AIL Technical Document

2IMP20, Domain Specific Language Design - Q4 (2022)

Group 17

Full Name	**Student ID**
Dmitrii Orlov	1068971
Mingzhe Shi	1665073

Department of Mathematics and Computer Science

Eindhoven, July 11, 2022

Contents

1 | Introduction

1.1 | Motivation

In agriculture, irrigation has always been a very critical factor. Precise control of irrigation resources and irrigation cycles can significantly improve the quality and yield of crops. However, as the most important link in the agricultural industry chain, not all farmers have enough experience and energy to control the irrigation of all crops.

For such and other purposes, emerging field of Internet of Things (IoT) provides the tools for increased automation and quality control on the farms [1]. To support the management of such systems, domain specific languages (DSL)s provide a gentle entrance into designing, implementing and managing the farm information and hence facilitate the adoption of IoT [2].

At the same time, software developed using the General programming language (GPL) often has too much redundancy, which increases the cost of learning [6]. Therefore, we think it is reasonable to design a domain-specific language (DSL) focusing on the field of agricultural precision irrigation.

1.2 | Inspiration

Our language includes the traits of two natures declarative, used to describe the configurations, and imperative, used to define the behavior.

We base our declarative syntax on other configuration languages such **JSON**, **YAML**, for their ease of use and popularity. Furthermore, when we describe specifications of each new element, we build upon the syntax used for *templating* in general purpose languages like **C++**, **Java**, **C#**, **TypeScript**.

For the behaviour specifications, we base the expression syntax on popular scripting languages such as **Python**, **Ruby** and **CoffeeScript**. We borrow their dynamic nature and the flexibility of mixing data declaration and behaviour. In order to keep our language syntax concise, we decided to borrow the iteration syntax widespread in Functional Programming languages instead of defining imperative **for**, **while**, or **until** loops. Namely, we specify our own syntax for the functional *map*, *filter*, and *reduce* operations. Such syntax allows to represent logically equivalent operations to what can be done with the imperative iteration constructs, and is essential in majority of general functional programming languages such as **Haskell**, **LISP**, **Elixir**. Furthermore, many imperative languages also have introduced these constructs as part of the standard library, such as **JavaScript**, **Python**, **C#**. The design of the operators representing the **map**, **filter**, and **reduce**, was based mostly on **Haskell**'s Functor operators and **Elixir**'s composition syntax.

1.3 | General information

1.3.1 | Name of the DSL

The name of this DSL: **AIL**, is extracted from the initials of Agriculture Irrigation Language as the name of our DSL. Since *ail* in French means *garlic*, we use this as a mascot and a logo for our language.

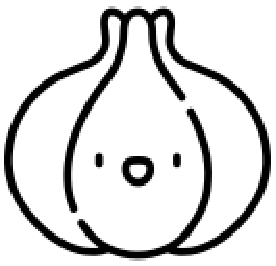


Figure 1.1: AIL Logo[5]

1.3.2 | About AIL purpose

AIL is a modular language for the collection and broadcast of farm irrigation information. AIL developers can develop targeted farm environmental monitoring programs for specific crop and hydrological conditions.In addition, developers can automatically monitor farm production by adding sensor components and functional function declarations with specific functions.

1.4 | AIL specification

AIL supports the modular definition and calling. Specifically, a complete AIL program is composed of the following five modules:

- Module
- Plants
- Actions
- Assets
- Controllers
- Observers

The following is a sample of a complete AIL program:

1.4.1 | Example code

```
Module : { Farm1 }
Plants : {
    Strawberry <<
        water_use : 5 mm,
        min_temp  : 20 C,
        max_water : 100 mm,
    >>
}
Actions : {
    notify <<%src, channels*>> (
        channels @> {c | send_msg : {c, %src.metadata}}
    ),
    check_water <<%src, condition, job, fail_job>> (
        if %src.water meets condition then do job else do fail_job
    ),
    check_avg_temp <<%src, condition, job>> (
        size = %src.items $> {a, _ | a + 1};
        total = %src.items $> {a, i | a + i.temp};
        if total/size meets condition then do job;
    ),
    send_report <<%src, channels*, name_match>> (
        channels  #> {c | c ~~ "abie"}} @> {c, send_msg : {c, %src.metadata}}
    )
}
Assets : {
    strawberry_plants : Plantation <<Strawberry>> : {
        desc: "Strawberry fruits along the South border",
        area: 300 m^2,
        plantation_date: "2022-03-02",
        sensors : {
            water_level : Sensor : {
                "daily 9 am",
                check_water : { > 30 mm, notify : { [support] } }
            }
        }
    }
}
```

```
Observers : {
    support <<support@farm.cool>>,
    admin <<admin@farm.cool>>
}
Controllers: {
    // Predefined construct, controls the sensor schedule.
    MONITOR
}
```

1.4.2 | Modules description

Among these modules, Module, Plants, Actions, Assets, and Observers are all declaration modules. Controllers is the execution module.

Module mainly records the program name and calls the external library in the future. Plant is the template definition module for farm crops and water tanks. These templates can be called in Assets to declare instances of entities.

Actions is the function definition module of AIL. The functions are mainly divided into two types: notification function and check function. The notification function can send information to the specified observer's contact address. The check function is used to detect the status parameters of the recorded assets and can also judge the status parameters and run other actions.

Assets module is used to declare specific farm properties and configure specific properties and corresponding monitoring and broadcasting commands. The parameters of Assets can be set by calling the Plants template and adding them manually. The receiver of the monitoring information sent by Assets in the AIL programs can be defined in the Observers module.

Controllers is the execution module of the AIL program. By adding predefined commands, the AIL program can execute the corresponding functions to monitor farm assets.

1.4.3 | Well-formedness description

- AIL supports expressions and basic operations, and adds unit numbers and conversions between unit numbers on this basis. Therefore, the well-formedness of AIL is concerned with the correctness of supporting expressions.
- AIL needs to execute the templates, functions, addresses and instances declared and defined in other modules in the Controllers module, so AIL prohibits the repeated use of module labels.
- AIL focuses on agricultural asset monitoring, so there are strict restrictions on the configuration of each module, thereby reducing contradictions and ambiguities when developing programs. For example, the template setting of Plants supports two template prototypes: $WaterSource$ and $Plantation$, and the template parameters must be in the specified parameter library.
- Since ALI needs to perform the actual message delivery and periodic status checking, it must be ensured that the addresses that receive messages conform to the specifications for mail, RSS, etc. In addition, due to the requirements of the scheduling job, it is also necessary to ensure the accuracy of the scheduled time setting.

1.4.4 | Python code generation

To make AIL executable, AIL code is translated into equivalent Python code. This happens at the same time with the parsing. The listing in Appendix C, demonstrates an intended Python translation from a sample AIL program.

1.5 | MkDocs

To show the syntax of AIL more clearly, MkDocs is used to build concrete syntax documentation website.

MkDocs is a fast, simple,downright gorgeous static site generator for building project documentation[7]. Documentation source files are written in Markdown and configured using a single YAML configuration file.

The detailed concrete syntax of AIL is documented in markdown files, which are integrated with MkDocs to generate AIL's documentation website. For a more specific look at AIL, following the instructions in section 2.4, the makefile provided can generate a local AIL website.

2 | Technical description

2.1 | Flex&bison tools

Lex and Yacc were the first popular and efficient lexer and parser generators, and Flex and Bison were the first widely open source versions compatible with the original software[3, 4]. By now, Flex and Bison have been used in many different settings and have a strong user base.

Flex and Bison sources need to first be compiled into C code. The result of this compilation can be used to build the executable *tokenizer* and *parser* respectively. [4] A combination of these two tools, allows to have a full suite in analysing the input source code.

Development of the surrounding features, extending the basic parsing and tokenizing purposes, can be done with any C-compatible dependencies. This allows to build arbitrary complex systems bounded only by the developer's skill.

Since Flex and Bison are technically environment agnostic and only depend on their own GNU CLI tools, there is no restriction in the development environment. Visual Studio Code IDE, as one of possibilities, can be easily adapted to support efficient development tasks for Flex and Bison projects by installing relevant extensions, and benefit from all other utilities offered by the IDE.

Combining all these features, using the Flex and Bison allowed the DSL developers to package together the **parsing** of the AIL code and **generation/compilation** of the corresponding Python code.

2.2 | Dependencies

To support the entire AIL development, the following dependencies are required:

- **gcc**: The C language compiler that development requires.
- **flex**: Lexical analyzer generator.
- **bison**: General-purpose parser generator.
- **Python3.9+**: Required for executing the generated Python code and serving the **MkDocs** site.
- **MkDocs**: Site generator for the documentation.

2.3 | Development details

In the AIL project development, intermediate, test, and generated files are stored separately to ensure that the project is orderly and neat. The following is an explanation of the important project folders and the execution details.

2.3.1 | Project directory

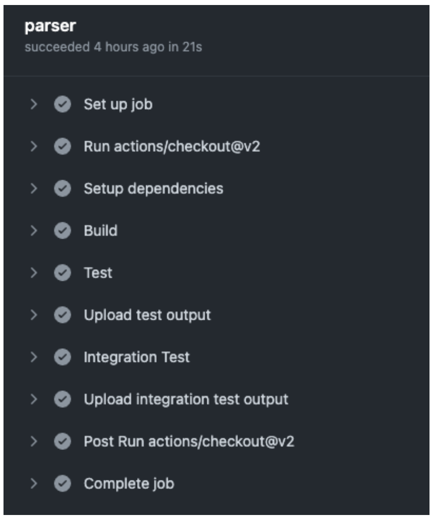
- **src/** - Containing the Flex and Bison tokenizer and parser sources.
- **lib/** - additional code supporting the AIL parser (wellformedness checks + Python code generation).
- **bin/** - is created during compilation and contains all executables..
- **dist/** - is created during the run of the AIL parser executable, and contains the generated Python code.

2.3.2 | Execution details

The compiled executable can be found (after compilation) in **bin/parser.out**. Each time this executable is ran on a AIL file, a new Python distribution code is generated in **dist/** folder. For example, running **bin/parser.out <foo.ail** will generate the following Python file: **dist/<foo's module identifier>.py**.

2.4 | Development flow

1) **Makefile** contains a bunch of convenient commands which are helpful for the development and testing.
2) Testing the parser with multiple test **.ail** files. Through the test results, the location of compilation errors is found, and targeted modifications can be made.
3) Integration testing involved running the generated Python code to verify that it is correct and can be executed.
4) GitHub Actions [2.1] was used to test the AIL project with on every new commit, building and testing the project according to the above flow in the Ubuntu environment to make sure the project is compatible.
5) IDE extensions



parser
succeeded 4 hours ago in 21s

- Set up job
- Run actions/checkout@v2
- Setup dependencies
- Build
- Test
- Upload test output
- Integration Test
- Upload integration test output
- Post Run actions/checkout@v2
- Complete job

Figure 2.1: CI/CD with Github

2.5 | Test cases

All test cases we set are in **test/** folder.

Test file	Test case
controllers_observers_and_plants.ail	if this configuration can be parsed
empty_module.ail	if module can be detected
garlic_and_potato.ail	two correct plants
observers_and_plants.ail	configuration with observers and plants only
only_garlic_and_comments.ail	comment support
only_garlic_observers_and_controllers.ail	only these specific constructs
only_garlic_with_collection.ail	collection value parsing test
only_garlic.ail	simple single plant module case
only_observers.ail	a module with only observers
plants_actions_assets_observers_controllers.ail	fully complete and extensive module
plants_actions_assets.ail	specific subset of the configuration
plants_actions_observers.ail	specific subset of the configuration
wellformedness_check.ail	annotated file for well-formedness violations

Table 2.1: Test Cases

Once evaluated by our parser, a corresponding Python file is written in the **dist/** folder.

3 | References

[1] Mahmoud Abbasi, Mohammad Hossein Yaghmaee, and Fereshteh Rahnama. Internet of things in agriculture: A survey. In *2019 3rd International Conference on Internet of Things and Applications (IoT)*, pages 1–12, 2019.

[2] Desirée Groeneveld, Bedir Tekinerdogan, Vahid Garousi, and Cagatay Catal. A domain-specific language framework for farm management information systems in precision agriculture. *Precision Agriculture*, 22(4):1067–1106, August 2021.

[3] Tony Mason John Levine, Doug Brown. *lex yacc, 2nd Edition.* O'Reilly Media, Inc., 1992.

[4] John Levine. *Flex & bison.* O'Reilly, Sebastopol, CA, 1st ed edition, 2009. OCLC: ocn321016664.

[5] Flaticon License. http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/. [Online].

[6] Marjan Mernik, Jan Heering, and Anthony Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37:316–, 12 2005.

[7] MkDocs Team. https://www.mkdocs.org. [Online].

A | Video

The video, along with other report content is included in the **reports** directory of the submission archive. If the video is not available at this address, please refer to a hosted file at a Google Drive https://drive.google.com/drive/folders/1cYkBtX88PPF4h1PnLYjtnDk5xqFSuiNK?usp=sharing

B | Archive README

A **README.md** is provided at the root of the submission archive to describe the submission structure and provide other relevant details.

C | Generated Python code

```
# Automatically compiled Python module.
# Module: TEST_GARLIC_ACTIONS_ASSETS_OBSERVERS_CONTROLLERS

####################
# Standard Imports #
####################

from abc import ABC, abstractmethod
from dataclasses import dataclass
from typing import Any, List, Dict, Callable, Union

#########################
# Defaults & Constructs #
#########################

@dataclass
class UnitNumber:
    value: Union[int, float]
    unit: str

class Plant:
    pass

@dataclass
class Sensor:
    trigger: str
    action: Callable

    def check(self):
        print(f"Sensor trigger: {self.trigger}")
        self.action()

@dataclass
class Observer:
    address: str

class Observable:
    sensors: Dict[str, Sensor]

class Asset:
    pass

class Plantation(Asset, Observable):
    pass
```

```
class WaterSource(Asset, Observable):
    pass

class Controller(ABC):
    @abstractmethod
    def bootstrap(self):
        pass

    @abstractmethod
    def run(self):
        pass

######################
# Built-in Utilities #
######################

@dataclass
class MonitorController(Controller):
    assets: List[Asset]
    __triggers = {}

    def __register_trigger(self, trigger, sensor):
        self.__triggers = {
            trigger: sensor,
            **self.__triggers
        }

    def bootstrap(self):
        print("Setting up the controller...")
        for asset in self.assets:
            for key, sensor in asset.sensors.items():
                self.__register_trigger(sensor.trigger, sensor)

    def run(self):
        triggers = self.__triggers.keys()
        print(f"Registered monitor triggers: {triggers}")
        for trigger, sensor in self.__triggers.items():
            print(f"Assume trigger happened: {trigger}...")
            sensor.check()

def Action_send_message(*args):
    def __internal_function__():
        message_body = ";;;".join([str(a) for a in args])
        print(f"INCOMING MESSAGE: {message_body}")
    return __internal_function__

###################
# Module contents #
###################

# Plants
# ~~~~~~~~

class Garlic(Plant):
    approx_volume = UnitNumber(1, "\cm^2")
    water_use = UnitNumber(5, "\mm")
    min_temp = UnitNumber(20, "\C")
```

```
    max_water = UnitNumber(100, "\C")

PLANTS = [Garlic]

# Actions
# ~~~~~~~

@dataclass
class Action_notify:
    s_src: Any
    channels: Any

    def __call__(self):
        list(map(lambda c: Action_send_message(c, "regular_update", self.s_src)(), self.channels))

ACTIONS = [Action_notify]

# Observers
# ~~~~~~~~~

support = Observer("support@farm.cool")
admin = Observer("admin@farm.cool")
rss = Observer("farm.feed.rss")
abbie = Observer("abie@service.farm.cool")

OBSERVERS = [support, admin, rss, abbie,]

# Assets
# ~~~~~~

class Asset_garlic_north(Plantation, Garlic):
    def __init__(self):
        self.desc = "Garlic plants on the north edge"
        self.area = UnitNumber(300, "\m^2")
        self.plantation_date = "2022-11-02"
        self.sensors = {
            "water_level": Sensor(
                "daily 9 am",
                Action_notify(self, [support, admin])
            )
        }
garlic_north = Asset_garlic_north()

class Asset_water_tank(WaterSource):
    def __init__(self):
        self.installation_date = "2020-02-02"
        self.max_temp = UnitNumber(60, "\C")
        self.capacity = UnitNumber(30, "\l")
        self.sensors = {
            "water_level": Sensor(
                "daily 10 am",
                Action_notify(self, [ support, admin, rss ])
            )
        }
water_tank = Asset_water_tank()

ASSETS = [garlic_north, water_tank]
```

```
# Controllers
# ~~~~~~~~~~~

MONITOR = MonitorController(ASSETS)
CONTROLLERS = [MONITOR]

# FIXME: add support for parallel processing
def main():
    # bootstrap controllers
    for controller in CONTROLLERS:
        controller.bootstrap()
    # run controllers
    for controller in CONTROLLERS:
        controller.run()


if __name__ == "__main__":
    main()
```