Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Talen en Compilers

2019 - 2020, period 2

Alejandro Serrano Mena

Department of Information and Computing Sciences
Utrecht University

16. Q&A, Summary, Farewell

This lecture

Q&A, Summary, Farewell

- Questions and Answers
- Summary and Farewell

16.1 Questions and Answers

Common pitfall 1: grammar vs. language

- A **language** L is a subset of the words of an alphabet X.
  - A **word** $w \in X^*$ is a sequence of symbols from X.
  - Our main question is **acceptance**: does word w belong to language L?
  - After some time, we want **efficient** acceptance.

Common pitfall 1: grammar vs. language

- A **language** L is a subset of the words of an alphabet X.
  - A **word** $w \in X^*$ is a sequence of symbols from X.
  - Our main question is **acceptance**: does word w belong to language L?
  - After some time, we want **efficient** acceptance.
- A **grammar** is a way to describe a language.
  - Some grammars are nicer than others:
    - Regular, context-free.

Common pitfall 1: grammar vs. language

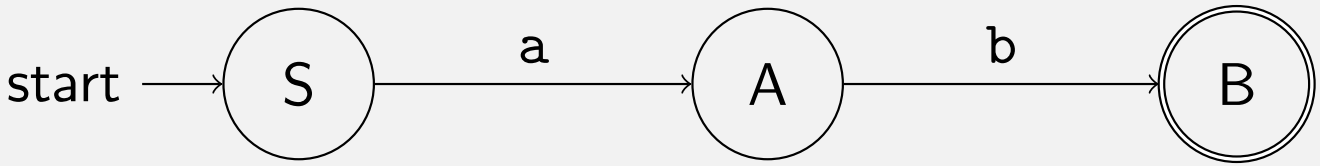
- A **language** L is a subset of the words of an alphabet X.
  - A **word** $w \in X^*$ is a sequence of symbols from X.
  - Our main question is **acceptance**: does word w belong to language L?
  - After some time, we want **efficient** acceptance.
- A **grammar** is a way to describe a language.
  - Some grammars are nicer than others:
    - Regular, context-free.
  - A language can be described by **many** different grammars.
  - **Warning!** A **regular** language can be described by at least one regular grammar, but many others (non-regular) may describe it, too.

Common pitfall 2: automata have an error state

Transitions are **total** functions, $d :: Q \to X \to Q$.

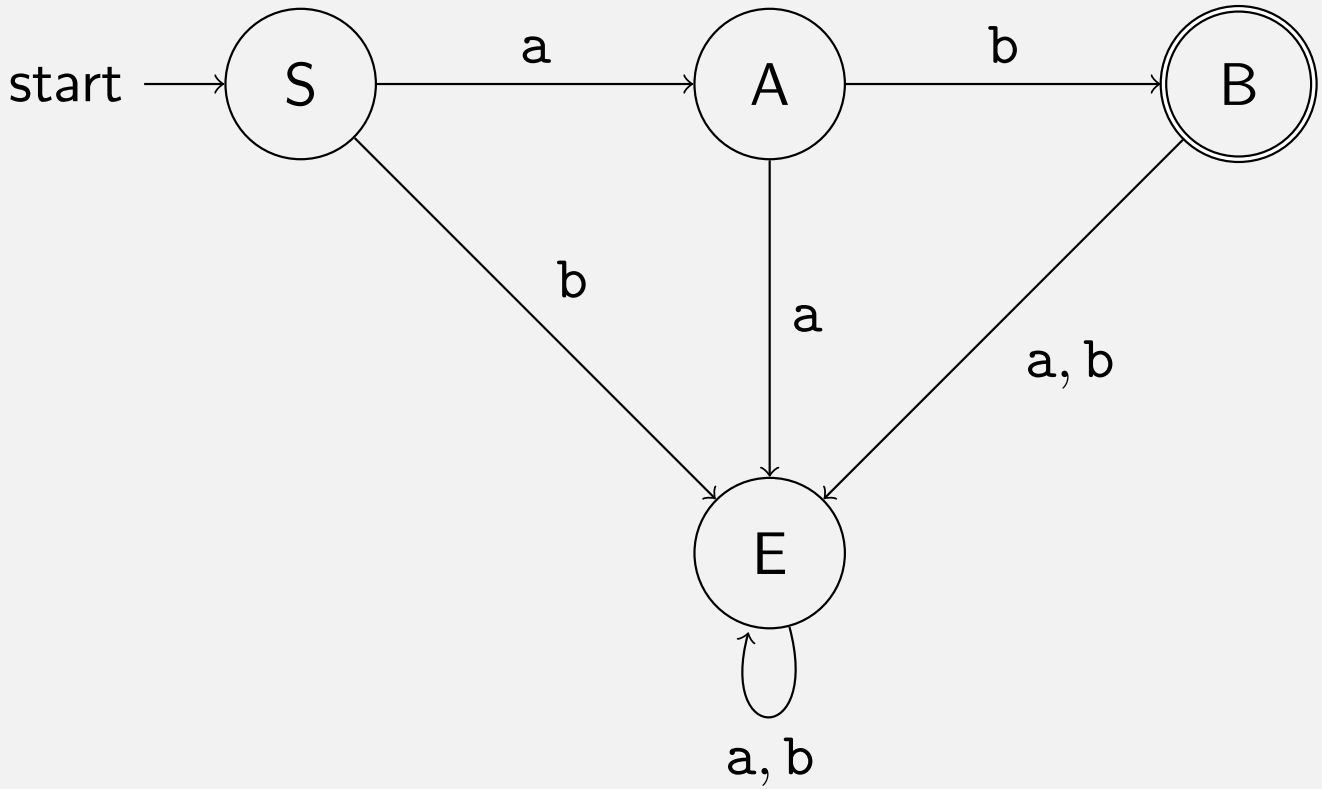
Common pitfall 2: automata have an error state

Transitions are **total** functions, $d :: Q \to X \to Q$.



Common pitfall 2: automata have an error state

Transitions are **total** functions, $d :: Q \to X \to Q$.



Common pitfall 3: pumping lemma

General structure of pumping lemma

L is reg./ctx.-free $\implies$ L has property P

Common pitfall 3: pumping lemma

General structure of pumping lemma

L is reg./ctx.-free $\implies$ L has property P

Small logic reminder

$A \implies B$ is logically equivalent to $\neg B \implies \neg A$

Common pitfall 3: pumping lemma

General structure of pumping lemma

L is reg./ctx.-free $\implies$ L has property P

Logically equivalent formulation

L does **not** have property P $\implies$ L is **not** reg./ctx.-free

Consequence

You can**not** use the lemma to prove that L **is** reg./ctx.-free

How to prove it then?

L is **regular**. Give me **one** of the following:

- Deterministic finite state automaton,
- Non-deterministic finite state automaton,
- Regular grammar, or
- Regular expression.

L is **context-free**. Give me **one** of the following:

- (Context-free) grammar, or
- Pushdown automaton.

How to prove it then?

L is **LL(1)**:

- Context-free grammar,
- Lookahead sets for each production, and
- The sets for each non-terminal must be disjoint.

L is **LR(0)**:

- Construct the LR(0) automaton,
  - The thing with markers • in the rules.
- There must be **no** conflicts.

16.2 Summary and Farewell

Brief course summary

- Build a simple compiler
  - Parsing using **combinators**
  - **Code generation** for a stack machine
  - Use **folds** to express the phases

Brief course summary

- Build a simple compiler
  - Parsing using **combinators**
  - **Code generation** for a stack machine
  - Use **folds** to express the phases
- A bit of **language and automata theory**
  - Grammars, languages, pumping lemmas
  - **Fixpoint** algorithms

Brief course summary

- Build a simple compiler
  - Parsing using **combinators**
  - **Code generation** for a stack machine
  - Use **folds** to express the phases
- A bit of **language and automata theory**
  - Grammars, languages, pumping lemmas
  - **Fixpoint** algorithms
- **Efficient** parsing algorithms
  - LL(1), LR(0), a bit or LR(1)
  - Key idea: **lookahead**

From a simple to a realistic compiler

1. Parsing
   - Usually dealt with with a parser generator

From a simple to a realistic compiler

1. Parsing
   - Usually dealt with with a parser generator
2. Static analyses
   - We want to catch errors before they happen
     - Compile-time vs. run-time
   - Type systems are an important asset

From a simple to a realistic compiler

1. Parsing
   - Usually dealt with with a parser generator
2. Static analyses
   - We want to catch errors before they happen
     - Compile-time vs. run-time
   - Type systems are an important asset
3. Program optimization

From a simple to a realistic compiler

1. Parsing
   - Usually dealt with with a parser generator
2. Static analyses
   - We want to catch errors before they happen
     - Compile-time vs. run-time
   - Type systems are an important asset
3. Program optimization
4. Code generation
   - Real architectures are more complex than SSM
   - Nowadays, you usually generate code in an intermediate language, like LLVM

Helium

- Helium (http://www.github.com/Helium4Haskell)
- Developed in Utrecht, since around 2002, revived a few years back
- (Very) close to the Haskel 2010 standard
- Focus on good type error diagnosis
- Moving from LVM (bytecode) backend to LLVM backend (C)
- Looking at advanced Haskell type system features: GADTs, type families, ...

Some technical details

- Handwritten parser using monadic parser combinators (Parsec)
- Haskell $\rightarrow$ Core $\rightarrow$ Iridium $\rightarrow$ LLVM
- Tree traversals/folds implemented using attribute grammars (11 datatypes, some with 15 or 22 constructors)
- Strictness analysis, region analysis, uniqueness typing

Computation theory

We have just scratched the surface of **computation** theory

- More **formal** treatment of languages

Computation theory

We have just scratched the surface of **computation** theory

- More **formal** treatment of languages
- What about **unrestricted** grammars?
  - Turing machines, λ-calculus, μ-recursive functions.
  - Is there anything outside? **Non-computable** functions!

Computation theory

We have just scratched the surface of **computation** theory

- More **formal** treatment of languages
- What about **unrestricted** grammars?
  - Turing machines, λ-calculus, μ-recursive functions.
  - Is there anything outside? **Non-computable** functions!
- How much time/space does it take to compute?
  - This is called **complexity** theory
  - Some algorithms are linear, other exponential...
  - The famous $P = NP$ problem belongs here

What more?

We happen to have a **master** program about this!

- Concepts of program design
  - How do programming languages look like?
  - How do we formally describe execution of a program?
- Program semantics and verification
  - How do you check that a program satisfies its specification?
  - Testing, model checking
- Automatic program analysis
  - The rest of the phases: static analyses and code generation
  - Monotone frameworks, type-and-effect systems

Success with your exams!