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
real	primitive

4 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 2: Data Type Table

Types	**Type of Data Type**
point	buffer
line	buffer
polygon	buffer

4.1 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:

4.1.1 int

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

4.1.2 real

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

4.1.3 point

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

4.1.4 line

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

5 Operations

Table 3: Operations

Special symbol	Action
@point	plotting the point
@line	plotting the line
@polygon	plotting the polygon
@view	view graph
@graph	plot graph

5.1 `a = a + c`

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

5.2 `a = c + a`

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

6 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.

6.1 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:

6.1.1 In-Line Comment

```
...

~ This is a single-line comment.

...
```

6.2 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 Line segments

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

6.3 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.

6.3.1 When using Check as if-else

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


check (condition) {
    ~ If gets true;
}
else (condition) {
   ~ else if gets true;
}
else{
   ~
}
```

6.4 loop Syntax

```
  loop(condition){
  ~  condition is true
  }
```

6.5 Main Function

NO main fuction requrement as in C++ ,if their is no main fuction it would not be compiled to executable C++ file

6.6 Functions

6.6.1 Function Declaration

```
func1(point a, lineseg b) => (bool) {
        ~ function body
    }
```

6.6.2 Function Call

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

6.6.3 Example Code

```
---------------------------------------!>
Problem Statement:
Plot a fibonacci sequence upto 20th fibonacci number
---------------------------------------!>
```

```
int=>main(){
    int=>i; ~ This is our y
    int=>j; ~ This is our x
    int=>prev;
    int=>current;
    int=>temp;

    i = 0;
    j = 0;
    prev = 0;
    current = 1;

    loop(i < 20){
        y = i;
        x = j;
        @point

        temp = current;
        current = current + prev;
        prev = temp;
        y =current;
        x = prev;
        @line
        i = i + 1;
        j = current;
    }

    @graph
    @view
    return 0;
}
```

7 Lexical Stage

7.1 Reserved Keywords

g1d	1-dimensional environment
g2d	2-dimensional environment
check	Check condition
else	Recheck the check
main	Main function
return	Return statement

show	Print
view	show the plotted geometry
graph	plot the graph

7.2 Punctuations

@	for calling special functions
;	End of line
:	End of scope
=	Assignment operator
(	Start of list
)	End of list
{	Start of scope
}	End of scope

7.3 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]*
```

8 Syntax

8.1 Sample Programs

1. **Sample Program 1**

```
---------------------------------------!>
Problem Statement:
Ploting the y= x^2

---------------------------------------!>

int=>main(){
    int=>i; ~ This is our y
    int=>j; ~ This is our x
    i = 0;
```

```
    j = 0;
    loop(i < 10){
        y = i;
        x = j;
        @line
        i = i + 1;
        j = i * i;
    }
    int=>m;
    int=>n;
    m = 3;
    n = 4;
    x = m;
    y = n;
    @point
    m = 9;
    n = 8;
    x = m;
    y = n;
    @point
    @graph
    @view
    return 0;
}
```

2. **Sample Program 2**

```
~ Some other function for calling...
void=====>main1(){
    real a;
    int b;
    show(a) newline;
    *****
    int k;
    a = b;
    return;
}

~ Some other function
void=====>main2(){
    real a;
    int b;
    b = 3;
    a = 5 + b;

    ~ Handling complex variables expressions
    a = 5 - b * 3 + b % b - a / a;
```

```
    show(a) newline;
    *****
}
int===>main3(){
    real a;
    int b;
    show(a);
    newline
    *****
    a = b;
    return 0;
}

~ This is a comment
~ =======> arrow can be used for better code readability, just use arrow length greater than =>
int=====>main(){
    real a;
    a = 6;
    int b;
    int x;
    x = 5 * 5 / 5;

    ~ Implementing cout statement as in C++
    ~ newline keyword to add a line break in console
    show(a) newline;

    ~ Eat five star! Do Nothing!
    *****
    a = 9;

    ~ Function Call with hello
    hello main2();

    ~ Similar to switch statement in C
    is (x){
    equals 5:
        show(x) newline;
    default:
        show(x - 5) newline;
    }

    ~ Similar to while loop in C
    loop (x > 2) {
        x = x - 1;
        show(x) newline;
    }
```

```
    for(x = 0; x < 7; x = x + 1){
        show(x) newline;
    }
    b = 9;
    x = a;
    y = b;
    return 0;
}
```

9 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.

10 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.

11 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.