Programming Paradigms
Lecture 2: Syntax (Part 1)

Join the course on Ilias! See link on
http://software-lab.org/teaching/winter2019/pp/

Prof. Dr. Michael Pradel
Software Lab, University of Stuttgart
Winter 2019/2020
Wake-up Exercise

What does the following C code print?

```c
#include <stdio.h>

int main() {
    int arr[3] = {5, 42, 23};
    int n = 2[arr];
    printf("%d", n);
}
```
Wake-up Exercise

What does the following C code print?

Result: 23

```c
#include <stdio.h>

int main()
{
    int arr[3] = {5, 42, 23};
    int n = 2[arr];
    printf("%d", n);
}
```

https://ilias3.uni-stuttgart.de/vote/0ZT9
Wake-up Exercise

What does the following C code print?

Result: 23

```c
#include <stdio.h>

int main () {
    int arr[3] = {5, 42, 23};
    int n = 2[arr];
    printf("%d", n);
}
```

Same as arr[2]: Computes a pointer *(arr + 2) and returns value stored at the address

https://ilias3.uni-stuttgart.de/vote/0ZT9
Motivation

- **Goal:** Specify a programming language
  - What code is part of the language?
  - What is the meaning of a piece of code?
- Important for both developers and tools
- **In contrast:** Natural languages not formally specified
Syntax vs. Semantics

Structure of code

Meaning of code

Example:

Grammar to define a language:  

digit → 0 | 1 | ... | 9  
non_zero_digit → 1 | ... | 9  
number → non_zero_digit digit*

Could mean

- Natural numbers
- Days of a 10-day week
- Colors
- ...
Syntax vs. Semantics

Structure of code

Meaning of code

Example:

Grammar to define a language:

digit → 0 | 1 | ... | 9
non_zero_digit → 1 | ... | 9
number → non_zero_digit digit*

Could mean

- Natural numbers
- Days of a 10-day week
- Colors
- ...

Focus of this and next lecture
Syntax of Different PLs

**Common:** Different syntax, same semantics

**Java:**

```java
if (foo > 100) {
  ...
}
```

**Bash:**

```bash
if [ $foo -gt 100 ]
then
  ...
fi
```
Syntax of Different PLs

Common: Different syntax, same semantics

Java:
```
if (foo > 100) {
    ...
}
```

Bash:
```
if [ $foo -gt 100 ]
then
    ...
fi
```

Sometimes: Same syntax, different semantics

Java:
```
if (foo > 100) {
    ...
}
```

JavaScript:
```
if ("abc" != 5) {
    ...
}
```

Branch is executed

Type error at compile time

Java:
```
if ("abc" != 5) {
    ...
}
```

JavaScript:
```
if ("abc" != 5) {
    ...
}
```

Branch is executed

Type error at compile time
4 concepts to specify syntax

- concatenation
- alternation / choice
- repetition ("Kleene closure")
- recursion

\[ \{ \text{regular expressions} \} \quad \text{recognized by scanners} \]
\[ \{ \text{context-free grammars (CFG)} \} \quad \text{recognized by parsers} \]
Plan for Today

- Specifying syntax
  - Regular expressions
  - Context-free grammars
- Scanning
- Parsing
Tokens

Basic **building blocks** of every PL

- Keywords, identifiers, constants, operators
- Think: ”Words” of the language

Example: C has more than 100 tokens

- Keywords, e.g., `double`, `if`, `return`, `struct`
- Identifiers, e.g., `my_var`, `printf`
- Literals, e.g., `6.022e23`, `'x'`
- Punctuators, e.g., `(`, `)`, `{`, `}`, `&&`
Regular Expressions

- Used to specify tokens
- A regular expression is one of:
  - A character
  - The empty string $\epsilon$
  - The concatenation of two regular expressions
    - Means a string generated by one or the other
  - A reg. expression followed by the Kleene star $*$
    - Means zero or more repetitions
Regular Expressions

- **Used to specify tokens**
- **A regular expression is one of:**
  - A character
  - The empty string $\epsilon$
  - The concatenation of two regular expressions
  - Two regular expressions separated by $|$ means a string generated by one or the other
  - A regular expression followed by the **Kleene star** $*$ means zero or more repetitions
  - No recursion!
Example

Numeric constants accepted by a calculator

number → integer | real
integer → digit*  
real → integer exponent | decimal (exponent 1E)
decimal → digit* (. digit | digit.) digit*
exponent → (e1E) (+1-1E) integer
digit → 0123456789
Quiz

Which of the following strings is accepted by the regular expression `number`?

https://ilias3.uni-stuttgart.de/vote/0ZT9
More compact syntax for regular expressions

Language of tokens: \{ c, cac, cbc, cacac, cbcbbc, cacbc, cbcac, etc. \}

Token → c More*

More → AorB c

AorB → a1 b

Shorter notation: Nest all in one

\[
\begin{array}{c}
\text{Token} \\
\hline
AorB \\
\hline
\text{More} \\
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{c} \\
\hline
( (a1b) c )^* \\
\hline
\end{array}
\]
Identifiers in Popular PLs

Different PLs allow different identifiers

- Case-sensitive vs. case-insensitive
  - E.g., foo, Foo, and FOO are the same in Ada and Common Lisp, but not in Perl and C

- Letters and digits: Almost always allowed

- Underscore: Allowed in most languages

In addition to syntax rules: Conventions

- E.g., Java: ClassName, variableName
Identifiers in Popular PLs

Different PLs allow different identifiers

- Case-sensitive vs. case-insensitive
  - E.g., foo, Foo, and FOO are the same in Ada and Common Lisp, but not in Perl and C
- Letters and digits: Almost always allowed
- Underscore: Allowed in most languages

In addition to syntax rules: Conventions

- E.g., Java: ClassName, variableName

Know the rules of the language you use!
White space in Popular PLs

Free format vs. formatting as syntax

- Spaces and tabs sometimes matter
  - E.g., in Python
- Line breaks sometimes matter
  - E.g., to separate statements in JavaScript or Python
Demo

demos/whitespace: show both stmts on one line; insert semi-colon; show not indenting print (after inverting condition)
Are regular expressions enough?

Specifying arithmetic expressions

E.g.,

\[
\begin{align*}
5 + 7 \\
(5 + 7) + 6 \\
(5 + 7) + 6 - 23
\end{align*}
\]

Each \boxed{\text{expression}} is an arithmetic expression

→ Want: Nesting

→ Need recursion
Context-free Grammars

≈ Regular expressions + Recursion

Example: Arithmetic expressions

\[ \text{expr} \rightarrow \text{id} \mid \text{number} \mid \text{expr op expr} \mid (\text{expr}) \]

\[ \text{op} \rightarrow + \mid - \mid * \mid / \]
Context-free Grammars

≈ Regular expressions + Recursion

Example: Arithmetic expressions

expr → id | number | expr op expr | ( expr )

op → + | - | * | /
Context-free Grammars

≈ Regular expressions + Recursion

Example: Arithmetic expressions

\[
\begin{align*}
\text{expr} & \rightarrow \text{id} \mid \text{number} \mid \text{expr} \ \text{op} \ \text{expr} \mid ( \ \text{expr} ) \\
\text{op} & \rightarrow + \mid - \mid \times \mid / \\
\end{align*}
\]

Non-terminals

Terminals = tokens of the PL
Context-free Grammars

≈ Regular expressions + Recursion

Example: Arithmetic expressions

\[ expr \rightarrow id \mid \text{number} \mid expr \ op \ expr \mid ( expr ) \]
\[ op \rightarrow + \mid - \mid * \mid / \]

Recursion allows for nesting expressions
Definition of CFGs

\[ G = (N, T, R, s) \]

- \( N \): finite set of non-terminals
- \( T \): finite set of terminals
  - alphabet of language
  - (for PPs) tokens of language
- \( R \): finite relation from \( N \) to \((N \cup T)^*\)
  - production rules
- \( s \): start symbol
Extensions of basic definition

- Kleene star
  \[ \text{id-list} \rightarrow \text{id} (, \text{id})^* \quad \text{means zero or more} \]

  \( \text{Short-hand for} \)
  \[ \text{id-list} \rightarrow \text{id} \]
  \[ \text{id-list} \rightarrow \text{id-list}, \text{id} \]

- Kleene plus
  \( \text{Same, but one or more} \)

- Vertical bar
  \[ \text{E.g., } \text{op} \rightarrow +|1-|1*|1/ \]

  \( \text{Short-hand for} \)
  \[ \text{op} \rightarrow + \]
  \[ \text{op} \rightarrow - \]
Derivations

Create concrete strings from the grammar

- Begin with start symbol
- Repeat until no non-terminals remain:
  - Choose production with a remaining non-terminal on the left-hand side
  - Replace it with right-hand side of the production (choose one option if multiple options)
Example:
\[ expr \rightarrow \text{id | number | expr | (expr) | expr \ op \ expr} \]
\[ op \rightarrow + | - | \times | / \]

Derivation of \( foo \times x + bar \)

\[ expr \quad \Rightarrow \quad expr \ op \ expr \]
\[ \quad \Rightarrow \quad expr \ op \ id \]
\[ \quad \Rightarrow \quad expr \ + \ id \]
\[ \quad \Rightarrow \quad expr \ op \ expr \ + \ id \]
\[ \quad \Rightarrow \quad expr \ \times \ id \ + \ id \]
\[ \quad \Rightarrow \quad id \ \times \ id \ + \ id \]

foo \quad x \quad bar
Parse Trees

Tree-structured representation of a derivation

- Root = Start symbol
- Leaf nodes = Tokens that result from derivation
- Intermediate nodes = Application of a production
Example

\[ \text{expr} \rightarrow \text{expr} \ast \text{expr} \]

\[ \text{id}(\text{foo}) \ast \text{id}(x) \]

\[ \text{expr} \rightarrow \text{expr} + \text{expr} \]

\[ \text{id}(\text{bar}) \]

\[ \text{expr} \rightarrow \text{id} \]
Not All Grammars are Equal

Each language has infinitely many grammars

Some grammars are ambiguous

- A single string may have multiple derivations
- Unambiguous grammars facilitate parsing

Grammar should reflect the internal structure of the PL

- E.g., associativity and precedence of operators
Example: Revised Grammar

A better version of the grammar of arithmetic expressions:

\[ \text{expr} \rightarrow \text{term} \mid \text{expr add_op term} \]
\[ \text{term} \rightarrow \text{factor} \mid \text{term mult_op factor} \]
\[ \text{factor} \rightarrow \text{id} \mid \text{number} \mid - \text{factor} \mid ( \text{expr} ) \]
\[ \text{add_op} \rightarrow + \mid - \]
\[ \text{mult_op} \rightarrow * \mid / \]
Quiz: Context-free Grammars

Draw the parse tree of $\text{foo} \ast \text{x} + \text{bar}$ with the revised grammar. How many nodes and edges does the tree have?

https://ilias3.uni-stuttgart.de/vote/0ZT9
Plan for Today

- Specifying syntax
  - Regular expressions
  - Context-free grammars
- Scanning
- Parsing
Implementing a Scanner

General idea

- Read **one character at a time**
- Whenever a **full token** is recognized, return it
- When no token can be recognized, report an **error**
- Sometimes, need to **look multiple characters ahead** to determine next token
Option 1: Ad-hoc Scanners

- Manually implemented
- Handle common tokens first
- Used in many production compilers

- Compact code
- Efficient scanning
Option 2: Finite Automata

- Each token specified by a regular expression
- Finite automata = Recognizers of regular expressions

Example:
\[ c ( ( a \mid b ) c )^* \]
Definition: DFA

Deterministic finite automaton (DFA):

\[ (Q, \Sigma, \delta, q_0, F) \]

- Finite set \( Q \) of states
- Finite set \( \Sigma \) of input symbols
- Transition function \( \delta : Q \times \Sigma \rightarrow Q \)
- Start state \( q_0 \)
- Set of accept states \( F \subseteq Q \)
DFA versus NFA

■ **Deterministic finite automaton (DFA):**
  □ At most one outgoing transition for each input symbol
  □ No $\epsilon$ transitions (empty word)

■ **Non-deterministic finite automaton (NFA)**
  □ Multiple outgoing transitions for same character
  □ May have $\epsilon$ transitions
See course on theoretical computer science or Chapter 2 of *Programming Language Pragmatics*
From NFA to Minimal DFA

- **NFA to DFA**
  - Why? To avoid exploring multiple possible next states during scanning
  - How? Powerset construction method (not covered here)

- **DFA to minimal DFA**
  - Why? Simplifies a DFA-based scanner
  - How? Remove unreachable and non-distiguishable states
From DFA to Scanner

Two popular options

- Implement the DFA using **switch statements**
  - Mostly in hand-written scanners

- **Table-based scanners**
  - Table represents states and transitions
  - Driver program indexes the table
  - Mostly in auto-generated scanners
Switch Statement Style

state = S1

Starting state: S1
Switch Statement Style

state = S1
token = ""
loop:

state = S1
token = ""
loop:
switch state:
case S1:
switch in_char:
case 'c': state = S2
else error
case S2:
switch in_char:
case 'a': state = S1
case 'b': state = S1
case ' ': return
else error
token = token + in_char
read next in_char

Loop reads one character at a time and builds the token
Switch Statement Style

state = S1
token = ""
loop:
    switch state:
        case S1:
            switch in_char:
                case 'c': state = S2
                else error
        case S2:
            switch in_char:
                case 'a': state = S1
                case 'b': state = S1
                case ' ': return
                else error
token = token + in_char
read next in_char

Switch statement that handles the current state
Switch Statement Style

state = S1
token = ""
loop:
    switch state:
        case S1:
            switch in_char:
                case 'c': state = S2
                else error
        case S2:
            switch in_char:
                case 'a': state = S1
                case 'b': state = S1
                case ' ': return
                else error
token = token + in_char
read next in_char

Switch statements to handle the current character
state = S1
token = ""
loop:
    switch state:
        case S1:
            switch in_char:
                case 'c': state = S2
                else error
        case S2:
            switch in_char:
                case 'a': state = S1
                case 'b': state = S1
                case ' ': return
                else error
    token = token + in_char
    read next in_char

Move to next state if character accepted
state = S1
token = ""

loop:
  switch state:
    case S1:
      switch in_char:
        case 'c': state = S2
        else error
    case S2:
      switch in_char:
        case 'a': state = S1
        case 'b': state = S1
        case ' ': return
        else error
  token = token + in_char
  read next in_char

Return the token when a space occurs
Switch Statement Style

state = S1
token = ""
loop:
    switch state:
        case S1:
            switch in_char:
                case 'c': state = S2
                else error
        case S2:
            switch in_char:
                case 'a': state = S1
                case 'b': state = S1
                case ' ': return
                else error
    token = token + in_char
    read next in_char

Raise an error for any illegal character
Table-based Scanning

Transition table indexed by state and input:

<table>
<thead>
<tr>
<th>State</th>
<th>'a'</th>
<th>'b'</th>
<th>'c'</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>-</td>
<td>-</td>
<td>S2</td>
<td>-</td>
</tr>
<tr>
<td>S2</td>
<td>S1</td>
<td>S1</td>
<td>-</td>
<td>token</td>
</tr>
</tbody>
</table>

Driver program

- moves to a new state,
- returns a token, or
- raises an error
Longest Possible Token Rule

- What if one token is a prefix of another?
  - Number 3.1 vs. number 3.141
  - Accept the longest possible token
    - 3.141 for the above example
- How to decide whether token has ended?
  - Scanner looks ahead (at least one character)
Quiz: Automata and Scanners

Which of these statements is true?

■ A scanner produces a sequence of tokens.
■ A scanner produces a syntax tree.
■ Efficient scanners are based on NFAs.
■ A scanner for C will turn ”ifStmt” into two tokens ”if” and ”Stmt”.

https://ilias3.uni-stuttgart.de/vote/0ZT9
Quiz: Automata and Scanners

Which of these statements is true?

■ A scanner produces a sequence of tokens.
■ A scanner produces a syntax tree.
■ Efficient scanners are based on NFAs.
■ A scanner for C will turn ”ifStmt” into two tokens ”if” and ”Stmt”.

https://ilias3.uni-stuttgart.de/vote/0ZT9
Plan for Today

- Specifying syntax
  - Regular expressions
  - Context-free grammars
- Scanning
- Parsing