical systems?

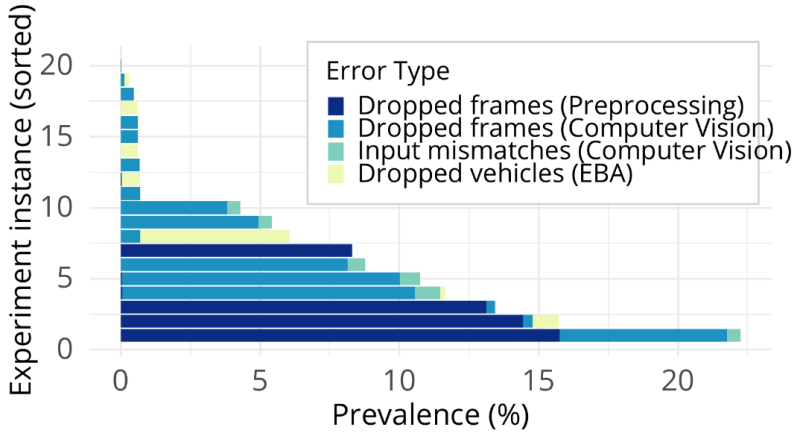
Contact: Christian Menard
Christian.Menard@tu-dresden.de

Example

- In todays distributed systems, nondeterminism is a big issue.
- Even the demonstrator application (brake assistant) of the upcoming automotive standard Adaptive AUTOSAR exposes nondeterminism:



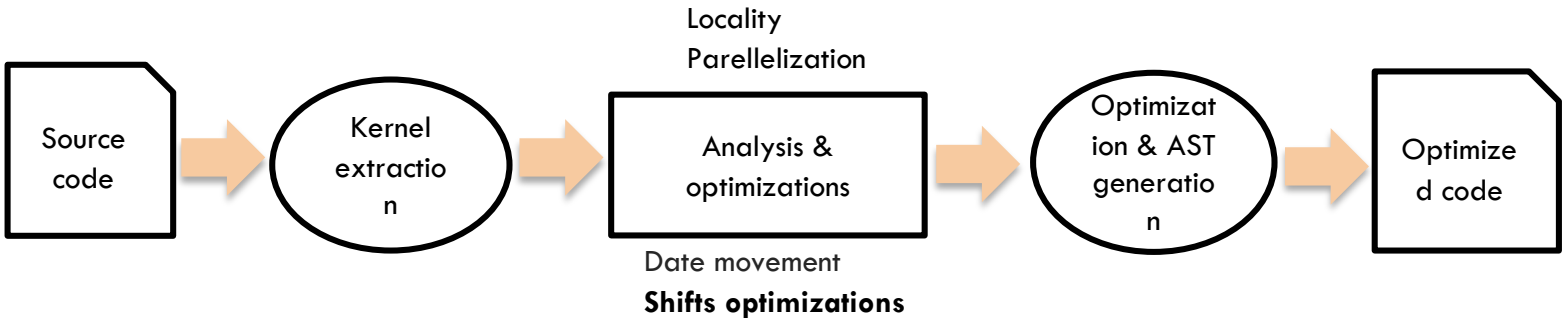
Data may be lost on any of the communication channels. This could have fatal consequences!



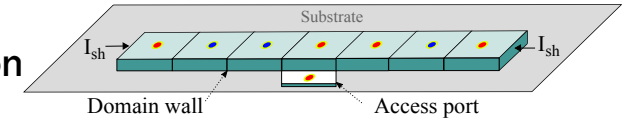
Compiler Optimizations for Racetrack Memories

- Context:
  - Racetrack Memory (RTM) is a promising new class of the emerging non-volatile memories
  - Shifts in RTMs are necessary but unwanted
  - Goal: Generate efficient code that minimizes shifts in RTMs

Contact: Asif Ali Khan
asif_ali.khan@tu-dresden.de



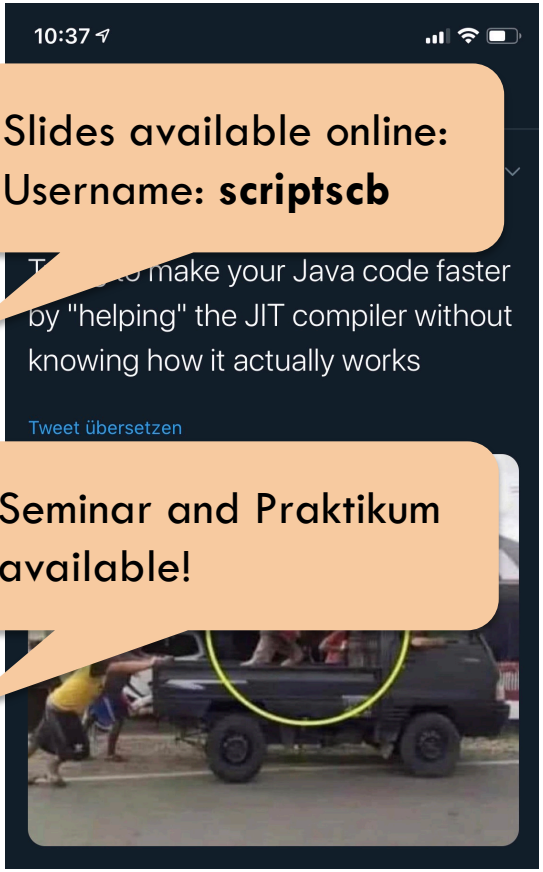
- Skills: Required C/C++; Preferable LLVM, MLIR, polyhedral compilation
- Suitable for DA or MA



About this course

- Classical compiler technology
  - Great "application" of computer science (more on this later)
  - Selected, representative optimization algorithms
- Content inspired by other lecturers
  - Prof. Leupers (RWTH), Prof. Franke (Edinburgh), Prof. Schwarz (Stanford) & others
  - Several books (more on this later)
  - Feedback is welcome!
- 2/2/0 – along **with exercise** (seminar in the website)

- **Examination:** Oral examination (normally)
- Hope to see some of you later for an *Abschlussarbeit!!*



About the exercise

- Main responsible persons: Lars Schütze
- Start: **next week**
- Structure
  - **Theoretical** and **practical** sessions
  - Topic-wise a week behind the lecture
  - Topic online a week before the exercise (prepare upfront)

About you

- Study programs: IST, EE, CS, CSE, DSE?
- Who knows what is a compiler, a preprocessor, a linker?
- Who has ever programmed in assembly?
- Who has ever written something like a compiler?
- Who knows what a deterministic finite state automaton is? And an LR parser?
- Who knows what is a cache? A pipeline? A register? An ALU?
- Who knows what RISC/VLIW/Superscalar are?

About compilers?

- Compilers are important, complex software systems
- Engineering: Optimization problems & approximation
- Great insight: Border between software and hardware (processor architecture)
- Learn how programming languages work & trade-offs in language design
- Ongoing: New architectures & goals ➔ Require new compilers

- Good application of different areas of Computer Science
  - Graph theory: Coloring, topological sort, …
  - Algorithms: Dynamic programming, greedy heuristics, …
  - Machine Learning: Heuristic search
  - Theory: Automata, parsers, pattern matching, fixed-point algorithms, …

Lecture overview

1. Introduction
2. Lexical analysis
3. Syntax analysis
4. Semantic analysis
5. Intermediate representation
6. Control & data-flow analysis
7. IR optimization
8. Target architectures
9. Code selection
10. Register allocation
11. Scheduling
12. Advanced topics

CHAIR FOR COMPILER CONSTRUCTION

cfaed CENTER FOR ADVANCING ELECTRONICS DRESDEN

Compiler Construction: Frontend

Prof. Dr.-Ing. Jeronimo Castrillon
TU Dresden – Cfaed
SS 2022

Chair for Compiler Construction
Helmholtzstrasse 18
jeronimo.castrillon@tu-dresden.de

Lecture overview – Frontend

1. Introduction
2. Lexical analysis
3. Syntax analysis
4. Semantic analysis
5. Intermediate representation
6. Control & data-flow analysis
7. IR optimization
8. Target architectures
9. Code selection
10. Register allocation
11. Scheduling
12. Advanced topics

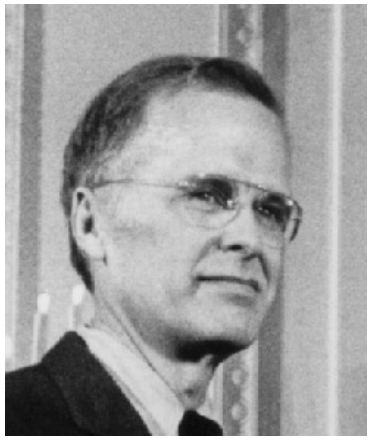
1. Introduction

- History
- Terminology
- Compiler structure
- Trends
- Literature

History of compilers

- < 1950 Programming in assembly
- 1950s First machine-independent languages (skepticism) – A-0 Language
  To compile: "put things together" – Not really what it is today
- 1959 Complete compiler – J. Backus @ IBM for Fortran (in assembly)
  Complexity: 15 man-year projects

Admiral
Grace Hopper
A-0 & COBOL
Source: wikipedia.org



John Backus
FORTRAN, IBM
Source:
columbia.edu/cU/computinghistory

History of compilers (2)

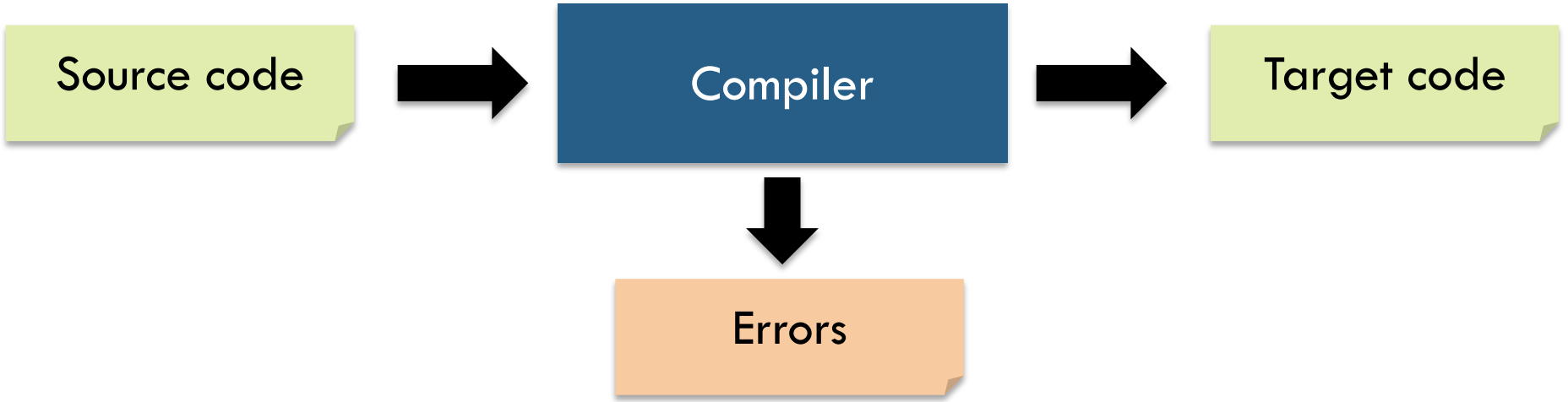
- 1960s — Theoretical work for code analysis
- 1970 — Bootstrapping becomes mainstream (C-compiler written in C)
  First ever: LISP in 1962
- 1970 — Tools to create parts of the compiler (lex & yacc)
- 1980s/90s — Code generator generators
- > 2000 — Optimizations (mostly backend)

- Today — SIMD, vectorization, continuous compilation, new goals, parallelizing compilers, domain-specific languages, skeletons, auto-tuning, …

1. Introduction

- History
- Terminology
- Compiler structure
- Trends
- Literature

What is a compiler



- Compiler translates an input source code to a target code
- Typical: target code closer to machine code (e.g., C ➔ assembly)
- Must recognize illegal code and generate correct code
- Must agree with lower layers (e.g., storage, linker and runtime)

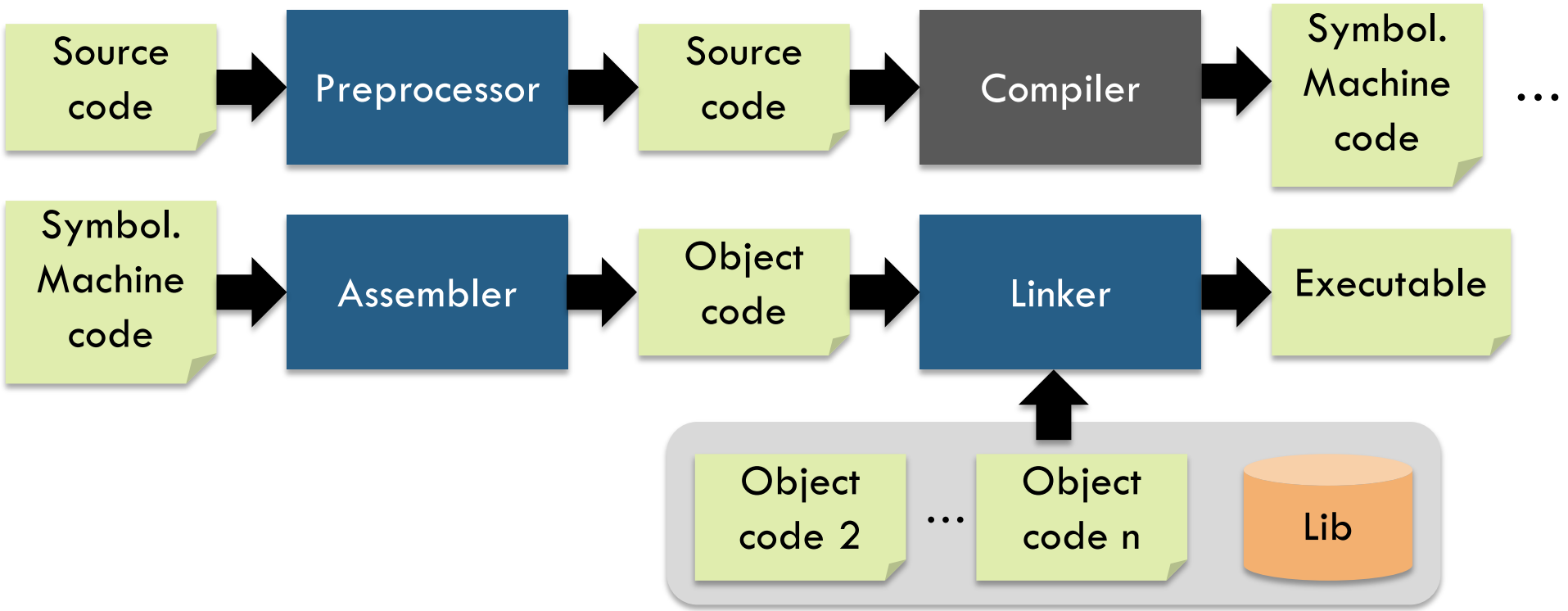
Compiler vs. Interpreter

- Interpreter: Does not generate another program – stepwise translation
  ➔ "Easier" to implement but less room for optimization
- Java byte-code is interpreted by the JVM (nowadays: just-in-time (JIT) compiled)

Compiler goals

- First and foremost: **Correctness**
  - Correct translation while preserving semantics
  - Incorrect code must be discarded
- Optimization goals
  - Typ.: Performance and code size (e.g., inlining)
  - New: Energy/power consumption, robustness
  - Optimality? – Undecidable, often NP complete (especially in the Backend)
- Off-line processing
  - Tolerable turn-around times – time complexity in O(n) – O(n^2)
  - Some domains are more patient (embedded)

Around the compiler



Around the compiler: Preprocessor

- Prepare code before actual compilation
- Examples: C-Preprocessor (try `gcc -E main.c`)
  - Constant definitions

```
#define NUM_ELEMENTS  1000
```

  - Macros

```
#define MIN(x,y) (x <= y ? x : y)
```

  - Include directives

```
#include <stdio.h>
```

Around the compiler: Assembler

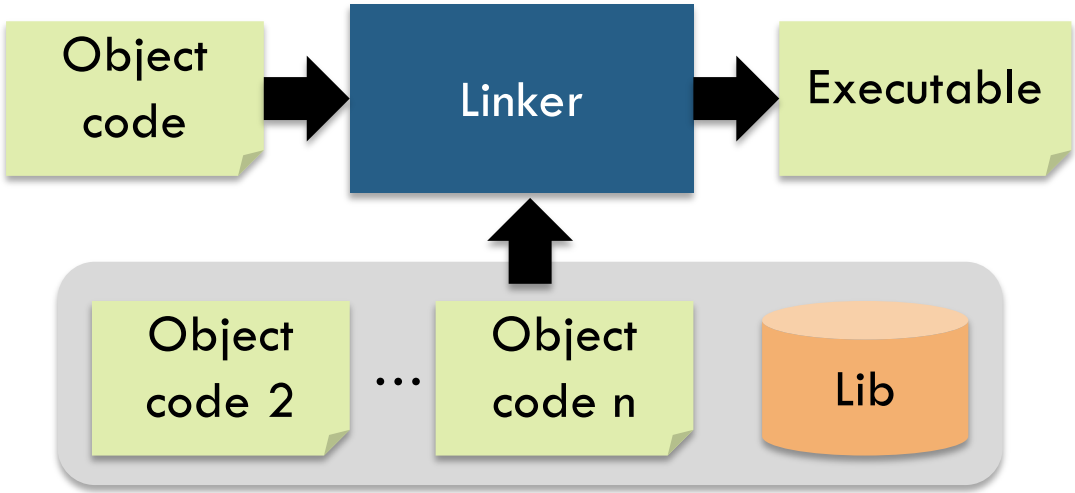
- Translates human-readable machine code into binary
  - Mnemonics (**ADD**) into opcodes (0011)
  - Symbolic addresses into numeric values
  - Generates re-locatable binary code

```
b = a + 2
```

```
MOV a,R1
ADD #2,R1
MOV R1,b
```

opcode	reg.	Immediates, (*) Relocatable code
0010	01	00000000 (*)
0011	01	00000010
0010	01	00000100 (*)

Around the compiler: Linker



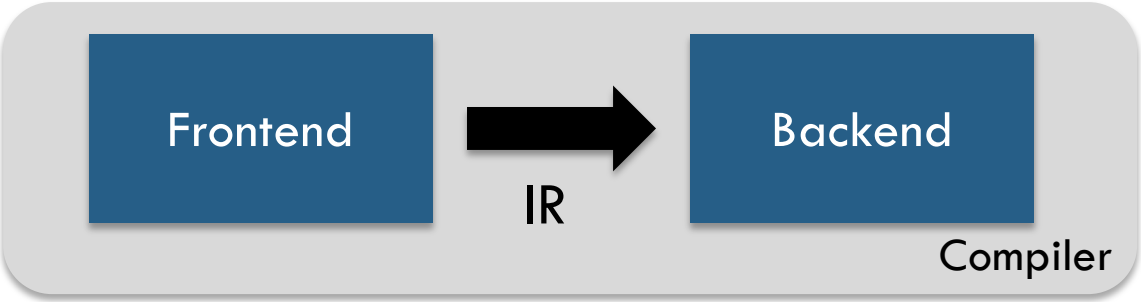
- Link different object codes & common libraries
- Move code segments (e.g., global variables & stack)
- Fix external references (e.g., library calls)

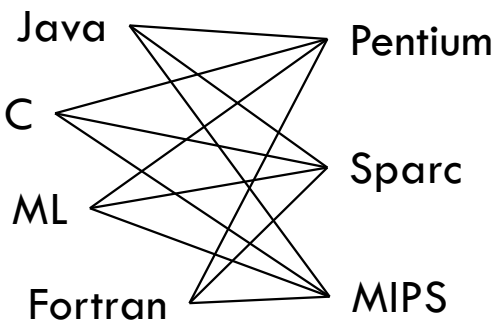
1. Introduction

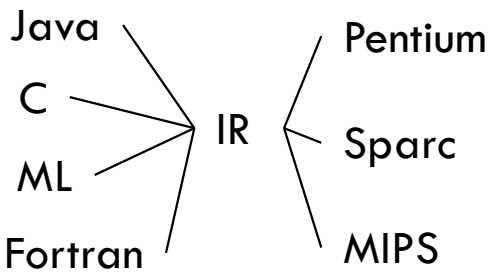
- History
- Terminology
- Compiler structure
- Trends
- Literature

Main Phases

- Two phases
  - IR: Intermediate representation
  - Frontend: Legal code ➔ IR
    - (Typ.) Target-independent
    - Complexity: O(n)-O(n log n)
  - Backend: IR ➔ Target code
    - Target-dependent
    - Complexity: (Typ.) NP complete
- Why IR? – Ideal case
  - Build n x m compilers with n + m components



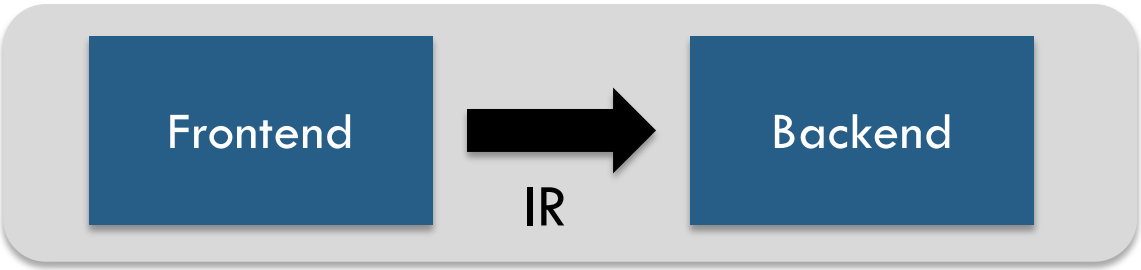




Adapted from: A. Appel: Modern Compiler Implementation in C.

Main Phases (2)

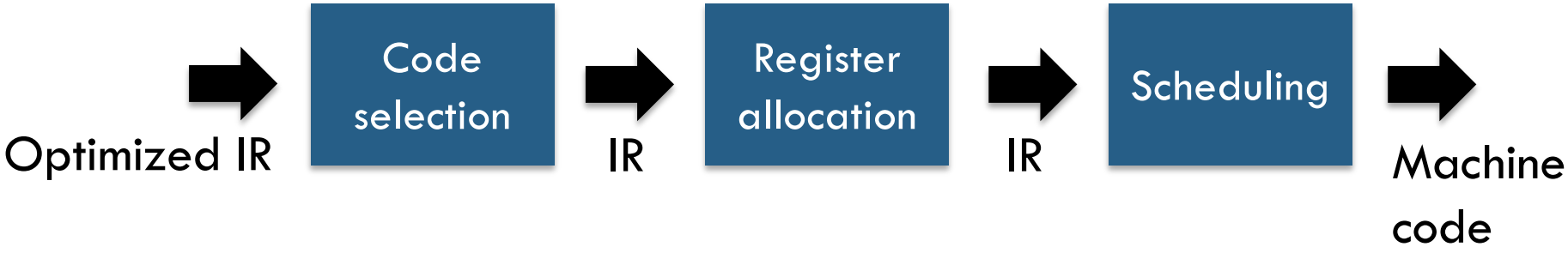
- Two phases
- Three phases
  - Middle-end: several optimization "passes"
- 4+ phases
  - Re-structure: almost source-level optimizations



Structure of a compiler: Frontend + Middle-end

- Lexical (scanner): Maps a character stream into words (tokens)
- Syntax (parser):
  - Recognizes "sentences of tokens" according to a grammar
  - Produces a representation of the application: **Syntax Tree** (ST)
- Semantic analysis: Adds information and checks – types, declarations, …
- IR-generation: Abstract representation of the program, amenable for code generation (backend) – A (abstract) syntax tree is a form of IR
- IR-optimization: Simplification & improvements (e.g., remove redundancies)

Structure of a compiler: Backend



- Code selection: Decide which instructions should implement the IR
- Register allocation: Decide in which register to place variables
- Scheduling: Decide when to execute the instructions (e.g., ordering in assembly program) & ensure conformance with interfaces and constraints

Lexical Analysis – Example

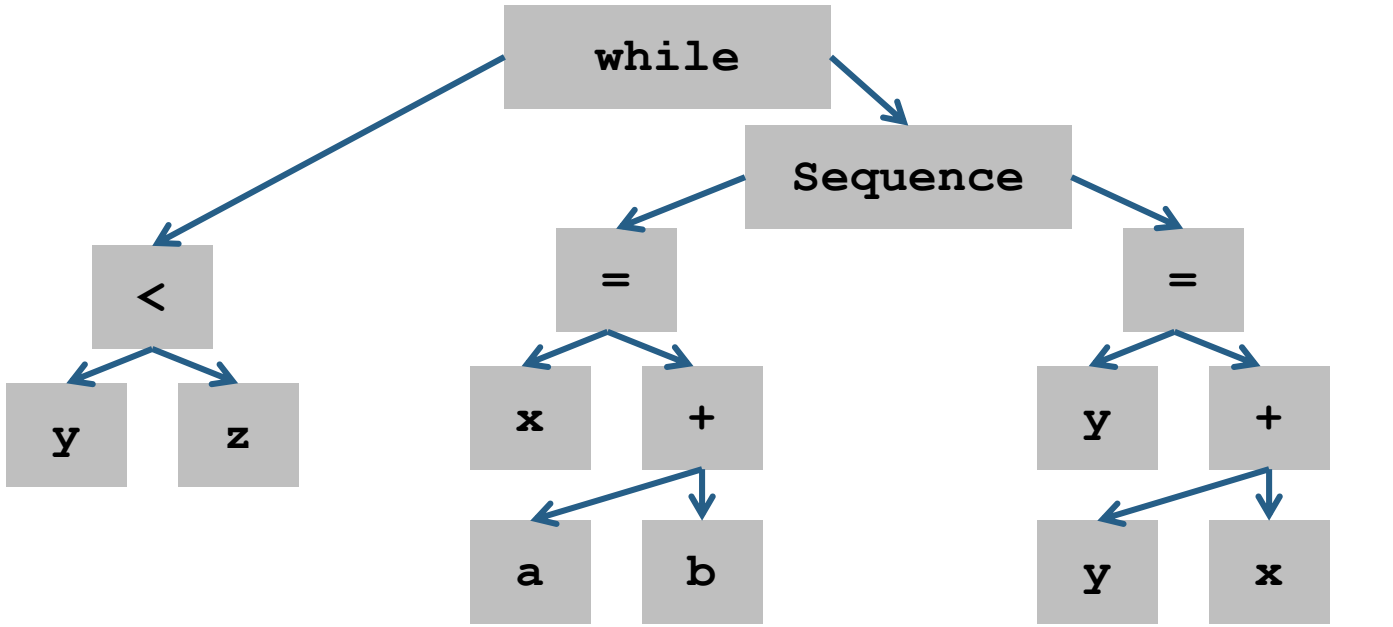

```
while (y < z)
{
  int x = a + b;
  y += x;
}
```

```
T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Int T_Identifier x
T_Assign T_Identifier a
T_Plus T_Identifier b
T_Semicolon
T_Identifier y
T_PlusAssign
T_Identifier x
T_Semicolon
T_CloseBrace
```

Adapted from: http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

Syntax Analysis – Example

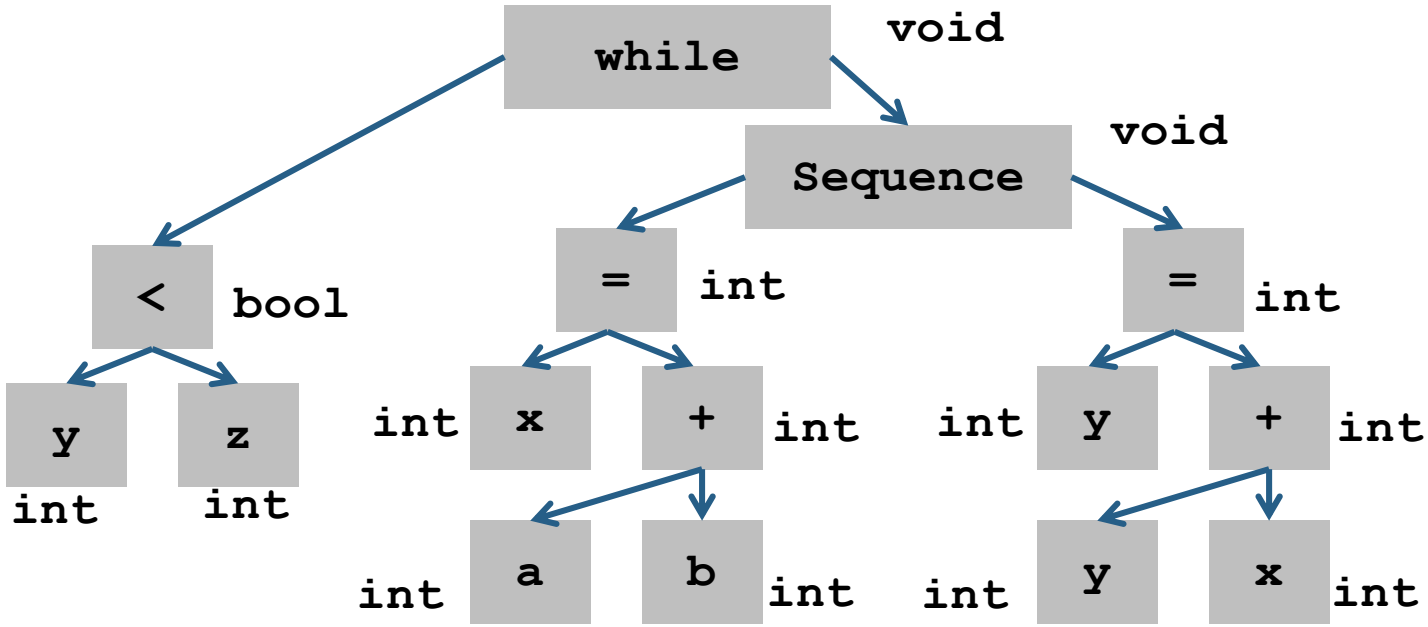

```
while (y < z)
{
   int x = a + b;
   y += x;
}
```



Adapted from: http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

Semantic Analysis – Example


```
while (y < z)
{
   int x = a + b;
   y += x;
}
```



Adapted from: http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

IR generation – Example


```
while (y < z)
{
  int x = a + b;
  y += x;
}
```

```
<...> // First exit condition
Loop: x = a + b
      y = x + y
      _t1 = y < z
      if _t1 goto Loop
```

Adapted from: http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

IR Optimization – Example


```
while (y < z)
{
  int x = a + b;
  y += x;
}
```

```
Loop: x = a + b
      y = x + y
      _t1 = y < z
      if _t1 goto Loop
```

```
      x = a + b
Loop: y = x + y
      _t1 = y < z
      if _t1 goto Loop
```

Adapted from: http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

Code generation – Example


```
while (y < z)
{
  int x = a + b;
  y += x;
}
```

```
      x = a + b
Loop: y = x + y
      _t1 = y < z
      if _t1 goto Loop
```

```
      ADD R1, R2, R3
Loop: ADD R4, R1, R4
      SLT R6, R4, R5
      BEQ R6, loop
```

Adapted from: http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

Why phases?

- The compiler is a complex software (>10^6 lines of code)
  - It must be very reliable for SW development!
  - ➔ Modularization is important
  - ➔ Good modularization allows to add more optimization passes later
- Mature theory and tools for automatic generation of phases (e.g., in frontend)

- Problem with phases: Phase-ordering problem limits optimization potential

1. Introduction

- History
- Terminology
- Compiler structure
- Trends
- Literature

Compiler trends

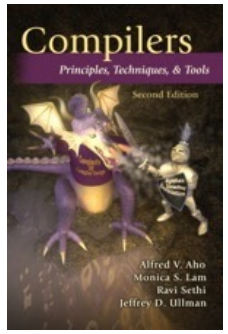
- Automation in compiler generation
  - Code generator generators
  - Retargetable compilers (from processor models)
- Code optimization: Whole program analysis
- New optimization goals: Real-time & energy constraints
- OOP Compilers for new architectures (VLIW & GPGPU)
- Parallelization for multi-core architectures
- Compilers for domain-specific languages
- Continuous compilation

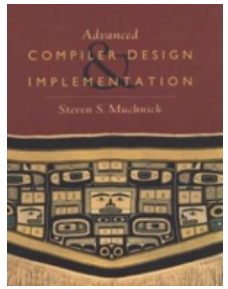
1. Introduction

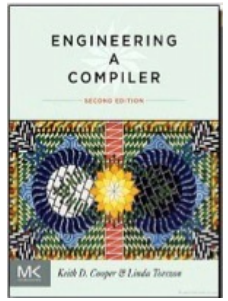
- History
- Terminology
- Compiler structure
- Trends
- Literature

Recommended books

- A. Aho, M. Lam, R. Sethi, J. Ullman: Compilers – Compilers: Principles, Techniques, and Tools, 2nd Ed., Addison-Wesley, 2008, ISBN 978-3-8273-7097-6
- A. Appel: Modern Compiler Implementation in C, Cambridge University Press, 1998, ISBN 0-521-58390-X
- S. Muchnik: Advanced Compiler Design and Implementation, Morgan Kaufmann Publishers, 1997, ISBN 1-55860-320-4
- K. Cooper, L. Torczon. Engineering a Compiler, Second Edition. Morgan-Kaufmann, ISBN 1-55860-698-X
- R. Wilhelm, D. Maurer: Übersetzerbau, 2. Auflage, Springer, 1997, ISBN 3-540-61692-6







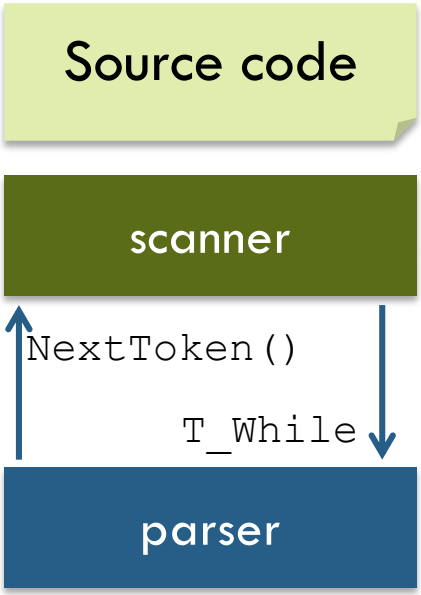
Where are we?

1. Introduction
2. Lexical analysis
3. Syntax analysis
4. Semantic analysis
5. Intermediate representation
6. Control & dataflow analysis
7. Compiler optimization
8. Target architectures
9. Code selection
10. Register allocation
11. Scheduling
12. Advanced topics

2. Lexical Analysis

- The scanner
- Regular expressions
- Finite automata
- Scanner generation with Lex

The scanner: Recall



Scanner: Compiler module in charge of the lexical analysis

```
while (y < z)
{
   int x = a + b;
   y += x;
}
```

```
T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Int T_Identifier x
…
```

Token: Represents a part of speech (includes the lexeme and other attributes)

The scanner: Implementation idea

- How to recognize a potentially infinite set of lexemes represented by a token?
  - Token: IDENTIFIER ➔ a, Email_Adress, b12345, ~~1234abc~~, …
- Need a formalism
  - Formal languages
  - Use regular expressions to identify the lexemes
  - Automata theory
  - ➔ Build recognizers automatically

2. Lexical Analysis

- The scanner
- Regular expressions
- Finite automata
- Scanner generation with Lex

Basics: Alphabet

Def.: An **alphabet** is a non-empty finite set of symbols

- Examples
  - Binary digits $\Sigma = \{0,1\}$
  - Decimal digits $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$
  - C keywords $\Sigma = \{\text{if, then, else, for, while, do, switch, ...}\}$

Basics: Strings

Def.: A **string** over an alphabet Σ is any finite sequence of symbols from Σ

- Examples
  - Binary strings Σ = {0,1} 0, 01010, …
  - Decimal numbers Σ = {0,1,2,3,4,5,6,7,8,9} 12, 1980, …
- Set of strings of any length over Σ is denoted Σ^* (Kleene closure of Σ)

$$\Sigma^* = \bigcup_{n \in N} \Sigma^n$$

ε: Empty word

  - Example: {0,1}*={ε,0,1,00,01,10,11,000,…}

Basics: Languages

Def.: A **formal language** is a set of strings over an alphabet Σ
(i.e., a subset of Σ*)

- Alphabet and strings
  - Binary strings: $\Sigma_1 = \{0,1\}$ 0, 01010, …
  - Decimal numbers: $\Sigma_2 = \{0,1,2,3,4,5,6,7,8,9\}$ 12, 1880, …
- Examples
  - L = {strings over Σ_1 of length 3} = {000,001,010,011,100,101,110,111}
  - L = {multiples of 3 over Σ_2} = {3,6,12,…}

Regular Expressions – regexp

- A regexp is sequence of symbols (constant and operator symbols) that forms a search pattern
- regexps are a family of descriptions that can be used to describe so-called **regular languages** (generated by Type-3 grammars)
- Given a regexp R, we denote the language associated with R, as L(R)

Def.: A **regexp** over an alphabet Σ is a sequence of symbols from (Σ U ε) and operators {*,°,|}, constructed according to the following rules:

1. Empty string: ε is a regexp, with L(ε) = {ε}
2. Literal: every symbol a in Σ is a regexp, with L(a) = {a}
3. …

Regular Expressions – regexp (2)

Def.: A **regexp** over an alphabet Σ is a sequence of symbols from (Σ U ε) and operators {*,°,|}, constructed according to the following rules:

1. Empty string: ε is a regexp, with L(ε) = {ε}
2. Literal: Every symbol a in Σ is a regexp, with L(a) = {a}
3. Union: Let M and N be regexp, then M|N is a regexp, with
   L(M|N)=L(M) U L(N)
4. Concatenation: Let M and N be regexp, then M°N (MN) is a regexp, with
   L(M°N) = $\{s_1 s_2; s_1 \in L(M) \wedge s_2 \in L(N)\}$
5. Kleene star: Let M be a regexp, then M* is a regexp, with
   L(M*) = (L(M))*

regexp: Precedence

- Precedence rules

  (R)

  R*

  R_1R_2

  $R_1 | R_2$

- Operators (*,°,|) are left-associative

- Example: 01*0|10 ➔ ((0(1*))0)|(10)

regexp: Examples

- R = (0|1)*1
- R=(0|1) (0|1) (0|1)
- R=aa*(b|ε)a*
- ??

- L(R): Odd binary numbers
- L(R): Binary strings of length 3
- L(R): Strings of ‘a’s, beginning with ‘a’ and with at most one ‘b’
- L(R): binary strings with a ‘0’ in the middle and the same number of ‘1’s left and right (e.g., 101, 11011, 111101111,…)

regexp: Algebra

Def.: Two regexps M and N are equivalent, M=N, if L(M) = L(N)

- Commutativity: M|N = N|M (MN ≠ NM)
- Associativity: (M|N)|O = M|(N|O), holds also for °
- Distributivity: M(N|O) = MN|MO

- Examples
  - a*(a*bb*|a*) = a*a*bb*|a*a* = a*bb*|a*
  - (a|b)*(b|a)* = (a|b)*(a|b)* = (a|b)*

2. Lexical Analysis

- The scanner
- Languages and regular expressions
- Finite automata
- Scanner generation with Lex

Match regexp: Scanner


```
while (y < z)
{
  int x = a + b;
  y += x;
}
```

```
T_While T_LeftP T_ID T_Less T_ID T_RightP
T_OpenBrace
  T_Int T_ID T_eq T_ID T_plus T_ID
  T_ID T_plus T_eq T_ID
T_CloseBrace
```

- Regular expressions can be implemented using finite automata (non-deterministic and deterministic)

Non-deterministic finite automata (NFA)

Def.: An **NFA** is a 5-tuple (Q,Σ,Δ,q_0,F), where Q is a finite set of states, Σ is an alphabet and

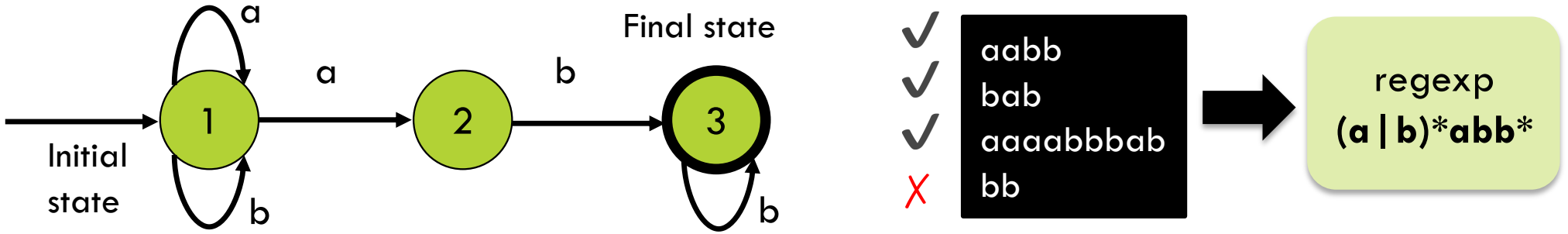
1. Δ : Transition **relation** $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$
2. q_0: Is an initial state ($q_0 \in Q$)
3. F : Set of final states ($F \subseteq Q$)

- Given an NFA in current state q, the next state is decided by
  - Input: Move to q' by reading $a \in \Sigma$ if $(q,a,q') \in \Delta$
  - ε-moves: By moving to state q', if $(q,\varepsilon,q') \in \Delta$
- An NFA accepts a string if a state can be reached that is in F
- The set of all strings accepted by an NFA Z defines a language L(Z)

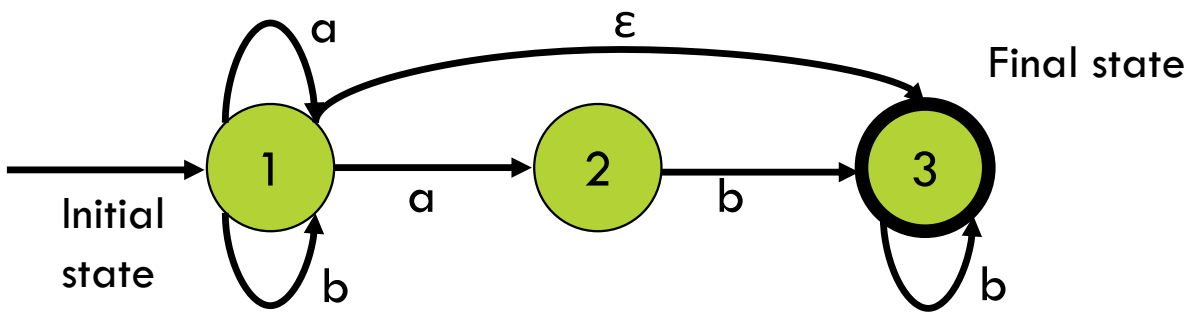
Graph representation

Def.: A **directed graph** (digraph) is a pair G=(V,E) where V is a set of elements called vertices and E $\subseteq$ VxV is a set of ordered pairs called edges (arcs)

- NFA digraph representation
  - States as graph nodes and transitions as graph edges
- Example with Q={1,2,3} , Σ = {a,b}, q_0=1 and F= {3}



Non-determinism



- On state q=1, not exactly defined what to do on arrival of symbol 'a'
- ε-moves introduce non-determinism as well
- Recall: An NFA **accepts** a string if there is at least a sequence of states that finishes with a state in F
- For **aaab**
  - **1**-a-**1**-a-**1**-a-**1**-ε-**3**-b-**3**
  - **1**-a-**1**-a-**2**-a-**Error**

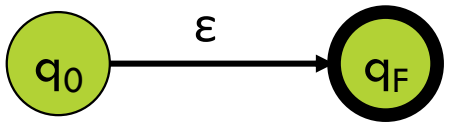
Relation regexp – NFA

For every regexp R there is a NFA Z, with L(R) = L(Z)

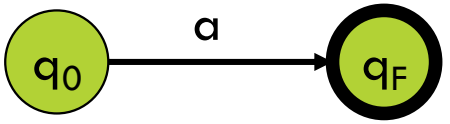
- Construction idea (Thompson's algorithm): Associate an NFA to each regexp construction rule, with
  - Exactly one accepting state q_F
  - No transitions out of q_F
  - No transitions into the starting state q_0

Thompson's algorithm: NFA construction rules

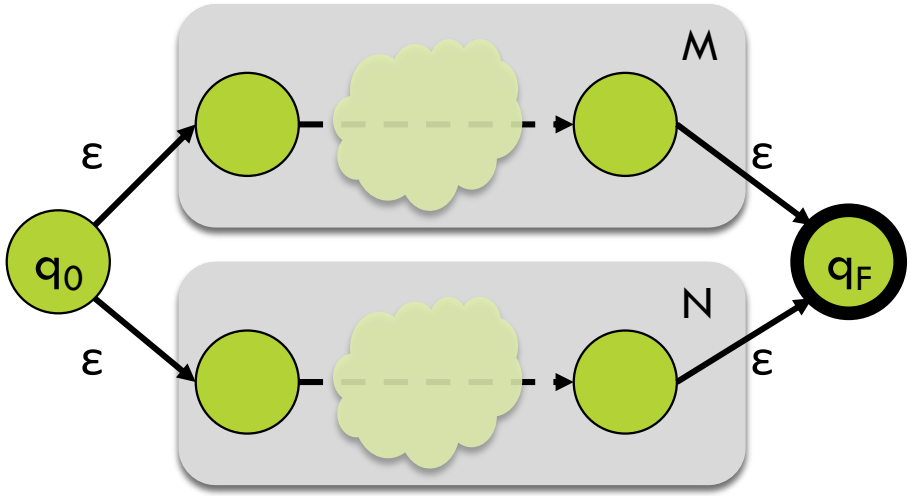
- Rule 1: Empty string R = ε



- Rule 2: Alphabet symbol R = a

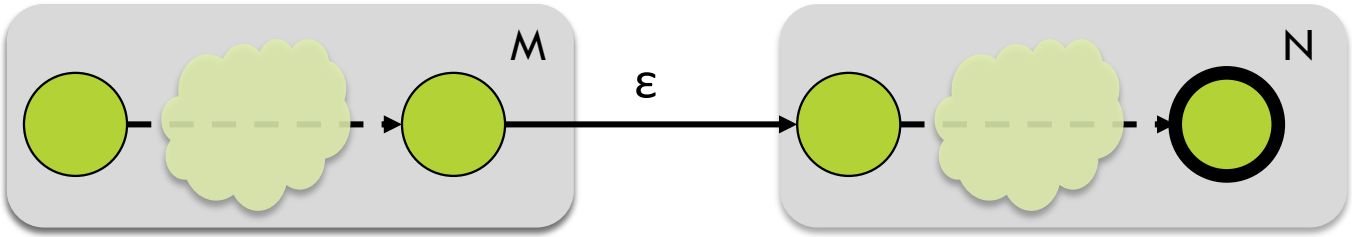


- Rule 3: Union R = M|N

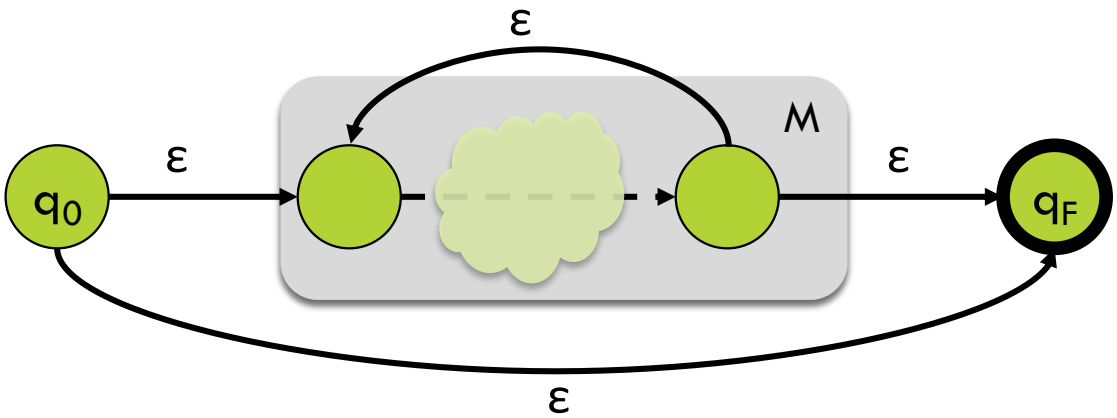


Thompson's algorithm: NFA construction rules (2)

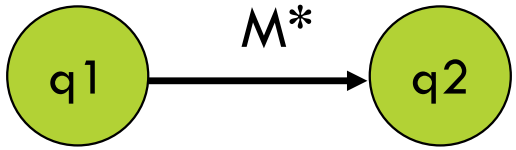
- Rule 4: Concatenation R = MN

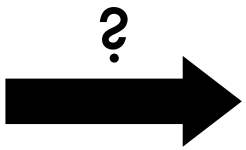


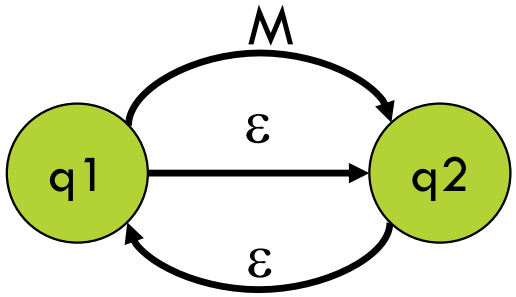
- Rule 5: Kleene star R = M*



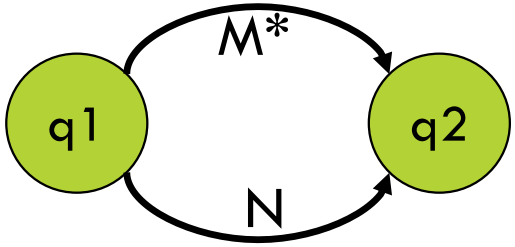
Kleene start: Easier way?

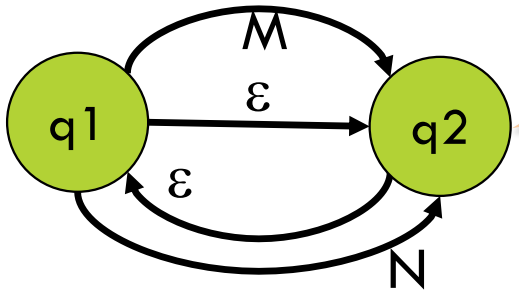






Applying rules for M*|N

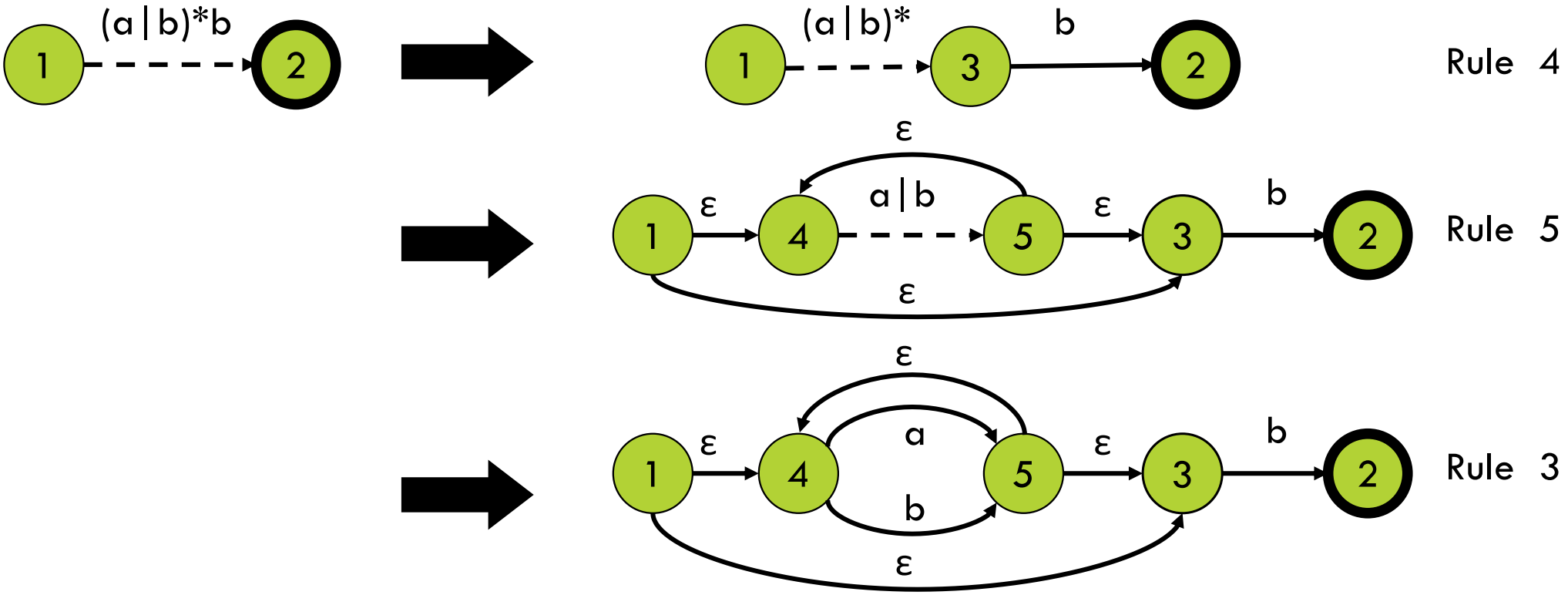




Recognizes (M|N)*

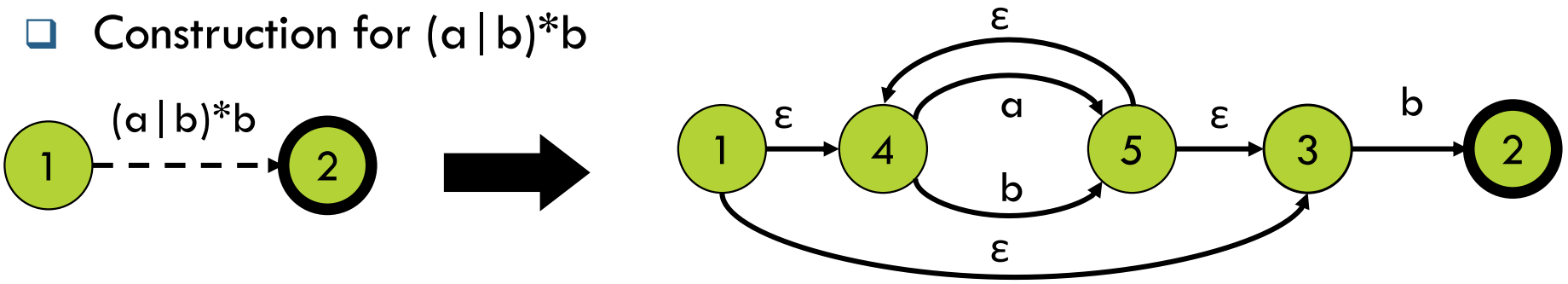
NFA construction: Example

- Construction for (a | b)*b

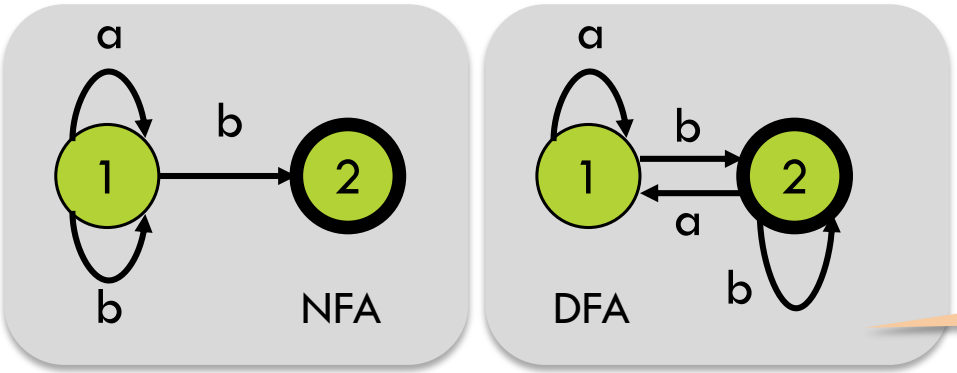


NFA construction: Example (2)

- Construction for (a | b)*b



- Equivalents



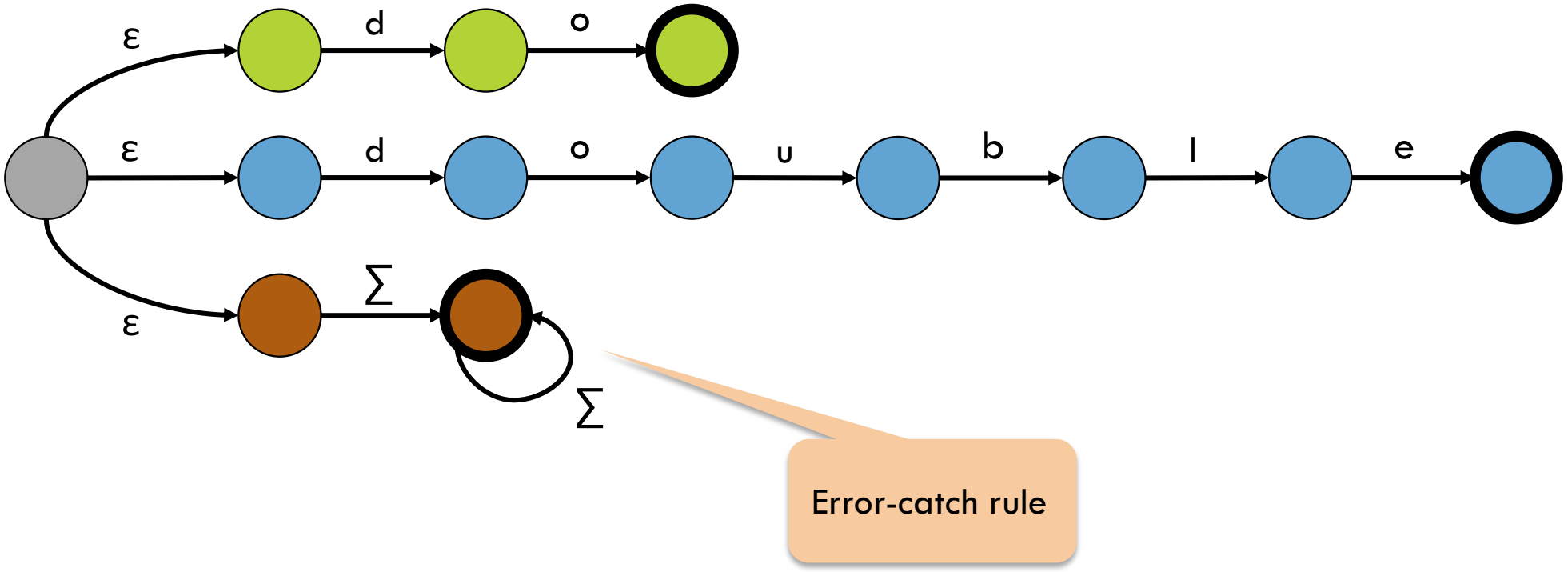
Different options:
Which one to implement?
What is the runtime complexity?
Important: **Automation**

DFA: **Deterministic** finite state automaton (upcoming)

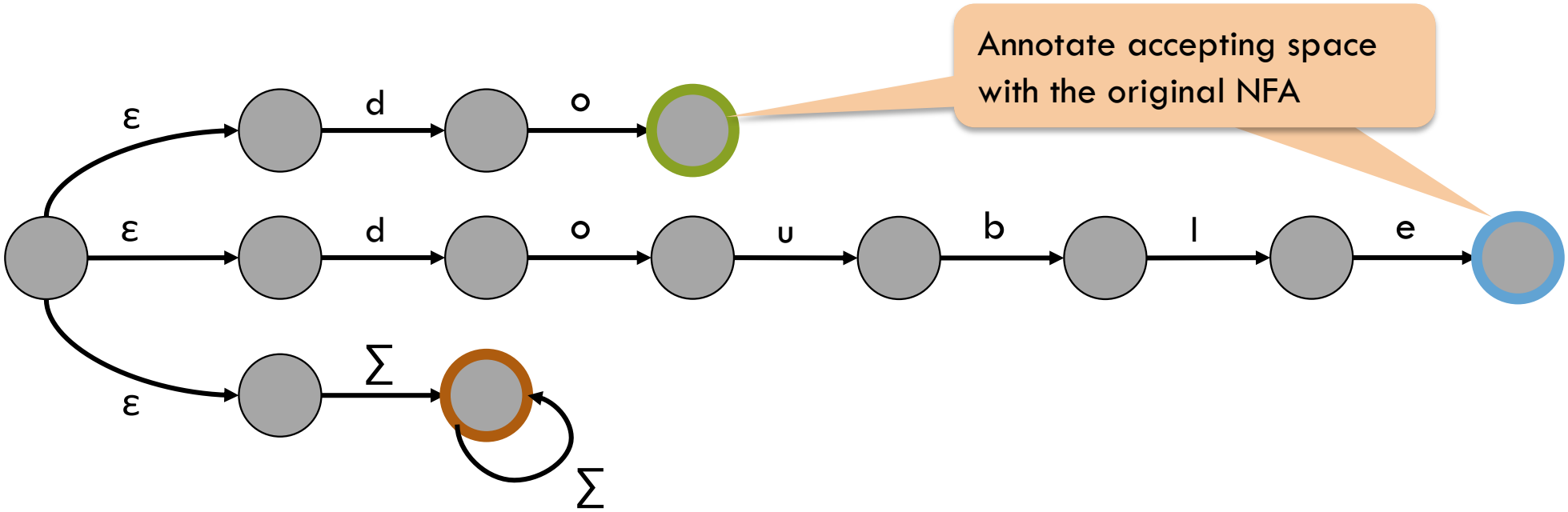
Scanner operation with NFAs

- A scanner = Multiple NFAs running in parallel
- Ambiguities
  - Tie-breaking rule: Maximal munch
  - More ties? – lexeme matches two tokens: Use priorities (first rule wins)
- If nothing matches?
  - Add a catch-up rule that reports an error

Scanner operation with NFAs: Example

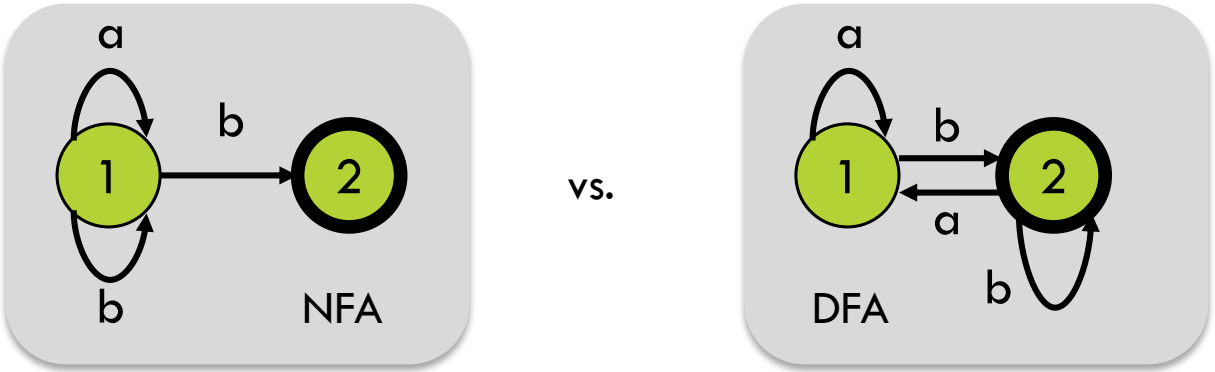


Scanner operation with NFAs: Example



Scanner operation with NFAs: Complexity

- What to do if there are multiple ways to scan the input?
  - Simulate the NFA with n states on string of length m: Complexity **$O(mn^2)$**
  - Option: Build a **deterministic** finite automaton (DFA)



Deterministic Finite Automata (DFA)

Def.: A **DFA** is a 5-tuple (Q,Σ,Δ,q_0,F), where Q is a finite set of states, Σ is an alphabet and

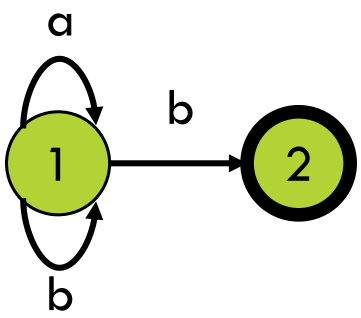
1. Δ: Transition **function** $\Delta : Q \times \Sigma \rightarrow Q$
2. q_0: Initial state ($q_0 \in Q$)
3. F: Set of final states ($F \subseteq Q$)

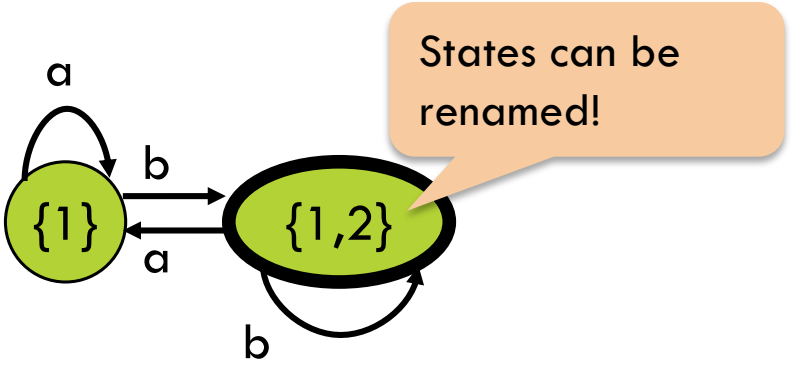
- DFA has no ε-moves and Δ is a function, i.e., Δ(q,a) = q' (no other options)
  - ➔ Simulating a DFA has complexity **O(m)**, with m the length of the input string

NFA → DFA: Subset construction

For every NFA Z there is a DFA Y, with L(Z) = L(Y)

- Construction idea: Make the DFA simulate the NFA
  - DFA states represent subsets of NFA states
  - Transitions in the DFA represent transitions between subsets in the NFA
  - Problem: For n NFA states potentially 2^n DFA states (power set)
- Example
  - ({1},a)→{1}
  - ({1},b)→{1,2}
  - ({1,2},a)→{1}
  - ({1,2},b)→{1,2}





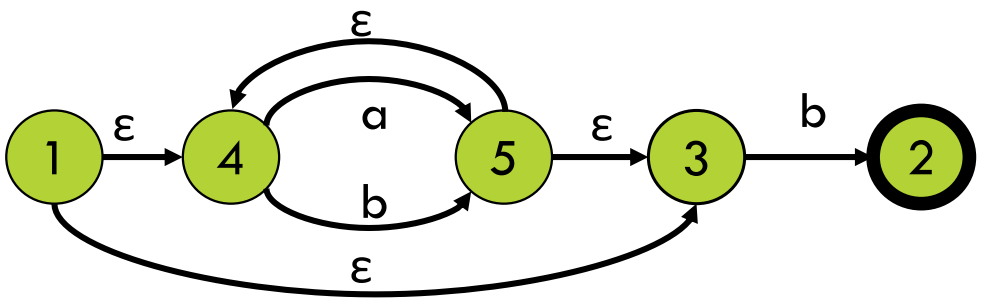
DFA Construction: ε-closure

Def.: Given an NFA $Z=(Q,\Sigma,\Delta,q_0,F)$, the ε-closure($q \in Q$) is the set of states reachable from q by ε-moves, including q itself

Def.: Given an NFA $Z=(Q,\Sigma,\Delta,q_0,F)$ and a subset $S \subseteq Q$, the **ε-closure**(S) is:

$$\bigcup_{q \in S} \epsilon\text{-closure}(q)$$

- Example
  - ε-closure(1)
    = {1,3,4}
  - ε-closure({1,5})
    = {1,3,4,5}



DFA Construction

Def.: Given an NFA Z=(Q,Σ,Δ,q0,F), the move(q ∈ Q, a ∈ Σ) is the set of states reachable from q when reading symbol 'a', i.e.,

move(q,a)= {p | (q,a,p) ∈ Δ}

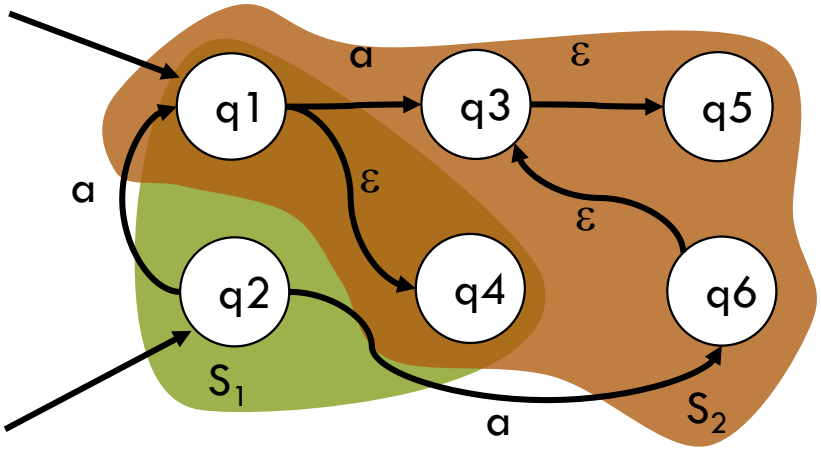
Def.: Similarly, for a subset S ⊆ Q, **move(S,a)** is: $\bigcup_{q \in S} \text{move}(q, a)$

- Given an NFA Z=(Q,Σ,Δ,q_0,F) we construct a DFA Y=(Q',Σ, Δ',q'_0,F'), where:
  - Q' ⊆ P(Q) – With P() the power set
  - q'_0 = ε-closure(q_0)
  - F' = {S ∈ Q'; S ∩ F ≠ ∅} – The DFA accepts when the NFA could have accepted
  - Δ'

$$\Delta'(S, a) = \epsilon\text{-closure}(\text{move}(S, a))$$

DFA construction: Example

- Consider following portion of an NFA and input ‘a’



Move(S_1,a)
= {q1,q3,q6}

ε-closure(move(S_1,a)) = ε-closure({q1,q3,q6})
= {q1,q3,q4,q5,q6} = S_2

$\Delta'(S_1,a) = S_2$

DFA Construction: Algorithm

Input NFA $Z = (\Sigma, Q, \Delta, q_0, F)$

Output DFA $Y = (\Sigma, Q', \Delta', q'_0, F')$

$q'_0 = \varepsilon\text{-closure}(q_0)$; $Q' = \{q'_0\}$; $\Delta' = \emptyset$;

while (Q' still changing)

  foreach $S \in Q'$

    foreach $a \in \Sigma$

      $S_? = \varepsilon\text{-closure}(\text{move}(S,a))$

      if $S_? \notin Q'$ **then**

        $Q' = Q' \cup \{S_?\}$

      $\Delta' = \Delta' \cup \{(S,a) \rightarrow S_?\}$

$F' = \{S \in Q'; S \cap F \neq \emptyset\}$

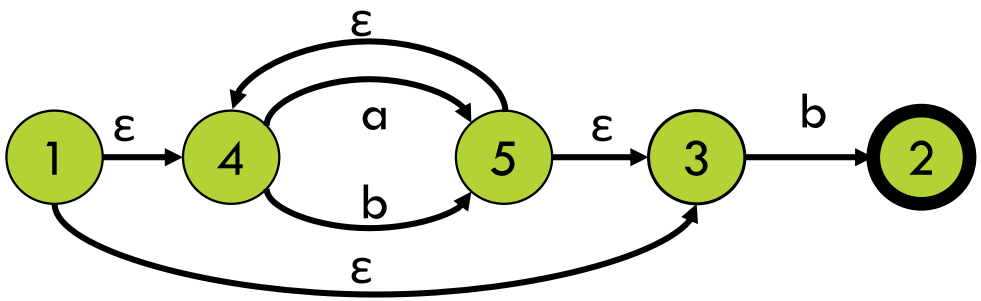
Fixed-point computations:

- Q' contains no duplicates
- Monotone while loop
- $2^{|Q|}$ is finite

Similar algorithms in parsing and dataflow analysis

DFA construction: Example

(a|b)*b ➔



Final states?

q'0 = {1,3,4}

ε-closure(Move(q'0,a)) = {3,4,5} ➔ q'1 (new)

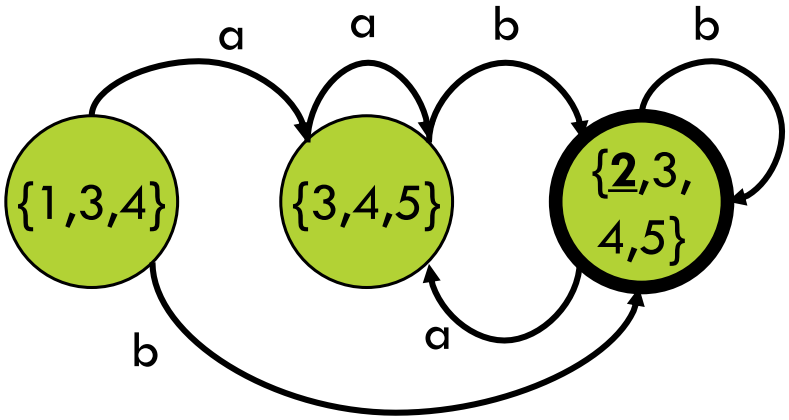
ε-closure(Move(q'0,b)) = {2,3,4,5} ➔ q'2 (new)

ε-closure(Move(q'1,a)) = {3,4,5} ➔ q'1

ε-closure(Move(q'1,b)) = {2,3,4,5} ➔ q'2

ε-closure(Move(q'2,a)) = {3,4,5} ➔ q'1

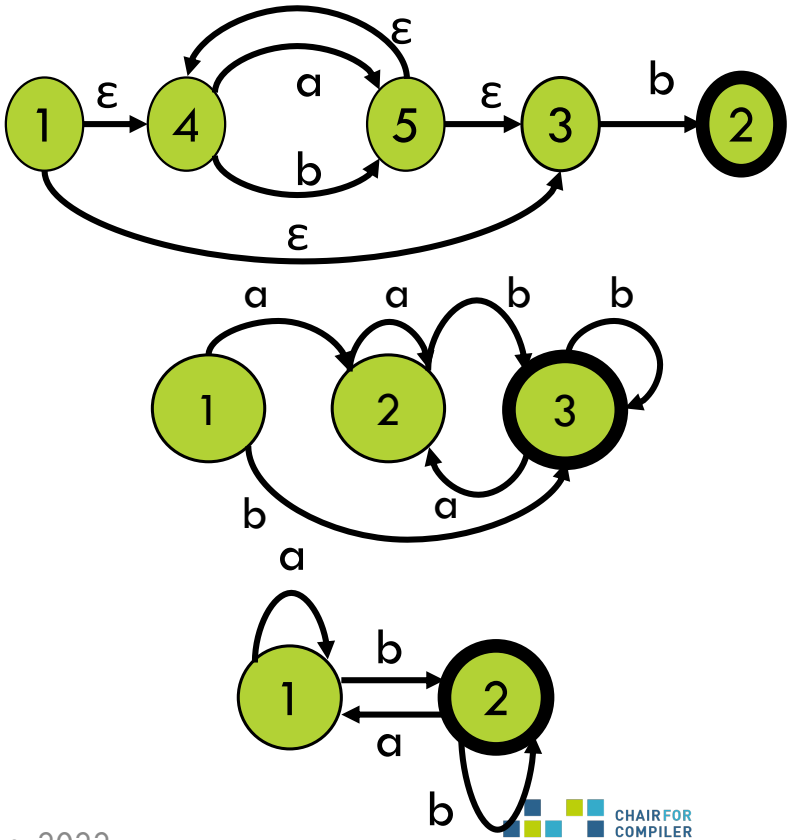
ε-closure(Move(q'2,b)) = {2,3,4,5} ➔ q'2



DFA minimization

- So far: regexp → NFA (Thompson) → DFA (Powerset construction)
  - Problem: State explosion – Time complexity? – Memory requirements?
- DFA minimization: Hopcroft's algorithm
  - Identify & merge equivalent states
    - Paths leading to states are equivalent
    - Transition on a given symbol 'a' lead to equivalent states
    - Transition on different symbols lead to distinct sets

(a | b)*b



2. Lexical Analysis

- The scanner
- Languages and regular expressions
- Finite automata
- Scanner generation with Lex

Lex (flex)

- Lex is the standard scanner generator in unix systems (1975)
  - Input: lex specification
  - Output: Scanner's C-code (produces tokens on demand for the parser)

- Input specification

```
Definition section
%%
Rules section
%%
C code section
```

flex: Example

```
/*** Definition section ***/

%{
/* C code to be copied verbatim */
#include <stdio.h>
%}

/* This tells flex to read only one input file */
%option noyywrap

%%
    /*** Rules section ***/

    /* [0-9]+ matches a string of one or more digits */
[0-9]+  {
            /* yytext is a string containing the matched text. */
            printf("Saw an integer: %s\n", yytext);
        }

.|\n    {   /* Ignore all other characters. */   }

%%
/*** C Code section ***/

int main(void)
{
    /* Call the lexer, then quit. */
    yylex();
    return 0;
}
```

Source: wikipedia.org

Regexp
(0|1...|9) (0|1...|9)*

Action: Executed in case of detection

Where are we?

1. Introduction
2. Lexical analysis
3. Syntax analysis
4. Semantic analysis
5. Intermediate representation
6. Control & data-flow analysis
7. IR optimization
8. Target architectures
9. Code selection
10. Register allocation
11. Scheduling
12. Advanced topics

3. Syntax Analysis

- Introduction
- Context free grammars
- Top-down parser
- Bottom-up parser
- Error handling
- Parser generation with yacc

The parser: Recall

Source code

scanner

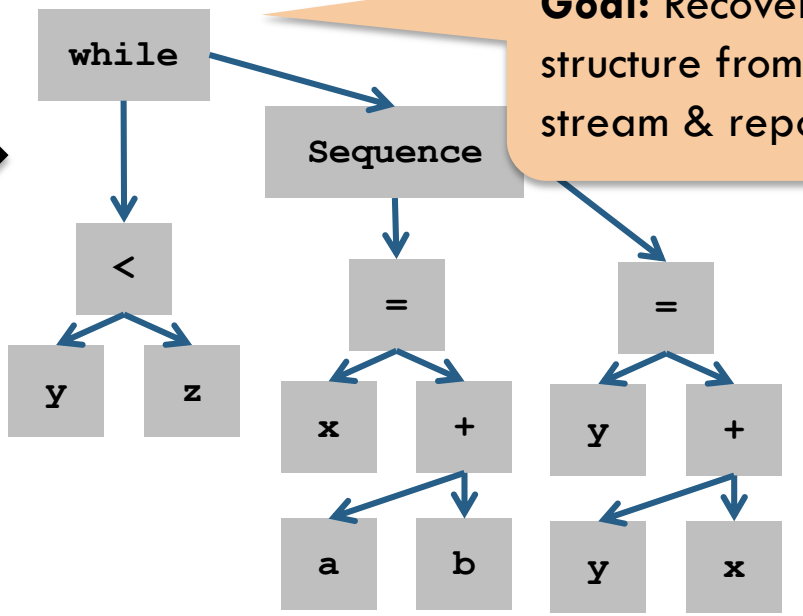
NextToken()

T_While

parser

Parser: Compiler module in charge of the syntax analysis

```
while (y < z)
{
  int x = a + b;
  y += x;
}
```



Goal: Recover the structure from the token stream & report errors

Why not use regexp and DFAs?

- Usually programming languages aren't regular languages
  - Balanced parenthesis & nested blocks in functions
- Example: $L = \{a^n b^n : n > 0\}$, regular?
  - If it is regular, then there is a regexp ➔ and a DFA, with k states (finite)
  - For an input with $n > k$, at least one repeated state while reading a^n
  - Let repeated state be s, after reading a^p and a^q
  - The DFA should accept $a^p b^p$ and $a^q b^q$
  - ➔ Consequently, the DFA would also accept $a^q b^p$ and $a^p b^q$

  (Alternative proof: By contradiction to the **pumping lemma**)

➔ Need more powerful formalism: **Context-free grammars**

Terminology

	Scanner	Parser
Alphabet	Characters, e.g., ASCII	Tokens, e.g., ID, numbers, keywords
String	Character string, e.g., keyword	Entire program
Language	Subset of strings, e.g., floating numbers	Programming language
Formalism	Regular expressions	Context-free grammars
Recognition	DFA	Parsers, e.g., LL(k) & LR(k)

3. Syntax Analysis

- Introduction
- Context free grammars
- Top-down parser
- Bottom-up parser
- Error handling
- Parser generation with yacc

Context-free grammar

Def.: A **context-free grammar (CFG)** is a 4-tuple (N,T,P,S), where

1. N: **Alphabet** of non-terminal symbols (help-symbols)
2. T: **Alphabet** of terminal symbols (of the language defined by the CFG), $N \cap T = \varnothing$
3. $P \subseteq N \times (N \cup T)^*$: Set **rules** or **productions**
4. $S \in N$: Is the start symbol

- Notation
  - A,B,C,…: Non-terminals
  - a,b,c,…: Terminals
  - α,β,γ,δ,…: Strings of $(N \cup T)^*$
  - Productions: A relation element $(A,\alpha) \in P$, is denoted $A \rightarrow \alpha$

Derivation

Def.: For any strings $\alpha, \beta \in (N \cup T)^*$, β can be **directly derived** from α, denoted $\alpha \rightarrow \beta$, if there are $\gamma_1, \gamma_2 \in (N \cup T)^*$ and $(A, \gamma_3) \in P$ such that $\alpha = \gamma_1 A \gamma_2$ and $\beta = \gamma_1 \gamma_3 \gamma_2$

Def.: A sequence of derivations $\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \ldots \rightarrow \alpha_n$ of zero or more steps is denoted $\alpha_1 \rightarrow^* \alpha_n$ (transitive closure of $\rightarrow$)

Def.: For a CFG G, a string $\alpha \in (N \cup T)^*$ is called a **sentential form** *iff* $S \rightarrow^* \alpha$

Deutsch: **Satzform**

Example: Chemistry

Form → Cmp | Cmp Ion

Cmp → Term | Term Num | Cmp Cmp

Term → Elem | (Cmp)

Elem → H | He | Li | Be | B | C | …

Ion → + | - | IonNum + | IonNum -

IonNum → 2 | 3 | 4 | …

Num → 1 | IonNum

P

G = (N,T,P,S)
N = {Form,Cmp,Ion,Term,Num,IonNum}
T = {H,He,…, +,-,1,2,3,…}
S=Form

"Form → Cmp | Cmp Ion" is a shortcut for:
Form → Cmp
Form → Cmp Ion

Adapted from:
http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

Example: Chemistry

Form → Cmp | Cmp Ion

Cmp → Term | Term Num | Cmp Cmp

Term → Elem | (Cmp)

Elem → H | He | Li | Be | B | C | …

Ion → + | - | IonNum + | IonNum -

IonNum → 2 | 3 | 4 | …

Num → 1 | IonNum

P

Sample derivation

Form

⇒ Cmp Ion

⇒ Cmp Cmp Ion

⇒ Cmp Term Num Ion

⇒ Term Term Num Ion

⇒ Elem Term Num Ion

⇒ Mn Term Num Ion

⇒ Mn Elem Num Ion

⇒ MnO Num Ion

⇒ MnO IonNum Ion

⇒ MnO_4 Ion

⇒ MnO_4^-

Adapted from:
http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

Example: Programming languages

```
BLOCK  → STMT
       | { STMTS }

STMTS  → ε
       | STMT STMTS

STMT   → EXPR;
       | if (EXPR) BLOCK
       | while (EXPR) BLOCK
       | do BLOCK while (EXPR);
       | BLOCK
       | ...

EXPR   → identifier
       | constant
       | EXPR + EXPR
       | EXPR – EXPR
       | EXPR * EXPR
       | ...
```

Adapted from:
http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

Context-free languages

Def.: Given a CFG G = (N,T,P,S) the associated context-free **language** is defined by L (G) = {w ∈ T*; S➔*w}

Def.: A language L is called **context-free** if there exists a CFG G, so that L = L(G)

CFGs and regexps

- All regular languages are also context-free languages, i.e., for every regexp R there is a CFG G so that L(R) = L(G)
- Simple rules
  - (a|b) → A → a; A→b
  - a* → A →aA; A→ε
- Example: (a|b)*b
  - S→AB
  - A→CA | ε
  - C→a
  - C→b
  - B→b

Derivations: Right and leftmost

Def. A **leftmost derivation** is a derivation in which each step expands the leftmost non-terminal (analogous for a rightmost derivation)

- Example: Expressions

(1) E → E + E
(2) E → E * E
(3) E → (E)
(4) E → -E
(5) E → id

- Leftmost

  E → -**E** → -(**E**) → -(**E**+E) → -(id + **E**) → -(id + id)

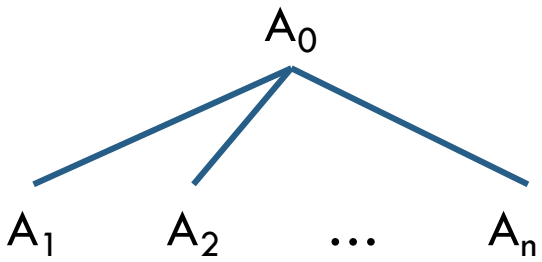
- Rightmost

  E → -**E** → -(**E**) → -(E+**E**) → -(**E** + id) → -(id + id)

Parse tree

Def.: A **parse tree (concrete syntax tree)** encodes the steps of a derivation. The root node represents the start-symbol, internal nodes represent non-terminal symbols and leaves represent terminal symbols

- In-order left-to-right traversal of the leaves represent the program
- Parse trees are invariant to leftmost and rightmost derivation
- When applying a production $A_0 \rightarrow A_1 A_2 .. A_n$, with $A_0 \in N$, $A_{i:1,\ldots,n} \in (N \cup T)$, the children $A_{i:1,\ldots,n}$ of node A_0 are placed from left to right

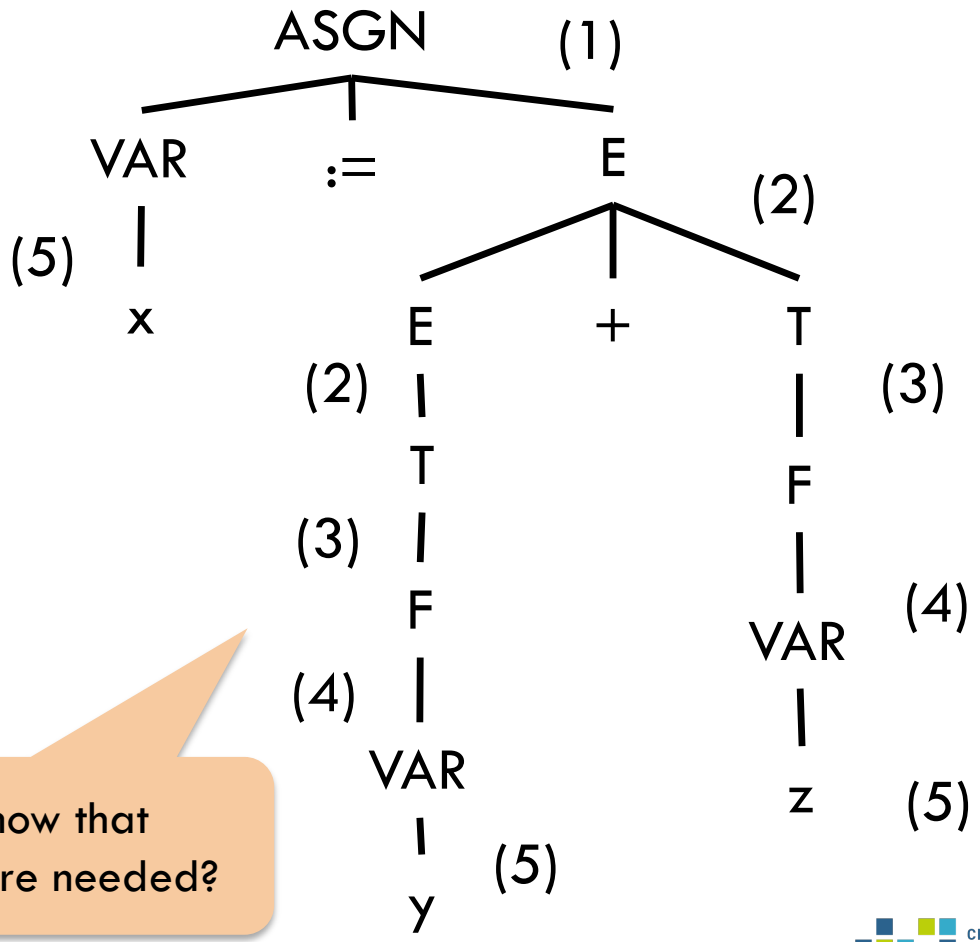


Parse tree: Example

- Given the rules

(1) ASGN → VAR := E
(2) E → E + T | T
(3) T → T x F | F
(4) F → (E) | VAR
(5) VAR → x | y | z

Parse tree for: x := y + z



How did we know that these steps were needed?

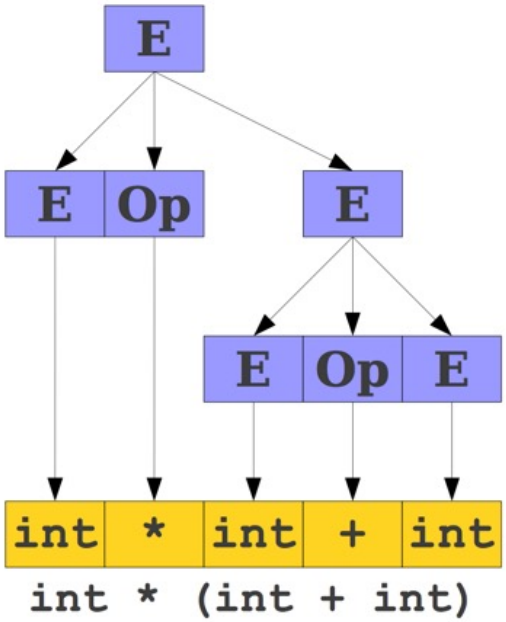
Parse challenge: Ambiguity

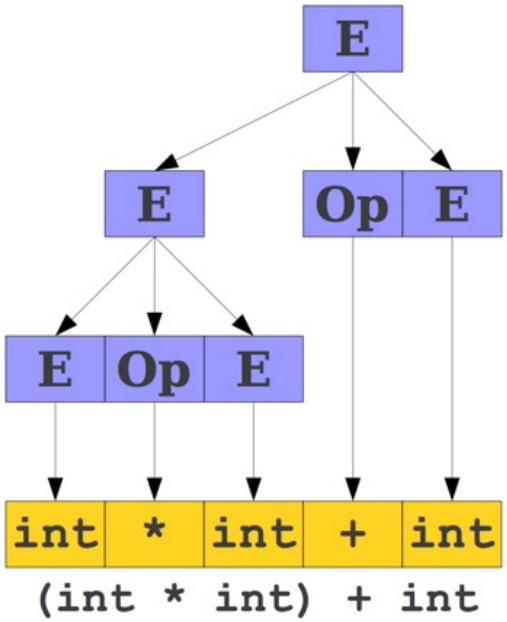
Def.: A CFG G is called ambiguous if there is at least one word w in L(G), i.e., S➔*w, with two or more parse trees

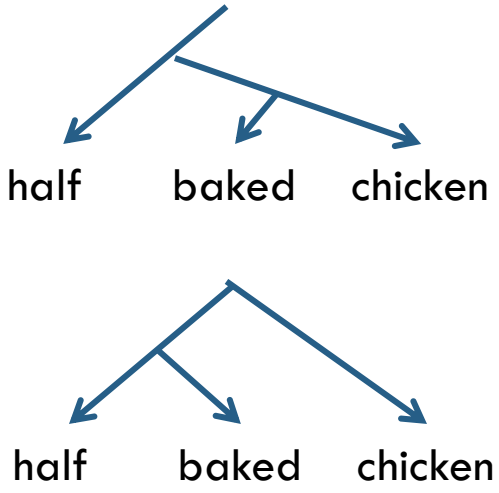
- Ambiguity is a property of the grammar
  - A language is **inherently ambiguous** if there is no unambiguous grammar
- Detecting ambiguity: Undecidable

Parse challenge: Ambiguity (2)

- Why is this a problem? – different semantics ➔ How to translate then?
- Examples





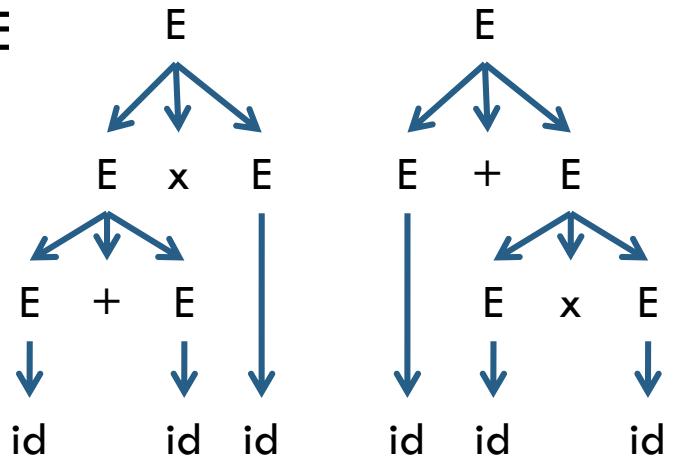


Adapted from: http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

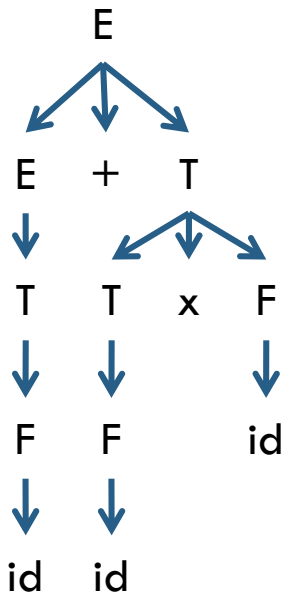
Resolving ambiguity

- Layering: Break derivation into different layers in the tree
  - Enforce precedence: id + id x id

(1) E → E + E
(2) E → E x E
(3) E → (E)
(4) E → -E
(5) E → id



(1) E → E + T
(2) E → T
(3) T → T x F
(4) T → F
(5) F → id

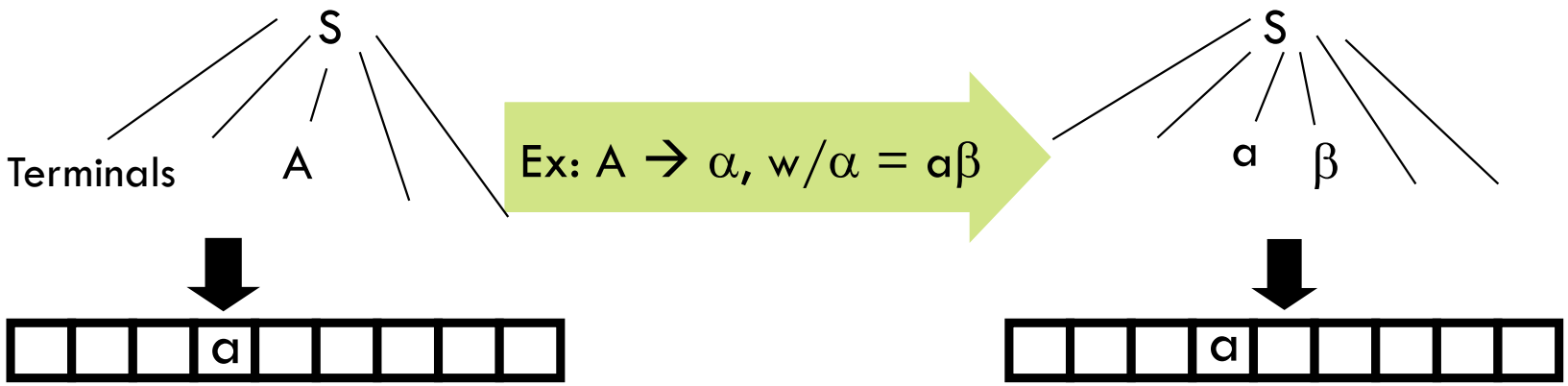


3. Syntax Analysis

- Introduction
- Context free grammars
- Top-down parser
- Bottom-up parser
- Error handling
- Parser generation with yacc

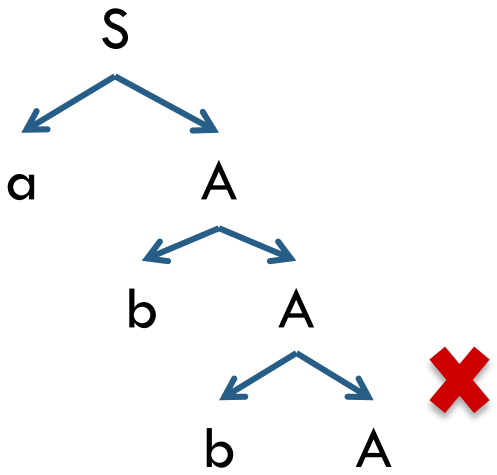
Top-down parser

- Recall – Parser goal: Given a program find a parse tree (or report error)
- Overview:
  - Start with the start symbol (parse tree root)
  - Grow towards the leaves using leftmost derivation, picking rules to match the input
  - Bad selection: backtrack (expensive)
    - Some grammars are backtrack-free (later: predictive parsing)
- Graphically



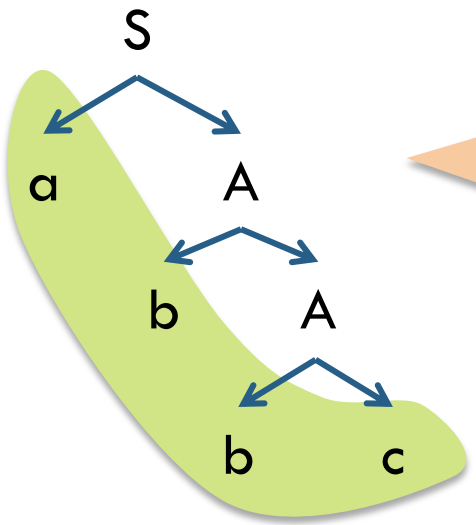
Top-down parser: Example

- Rules
  - S→aA
  - A→bA
  - A→bc
- Input: **abbc**
  - Need a: S→aA
  - Need b: A→bA
  - Need b: A→bA
  - Need c: Impossible ➔ backtrack



Top-down parser: Example

- Rules
  - S→aA
  - A→bA
  - A→bc
- Input: **abbc**
  - Need a: S→aA
  - Need b: A→bA
  - Need b: A→bA
  - Need c: Impossible ➔ backtrack
  - Take A→bc (done!)



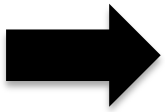
What kinds of grammars would be problematic?

Problem 1: Left recursion

Def.: A CFG G is left recursive if there is a non-terminal symbol A and a sentential $\alpha \in (N \cup T)^*$ so that $A \rightarrow^* A\alpha$ (analogous for right recursion)

- Top-down parsers cannot handle left recursion → leads to non-termination!
- Solution: Grammar transformation into right recursive

(1) $E \rightarrow E\ \alpha$
(2) $E \rightarrow \beta$

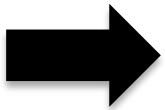


(1) $E \rightarrow \beta\ T$
(2) $T \rightarrow \alpha\ T$
(3) $T \rightarrow \varepsilon$

Problem 2: Multiple options

- Having multiple rules for a non-terminal in the partial parse tree may lead to backtracking
- Solution: Factorization

(1) E → AB
(2) E → ABC

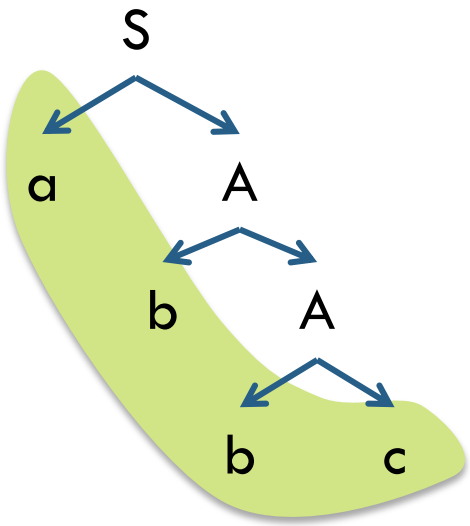


(1) E → ABX
(2) X → C
(3) X → ε

Predictive parsing (recursive decent parser)

- Idea: Use the input to better predict what rule to use and prevent backtracking
  - Use **look-ahead tokens** from the input stream
  - The more tokens, the better
- Tradeoff: Easy parser construction, but low expressiveness
- Example with **two** look-ahead tokens

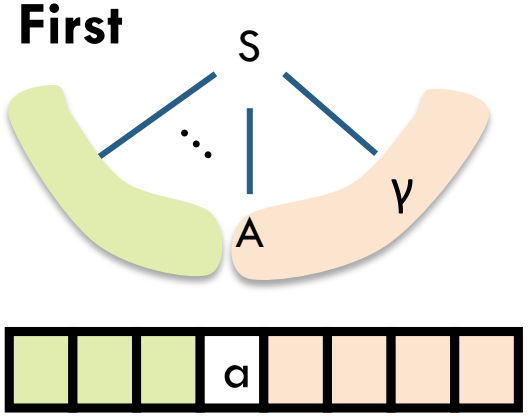
  Input **abbc**
  - S→aA → Input: ab: S→aA
  - A→bA → Input: bb: A→bA
  - A→bc → Input: bc: A→bc



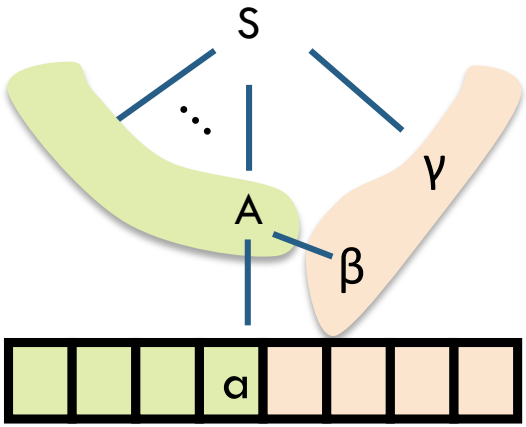
First and Follow sets: Intuition

- When to apply a rule **A→α** when looking at a token 'a'?

1. **First**

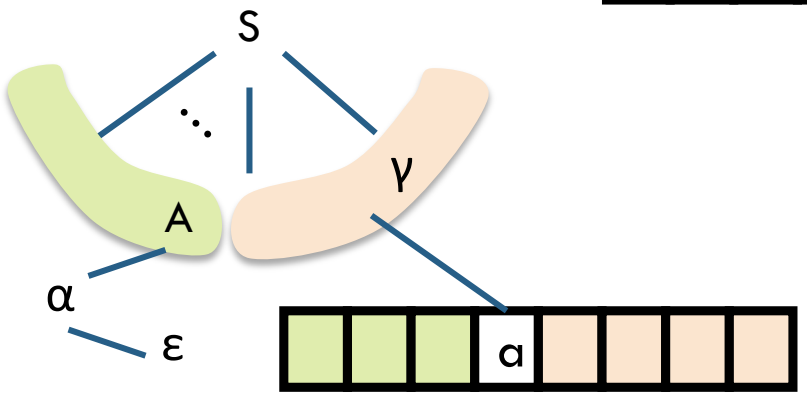


If α can **start** with 'a', e.g., A→aβ, then use rule and derive the rest from βγ



2. **Follow**

If α→*ε and S→*βAaγ', then use rule and derive the rest from aγ'



First and Follow sets: Definition

Def.: Given a CFG $G = (N,T,P,S)$ and a string $\alpha \in (N \cup T)^*$ the **first set** is defined as $FIRST(\alpha) = \{a \in T;\ \alpha \rightarrow^* a\beta,\ \beta \in (N \cup T)^*\} \cup \{\varepsilon \mid \alpha \rightarrow^* \varepsilon\}$

Def.: Given a CFG $G = (N,T,P,S)$ and non-terminal $A \in N$ the **follow set** is $FOLLOW(A) = \{a \in T;\ S \rightarrow^* \beta A a \gamma,\ \beta, \gamma \in (N \cup T)^*\}$

First and Follow sets: Example

$$FIRST(\alpha) = \{a \in T;\ \alpha \rightarrow^* a\beta,\ \beta \in (N \cup T)^*\} \cup \{\varepsilon \mid \alpha \rightarrow^* \varepsilon\}$$
$$FOLLOW(A) = \{a \in T;\ S \rightarrow^* \beta A a \gamma,\ \beta, \gamma \in (N \cup T)^*\}$$

S → uBDz
B → Bv
B → w
D → EF
E → y
E → ε
F → x
F → ε

FIRST(uBDz) = { u }
FIRST(Bv) = { w }
FIRST(w) = { w }
FIRST(EF) = { ε, x, y }
FIRST(y) = { y }
FIRST(x) = { x }

FIRST(α) for all right-hand sides

FOLLOW(D) = { z }
FOLLOW(B) = ??

Easy in "Bv", but in "uBDz"
FOLLOW(B) ⊃ FIRST(Dz)-{ε}

First and Follow sets: Example

$$FIRST(\alpha) = \{a \in T;\ \alpha \rightarrow^* a\beta,\ \beta \in (N \cup T)^*\} \cup \{\varepsilon \mid \alpha \rightarrow^* \varepsilon\}$$
$$FOLLOW(A) = \{a \in T;\ S \rightarrow^* \beta A a \gamma,\ \beta, \gamma \in (N \cup T)^*\}$$

S → uBDz
B → Bv
B → w
D → EF
E → y
E → ε
F → x
F → ε

FIRST(uBDz) = { u }
FIRST(Bv) = { w }
FIRST(EF) = { ε, x, y }

FIRST(a∈ T) = { a }
FIRST(D) = {ε, x, y }
FIRST(B) = { w }
FIRST(E) = { ε, y }
FIRST(F) = { ε, x }

FOLLOW(D) = { z }
FOLLOW(B) = ??

Easy in "Bv", but in "uBDz"
FOLLOW(B) ⊃ FIRST(Dz)-{ε}

FIRST(D) = FIRST(EF)

First and Follow sets: Example

$$FIRST(\alpha) = \{a \in T;\ \alpha \rightarrow^* a\beta,\ \beta \in (N \cup T)^*\} \cup \{\varepsilon \mid \alpha \rightarrow^* \varepsilon\}$$
$$FOLLOW(A) = \{a \in T;\ S \rightarrow^* \beta A a \gamma,\ \beta, \gamma \in (N \cup T)^*\}$$

S → uBDz
B → Bv
B → w
D → EF
E → y
E → ε
F → x
F → ε

FIRST(uBDz) = { u }
FIRST(Bv) = { w }
FIRST(EF) = { ε, x, y }

FIRST(a∈ T) = { a }
FIRST(D) = {ε, x, y }
FIRST(B) = { w }
FIRST(E) = { ε, y }
FIRST(F) = { ε, x }

FOLLOW(D) = { z }
FOLLOW(B) = **{ v, x, y, z }**

Easy in "Bv", but in "uBDz"
FOLLOW(B) ⊃ FIRST(Dz)-{ε}

FIRST(D) = FIRST(EF)

First and Follow sets: Example

$$FIRST(\alpha) = \{a \in T;\ \alpha \rightarrow^* a\beta,\ \beta \in (N \cup T)^*\} \cup \{\varepsilon \mid \alpha \rightarrow^* \varepsilon\}$$
$$FOLLOW(A) = \{a \in T;\ S \rightarrow^* \beta A a \gamma,\ \beta, \gamma \in (N \cup T)^*\}$$

S → uBDz
B → Bv
B → w
D → EF
E → y
E → ε
F → x
F → ε

FIRST(uBDz) = { u }
FIRST(Bv) = { w }
FIRST(EF) = { ε, x, y }

FIRST(a∈ T) = { a }
FIRST(D) = {ε, x, y }
FIRST(B) = { w }
FIRST(E) = { ε, y }
FIRST(F) = { ε, x }

FOLLOW(D) = { z }
FOLLOW(B) = { v, x, y, z }
FOLLOW(E) =
FIRST(F)-{ε} ∪ FOLLOW(D)

Since F →*ε

First and Follow sets: Example

$$FIRST(\alpha) = \{a \in T;\ \alpha \rightarrow^* a\beta,\ \beta \in (N \cup T)^*\} \cup \{\varepsilon \mid \alpha \rightarrow^* \varepsilon\}$$
$$FOLLOW(A) = \{a \in T;\ S \rightarrow^* \beta A a \gamma,\ \beta, \gamma \in (N \cup T)^*\}$$

S → uBDz
B → Bv
B → w
D → EF
E → y
E → ε
F → x
F → ε

FIRST(uBDz) = { u }
FIRST(Bv) = { w }
FIRST(EF) = { ε, x, y }

FIRST(a∈ T) = { a }
FIRST(D) = {ε, x, y }
FIRST(B) = { w }
FIRST(E) = { ε, y }
FIRST(F) = { ε, x }

FOLLOW(D) = { z }
FOLLOW(B) = { v, x, y, z }
FOLLOW(E) = { x, z }
FOLLOW(F) = { z }
FOLLOW(S) = { $ }

Help symbol: End of string (EOF)

First and Follow sets: Systematic

- FIRST(X): for $X \in T$, FIRST(X) = {X}, for $X \in N$ and $X \rightarrow \alpha$:
  - If $\alpha \rightarrow^* \varepsilon$, **then $\varepsilon \in$ FIRST(X)**
  - If $\alpha = Y_1 Y_2 ... Y_n$ and there is a $k \in \{1,...,n-1\}$ for which $Y_i \rightarrow^* \varepsilon$ for all $i \in \{1,...,k\}$, **then (FIRST(Y_i) – {ε}) ⊂ FIRST(X), for j∈ {1,...,k+1}**

- FOLLOW(X): for $X \in N$,
  - Look at every rule $A \rightarrow \alpha X \beta$ where A is reachable from S:
    - **(FIRST(β) – {ε}) ⊂ FOLLOW(X)**
    - If $\beta \rightarrow^* \varepsilon$, then **FOLLOW(A) ⊂ FOLLOW(X)**
  - $ ∈ FOLLOW(S)

Table-based predictive parser

- Given a CFG G = (N,T,P,S), build a table M[A ∈ N, a ∈ T] = r ∈ P
  - Given the leftmost non-terminal A and look-ahead 'a' use rule 'r'
- Construction: Entry for M[A, a]
  - Entry r: A → α if a ∈ FIRST(α)
  - Entry r: A → α with α →* ε if a ∈ FOLLOW(A)
  - Entry is error otherwise

Def.: A so constructed parser is called **LL(1) parser** – it parses the input from **L**eft to right, performing a **L**eftmost derivation with **1** look-ahead token

Table-based predictive parser: Example

S → uBDz
B → Bv | w
D → EF
E → y | ε
F → x | ε

FIRST(uBDz) = { u }
FIRST(Bv) = { w }
FIRST(EF)/FIRST(D) = { ε, x, y }
FIRST(a∈ T) = { a }

FOLLOW(D) = { z }
FOLLOW(E) = { x , z }
FOLLOW(F) = { z }
FOLLOW(B) = { x, y, v, z }
FOLLOW(S) = { $ }

M	u	v	w	x	y	z
D				D → EF	D → EF	D → EF
E				E → ε	E → y	E → ε
F				F → x		F → ε
B			B → Bv B → w			
S	S → uBDz					

Table-based predictive parser: Example

S → uBDz
B → Bv | w
D → EF
E → y | ε
F → x | ε

M	u	v	w	x	y	z
D				D → EF	D → EF	D → EF
E				E → ε	E → y	E → ε
F				F → x		F → ε
B			B → Bv B → w			
S	S → uBDz					

LL(1) parser not enough for this grammar!
Reason: Left-recursion
"Homework": Rewrite grammar and build table again

LL(k) grammars

Def.: A CFG G is in the class LL(1), denoted $G \in LL(1)$ if the parse table has at most one entry per cell

- The concept of LL(1) can be extended to k look-ahead tokens: LL(k)
  - The table grows exponentially with k ($|T|^k$)
  - The more tokens, the more expressive the grammar, i.e.,

$$LL(0) \subset LL(1) \subset \ldots \subset LL(k) \ldots \subset \text{unambiguous grammars} \subset \text{all grammars}$$

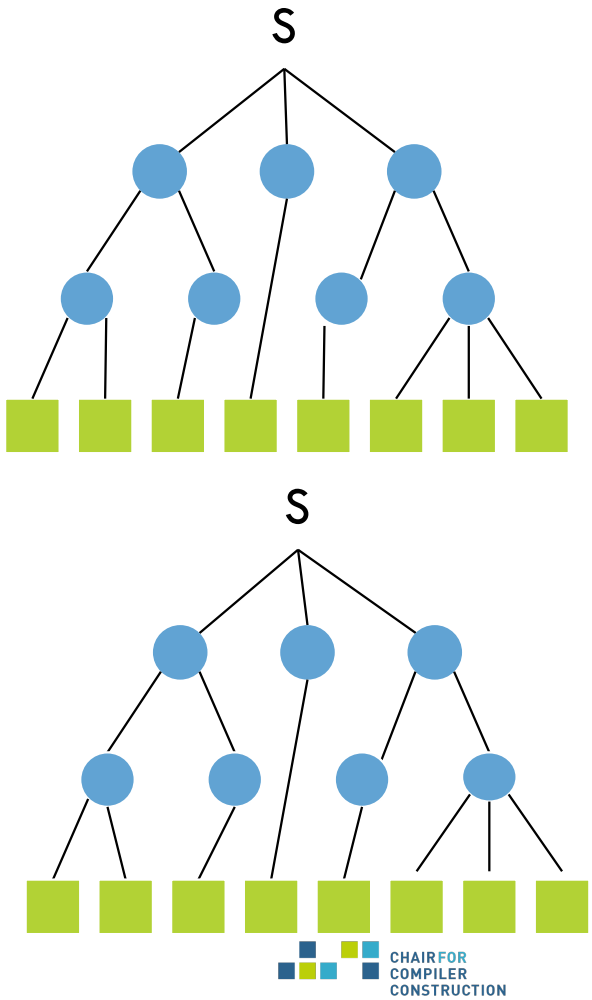
What kind of language would this be?

3. Syntax Analysis

- Introduction
- Context free grammars
- Top-down parser
- Bottom-up parser
- Error handling
- Parser generation with yacc

Top-down vs. Bottom-up

- Top-down
  - Start from the root and grow toward the leaves
  - Pick a rule and try to match input (backtrack on mismatch)
- Bottom-up
  - Start from the leaves and grow toward the root
  - Try to match right-hand side of rules
  - Finish once the start symbol is reached



LR: Inverse Left-to-right Rightmost derivation

- Why the name? – Example

CFG:

S → aABe

A → Abc | b

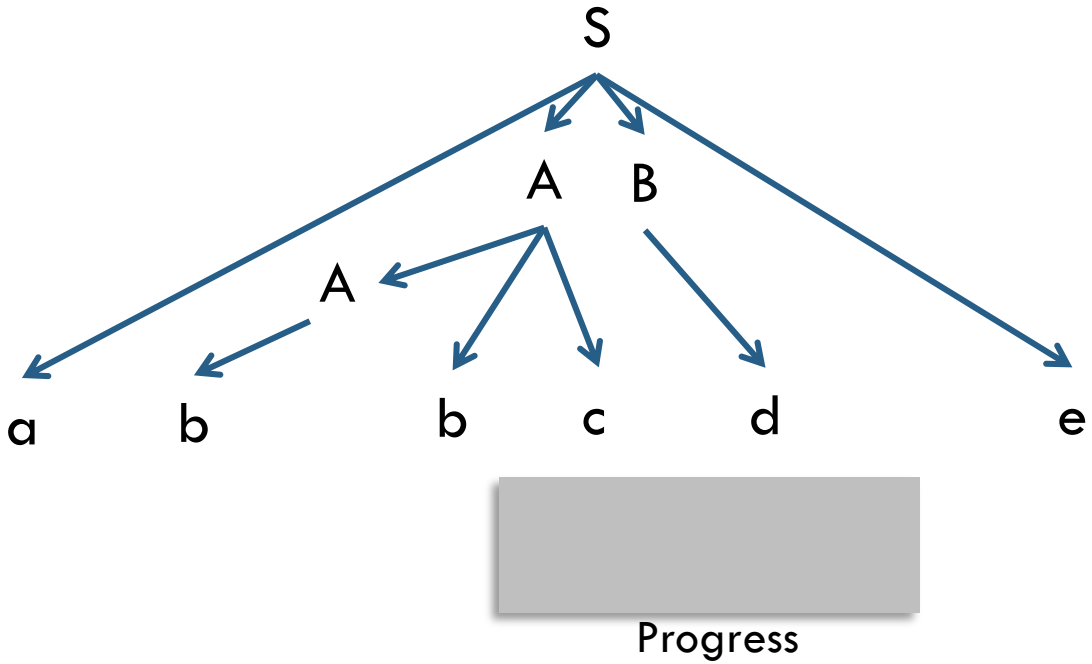
B → d

A → b

A → Abc

B → d

S → aABe

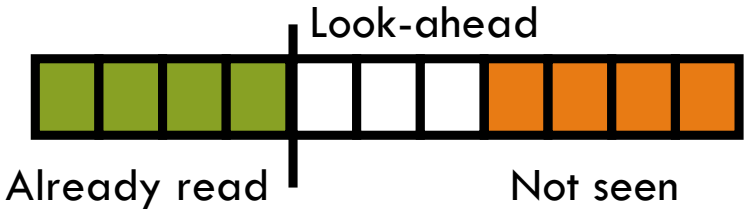


Input:

a	b	b	c	d	e

Construction idea

- The string is split into three parts

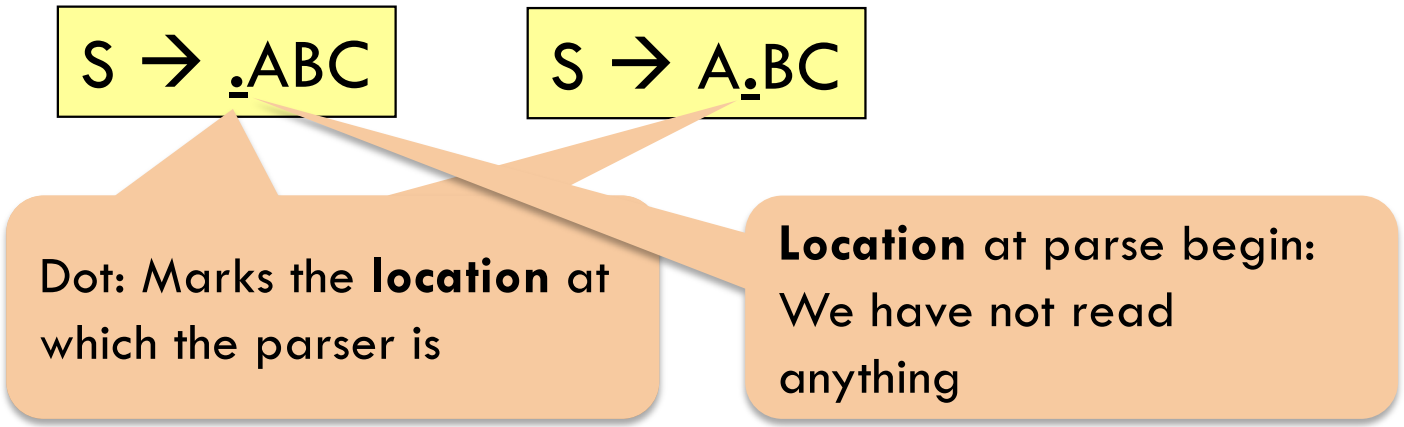


- "Already read": Encode as state
  - Partial derivation (recall from example aAbcde)
  - Options to continue the current parsing (e.g., tokens expected to complete a rule)
- Look-ahead + "Not seen": String made only of terminals

- Parser (main) actions: **Shift/Reduce**
  - **Shift**: No rule completed ➔ Move terminals from right to the left
  - **Reduce**: Rule A→α completed ➔ Replace partial sentential form on the left with A

Construction idea: Encoding the state

- Idea
  - Do not store the entire partial sentential form
  - Use a stack of partial derivations, i.e. using **locations** in the rules
- Example

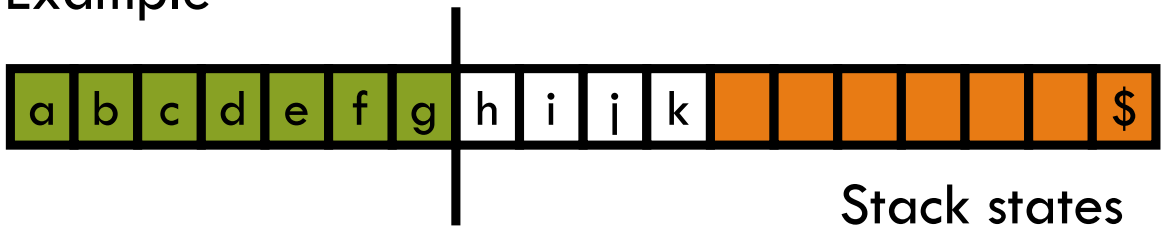


CFG rules

1. S → ABC
2. A → abcd
3. B → efghX
4. B → efgD
5. …

Construction idea: Encoding the state

- Idea
  - Do not store the entire partial sentential form
  - Use a stack of partial derivations, i.e. using **locations** in the rules
- Example



CFG rules

1. S → ABC
2. A → abcd
3. B → efghX
4. B → efgD
5. …

Stack states

B → efg.hX
B → efg.D

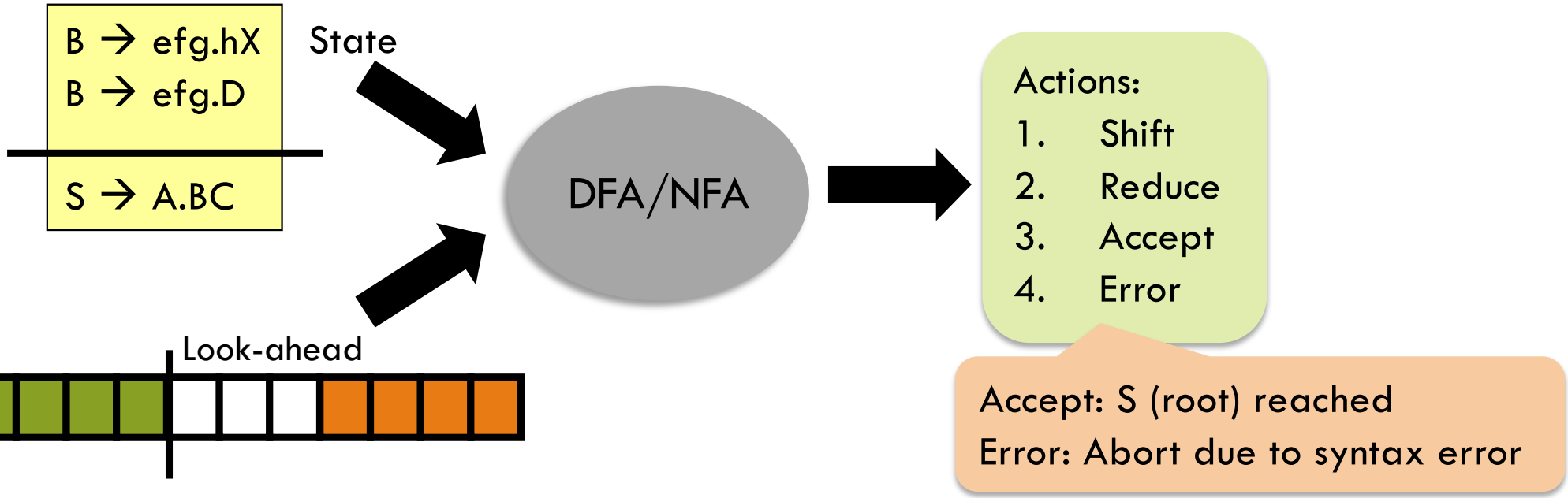
S → A.BC

"Trying" to match right-hand side for B: Read efg, the rest from hX or D

"abcd" can be reduced to A, rest should come from BC

Construction idea: Deciding actions (shift/reduce)

- Idea
  - There are finite productions, and finite locations within them
  - ➔ Use a DFA (or NFA) to track state changes and select **actions**



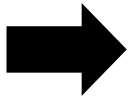
States: LR(0)-Items

Def.: Given a CFG G = (N,T,P,S), an **LR(0)-item** is a rule from P with a special dot added somewhere in the right-hand side

- Example

CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x



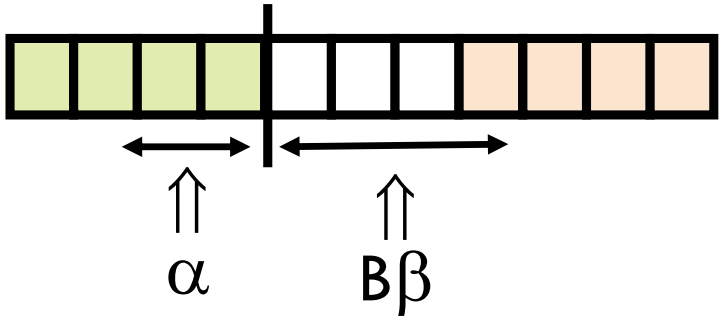
For rule E → T + E

1. [E → .T + E]
2. [E → T. + E]
3. [E → T + .E]
4. [E → T + E.]

Content to the left represents input that has already been read

States: Sets of LR(0)-Items

- Given an LR(0)-item [A → α.Bβ]
  - Part of the stack was derived from α
  - The rest should have as prefix something derivable from Bβ
  - Since B ∈ N ➔ a prefix of the remaining input could also be derived from r_i: B→β_i for all r_i ∈ P
  - ➔ This corresponds to a closure operator



- Example

CFG rules

1. S → E$
2. E → T
3. E → T + E
4. T → x

[S → .E$]
[E → .T + E]
[E → .T]
[T → .x]

Closure for LR(0)-items

Def.: Given a CFG G=(N,T,P,S) and a set I of LR(0)-items, the closure of the set is defined by $cl(I) = I \cup \{[B \rightarrow .\gamma] \mid [A \rightarrow \alpha.B\beta] \in cl(I) \text{ and } B \rightarrow \gamma \in P\}$

- Algorithm

```
Input     I
Output    cl(I) = I
While  cl(I) still changing
   foreach [A → α.Bβ] ∈ cl(I)
      foreach B → γ ∈ P
         cl(I) = cl(I) U {[B → .γ]}
return cl(I)
```

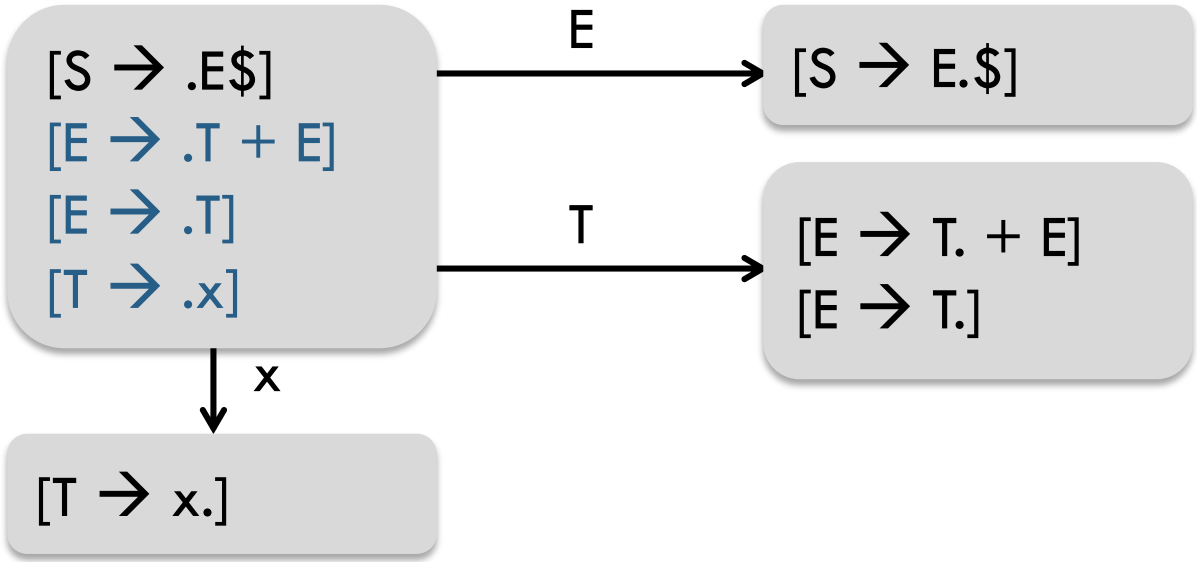
Transitions

Def.: Given a set I of LR(0)-items and a symbol X ∈ (N U T), we define the transition function GOTO(I,X) = cl({ [A→ αX.β] | [A → α.Xβ] ∈ I })

- Intuition: Just move the cursor "." to the right and calculate the closure
- Example

CFG rules

1. S → E$
2. E → T
3. E → T + E
4. T → x



DFA for LR Parser: Example

CFG rules
1. S → E$
2. E → T
3. E → T+E
4. T → x

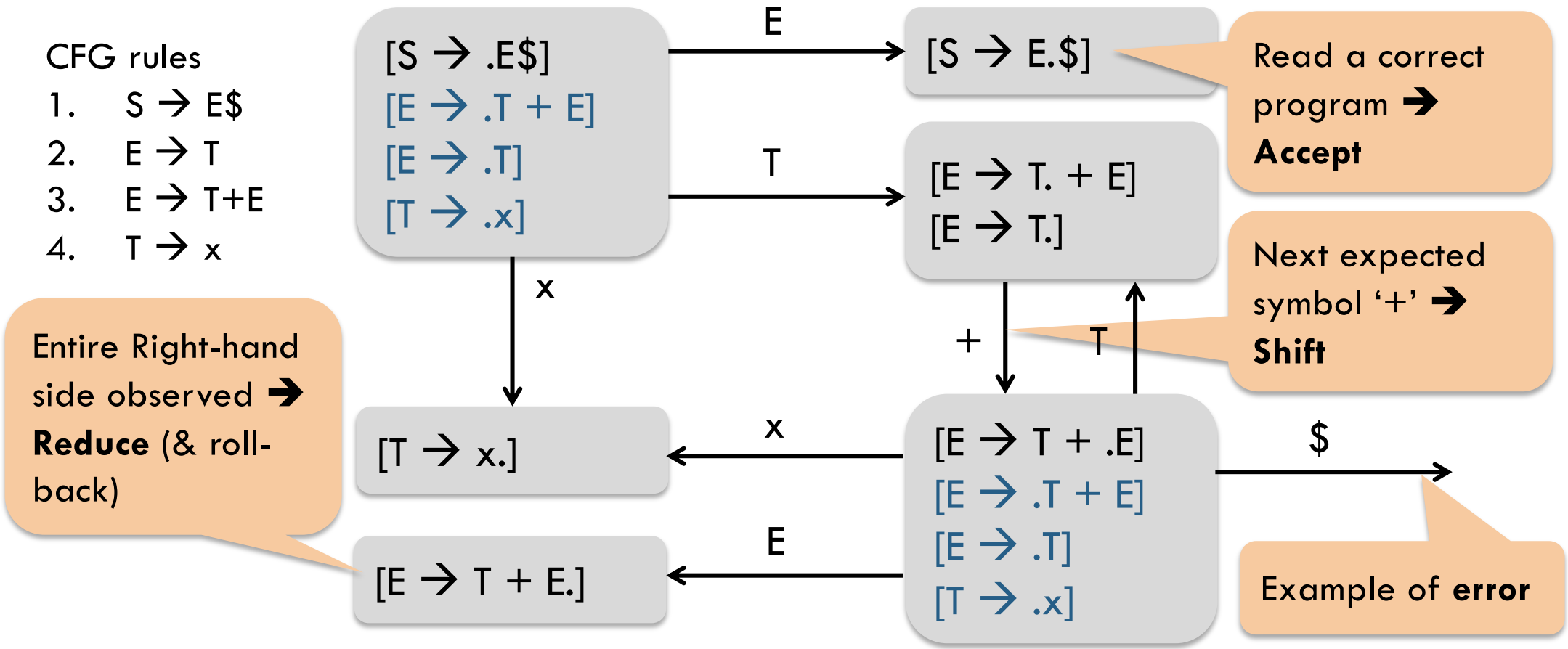


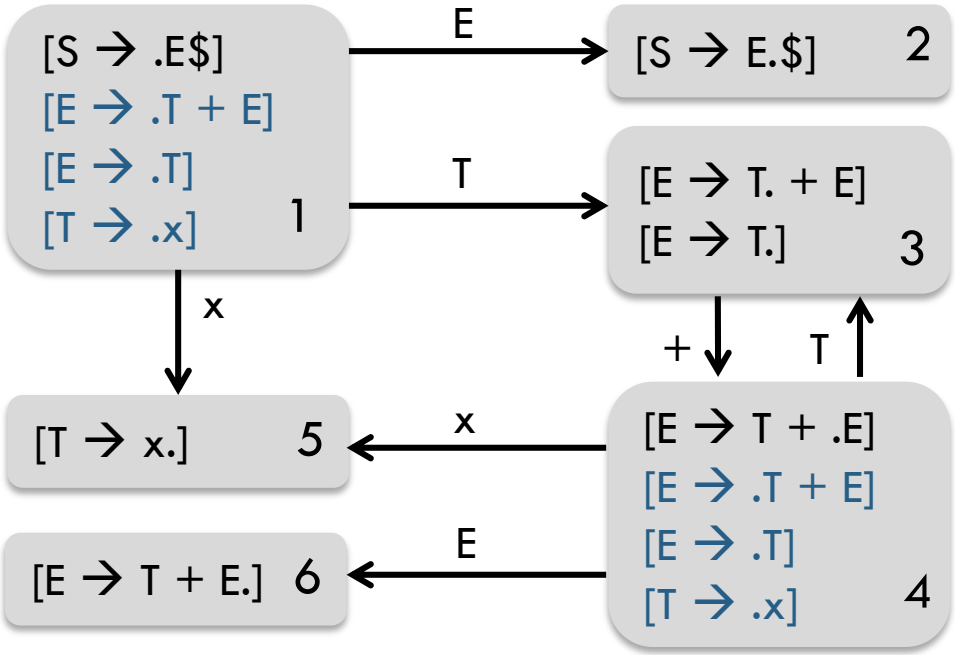
Table-based LR-parser

- Parser implementation through table (similar to LL(k) parsers)
- Given a CFG G = (N,T,P,S), and a transition (I,X,J) of the DFA – I,J: Sets of LR(0)-items & X ∈ (N ∪ T), build table M[I,X] as follows:
  - If X ∈ T — **M[I,X] = shift state J**
  - If X ∈ N — **M[I,X] = goto state J**
  - If [S'→S.$] ∈ I — **M[I,$] = accept**
  - If [A→α.] ∈ I — **M[I,X] = reduce rule n** (with $\mathbf{r_n}$: **A→α** ∈ P)
  - Otherwise — **M[I,X] = error**

LR table: Example

CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

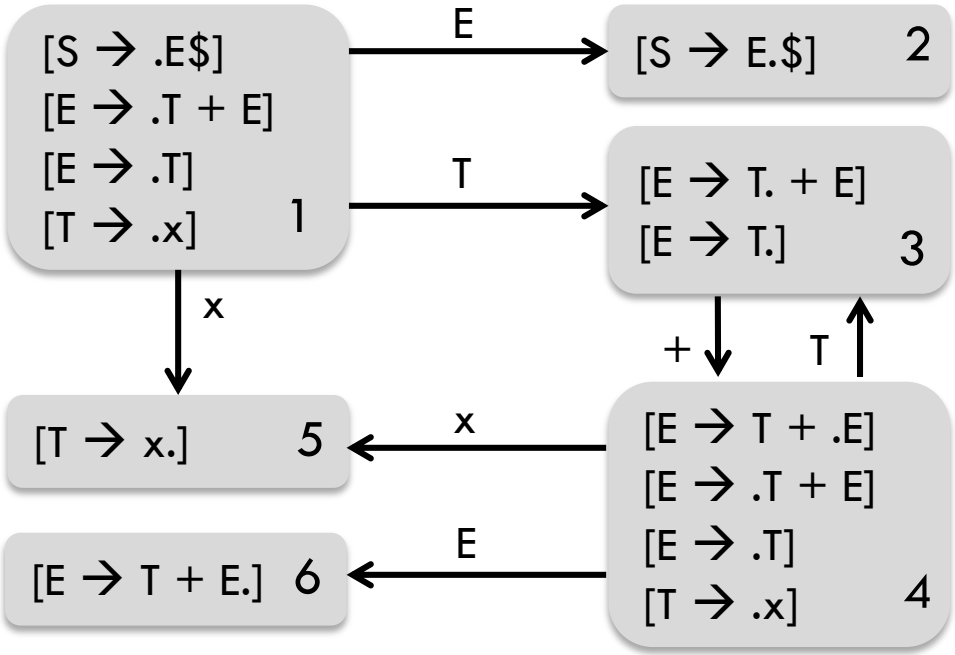


M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		

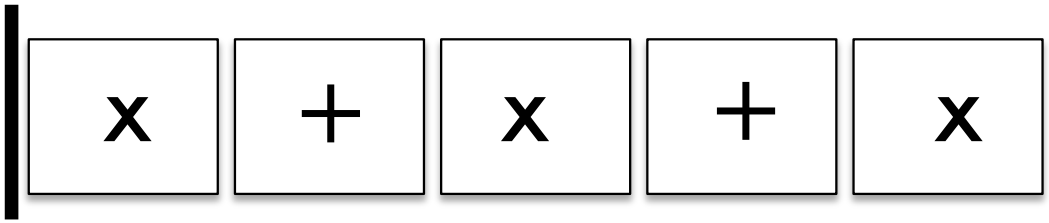
Prefer shift over reduce

s/g n: shift/goto state n
r n: reduce rule n

LR Parser operation: Intuition



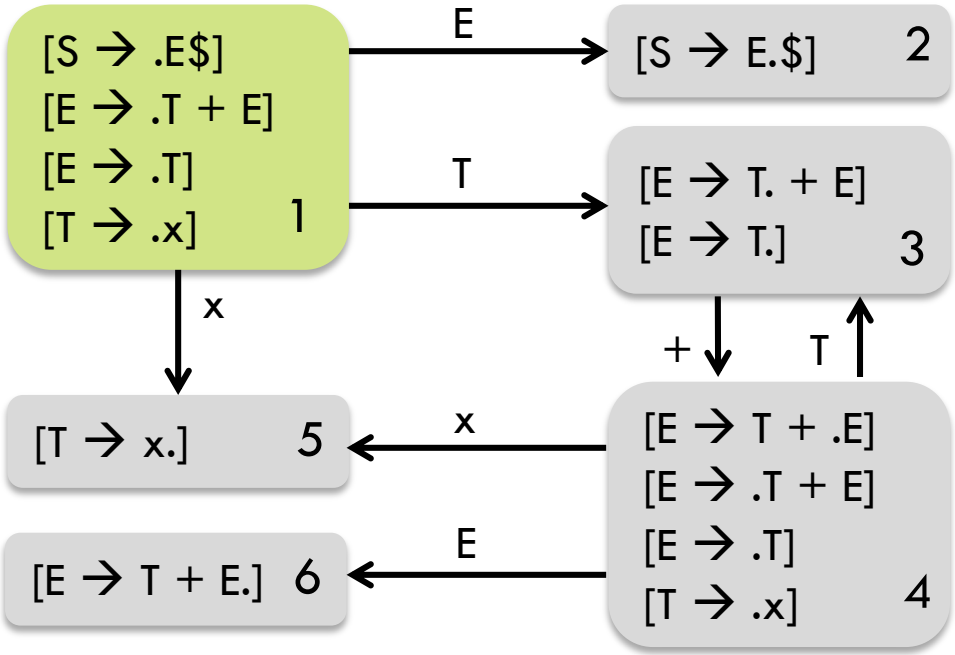
M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		



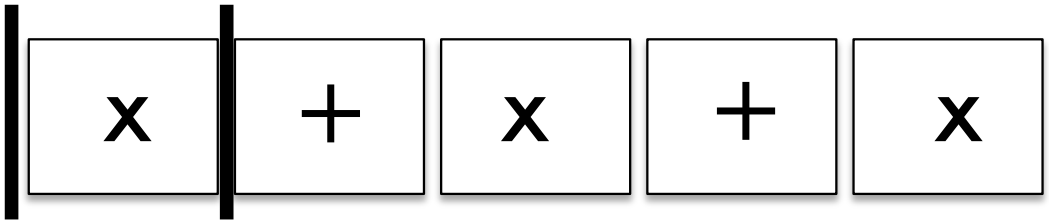
CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

LR Parser operation: Intuition



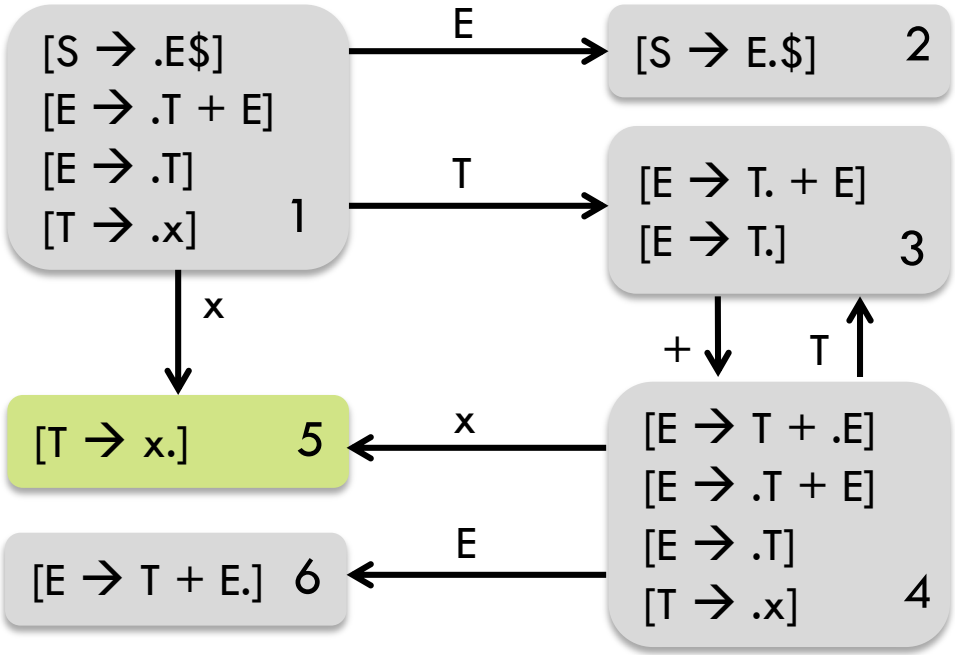
M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		



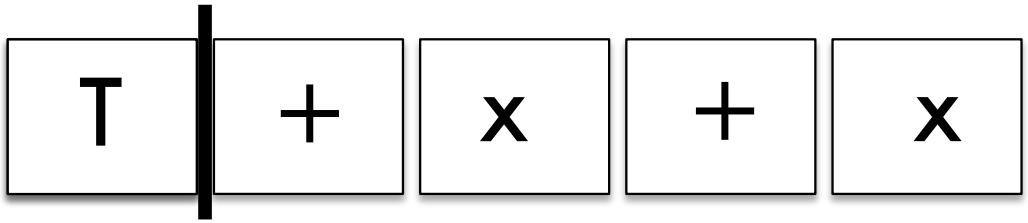
CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

LR Parser operation: Intuition



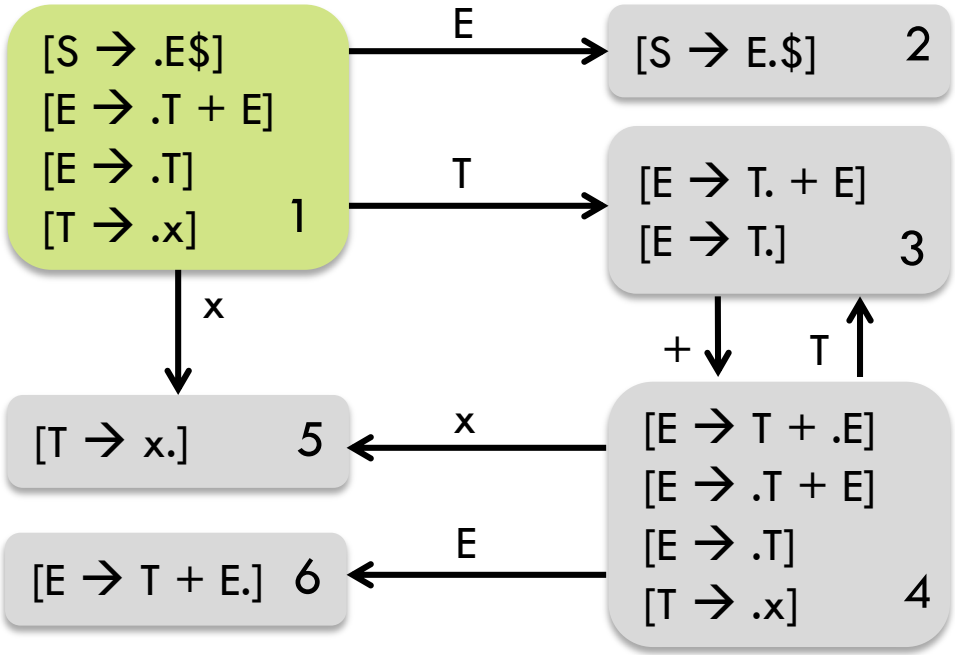
M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		



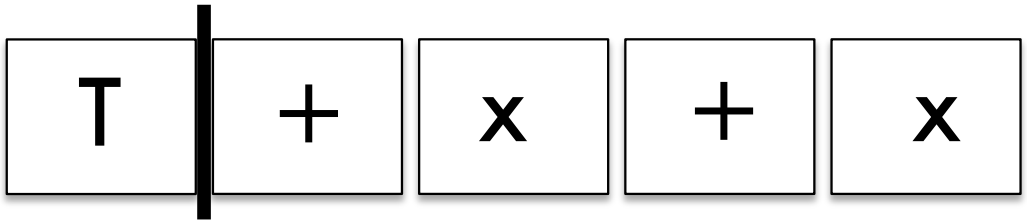
CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

LR Parser operation: Intuition



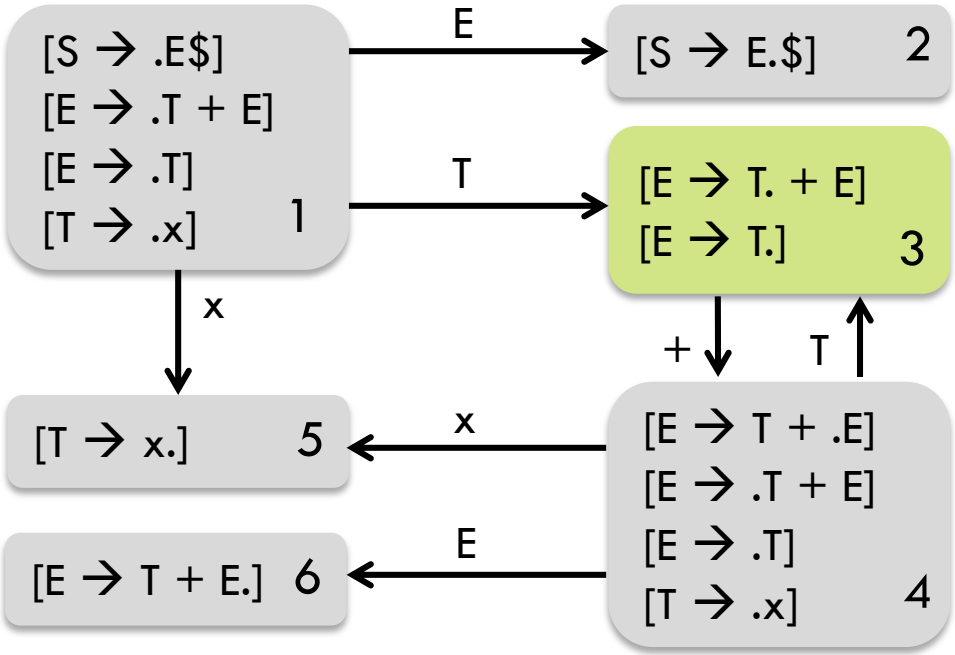
M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		



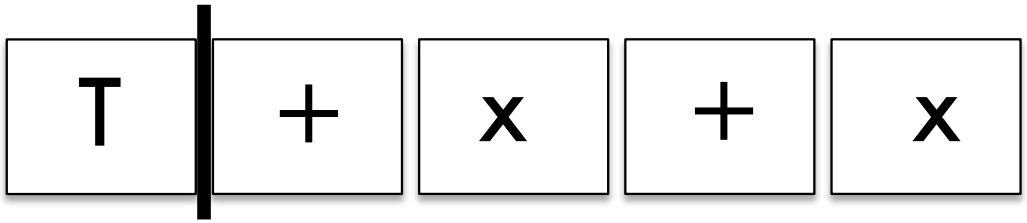
CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

LR Parser operation: Intuition



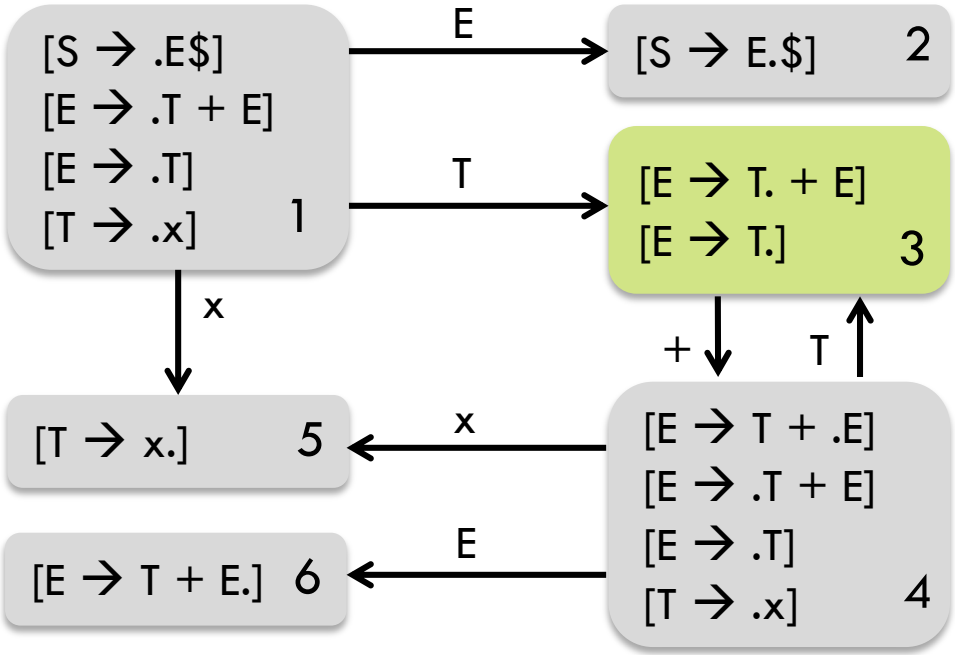
M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		



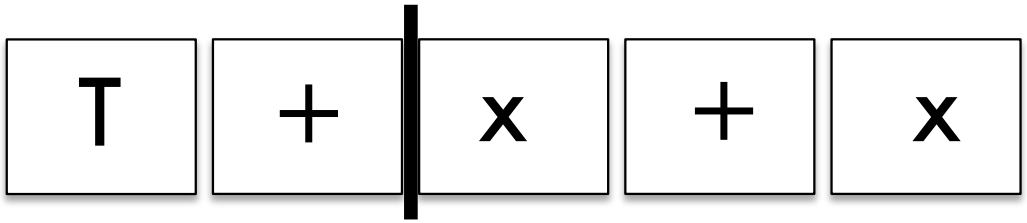
CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

LR Parser operation: Intuition



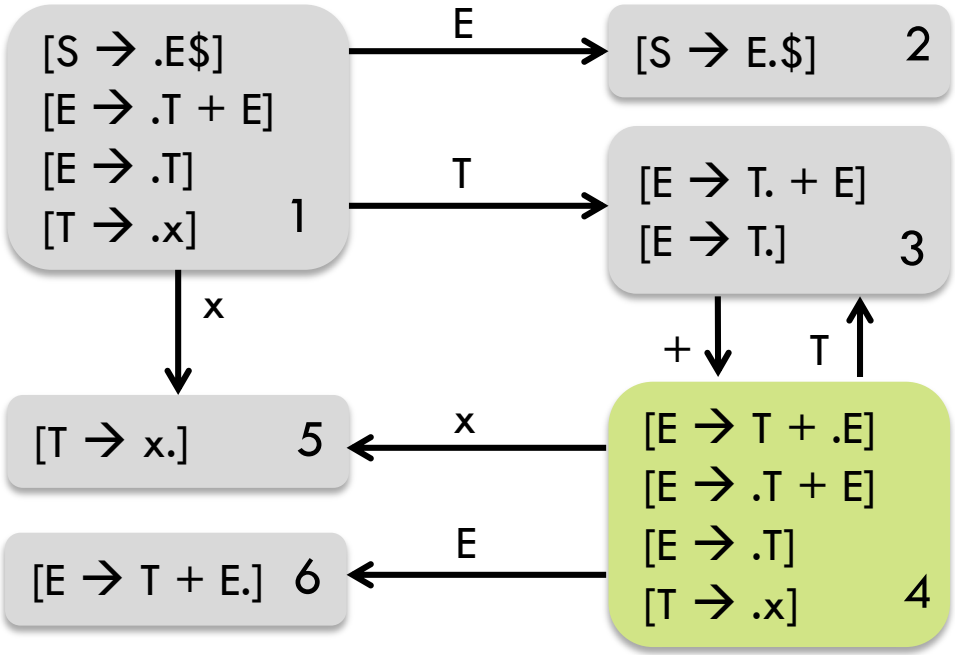
M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		



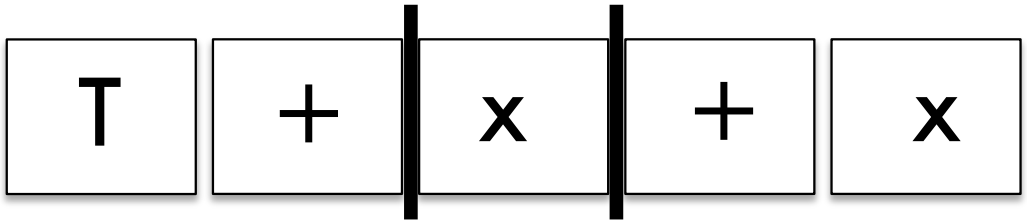
CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

LR Parser operation: Intuition



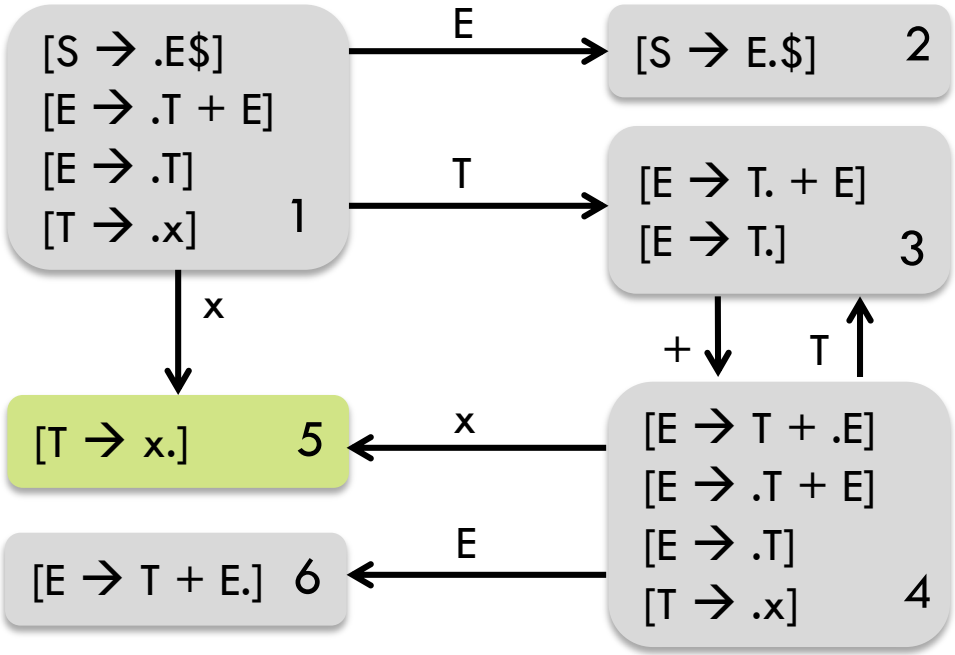
M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		



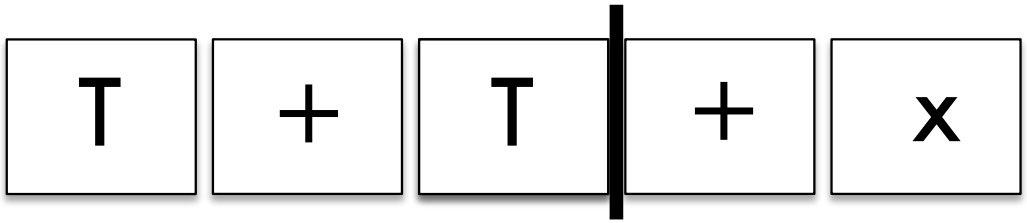
CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

LR Parser operation: Intuition



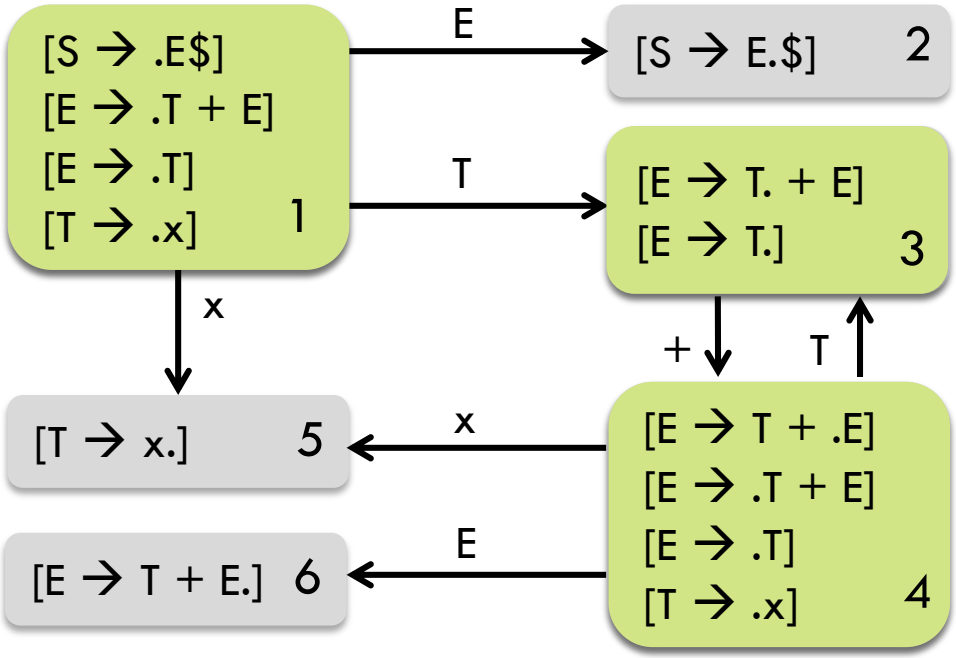
M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		



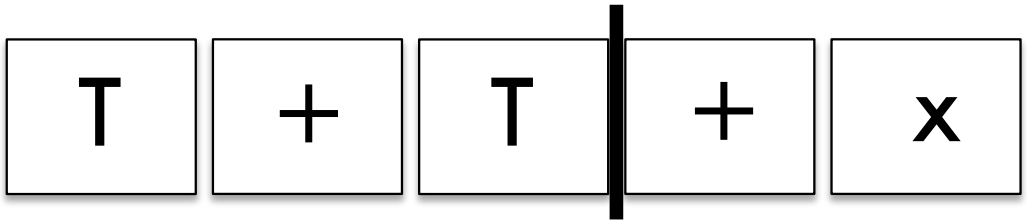
CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

LR Parser operation: Intuition



M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		

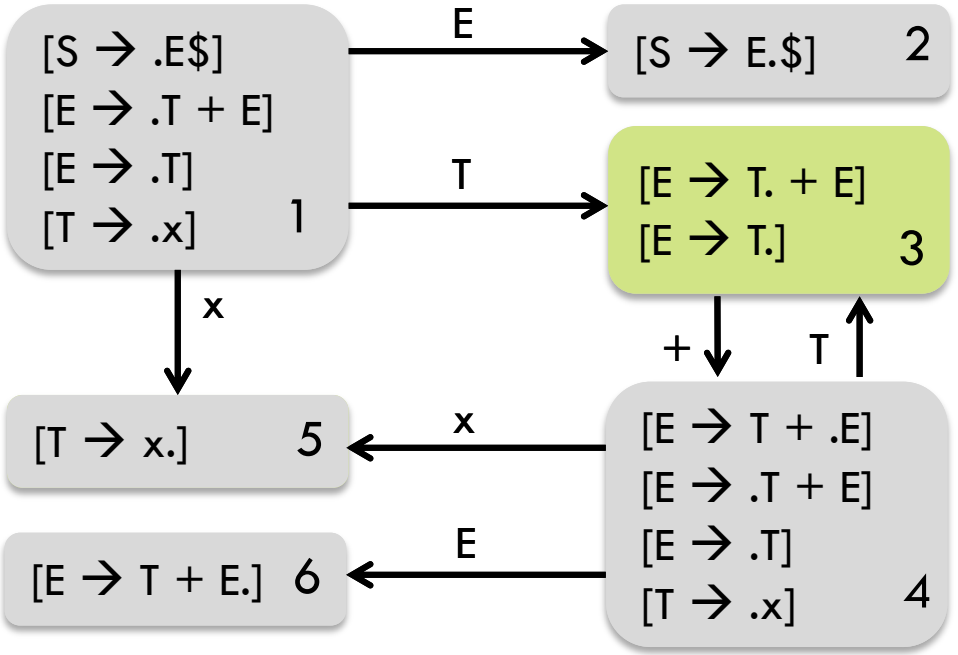


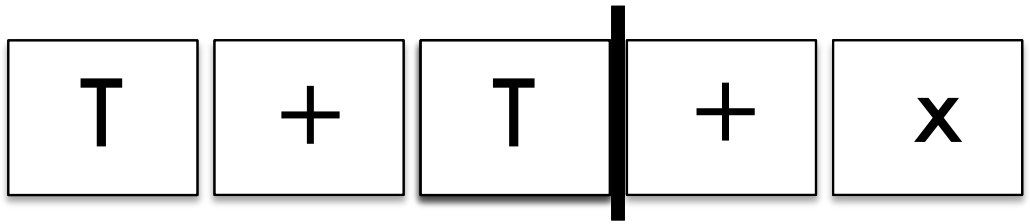
CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

Optimization

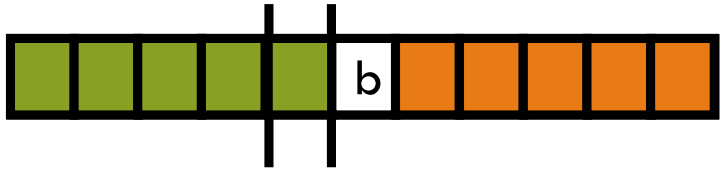
- Idea: Do not restart the automaton after every reduce
  - Keep track of the states of the DFA with a stack
  - On reduce A → α: Go back to the previous state, i.e., roll-back as many transitions as symbols are in α
- Example:
  - Reduce [T → x]
  - Roll back 1 state (back to 4)
  - Apply GOTO(4,T) = 3

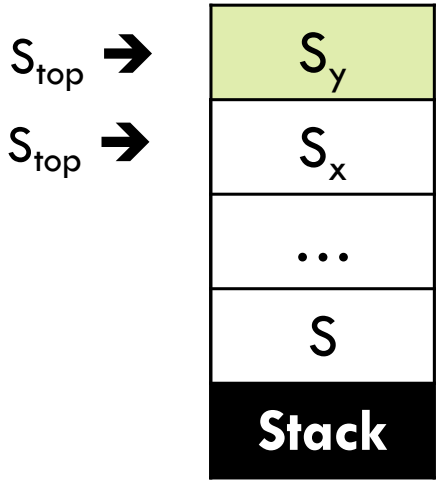




LR operation

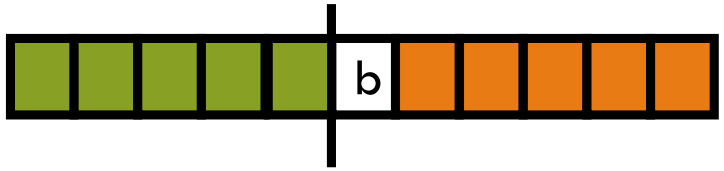
- Init: Push start symbol to the stack
- Repeat until "accept" or "error"
  - Let S_{top} be the top of the stack and 'a' the next input
  - Compute action from table act = M[S_{top},a]
  - If act = Shift S_y
    - Push S_y to stack and read next symbol

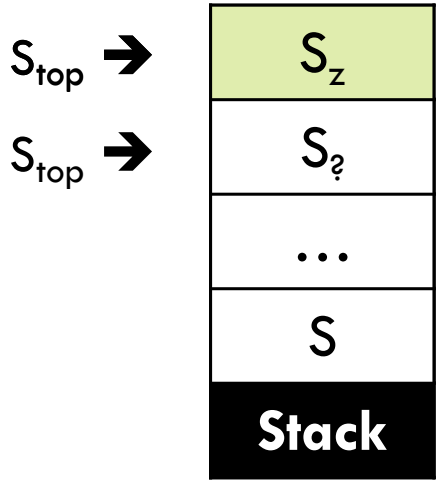




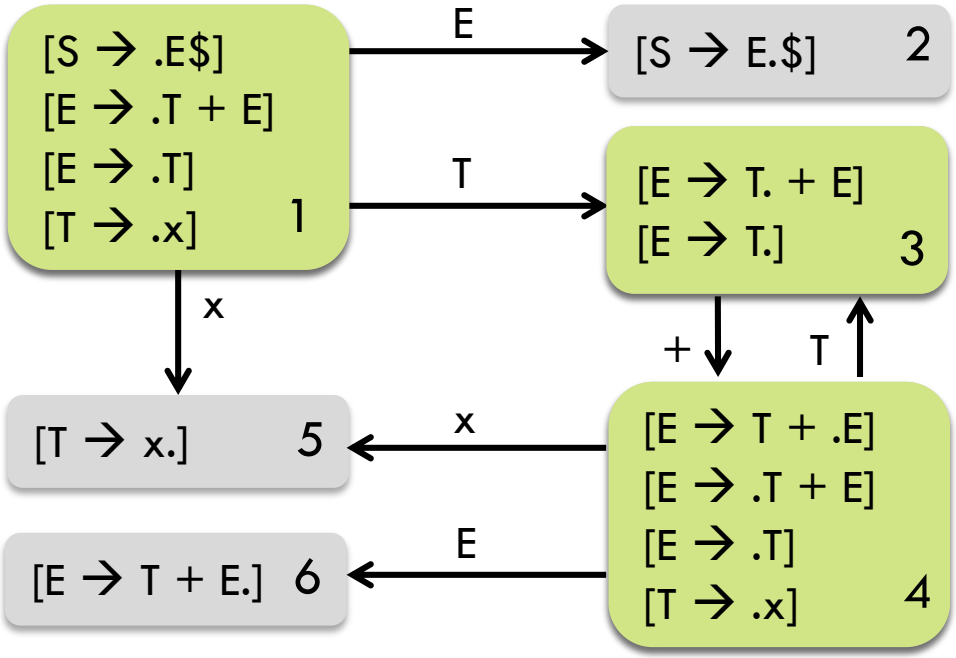
LR operation

- Init: Push start symbol to the stack
- Repeat until "accept" or "error"
  - Let S_{top} be the top of the stack and 'a' the next input
  - Compute action from table act = $M[S_{top},a]$
  - If act = Shift S_y
    - Push S_y to stack and read next symbol
  - If act = Reduce n (with r_n: A → α)
    - Remove $|\alpha|$ states from stack (i.e., roll back)
    - **Do not advance** input cursor
    - Compute new state $M[S_{top}, A]$ = goto S_z, and push to stack
  - If act = error/accept: Report and stop

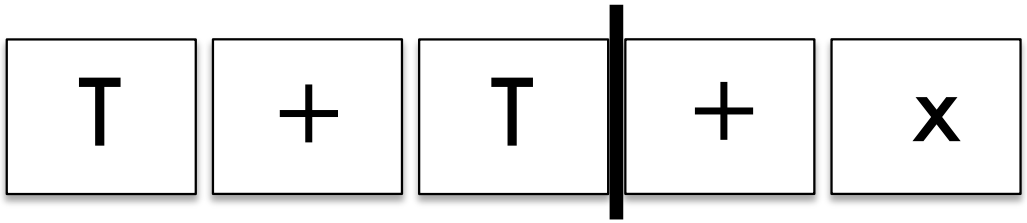




LR operation: Example



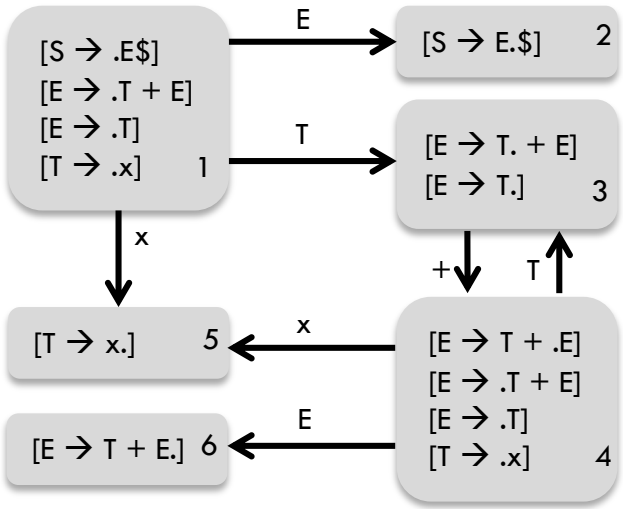
M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		



CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

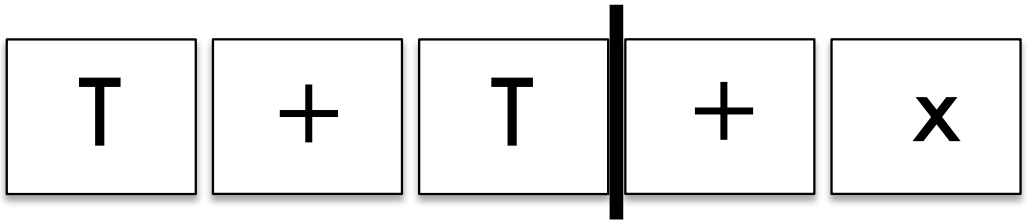
LR operation: Example



S_{top} →

3
4
3
1
Stack

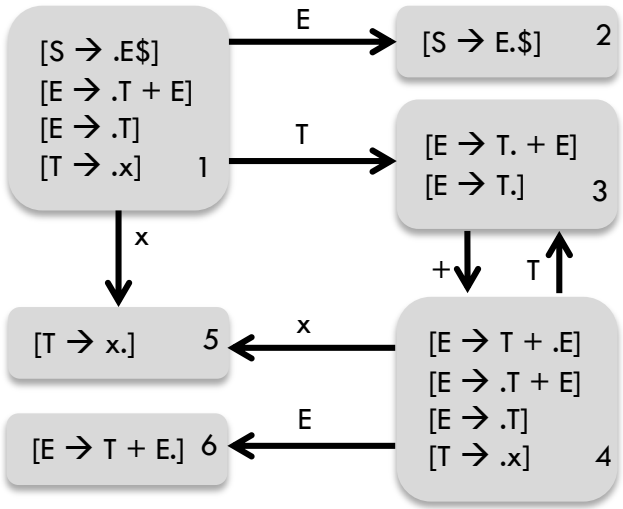
M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		



CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

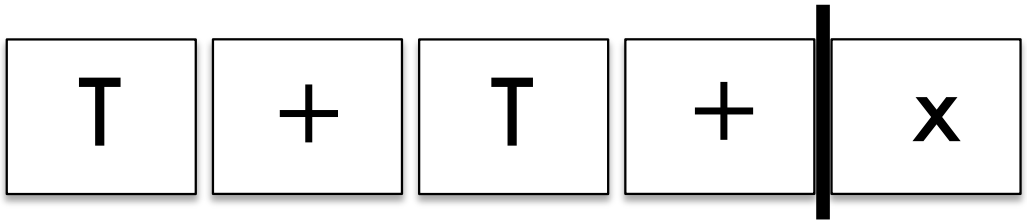
LR operation: Example



S_{top} →

4
3
4
3
1
Stack

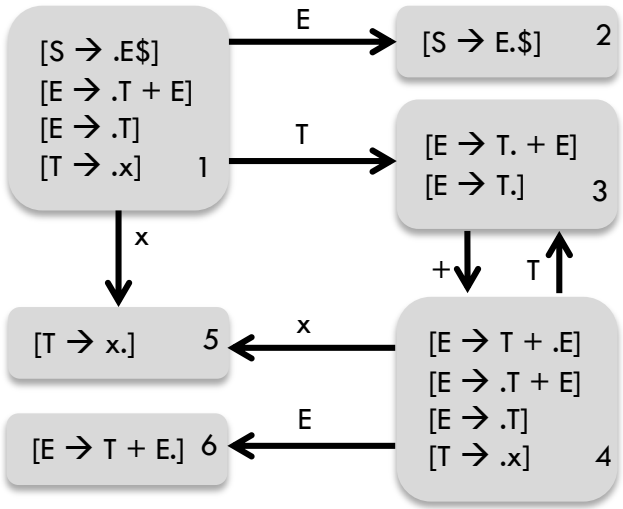
M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		



CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

LR operation: Example



S_{top} →

5
4
3
4
3
1
Stack

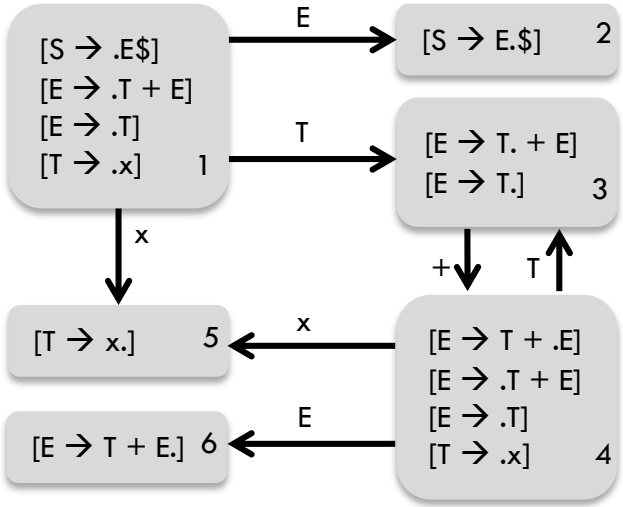
M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		

T	+	T	+	x

CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

LR operation: Example

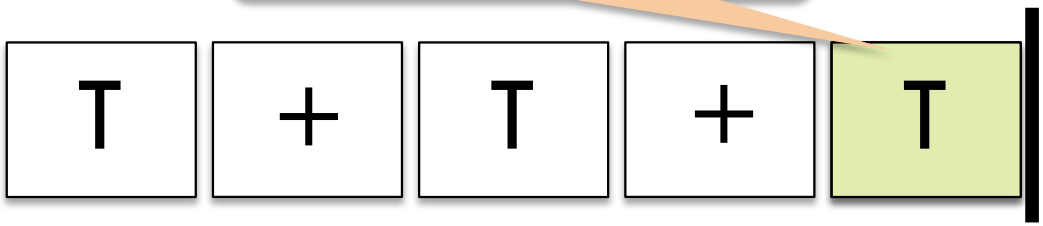


S_{top} →

5
4
3
4
3
1
Stack

M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		

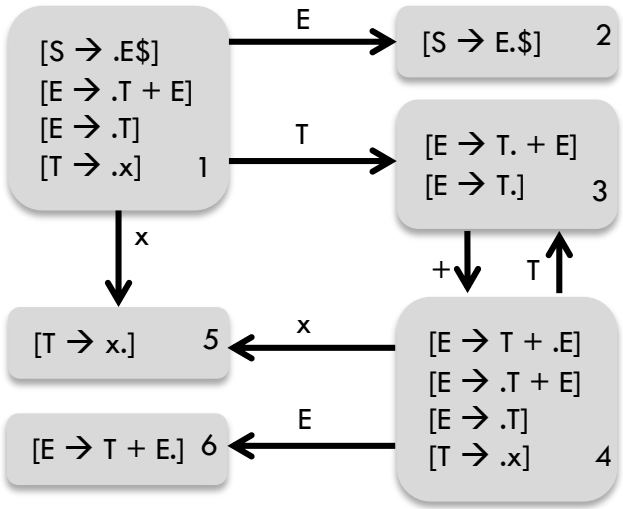
T → x: Pop **one** state



CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

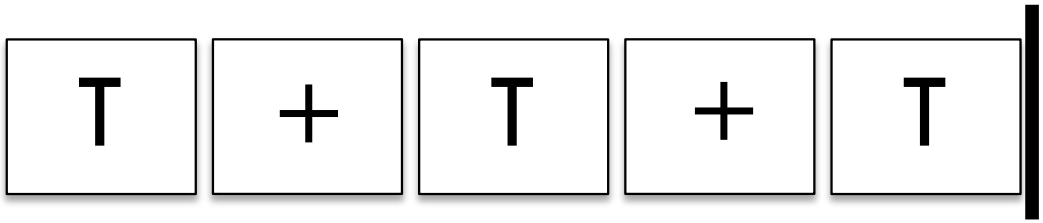
LR operation: Example



S_{top} →

4
3
4
3
1
Stack

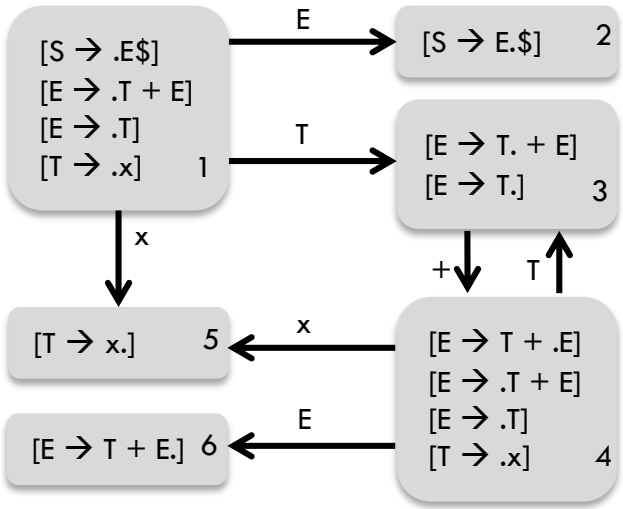
M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		



CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

LR operation: Example

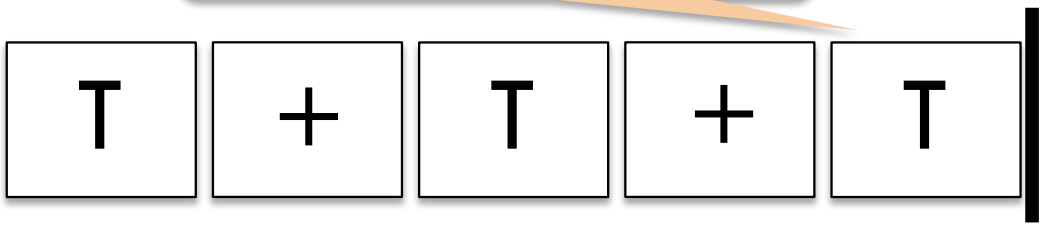


S_{top} →

3
4
3
4
3
1
Stack

M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		

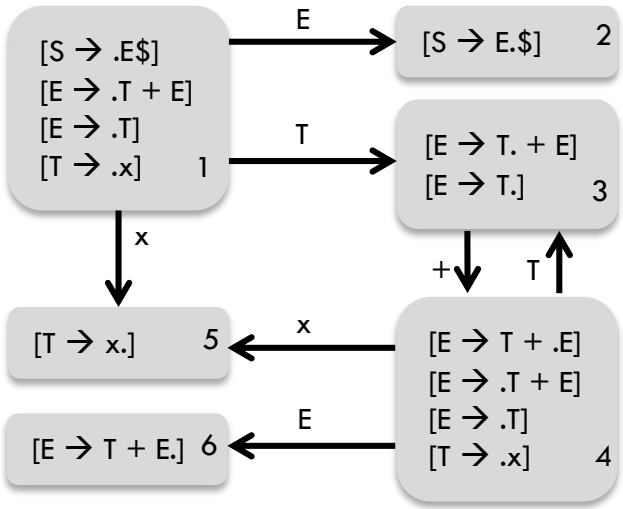
E → T: Pop **one** state



CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

LR operation: Example



S_{top} →

4
3
4
3
1
Stack

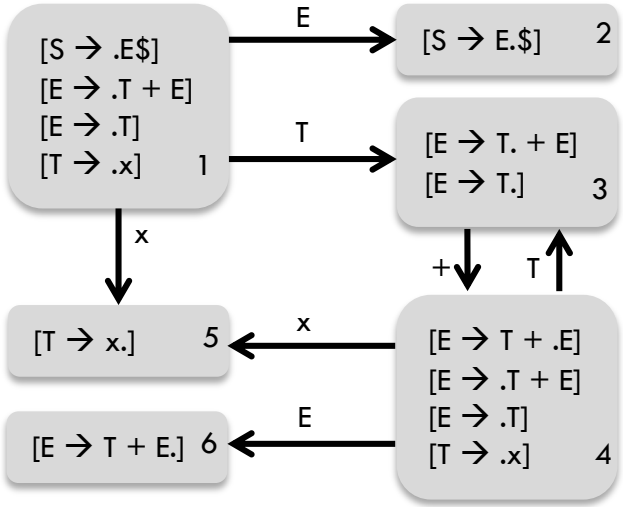
M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		

T	+	T	+	E

CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

LR operation: Example

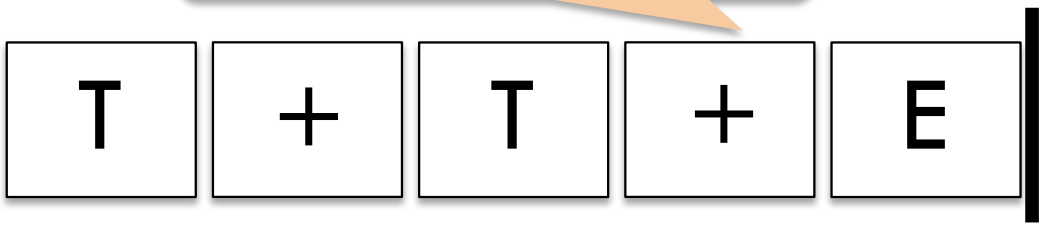


S_{top} →

6
4
3
4
3
1
Stack

M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		

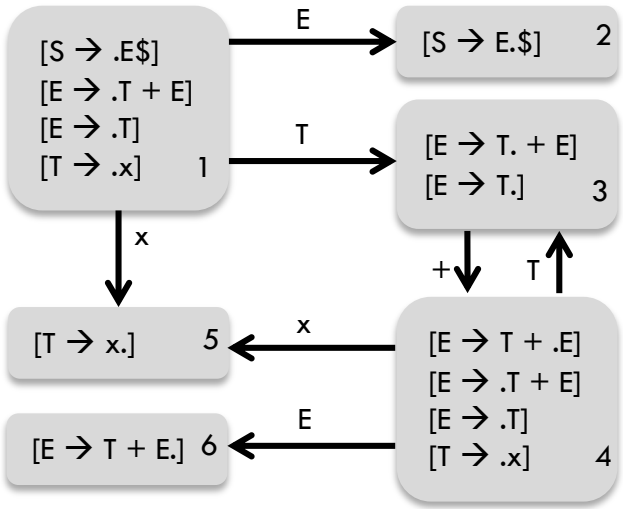
E → T+E: Pop **three** states



CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

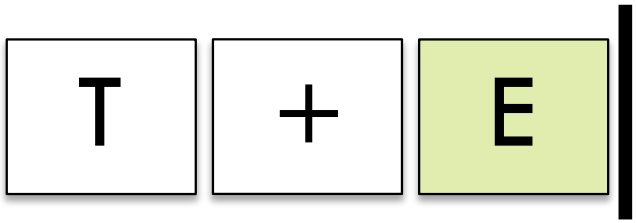
LR operation: Example



S_{top} →

4
3
1
Stack

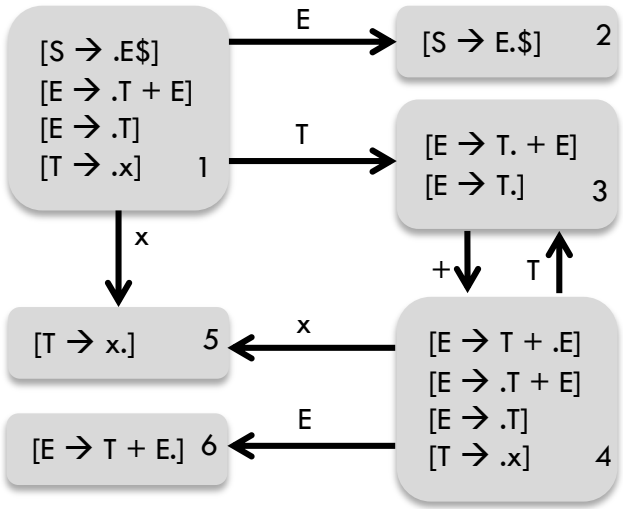
M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		



CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

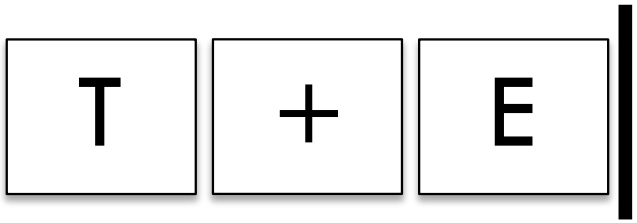
LR operation: Example



S_{top} →

6
4
3
1
Stack

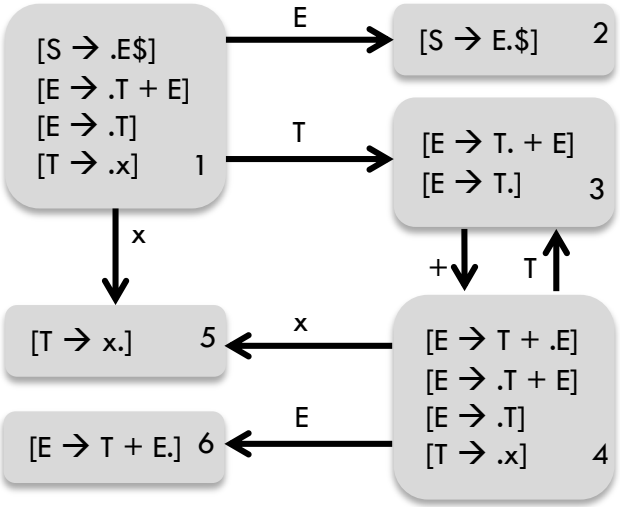
M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		



CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

LR operation: Example



S_{top} →

1
Stack

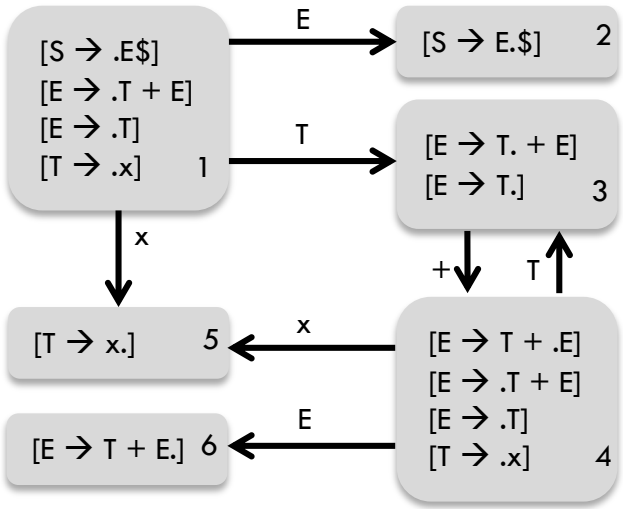
M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		

E

CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

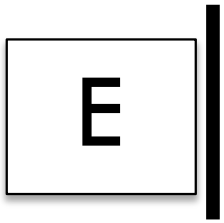
LR operation: Example



S_{top} →

1
Stack

M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		



CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x

LR(k) grammars

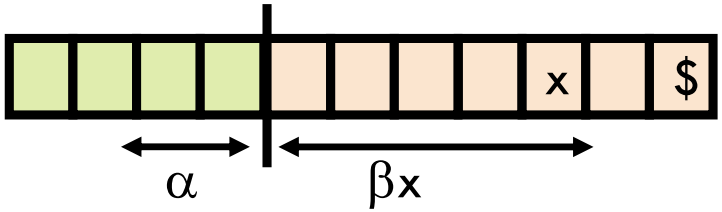
Def.: A CFG G is in the class LR(0), denoted G ∈ LR(0) if the parse table has at most one entry per cell (no conflicts shift/reduce or reduce/reduce)

- From the example…
- Shift-reduce conflicts typically resolved by choosing shift (yacc)
- Reduce-reduce are more complex
- Solution: As with LL(k), use look-ahead tokens LR(k)
  - Table grows exponentially with k
- Why is it call LR(**0**)?

M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4, r2	r2		
4	s5			g6	g3
5	r4		r4		
6					

Would it help to use semi-colon?

LR(k) and other grammars



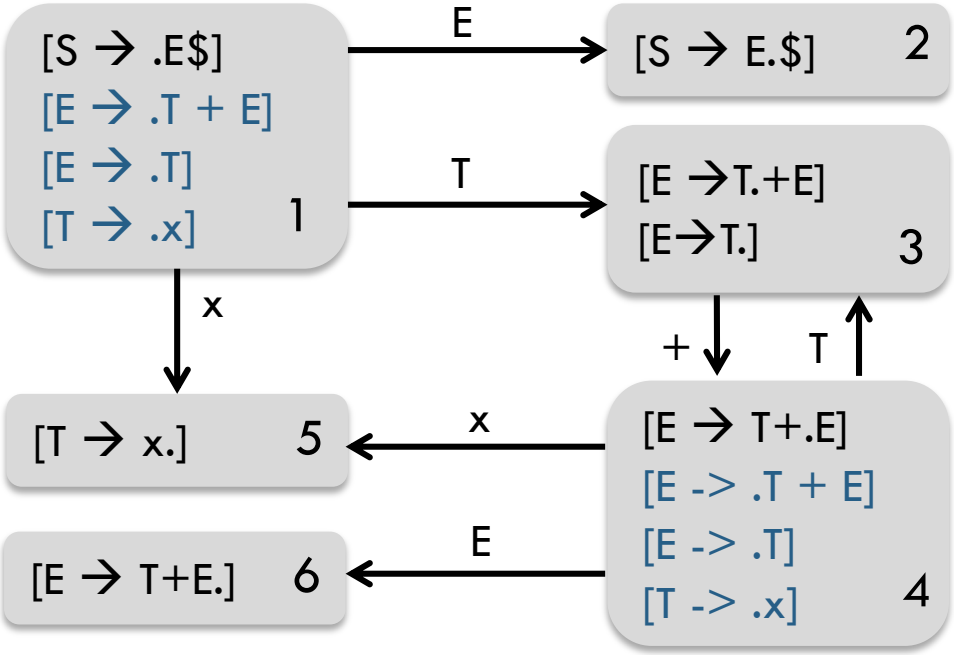
- For practical languages LR(1) is usually enough
  - New definitions: LR(1)-items & GOTO function
  - LR(1)-item are of the form [A → α.β, x]
  - Intuition: Use the look-ahead to disambiguate in case of conflicts
- Complexity reduction
  - SLR: Simple LR
    - Extension to LR(0): Reduce A→α only for a ∈ FOLLOW(A)
  - LALR(1): Look-Ahead(1) LR(0) – (The one used in yacc)
    - Start from LR(1) and merge states with the same **core**

$$\boxed{\begin{array}{l}[A \rightarrow \alpha_1.\beta_1, x]\\ [A \rightarrow \alpha_2.\beta_2, x]\end{array}}\ \boxed{\begin{array}{l}[A \rightarrow \alpha_1.\beta_1, y]\\ [A \rightarrow \alpha_2.\beta_2, y]\end{array}} \Rightarrow \boxed{\begin{array}{l}[A \rightarrow \alpha_1.\beta_1, x/y]\\ [A \rightarrow \alpha_2.\beta_2, x/y]\end{array}}$$

Example: SLR for the example

CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x



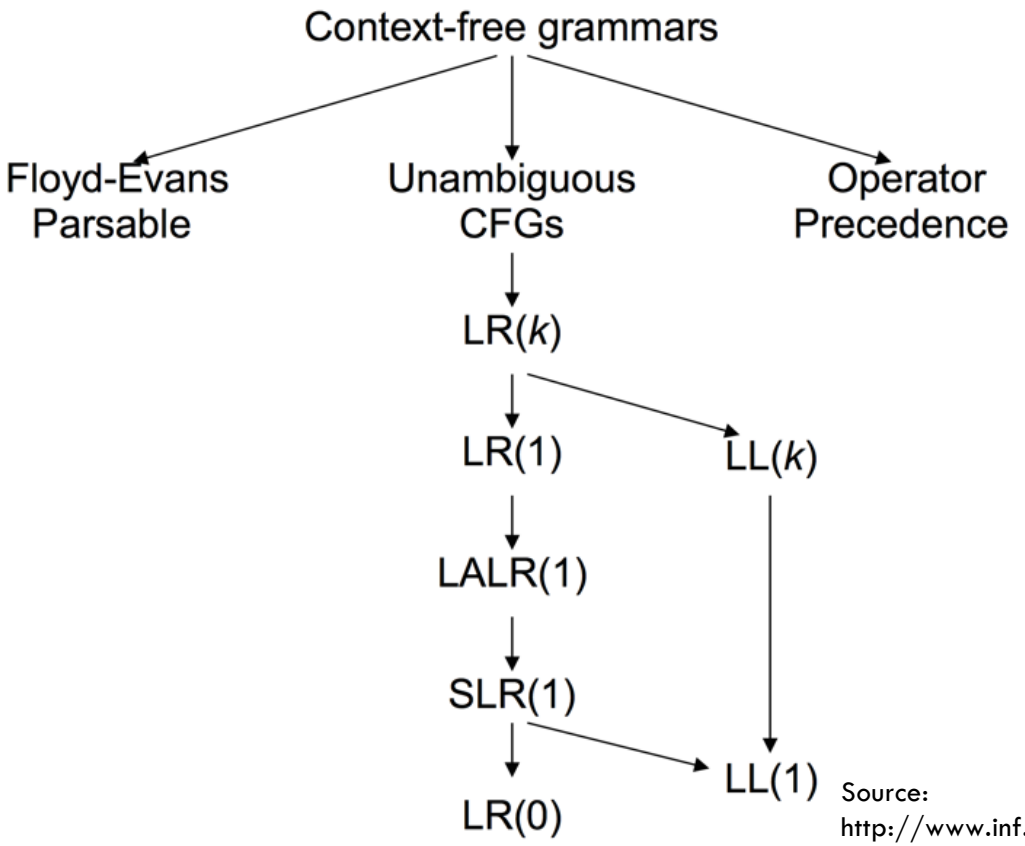
[E →T.+E]
[E→T.] 3

FOLLOW(E) = { $ }
FOLLOW(T) = { +, $ }

M	x	+	$	E	T
1	s5			g2	g3
2			accept		
3	r2	s4 r2	r2		
4	s5			g6	g3
5	r4	r4	r4		
6	r3	r3	r3		

Grammars so far and some others

- Inclusion hierarchy of context free grammars



Source: http://www.inf.ed.ac.uk/teaching/courses/ct/slides/Lecture07.pdf

3. Syntax Analysis

- Introduction
- Context free grammars
- Top-down parser
- Bottom-up parser
- Error handling
- Parser generation with yacc

Error handling so far

M	x	+	$	E	T
1	s5	E	E	g2	g3
2	E	E	**accept**	E	E
3	E	s4	r2	E	E
4	s5	E	E	g6	g3
5	E	r4	r4	E	E
6	E	E	r3	E	E

Error handling so far

- Desired
  - More information about the error
  - Resume after finding one error
- Ideas
  - Add common error rules to the grammar
  - Add error token to grammar and synchronization tokens to restart

M	x	+	$	E	T
1	s5	E	E	g2	g3
2	E	E	**accept**	E	E
3	E	s4	r2	E	E
4	s5	E	E	g6	g3
5	E	r4	r4	E	E
6	E	E	r3	E	E

Error rules

- Useful to catch user programming mistakes
- Extending the grammar
  - Introduce sources of ambiguity
  - Make language fall outside of class, e.g., LALR(1)

CFG rules

1. S → E$
2. E → T
3. E → T+E
4. T → x
5. E → EE
6. E → E+

Synchronization tokens

- Synchronization tokens: Those after error token
- Technique used by yacc

- Behavior on error
  - Set error flag
  - Pop from stack until M[s,error] = shift
  - Jump in the input until the synchronization token is found
  - Report error
  - Continue parsing

CFG rules

1. S → E$
2. E → T
3. E → (E)
4. E → T+E
5. T → x
6. E → (error)
7. T → error x

3. Syntax Analysis

- Introduction
- Context free grammars
- Top-down parser
- Bottom-up parser
- Error handling
- Parser generation with yacc

yacc (bison)

- yacc: “yet another compiler compiler” developed in 1970 by S. Johnson, AT&T
  - Input: yacc specification of the CFG with semantic actions
  - Output: LALR(1) parser in C code – call it with **`yyparse()`**
  - Output: Conflict reports
  - Requisite: yacc expects the lexer to be available **`yylex()`**
- Input format

```
%{
C-declarations
%}
Tokens
%%
CFG Rules
%%
C code section
```

yacc example

- Sample rule for expressions

```
expr
     : expr[L] TOKEN_PLUS expr[R]
       {
         $$ = createOperation( ePLUS, $L, $R );
       }
     | expr[L] TOKEN_MULTIPLY expr[R]
       {
         $$ = createOperation( eMULTIPLY, $L, $R );
       }
     | TOKEN_LPAREN expr[E] TOKEN_RPAREN { $$ = $E; }
     | TOKEN_NUMBER { $$ = createNumber($1); }
     ;
```

Where are we?

1. Introduction
2. Lexical analysis
3. Syntax analysis
4. Semantic analysis
5. Intermediate representation
6. Control & data-flow analysis
7. IR optimization
8. Target architectures
9. Code selection
10. Register allocation
11. Scheduling
12. Advanced topics

4. Semantic Analysis

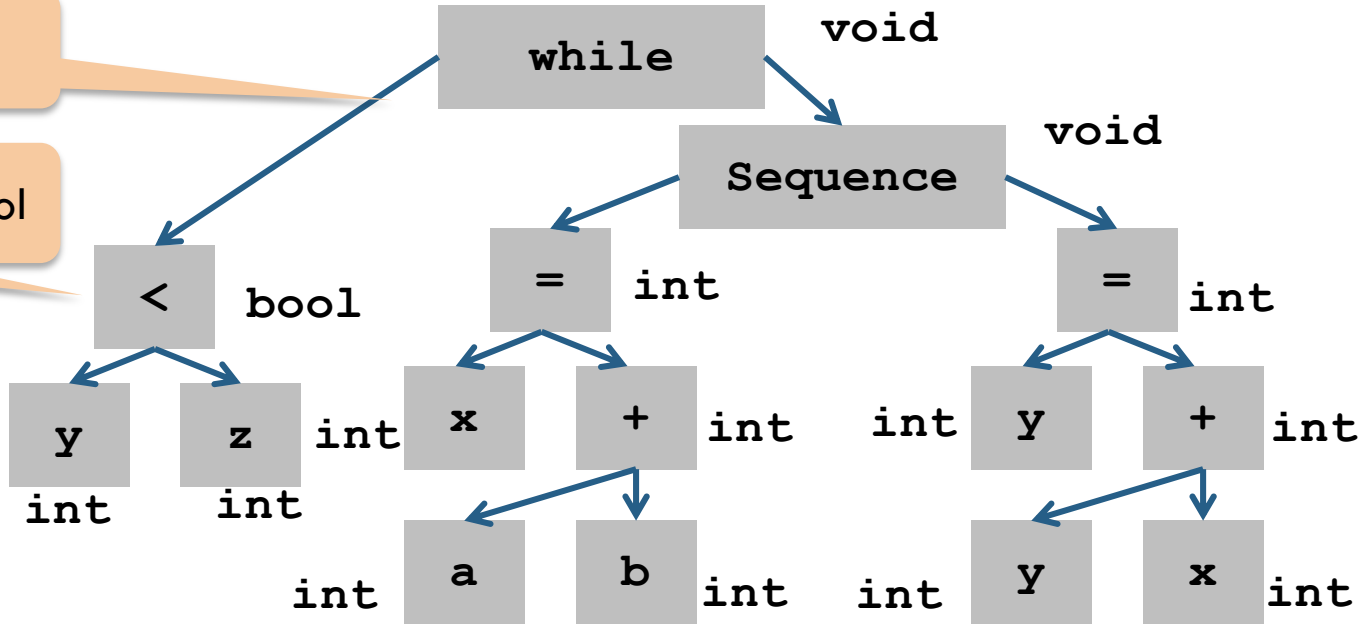
- Introduction
- Attribute grammars
- Applications

Semantic analysis recall

ST: Syntax Tree (= parse tree)

Semantic: int < int ➔ produces a bool

```
while (y < z)
{
   int x = a + b;
   y += x;
}
```



Why semantic analysis?

```
foo(int a,int b,int c,int d)
{...}
bar()
{
  int f[3],g[0], h, i, j, k;
  char *p;
  foo(h,i,"ab",j, k);
  k = f * i + j;
  h = g[17];
  printf("<%s,%s>.\n", p,q);
  p = 10;
}
```

To generate code, the compiler must understand meaning

Wrong number of arguments and type

Wrong dimension for f

Accessing g[17], but declared g[0]

q unknown

p is "char*"

Adapted from: Keith D. Cooper, Ken Kennedy & Linda Torczon

Can't we use CFGs for all this?

- Declaration, code and uses

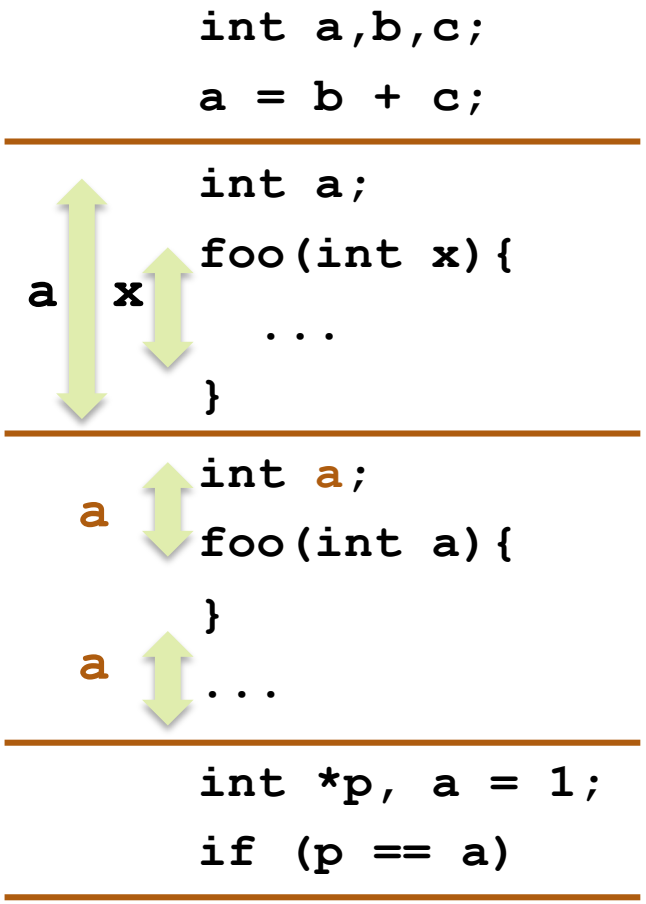
```
int a,b,c;
...
a = b + c;
```

- Language abstraction: L = {wcw; w ∈ L((a|b)*)}
  - Example: aaaabbbababab<u>c</u>aaaabbbababab
  - Context sensitive: Whether the second 'w' is valid depends on the first occurrence
  - It is more difficult to parse context sensitive grammars

- There are similar abstractions for checking function parameters

Semantic analysis: Sample tasks

- **Declaration:** Check that all identifiers were already declared
- **Validity:** Check whether in a portion of a program it is valid to use an identifier
- **Visibility:** Decide which valid "object" is visible when accessed
- **Type-consistency:** Check that the operands obey the type rules of operators



Approaches to semantic analysis

- Formal approach (apart from context-sensitive grammars): Attribute grammars
  - Automatic evaluation of attributes (declarative)
  - But: No globals & too many copies
  - Automatic generation of analyzer: ox
- Ad-hoc approach:
  - Associate code snippet with each production (yacc/bison)
  - Simple, functional but more rigid evaluation (actions given by parsing)
- Modern approach: Tree walk (used in LLVM)
  - Build the *abstract* syntax tree
  - Use virtual functions to walk through it (e.g., visitor pattern)
  - Need to associate rules to every tree-node type: more complicated

We will talk about this: Helps to understand the principles

4. Semantic Analysis

- Introduction
- Attribute grammars
- Applications

Attribute grammars: Intuition

- Idea
  - Syntax-controlled evaluation: semantic actions are associated to grammar rules
  - Symbols (N U T) are given **attributes** (variables)
  - According to the semantic actions concrete values are computed for the attributes
- Example

$$E \rightarrow E + E$$

Attributes	type	type	type
Semantic action	type = max (type, type)		
After parsing	float = max (int, float)		

Assume: Type total order

...

↑

float

↑

int

↑

char

Attribute grammars: Definition

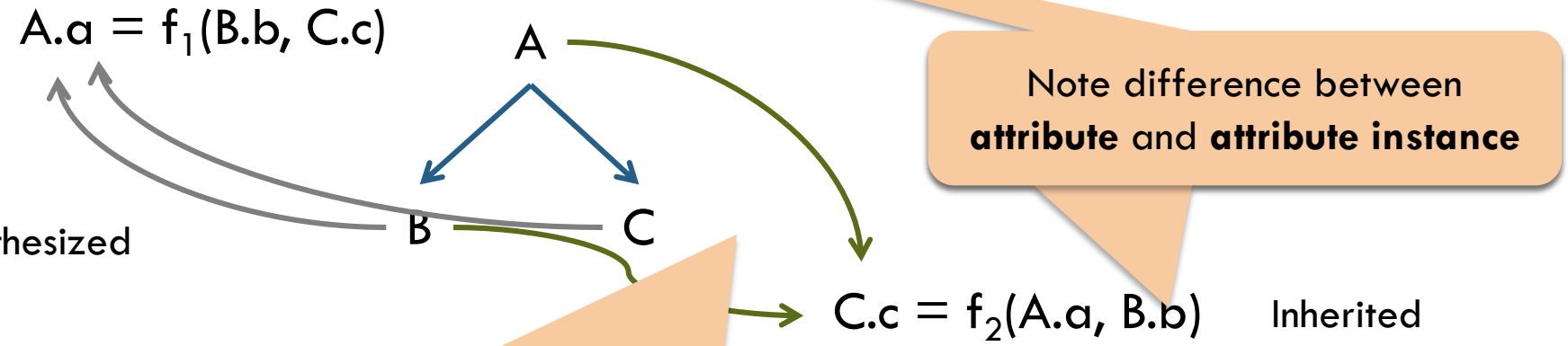
Def.: An **attribute grammar** is composed of a grammar CFG G = (N,T,P,S) and

- Two disjoint sets of attributes SYN (synthesized) and INH (inherited)
  - SYN: Attributes that depend on children of the syntax tree
  - INH: Attributes that depend on parents and siblings of the syntax tree

  Notation: For $X \in (N \cup T)$, $SYN(X) \subseteq SYN$ and $INH(X) \subseteq INH$ denote the attributes associated with symbol X. $ATTR(X) = SYN(X) \cup INH(X)$
- A domain D(a) for each attribute $a \in (SYN \cup INH)$
- Semantic rules for every *computable* attribute in every production $p \in P$, i.e., given $X_0 \rightarrow X_1 X_2 \ldots X_n$, with $X_0 \in N$, $X_{i=1,\ldots,n} \in (N \cup T)$, a rule $a = f(c_1,\ldots c_m)$ with $a \in SYN(X_0) \cup \{\cup^n_{i=1} INH(X_i)\}$, and $c_{k=1,\ldots,m} \in \cup^n_{i=0} ATTR(X_i)$

Attribute grammars: SYN & INH

- Example:
  - Grammar symbols A, B, C
  - Attributes a ∈ ATTR(A), b ∈ ATTR(B), c ∈ ATTR(C)
  - Sample production: A → BC



Note difference between **attribute** and **attribute instance**

Easy to do during LR-parsing

Directly expresses context, difficult at parse time

Computing attributes: Example

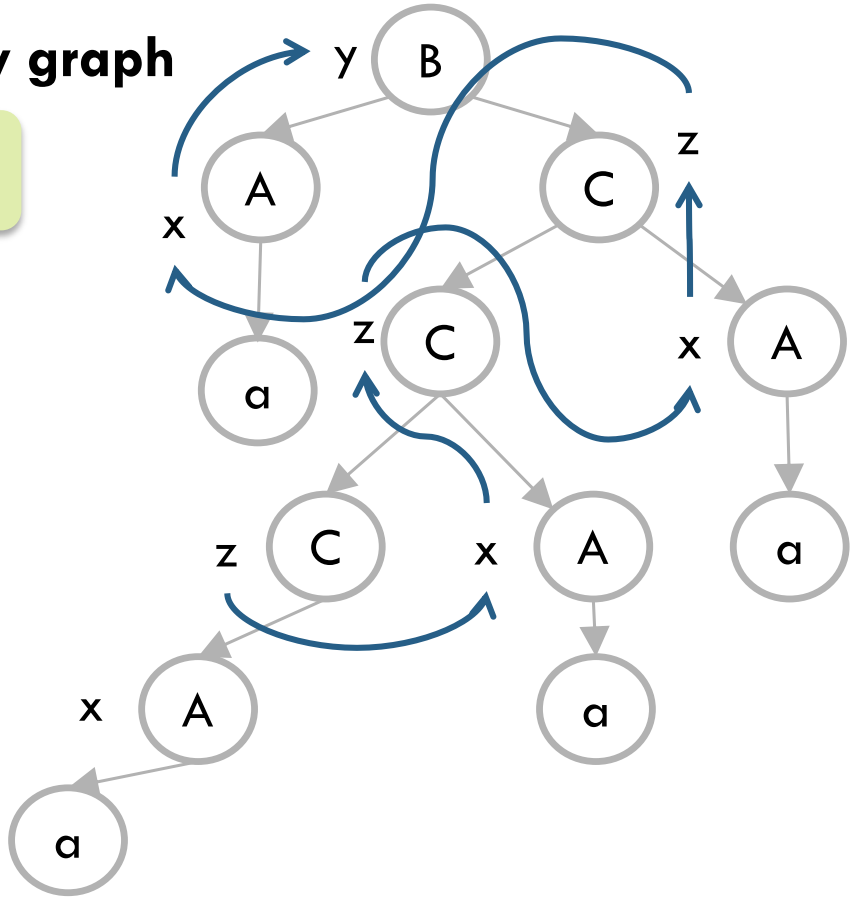
- Semantic rules and parse tree →**dependency graph**
- Example

Production	Semantic rules
1. B → AC	A.x=C.z B.y = A.x + 1
2. B →A	A.x=0 B.y = 1
3. C → CA	A.x=C1.z C0.z = A.x + 1
4. C → A	A.x=0 C.z = 1
5. A → a	

Input: aaaa

A.x: INH and
B.y & C.z: SYN

Numbers to distinguish instance

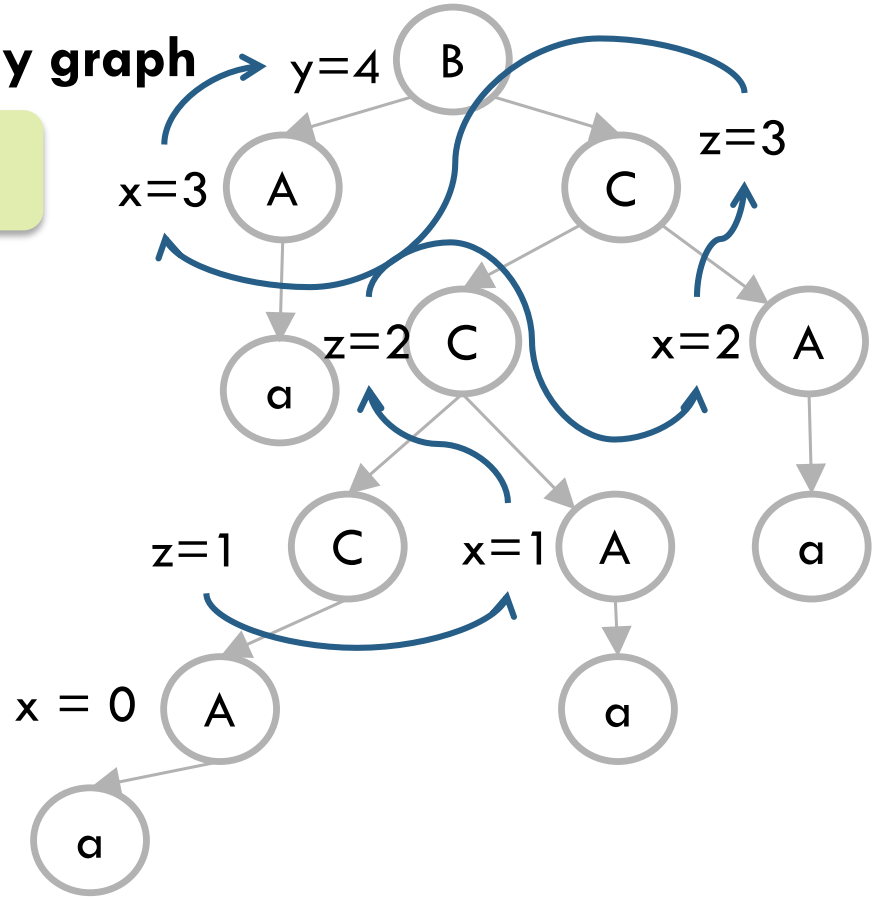


Computing attributes: Example

- Semantic rules and parse tree →**dependency graph**
- Example

Production	Semantic rules
1. B → AC	A.x=C.z B.y = A.x + 1
2. B →A	A.x=0 B.y = 1
3. C → CA	A.x=C1.z C0.z = A.x + 1
4. C → A	A.x=0 C.z = 1
5. A → a	

Input: aaaa



Attribute dependency graph

Def.: An **attribute dependency graph** is a directed graph G= (V,E) where nodes represent attribute instances and edges dependencies. There is an edge (c_i,a) for every attribute and every semantic rule $a = f(c_1,\ldots c_m)$

- Approach: Compute the attributes in the order given by a topological sort
- Cannot handle cyclic graphs
  - Largest class: Strongly non-circular grammars (SNC)
  - SNC-test: Polynomial-time (failing is not conclusive)

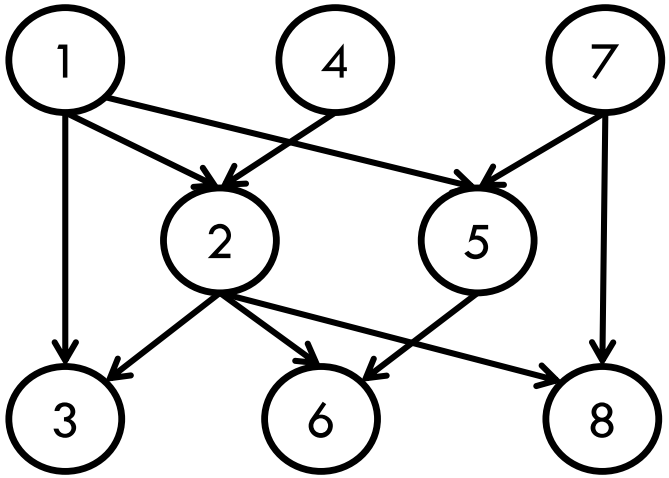
Topological sort

- DFS - Algorithm

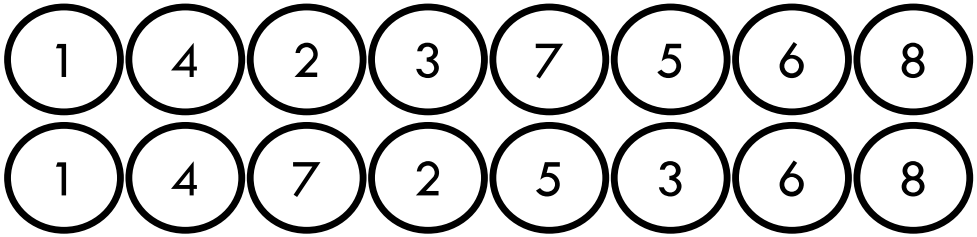
```
Input       acyclic graph G = (V,E)
Output      Node list L
L = {}
While |L| != |V| do
    select n ∈ (V – L)
    append(n,L)
return L

Function append(n, L)
    foreach (m,n) ∈ E, m ∉ L
        append(m,L)
     L = {L, n}
```

Example:



Possible orders:



4. Semantic Analysis

- Introduction
- Attribute grammars
- Applications

Application (examples) of attribute grammars

- In the coming slides
  - Synthetic example: Binary numbers
  - Abstract syntax trees
  - Label/goto correctness
  - Type checking

Binary numbers: CFG & attributes

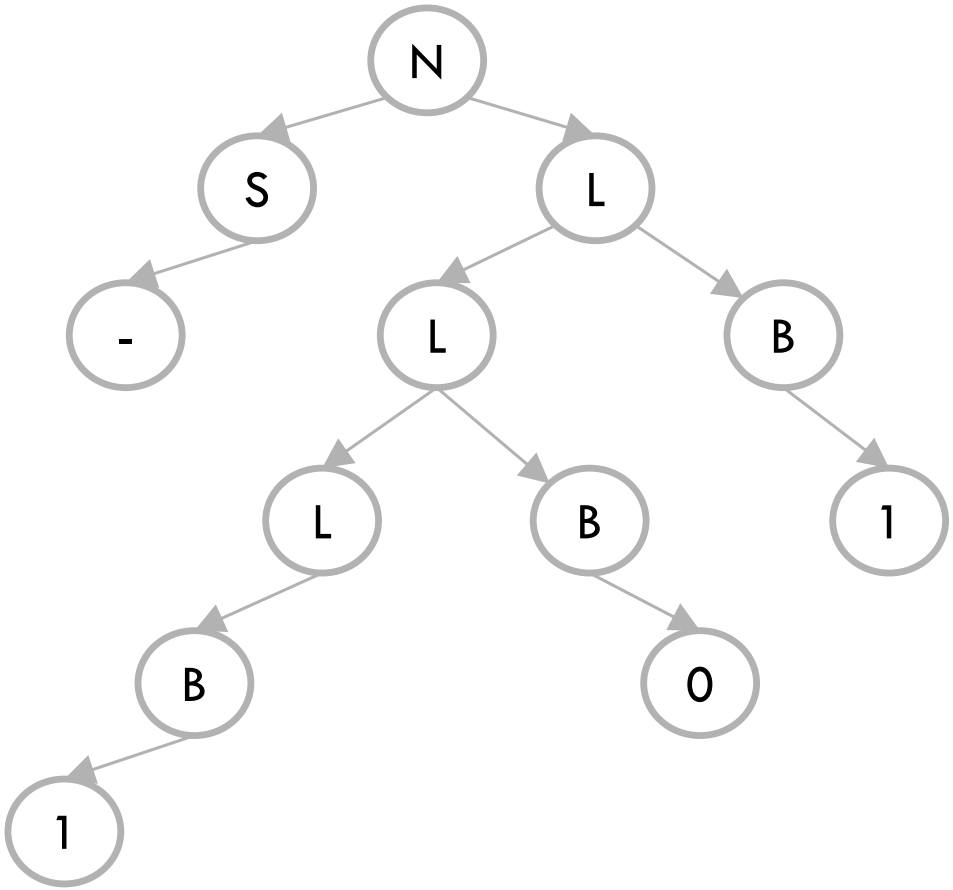
- Binary representations of integers

CFG rules

1. N → S L
2. L → L B
3. L → B
4. B → 0
5. B → 1
6. S → +
7. S → -

(N: number, S: sign, L: list, B: bit)

Input: -101



Binary numbers: CFG & attributes

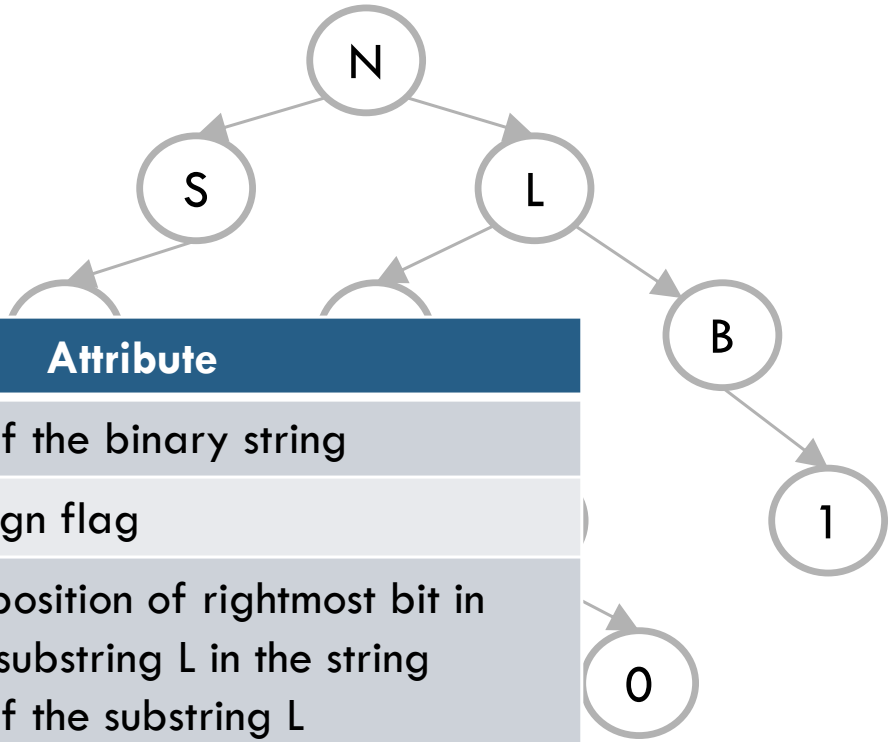
- Binary representations of integers

CFG rules

1. N → S L
2. L → L B
3. L → B
4. B → 0
5. B → 1
6. S → +
7. S → -

(N: number, S: sign, L: list, B: bit)

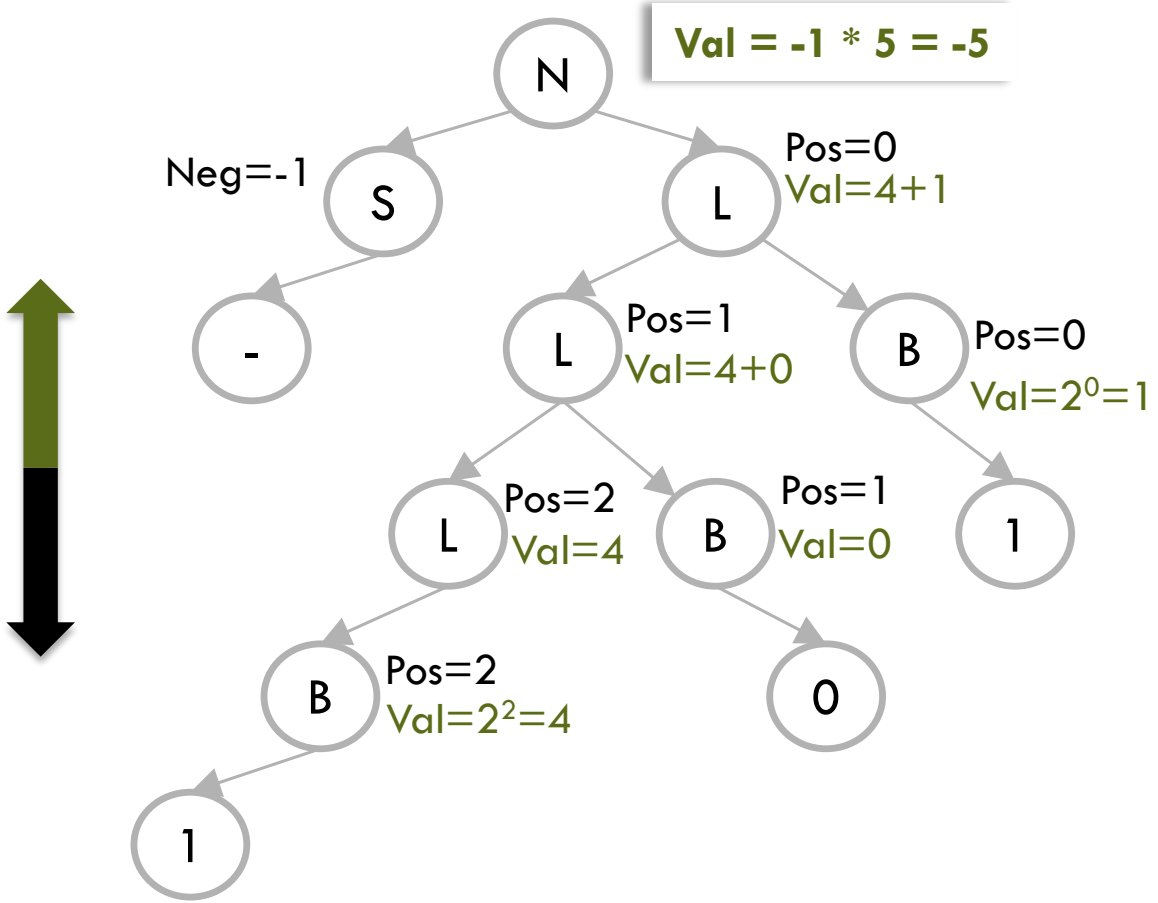
Input: -101



Symbol	Attribute
N	Val (Z): Value of the binary string
S	Neg ({-1,1}): Sign flag
L	Pos (Z^+ U {0}): position of rightmost bit in substring L in the string Val (Z): Value of the substring L
B	Pos(Z^+ U {0}): bit position in string Val (Z): Value of the bit (2^{Pos})

Binary numbers: Semantic rules

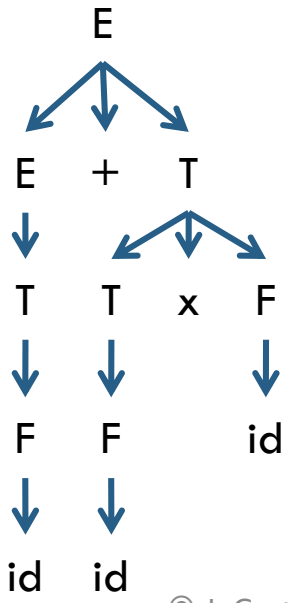
CFG rule	Semantic rule
N → S L	N.Val = S.Neg * L.Val L.Pos = 0
L_0 → L_1 B	L_1.Pos = L_0.Pos + 1 B.Pos = L_0.Pos L_0.Val = L_1.Val + B.Val
L → B	L.Val = B.Val B.Pos = L.Pos
B → 0	B.Val = 0
B → 1	B.Val = $2^{B.Pos}$
S → +	S.Neg = +1
S → -	S.Neg = -1



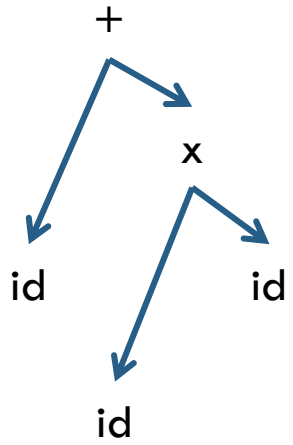
Abstract syntax tree

- Parse tree = Concrete syntax tree
  - Contains symbols from the grammar
  - Symbols are irrelevant after parsing
- Abstract syntax tree (AST): Drop symbols and retain only what matters

(1) E → E + T
(2) E → T
(3) T → T x F
(4) T → F
(5) F → id



Concrete syntax tree



Abstract syntax tree

AST: Attribute CFG

Symbol	Attribute
E	ptr: pointer to AST-node
T	
F	
id	name: actual lexeme

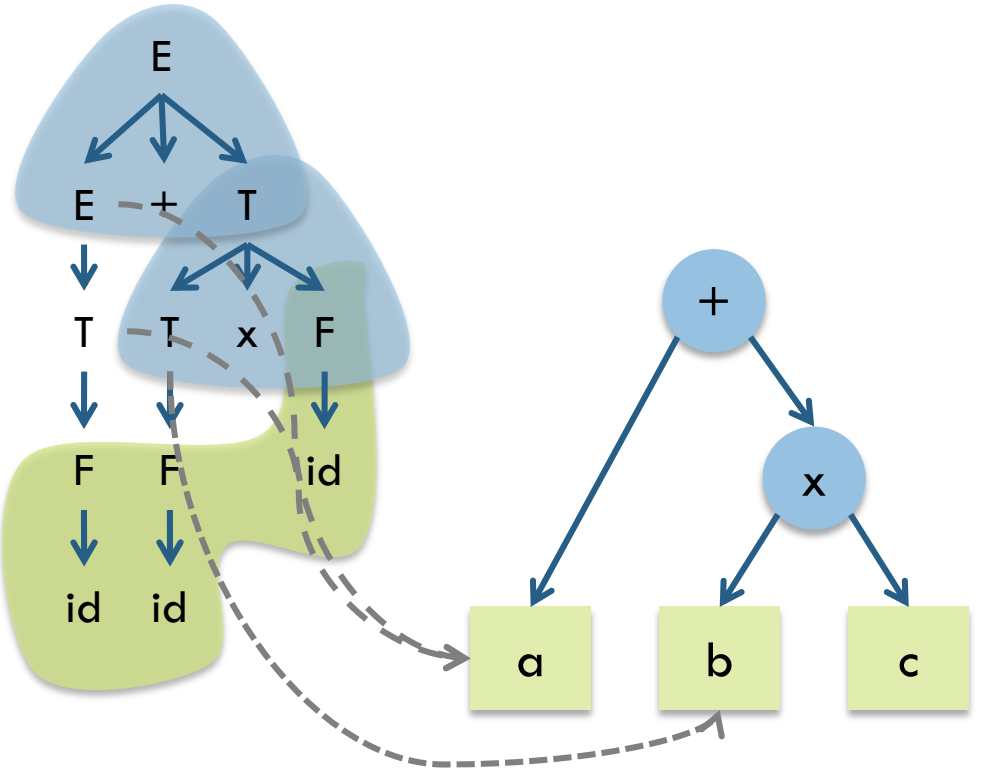
CFG rule	Semantic rule
$E_0 \rightarrow E_1 + T$	E_0.ptr = mknode(´+´,E_1.ptr,T.ptr)
$E \rightarrow T$	E.ptr = T.ptr
$T_0 \rightarrow T_1 \times F$	T_0.ptr = mknode(´x´,T_1.ptr, F.ptr)
$T \rightarrow F$	T.ptr = F.ptr
$F \rightarrow id$	F.ptr = mkleaf(id.name)

- **mknode(op,left,right)**: Help function to create internal AST nodes
- **mkleaf(name)**: Help function to create leaf nodes for identifiers

AST: Example

Input: a+bxc

CFG rule	Semantic rule
$E_0 \rightarrow E_1 + T$	E_0.ptr = mknode(´+´,E_1.ptr,T.ptr)
$T \rightarrow T \times F$	T_0.ptr = mknode(´x´,T_1.ptr, F.ptr)
$E \rightarrow T$	E.ptr = T.ptr
$T \rightarrow F$	T.ptr = F.ptr
$F \rightarrow id$	F.ptr = mkleaf(id.name)

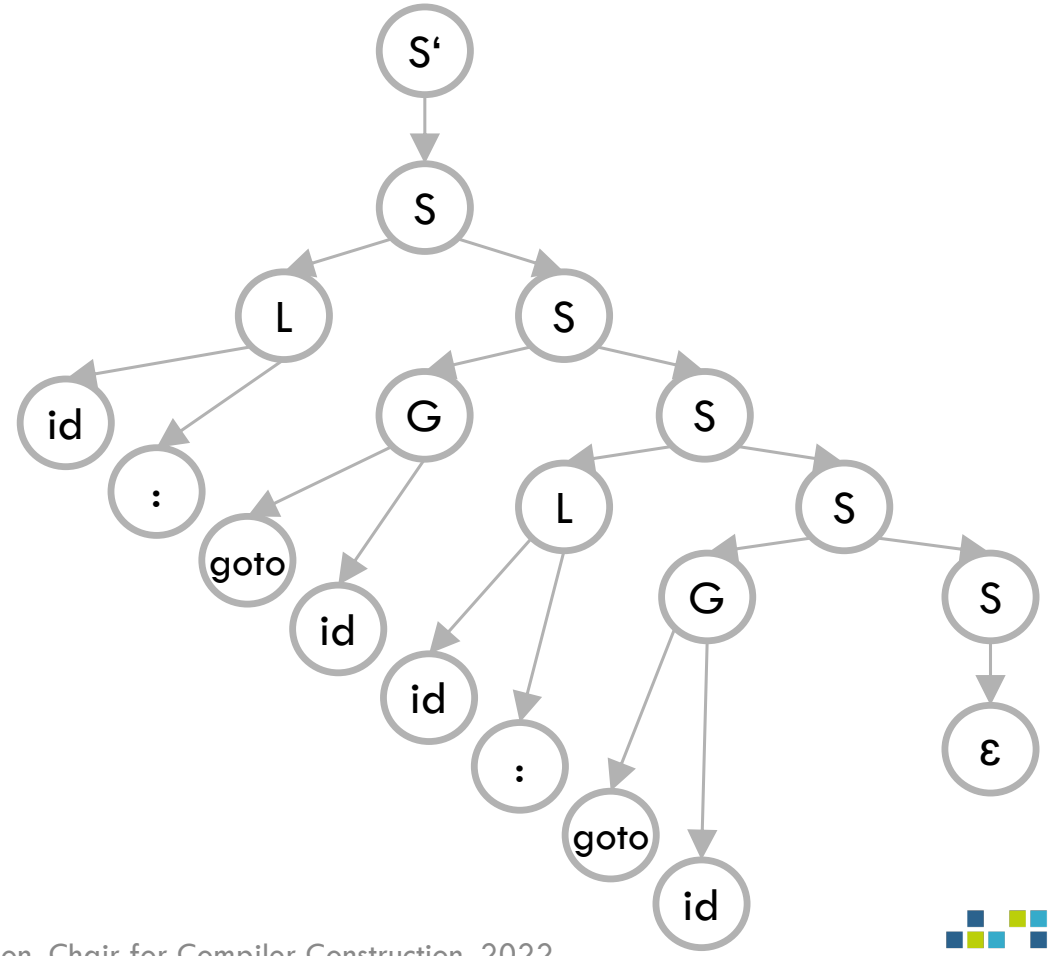


Label/goto correctness

- Grammar
  - (1) S' → S
  - (2) S → L S
  - (3) S → G S
  - (4) S → ε
  - (5) L → id :
  - (6) G → goto id;

Input:

```
L1:
goto L2;
L2:
goto L1;
```



Label/goto: Attribute grammar

Symbol	Attribute
S'	S_{lab}: Set of all declared labels OK: flag ➔ program correct
S	S_{lab}: Set of all declared labels S'_{lab}: Set of declared labels below OK: flag ➔ statement correct
G	S_{lab}: Set of all declared labels OK: flag ➔ statement correct
L	name
id	name

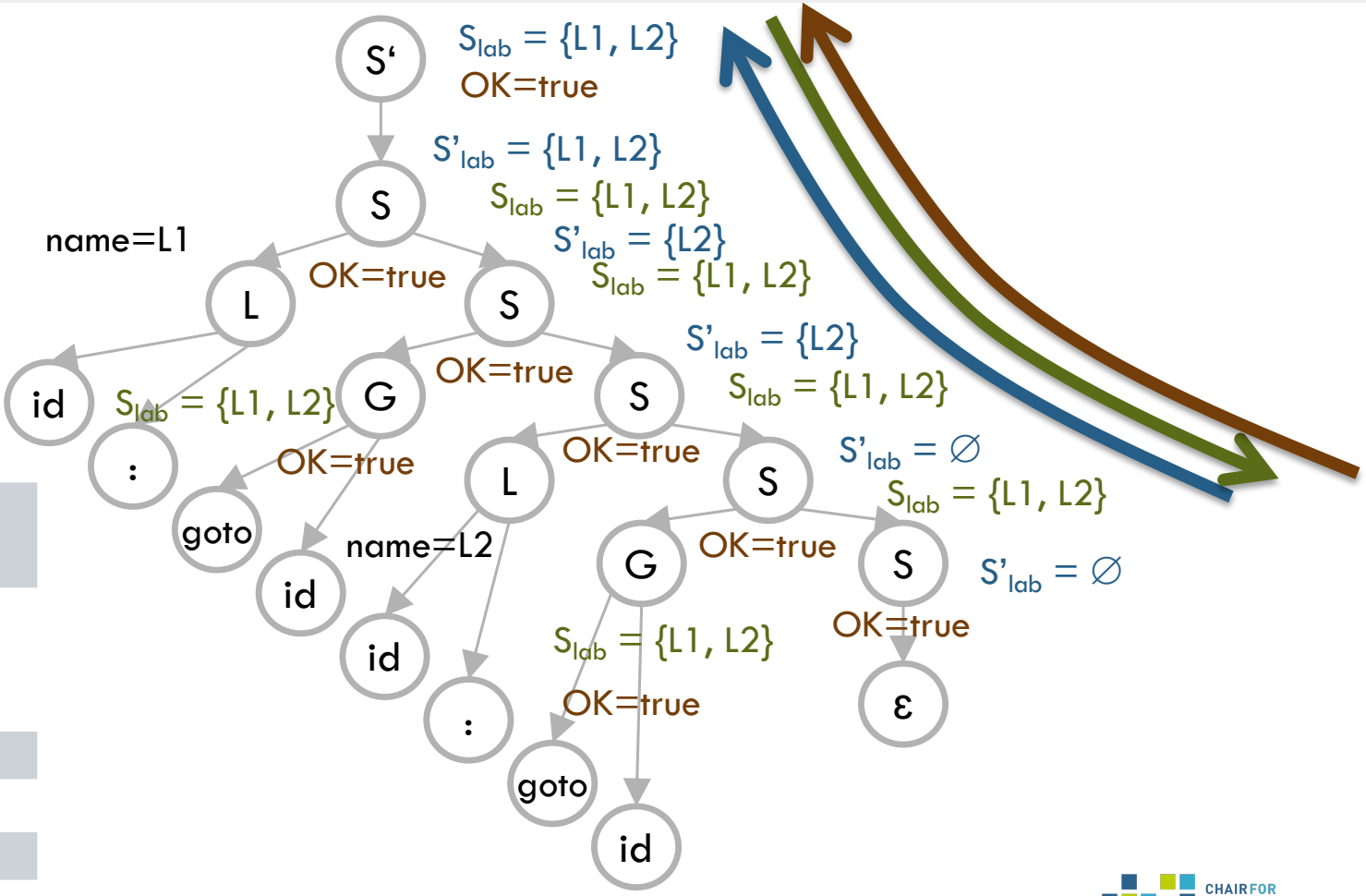
CFG rule	Semantic rule
S' → S	$S'.S_{lab} = S.\ S'_{lab}$ $S.S_{lab} = S'.S_{lab}$ $S'.OK = S.OK$
S_0 → L S_1	$S_0.S'_{lab} = S_1.\ S'_{lab}\ U\ \{L.name\}$ $S_1.\ S_{lab} = S_0.S_{lab}$ $S_0.OK = S_1.OK$
S_0 → G S_1	$S_0.S'_{lab} = S_1.\ S'_{lab}$ $S_1.\ S_{lab} = S_0.S_{lab}$ $G.S_{lab} = S_0.S_{lab}$ $S_0.OK = S_1.OK$ AND G.OK
S → ε	$S.S'_{lab}=\varnothing$, S.OK = true
L → id :	L.name= id.name
G → goto id;	G.OK = id.name $\in$ $G.S_{lab}$

Label/goto correctness: Example

Input:

```
L1:
goto L2;
L2:
goto L1;
```

S' → S	$S'.S_{lab} = S.S'_{lab}$ $S.S_{lab} = S'.S_{lab}$ S'.OK = S.OK
S_0 → L S_1	$S_0.S'_{lab} = S_1.S'_{lab} \cup \{L.name\}$ $S_1.S_{lab} = S_0.S_{lab}$ S_0.OK = S_1.OK
S_0 → G S_1	$S_0.S'_{lab} = S_1.S'_{lab}$ $S_1.S_{lab} = S_0.S_{lab}$ $G.S_{lab} = S_0.S_{lab}$ S_0.OK = S_1.OK AND G.OK
S → ε	$S.S'_{lab} = \emptyset$, S.OK = true
L → id :	L.name= id.name
G → goto id;	G.OK = id.name ∈ $G.S_{lab}$



Approaches to semantic analysis: Recall

- Formal approach (apart from context-sensitive grammars): Attribute grammars
  - Automatic evaluation of attributes (declarative)
  - **But: No globals & too many copies**
  - Automatic generation of analyzer: ox
- Ad-hoc approach:
  - Associate code snippet with each production (yacc/bison)
  - Simple, functional but more rigid evaluation (actions given by parsing)
- Modern approach: Tree walk (used in LLVM)
  - Build the *abstract* syntax tree
  - Use virtual functions to walk through it (e.g., visitor pattern)
  - Need to associate rules to every tree-node type: more complicated

Type-checking: Introduction

- Recall motivation

```
int x,y;
float foo(float y){
  return x + y;}
```

E → E + E

Attributes	type	type	type
Semantic action	type = max (type, type)		
After parsing	float = max (int, float)		

- What the compiler needs to know
  - Scope information: Which 'x' is meant in "x + y"
  - Type information (& other attributes) of the variables
  - Type checking rules for '+'

Use so-called **symbol tables**

Use so-called **type systems**

Symbol tables: Intuition

```
0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x, y, z);
4:   {
5:     int x, z;
6:     z = y;
7:     x = z;
8:     {
9:       int y = x;
10:      printf("%d,%d,%d\n", x, y, z);
14:    }
15:    printf("%d,%d,%d\n", x, y, z);
16:  }
17:}
```

Symbol table	
x	0
z	1

Symbol

Line number (& attributes in reality)

Adapted from: http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

Symbol tables: Intuition

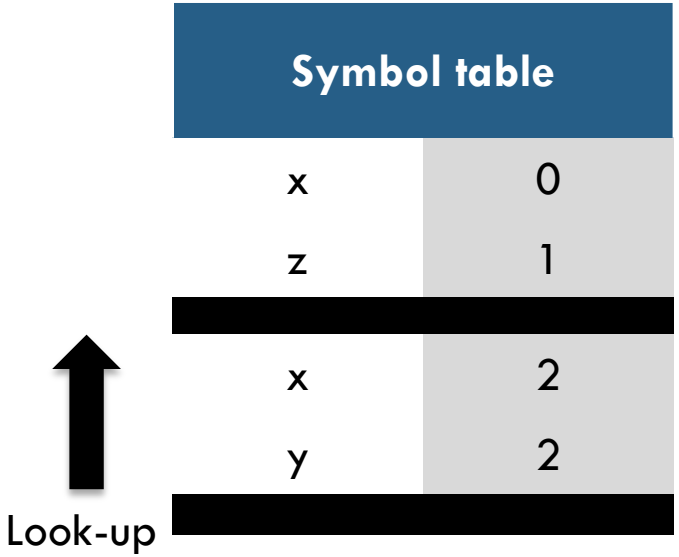
```
0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x, y, z);
4:   {
5:     int x, z;
6:     z = y;
7:     x = z;
8:     {
9:       int y = x;
10:      printf("%d,%d,%d\n", x, y, z);
14:    }
15:    printf("%d,%d,%d\n", x, y, z);
16:  }
17:}
```

New scope

Symbol table	
x	0
z	1
x	2
y	2

Symbol tables: Intuition

```
0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
4:   {
5:     int x, z;
6:     z = y;
7:     x = z;
8:     {
9:       int y = x;
10:      printf("%d,%d,%d\n", x, y, z);
14:    }
15:    printf("%d,%d,%d\n", x, y, z);
16:  }
17:}
```



Symbol tables: Intuition

```
0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
4:   {
5:     int x, z;
6:     z@5 = y@2;
7:     x@5 = z@5;
8:     {
9:       int y = x;
10:      printf("%d,%d,%d\n", x, y, z);
14:    }
15:    printf("%d,%d,%d\n", x, y, z);
16:  }
17:}
```

Symbol table	
x	0
z	1
x	2
y	2
x	5
z	5

Symbol tables: Intuition

```
0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
4:   {
5:     int x, z;
6:     z@5 = y@2;
7:     x@5 = z@5;
8:     {
9:       int y = x@5;
10:      printf("%d,%d,%d\n", x@5, y@9, z@5);
14:    }
15:    printf("%d,%d,%d\n", x, y, z);
16:  }
17:}
```

Symbol table	
x	0
z	1
x	2
y	2
x	5
z	5
y	9

Symbol tables: Intuition

```
0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
4:   {
5:     int x, z;
6:     z@5 = y@2;
7:     x@5 = z@5;
8:     {
9:       int y = x@5;
10:      printf("%d,%d,%d\n", x@5, y@9, z@5);
14:    }
15:    printf("%d,%d,%d\n", x@5, y@2, z@5);
16:  }
17:}
```

Symbol table	
x	0
z	1
x	2
y	2
x	5
z	5

Symbol tables: Intuition

```
0: int x = 137;
1: int z = 42;
2: int MyFunction(int x, int y) {
3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
4:   {
5:     int x, z;
6:     z@5 = y@2;
7:     x@5 = z@5;
8:     {
9:       int y = x@5;
10:      printf("%d,%d,%d\n", x@5, y@9, z@5);
14:    }
15:    printf("%d,%d,%d\n", x@5, y@2, z@5);
16:  }
17:}
```

Symbol table	
x	0
z	1

Symbol tables

- Symbol table: Mapping from a name to the object the name refers to
  - Filled during semantic analysis (with info from lexical and syntax analysis)
  - *Stack of maps*: Each map represents a scope
- Object attributes: Name, type, size, alignment, … (see next slide)

- Operations
  - Push scope: Enter a new scope
  - Pop scope: Leave a scope and delete its declarations
  - Insert symbol
  - Look-up symbol

Symbol tables: Attributes

- Attributes depend on the language, e.g., for C:

Attribute	Representation
Name	String
Type	Type-expression, e.g., int, float, array(100,int), struct(float,…), (float, short) → pointer(double), …
Size	Integer
Alignment	Integer
Qualifier	{const, volatile}
Storage class	{extern, static, auto, register}

Compiler needs to insert *memcopy* when assigning structs

```
struct { ... } S, T;
S = T;
```

Compiler has to know what are correct addresses in HW (0x0, 0x4, 0x8, …)

Recognize illegal assignment, e.g.,

```
const int x = 1;
x = 2;
```

Symbol tables: Implementation

- Need for speed: **n** insertions and **m** look-ups (m > n)
  - On every insert: Check for existence
- Linked lists
  - Insert: Existence check O(n), actual insertion O(1)
  - Look-up: O(n)
  - ➔ Time complexity $O(n^2 + mn)$
- Hash tables (assumption: open hashing, balanced)
  - Insert: Existence check O(1), actual insertion O(1)
  - Look-up: O(1)
  - ➔ Time complexity O(n + m)

Type-checking: Reloaded

- Recall motivation

```
int x,y;
float foo(float y){
  return x + y;}
```

E → E + E

Attributes	type	type	type
Semantic action	type = max (type, type)		
After parsing	float = max (int, float)		

- What the compiler needs to know
  - Scope information: Which 'x' is meant in "x + y"
  - Type information (& other attributes) of the variables
  - Type checking rules for '+'

Type-checking: Intuition

- Include checking rules in the semantic actions of the attribute grammar
- Example

(1) S → D ; E
(2) D → D ; D | T id
(3) T → int | float | T[num] | T*
(4) E → num | num.num | id | E + E | E[E] | *E

```
int a;
int[10] b;
int* c;
A + b[1] + *c + 42;
```

Type-checking: Sample attribute grammar

CFG rule	Semantic rule
D → T id	
T → int | float	
T_0 → T_1 [num]	
T_0 → T_1*	
E → num | num.num	
E → id	

Type-checking: Sample attribute grammar

CFG rule	Semantic rule
D → T id	symtab[id.name] = T.type
T → int | float	T.type = int | float
T_0 → T_1 [num]	T_0.type = array(num.val, T_1.type)
T_0 → T_1*	T_0.type = ptr(T_1.type)
E → num | num.num	E.type = int | float
E → id	E.type = symtab[id.name]

Type-checking: Sample attribute grammar (2)

CFG rule	Semantic rule
	…
$E_0 \rightarrow E_1 + E_2$	
$E_0 \rightarrow E_1[E_2]$	
$E_0 \rightarrow *E_1$	

Type-checking: Sample attribute grammar (2)

CFG rule	Semantic rule
	…
$E_0 \rightarrow E_1 + E_2$	E_0.type = if (E_1.type == **int** && E_2.type == **int**) then **int** else if (E_1.type == **float** && E_2.type == **float**) then **float** else **errortype**
$E_0 \rightarrow E_1[E_2]$	E_0.type = if (E_1.type == **array(s,t)** && E_2.type == **int**) then **t** else **errortype**
$E_0 \rightarrow *E_1$	E_0.type = if E_1.type == **pointer(t)** then **t** else **errortype**

Type-checking as proofs: Inference rules

- Define inference rules to determine the type
- Notation

$$\frac{S \vdash e1: \text{int} \qquad S \vdash e2: \text{int}}{S \vdash e1 + e2: \text{int}}$$

Read: If we can show that e1 and e2 are of **type int** in **scope S**, then we can show that e1+e2 has **type int** as well

- Other examples

$$\frac{S \vdash e1: T[] \qquad S \vdash e2: \text{int}}{S \vdash e1[e2]: T}$$

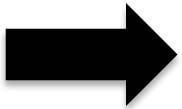
$$\frac{f \text{ is an identifier for a function in scope } S \qquad f \text{ has type } (T1, \ldots, Tn) \rightarrow U \qquad S \vdash e_i: T_i \text{ for } 1 \leq i \leq n}{S \vdash f(e_1, \ldots, e_n): U}$$

Partial order for types

- Introduce partial order of types (for example for class hierarchies)
- Given two types A and B, we say $A \leq B$ if A is convertible to B
  - $A \leq A$
  - If $A \leq B$ and $B \leq C$ ➔ $A \leq C$
  - If $A \leq B$ and $B \leq A$ ➔ $A = B$

- Example: Rule for comparison

$$\frac{S \vdash e1: T \quad S \vdash e2: T \quad T \text{ is primitive}}{S \vdash e1 == e2: \text{bool}}$$



$$\frac{S \vdash e1: T1 \quad S \vdash e2: T2 \quad T1 \leq T2 \text{ or } T2 \leq T1}{S \vdash e1 == e2: \text{bool}}$$

Type-checking: Example

```
int x, y, z;
if (((x == y) > 5 && x + y < z)) { /* ... */ }
```

Adapted from: http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

Type-checking: Example

```
int x, y, z;
if (((x == y) > 5 && x + y < z)) { /* ... */ }
```

$$\frac{\begin{array}{c}a \text{ is identifier} \\ a \text{ is a variable in scope } S \text{ of type } T\end{array}}{S \vdash a: T}$$

Facts
$S \vdash x: \text{int}$
$S \vdash y: \text{int}$
$S \vdash z: \text{int}$

Type-checking: Example

```
int x, y, z;
if (((x == y) > 5 && x + y < z)) { /* ... */ }
```

$$\frac{S \vdash e1: T1 \quad S \vdash e2: T2 \quad T1 \leq T2 \text{ or } T2 \leq T1}{S \vdash e1 == e2: \text{bool}}$$

Facts
$S \vdash x: \text{int}$
$S \vdash y: \text{int}$
$S \vdash z: \text{int}$
$S \vdash x == y: \text{bool}$

Type-checking: Example

```
int x, y, z;
if (((x == y) > 5 && x + y < z)) { /* ... */ }
```

$$\frac{S \vdash e1: \text{int} \quad S \vdash e2: \text{int}}{S \vdash e1 + e2: \text{int}}$$

Facts
$S \vdash x: \text{int}$
$S \vdash y: \text{int}$
$S \vdash z: \text{int}$
$S \vdash x == y: \text{bool}$
$S \vdash x + y: \text{int}$

Type-checking: Example

```
int x, y, z;
if (((x == y) > 5 && x + y < z)) { /* ... */ }
```

$$\frac{S \vdash e1: T1 \quad S \vdash e2: T2 \quad T1 \leq T2 \text{ or } T2 \leq T1}{S \vdash e1 < e2: \text{bool}}$$

Facts
$S \vdash x: \text{int}$
$S \vdash y: \text{int}$
$S \vdash z: \text{int}$
$S \vdash x == y: \text{bool}$
$S \vdash x + y: \text{int}$
$S \vdash x + y < z: \text{bool}$

Type-checking: Example

```
int x, y, z;
if (((x == y) > 5 && x + y < z)) { /* ... */ }
```

$$\frac{S \vdash e1: \text{int} \quad S \vdash e2: \text{int}}{S \vdash e1 > e2: \text{bool}}$$

```
> Error: Cannot compare int and bool
```

Facts
$S \vdash x: \text{int}$
$S \vdash y: \text{int}$
$S \vdash z: \text{int}$
$S \vdash x == y: \text{bool}$
$S \vdash x + y: \text{int}$
$S \vdash x + y < z: \text{bool}$

Type-checking: Example

```
int x, y, z;
if (((x == y) > 5 && x + y < z)) { /* ... */ }
```

$$\frac{S \vdash e1: \text{bool} \quad S \vdash e2: \text{bool}}{S \vdash e1 \;\&\&\; e2: \text{bool}}$$

Facts
$S \vdash x: \text{int}$
$S \vdash y: \text{int}$
$S \vdash z: \text{int}$
$S \vdash x == y: \text{bool}$
$S \vdash x + y: \text{int}$
$S \vdash x + y < z: \text{bool}$

```
> Error: Cannot compare int and bool
> Error: Unrecognized ??? && bool
```

Error type (bottom type) to prevent errors from propagating (analog to error token in parsing)

Where are we?

1. Introduction
2. Lexical analysis
3. Syntax analysis
4. Semantic analysis
5. Intermediate representation
6. Control & data-flow analysis
7. IR optimization
8. Target architectures
9. Code selection
10. Register allocation
11. Scheduling
12. Advanced topics