CS3423 - Compilers II

GD

A language for Geometry-Graph Operations

Sunday 22nd October, 2023
Team Number: 10

Name	Roll Number
K Vivek Kumar	CS21BTECH11026
Bhende Adarsh Suresh	CS21BTECH11008
Jarupula Sai Kumar	CS21BTECH11023
Pathlavath Shankar	CS21BTECH11064

Contents

1 Introduction

This document introduces our project, which focuses on developing a Domain Specific Language (DSL) and overview of its associated compiler as part of the final course project. The DSL is designed to address specific needs within a particular domain or problem space. The compiler is the core component responsible for translating DSL code into executable instructions.

2 Motivation

In our quest for a groundbreaking solution, we sought to revolutionize the way we approach a recurring challenge.

2.1 Identifying the Problem

The conventional method of tackling geometry and graph-based problems typically involves the laborious use of pen and paper. Translating these problems into programming languages such as C and C++ can be cumbersome. To simplify and streamline this process, we embarked on a journey to craft a specialized programming language tailored specifically for solving geometric problems on graphs.

2.2 Expanding the Concept

Geometry, a multidimensional realm, presents a unique challenge. Operations across different dimensions share similarities, but they also exhibit significant differences. To address this issue comprehensively, we introduce a structured approach. It is essential to define the geometric context for your program one dimension at a time. This approach is pivotal, as it allows the compiler to comprehend the syntax and supported operations within each dimensional geometry.

For instance, every program intended for the GD Compiler should commence with an environment declaration, following the syntax provided below:

```
To define the geometry environment:
env:
    g1d; -------------> 1-Dimensional Geometry


env: g1d; ------------> 1-Dimensional Geometry


env:
    g2d; -------------> 2-Dimensional Geometry


env: g2d; ------------> 2-Dimensional Geometry
```

2.3 Goals

Our DSL program aspires to provide the following key features:

1. Simplification of program development.
2. An elegant and easily comprehensible syntax.
3. Efficient translation of geometric concepts into code.

With these objectives in mind, we aim to redefine the way we approach and solve geometry and graph problems in the realm of programming.

3 Data Types

Data types can be broadly categorized into two main groups: primitive data types and non-primitive data types. In our case, we exclusively employ primitive data types.

Table 1: Data Type Table

Types	**Type of Data Type**
int	primitive
bool	primitive
real	primitive
point	primitive
lineseg	primitive

3.1 Data Type Selection

In the design and development of our domain-specific language, GD Language, for geometric operations, we have meticulously selected specific data types to represent various aspects of our language's domain. These choices are rooted in the functional requirements. Below, we provide justifications for the selection of each data type:

3.1.1 int

Usage: Int data types are employed as the return type for various operations.

3.1.2 bool

Usage: The "bool" data type serves as a representation for boolean values.

3.1.3 real

Usage: The "real" data type is used to handle real values.

3.1.4 point

Usage: The "point" data type is chosen to denote the location of data on a 1-dimensional line.

3.1.5 lineseg

Usage: The "lineseg" data type is the selected representation for declaring a line segment on a 1-dimensional line.

4 Operations

Table 2: Operations

Special symbol	Action
star	check if two linesegment or two point touch each other
hashtag	check the overlape of two linesegment or two point
arrrow	check if one point or linesegment is on another one
addition	for incrementing the length of line segment
pipe	for check if first point or linesegment is on left of second
negation	used for negating the conditions

4.1 `l1 * l2`

l1 and l2 are linegements This command is used to check the touch between two linesegment.

4.2 `a # b`

a and b are declared linesegments This command check the overlap length between two linesegments

4.3 `a$->b`

(single doller represent the first point of linesegment whereas double doller represent the second point of linesegment)

check if the first point of linesegment a is on linesegment b

4.4 `a = a + c`

increase the length of linesegment a by c in right direction,where c is a constant

4.5 `a = c + a`

increase the length of linesegment a by c in left direction,where c is a constant

This method to write new content into the file.

4.6 `a | b`

this will check if the linesegment a is on left of linesegment b

4.7 `~(a*b)`

This operator is for negation

5 Statements

Within the DSL, numerous statements play essential roles in its operations. In the following subsections, we introduce each of these statements and their specific functions.

5.1 Comments

Comments play a vital role in any programming language, offering a valuable way to provide program visitors with an overview of the program's purpose and functionality. In our DSL, we've incorporated the concept of comments, allowing for clear program documentation. A sample program snippet is presented below:

5.1.1 In-Line Comment

```
...

---------> This is a single-line comment.
------> Arrow should be atleast of size3 "-->"

...
```

5.1.2 Multi-Line Comment

Arrow should be atleast of size `"-!>"`

```
...

---------!>
This is a multi-line comment.
It can span across multiple lines.
!>

...
```

5.2 Declaration Statements

Declaration statements involves introducing variables of various datatypes. This can include one of the following:

1. Declaring number constants

```
-!> This declares a variable which can hold integer constant.!>
int <identifier>;
int i;
```

2. Declaring float constants

```
-!> This declares a variable which can hold decimal constant.!>
real <identifier>;
real i;
```

3. Declaring Points

```
-!> This declares a variable which can a point data on 1-dimensional axis.!>
point <identifier>;
point p;
```

4. Declaring Boolean

```
-!> This declares a variable which can store a boolean variable.!>
bool <identifier>;
bool b;
```

5. Declaring Line segments

```
-!> This declares a variable which can hold end points of a line segment. !>
lineseg <identifier>;
lineseg l;
```

5.3 Check Section

Its a combination of 'if-else' and 'loop' across other programming language. There is a common syntax for both these operation with a small difference in approach of implementing this.

5.3.1 When using Check as if-else

```
----!>
Using Check as if-else
Same like if-else ladder in C/C++.
----!>

check (condition) {
    ... ----------------------> If gets true;
} : check (condition) {
    ... ----------------------> else if gets true;
} : {
    ... ----------------------> else
}
```

5.3.2 When using Check as loop

```
----!>
Using Check as loop
Same like while in C/C++.
----!>

check (condition) {
    ... ----------------------> If gets true;
    recheck; -----------------> Goes into as a loop to re
}
```

5.4 Main Function

Our main function should be declared with a special keyword as `main` only.

5.5 Functions

5.5.1 Function Declaration

```
func1:
    (point a, lineseg b) => (bool) {
        ...
    }
```

5.5.2 Function Call

```
call func1(a, b);
```

5.5.3 Example Code

```
---------------------------------------!>
Problem Statement:
Code to 'Check in how many steps two line segments shall meet if first line segment
(length l1) started to grow (towards L2) from point x and second line segment
(length l2) started to grow (towards L1) from point y'.
---------------------------------------!>

env:
    g1d;

main:
    () => () {
        point x = 3; ------------------> Declaring the first point
        point y = 12; -----------------> Declaring the second point
        lineseg l1 = x.toright(5); ----> Declaring line segment L1
        lineseg l2 = y.toright(4); ----> Declaring line segment L2
        int a = 0; --------------------> Declaring initial step count
```

```
        check(l1 * l2) { --------------> Check if they touch
            l1 = l1 + 1; --------------> Increment 1 length from right of L1
            l2 = 1 + l2; --------------> Increment 1 length from left of L2
            a = a + 1; ----------------> Increment step count
            check; --------------------> Recheck the condition
        } ----------------------------> Shall break if check failed
        show(a); ----------------------> Print the output
    }
```

6 Lexical Stage

6.1 Reserved Keywords

env	Describing the environment
g1d	1-dimensional environment
g2d	2-dimensional environment
check	Check condition
recheck	Recheck the check
main	Main function
return	Return statement
show	Print
export	Exporting a function from the file
call	Calling a function

6.2 Punctuations

;	End of line
:	End of scope
=	Assignment operator
(	Start of list
)	End of list
{	Start of scope
}	End of scope

6.3 Identifiers

An identifier is a sequence of characters that can consist of a combination of letters (alphabets) and numbers. The identifier must begin with a letter (alphabet). It is not allowed for the identifier to be composed solely of numbers.

Regular expression for an identifier in GD Language is:

```
[a-zA-Z][a-zA-Z0-9]*
```

7 Syntax

7.1 Sample Programs

1. **Sample Program 1**

```
---------------------------------------!>
Problem Statement:
Code to 'Check in how many steps two line segments shall meet if first line segment
(length l1) started to grow (towards L2) from point x and second line segment
(length l2) started to grow (towards L1) from point y'.
---------------------------------------!>

env:
    g1d;

main:
    () => () {
        point x = 3; ------------------> Declaring the first point
        point y = 12; -----------------> Declaring the second point
        lineseg l1 = x.toright(5); ----> Declaring line segment L1
        lineseg l2 = y.toright(4); ----> Declaring line segment L2
        int a = 0; --------------------> Declaring initial step count
        check(l1 * l2) { --------------> Check if they touch
            l1 = l1 + 1; --------------> Increment 1 length from right of L1
            l2 = 1 + l2; --------------> Increment 1 length from left of L2
            a = a + 1; ----------------> Increment step count
            check; --------------------> Recheck the condition
        } ----------------------------> Shall break if check failed
        show(a); ----------------------> Print the output
    }
```

2. **Sample Program 2**

```
-----------------------------------------------!>
Problem Statement:
Code to 'Find the common length as a line segment'.
-----------------------------------------------!>
```

```
env: ----------------------------------------------> Setting up the environment
    g1d; ------------------------------------------> Environment in 1-dimensional geometry

about commonLineSegment: ------------------------> Function declared
    (lineseg a, lineseg b) => (lineseg) { -------> arguments and returns
        check (~(a * b)) { ----------------------> Checks if a and b touches each other
            return null; ------------------------> Returns null, any-datatype
        } : check(a # b == 0) { -----------------> Length of the line segment crossed
            check(a$ -> b) { --------------------> If the first point is on b
                lineseg c = (a$, a$); -----------> Make line segment, from a to itself
                return c; ---------------------> Return that line segment
            } : { -------------------------------> If the check fails
                return (a$$, a$$); --------------> Line segment from a[2] to a[2]
            } ----------------------------------> Come out
        } : { ------------------------------------> If the check fails
            check(a | b) { ----------------------> Check if a is on the left of b
                return (b$, a$$); ---------------> Return the appropriate line
            } : { -------------------------------> If the check fails
                return (b$$, a$); ---------------> Return the appropriate line
            } ----------------------------------> Come out
        } ----------------------------------------> Come out
        return null; ----------------------------> Returns the universe datatype
    } ------------------------------------------> Come out

about main: -------------------------------------> Main function
    () => () { --------------------------------> Arguments
        lineseg l1 = (3, 4); --------------------> Declaring a line segment
        lineseg l2 = (4, 5); --------------------> Declaring a line segment
        lineseg l = call commonLineSegment(l1, l2); ----> Calling the function
        show("Common line is ", l); -------------> Print the result out
    }
```

8 Advantages

The GD Language offers several advantages due to its unique design:

- **User-Friendly Syntax:** GD prioritizes a user-friendly experience, making the language easy to learn and work with.
- **Optimized Operations:** Our compiler shall be designed to optimize each operation, ensuring efficient execution.

9 Challenges

Despite its advantages, GD may encounter certain challenges while development, including:

- **Expanding the idea:** Introducing `g2d` and further dimensions on the language.
- **Loop and if-else combination:** Translation of this combination can be a difficult part.
- **Completeness Considerations:** Though we try to cover for nested expression workings, still the completeness can be a doubt.

10 Conclusion

Providing an overview of the language has been a valuable exercise in understanding the compiler's inner workings and the language's constraints. We are committed to implementing as many features as possible from this document. Please note that the features outlined are subject to refinement and further development.