Talen en Compilers
2018 - 2019, 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
More exercises, please
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 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 grammars, but many others (non-regular) may describe it, too.
Common pitfall 2: automata have an error state

Transitions are **total** functions, \( d : Q \rightarrow X \rightarrow Q \).
Common pitfall 2: automata have an error state

Transitions are total functions, \( d : Q \rightarrow X \rightarrow Q \).

![Diagram of a simple automaton](image-url)
Common pitfall 2: automata have an error state

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

![Diagram](image_url)

- Start state: $S$
- States: $S$, $A$, $B$, $E$
- Transitions:
  - $S \xrightarrow{a} A$
  - $A \xrightarrow{b} B$
  - $A \xrightarrow{a, b} E$
  - $E \xrightarrow{a, b} A$
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 $\neg A \implies \neg B$
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 cannot 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.
Environments, what the ???

Simplest language ever

data SLE = Literal Int | Add SLE SLE

eval :: SLE → Int

eval (Literal n) = n

eval (Add t₁ t₂) = eval t₁ + eval t₂
Simplest language ever

```haskell
data SLE = Literal Int | Add SLE SLE

eval :: SLE -> Int
eval (Literal n) = n
eval (Add t1 t2) = eval t1 + eval t2

type SLEAlg r = (Int -> r, r -> r -> r)

foldSLE :: SLEAlg r -> SLE -> r
foldSLE (l, a) (Literal n) = l n
foldSLE (l, a) (Add t1 t2) = a (foldSLE (l, a) t1) (foldSLE (l, a) t2)

evalAlg :: SLEAlg Int
evalAlg = (id, (+))
```
Environments, what the ???

Simplest language ever, with variables

```haskell
data SLE = Var Id | Literal Int | Add SLETerm SLETerm

type Env = Map Id Int  -- or something similar

eval :: SLE -> Env -> Int

eval (Var i) env = lookup i env
eval (Literal n) env = n
eval (Add t1 t2) env = eval t1 + eval t2
```
Environments, what the ???

Simplest language ever, with variables

```haskell
data SLE = Var Id | Literal Int | Add SLETerm SLETerm

type Env = Map Id Int -- or something similar

eval :: SLE → Env → Int
eval (Var i) env = lookup i env
eval (Literal n) env = n
eval (Add t1 t2) env = eval t1 + eval t2
```

How to write this as a fold?

```haskell

type SLEAlg r = (Id → r, Int → r, r → r → r)
```
Environments, what the ???

Solution 1 (cheating)
Notice that the environment never changes!

\[
evalAlg :: \text{Env} \rightarrow \text{SLEAlg Int}
\]
\[
evalAlg \text{ env} = (\text{flip lookup env}, \text{id}, (+))
\]
Environments, what the ???

Solution 1 (cheating)
Notice that the environment never changes!

| evalAlg :: Env → SLEAlg Int
| evalAlg env = (flip lookup env, id, (+))

Solution 2
Compare the following two type signatures:

| foldSLE :: SLEAlg r → SLE → r
| eval :: SLE → (Env → Int)
Environments, what the ???

Solution 1 (cheating)
Notice that the environment never changes!

\[
\text{evalAlg} :: \text{Env} \rightarrow \text{SLEAlg} \text{ Int} \\
\text{evalAlg} \ env = (\text{flip lookup} \ env, \text{id}, (+))
\]

Solution 2
Compare the following two type signatures:

\[
\text{foldSLE} :: \text{SLEAlg} \ r \rightarrow \text{SLE} \rightarrow r \\
\text{eval} :: \text{SLE} \rightarrow (\text{Env} \rightarrow \text{Int})
\]

\[
\text{evalAlg} :: \text{SLEAlg} \ (\text{Env} \rightarrow \text{Int}) \\
\text{evalAlg} = (\lambda i \ env \rightarrow \text{lookup} \ i \ env \ , \lambda n \ env \rightarrow n \ , \lambda n_1 \ n_2 \ env \rightarrow n_1 \ env + n_2 \ env) \quad -- \ (1)
\]
Environments, what the ???

Solution 2 is more extensible

```
data SLE = Var Id | Literal Int | Add SLETerm SLETerm
          | Let Id SLE SLE   -- let x = ... in ...
```

The environment must change in the `let`!
Solution 2 is more extensible

\[
\text{data SLE = Var Id | Literal Int | Add SLETerm SLETerm}
\]

| | Let Id SLE SLE -- let x = ... in ...

The environment must change in the \texttt{let}!

\[
\text{evalAlg :: SLEAlg (Env \rightarrow \text{Int})}
\]

\[
\text{evalAlg} = (\lambda i \text{ env} \rightarrow \text{lookup } i \text{ env} , \lambda n \text{ env} \rightarrow n , \lambda n_1 \ n_2 \text{ env} \rightarrow n_1 \text{ env} + n_2 \text{ env} , \lambda i \ n_1 \ n_2 \text{ env} \rightarrow \text{let } r_1 = n_1 \text{ env} \text{ in } n_2 (\text{insert } (i, r_1) \text{ env})
\]
More exercises, please
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

- 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
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**
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 subset
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 subset

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 subset

3. Program optimization

4. Code generation
   - Real architectures are more complex than SSM
   - Nowadays, you usually generate code in an intermediate language, like LLVM
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
Computation theory

We have just scratched the surface of computation theory

- More **formal** treatment of languages
- What about **unrestricted** grammars?
  - Turing machines, $\lambda$-calculus, $\mu$-recursive functions.
  - Is there anything outside? **Non-computable** functions!
We have just scratched the surface of computation theory

- More **formal** treatment of languages
- What about **unrestricted** grammars?
  - Turing machines, $\lambda$-calculus, $\mu$-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
▶ Compiler construction
  ▶ The rest of the phases: static analyses and code generation
  ▶ Monotone frameworks, type-and-effect systems
What I’ve learnt

You don’t take the union of languages L and L’ in The Netherlands, you use A and B.
What I’ve learnt

You don’t take the union of languages $L$ and $L'$ in The Netherlands, you use $A$ and $B$

This man here is **Ramon Llull**

- He lived in the 13th c.
- One of the first writers in Catalaan language
  - The language spoken in the East part of Spain (Calalunya, València...)
- Brought islamic knowledge to Europe
What I’ve learnt

You don’t take the union of languages L and L’ in The Netherlands, you use A and B

This man here is Ramon Llull

- He lived in the 13th c.
- One of the first writers in Catalan language
  - The language spoken in the East part of Spain (Calalunya, València…)
- Brought islamic knowledge to Europe

I can no longer see his name and not laugh!
Success with your exams!