The Setup Programming Language

Ian Erb (ire2102)
Bill Warner (whw2108)
Adam Weis (ajw2137)
Andrew Ingraham (aci2110)

December 22, 2011

Contents

Chapter 1

Introduction

The SETUP language was created to leverage the clear and concise syntax of set formalisms in mathematics. With SETUP , users can generate sets using a variety of logic and combinatorial techniques that would require significantly more lines of code in other imperative languages such as Java, C or C++. SETUP was designed with the following goals in mind:

- **Set Theory Abstraction.** SETUP provides a framework for handling data which many mathematicians and scientists will find familiar and easy to use. By maintaining a sufficient level of abstraction, SETUP can minimize the time needed to go from concept to concrete, working code.
- **Minimal Code Generation.** SETUP greatly reduces the amount of code needed to perform routine tasks. The user will find that nested loops can be virtually eliminated from code, making program files lighter and debugging easier.
- **High Level of Readability.** SETUP aims to be a clear and concise programming language with intuitive commands and syntax which mirror the mathematical underpinnings of set theory.

Chapter 2

Setup Tutorial

Each program you write in SETUP should be placed in a plain text file. By convention, the `.su` suffix is used to denote programs written in valid SETUP code. Let's write a simple program to get the hang for how SETUP works.

2.1 A Simple Scheduling Program

2.1.1 Writing the Code

Create a text file entitled `schedule.su` and open it in your text editor. In this file, place the following text:

```
/* Our First Program */
function main[] returns int {
   set days = {"M","T","W","Th","F"};
   set hrs = {1...24};
   set week = Days cross hrs;
   return 0;
   }
```

We use the keyword `set` to indicate that we are defining a variable name for a set

2.1.2 Compiling a Program

```
-$ ./cWriter.native < hello.su
```

2.2 Sets and Tuples

2.3 Using Set-Builder Notation

Chapter 3

Language Reference Manual

3.1 Overview

3.1.1 Introduction

The language of set theory is widely used by mathematicians and scientists to construct and manipulate large collections of objects in an abstract setting. The SETUP language implements the most useful and commonly used abstractions (e.g., union, intersections, direct products) from set theory in a language which users of other common imperative languages such as C, C++ and Java will find very simple and intuitive.

SETUP was created with three goals in mind:

1. Construct sets in a logical way
2. Perform operations on large sets with minimal code generation
3. Maintain a high level of readability

We anticipate users will solve simple set-oriented problems like scheduling, logic, and probability problems.

3.1.2 A Motivating Example

SETUP greatly reduces the amount of code required to generate and manipulate sets. A level of abstraction is provided which allows the user to work with sets

in an intuitive way – simultaneously improving readability. For example, the need for writing successive nested loops is practically eliminated.

Suppose the user wanted to generate all possible ordered pairs of numbers from 1 to 10. In C++, we would write:

```
struct { int first; int second; } pair;
vector<pair> pairs;
for(int i = 1; i!= 11; i++){
    for(int j = 1; j!=11; j++){
                pair temp;
                temp.first = i;
                temp.second = j;
        vector.push_back(temp);
    }
}
```

In SETUP , the equivalent code can be written:

```
set A = {1...10};
set pairs = {(x,y) | x in A, y in A};
```

Making use of `cross` – a built-in cartesian product operator – we can further simplify our code to a single line:

```
set pairs = {1...10} cross {1...10};
```

Though somewhat trivial, the above example shows how we can greatly reduce code and improve readability at the same time. Suppose we wanted to remove duplicates `(1,1) ... (10,10)`. We would simply write

```
set unique_pairs = pairs minus {(x,x) | x in {1...10}};
```

Here we highlight another useful built in operator – `minus` – which returns the complement of the second set within the first set. That is, `A minus B` $= A \cap B^c$.

Working with SETUP is simple and getting started is easy.

3.1.3 Lexical Conventions

The language has 5 basic tokens:

- Identifiers

- Keywords
- Literals
- Operators
- Punctuation

Whitespace characters (blanks, tabs and newlines) are ignored and used only to separate tokens. At least one whitespace character is required to separate adjacent tokens.

Comments

Block comments are introduced with `/*` and terminated with `*/`. Nesting of comments is not permitted. Comments are in general ignored by the compiler.

Identifiers

An identifier is a sequence of letters and digits. The first character must be alphabetic. Names are case-sensitive.

Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

`set`	`int`	`bool`	`float`	`string`
`if`	`tuple`	`in`	`then`	`union`
`else`	`intersect`	`minus`	`while`	`cross`
`true`	`function`	`returns`	`false`	`return`

Primitive Types

There are four primitive types in SETUP :

Integers

An integer is a sequence of digits. All integers are lexed as a sequence of digits with an optional leading minus sign for negative integers. They are represented

internally using architecture native integer representation.

Strings

A string is a sequence of characters enclosed in double quotes as in `"string"`. Two adjacent strings are concatenated using the "+" sign. As in

```
"string" + "concat" -> "stringconcat".
```

Floats

We adopt the *C Reference Manual* definition of a floating point number:

> A floating constant consists of an integer part, a decimal point, a fraction part, an `e` and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the `e` and the exponent (not both) may be missing.

All floating point numbers will be 64-bit double precision.

Booleans

A Boolean value can take either `true` or `false`.

3.2 Syntax Notation

In this manual, elements of language syntax are indicated by *italic* type. Literal words and characters are written in `verbatim`. Alternatives are listed using the "|" character: *item* | *item*.

3.3 Objects

3.3.1 Variables

A variable token is an identifier to a stored primitive, tuple or set. A variable must begin with an alphabetic character followed by zero or more letters and digits. Variables are declared by referencing their type followed by the associated token and an optional initializer, as in `int a;` or `int a = 3;`. All global variables must be declared outside of any function definition, and cannot include an initializer.

A variable, once declared, may be reassigned but cannot be declared again in the same scope.

```
int a = 3;
a=4;          /* ok    */
int a = 5;    /* error */
```

Variable Initialization

Variables for primitives not explicitly initialized when declared will be initialized as follows:

$$int \rightarrow 0$$
$$string \rightarrow \text{""}$$
$$float \rightarrow 0.0$$
$$bool \rightarrow false$$
$$set \rightarrow \{\}$$

Uninitialized tuples are not permitted.

3.3.2 Tuples

A *n-tuple* is an ordered collection of n comma-delimited *expressions* enclosed in parentheses. An *element* is either a primitive, set or tuple. An *n-tuple* and *m-tuple* are considered of the same type if the following two conditions are satisfied:

- $n = m$.

- The type of each coordinate *element* is of the same type.

For example, the following tuple elements are not the same type because the first coordinate *tuples* are not of the same *type*:

```
((1,"a"),2) //type: ((int,str),int)
((1,2),3) //type: ((int,int),int)
```

3.3.3 Sets

A set is an unordered collection (potentially empty) of terminals, sets or tuples. Every element of a set is unique (duplicate elements are discarded). A set must be **homogeneous in type**, meaning that every element within the set has matching type. All sets are typed as `set`, regardless of their contents. This means that a set containing sets of varying types is allowed.

```
{1,2,3,4} /* valid: homogeneous in type */
{1,2,3,"f"} /* invalid: not homogeneous in type */
{(1,2),(3.0,4.0)} /* invalid: tuple types are deep */
{{1,2,3},{"a","b"}} /* valid: set types are shallow */
```

Sets can be initialized in various ways:

Literal Initialization

A set may be initialized with a comma-delimited list of elements or identifiers of matching type within curly braces:

```
set A = {1, 2, 3, 4, 5, 6}; /* ok */
string b = "name";
set B = {"this", "works", b}; /* ok */
set C = {1, 2, 3, b}; /* error: type mismatch */
```

Range Initialization

For sets of integer type, we allow the following range initialization using " . . . ":

```
set A = {1 ... 6}; /* ok: returns {1, 2, 3, 4, 5, 6} */
set B = {1 ... 3, 5...7}; /* ok: returns {1, 2, 3, 5, 6, 7} */
set C = {3 ... -1}; /* ok: returns {3, 2, 1, 0, -1} */
```

Set-Builder Initialization

The following syntax is used for initializing a set through set-builder notation:

`{` *expr pipe sourcelist* `}`

The source list is a sequence of comma-delimited expressions of the form

id `in` *set*

Each *id* represents a local variable which may be used to construct new elements for a set via the expression appearing on the left hand side of the | symbol. The sequenced sourcelist expressions are evaluated left-to-right.

The `in` operator is right associative, and the *id* is not assigned a value until the right operand has been resolved. The resulting set will include the resulting left side expression evaluated for all potential *id* values, with duplicates removed.

```
set A = { (x,y) | x in {1,2,3}, y in {"a","b","c"} };
B = { (1,"a"),(1,"b"),(1,"c"),
      (2,"a"),(2,"b"),(2,"c"),
      (3,"a"),(3,"b"),(3,"c") };
A == B; /* returns yup */
```

In addition, the left side expression may contain references to variables in enclosing scopes (local sourcelist variables shadow enclosing scopes). For example:

```
int a = 0;
int x = 5;
set A = { (a,x) | x in {1,2,3} }; /*x shadows enclosing x = 5*/
B = { (0,1),(0,2),(0,3) };
A == B; /*returns yup*/
```

3.4 Operators

3.4.1 Arithmetic Operations

The following arithmetic operations are provided: `+,-,*` for types `int` and `float`. The return type is as follows:

$$\textit{int binop int} \rightarrow \textit{int}$$
$$\textit{float binop float} \rightarrow \textit{float}$$
$$\textit{float binop int} \rightarrow \textit{float}$$
$$\textit{int binop float} \rightarrow \textit{float}$$

There are two division operators:

1. `/`, which accepts as arguments any two numerical values and is guaranteed to return a `float`
2. `//` accepts as arguments any two numerical values and is guaranteed to return an `int` truncated toward zero.

The following relational operators are provided for all primitive types: `<,>,<=,>=,==,!=`. Types passed to these operators must match, except in the case of `<,>` when comparing `int` and `float`. String comparison is done lexicographically as per `strcmp` in the *C* standard library. The only character set supported is 8-bit ASCII.

3.4.2 Set Operations

The following set operations are provided:

```
union  intersect  minus  cross  #
```

union

`union` is a binary operator which returns a union of two sets. Both sets must contain elements of the same type.

intersect

`intersect` is a binary operator which returns the intersection of two sets. Both sets must contain elements of the same type.

minus

`minus` is a binary operator which returns a set of elements which are present in the first set but **not** in the second set. Both sets must contain elements of the same type.

cross

`cross` takes as left operand a set with elements of type a and as right operand a set of elements with type b and returns an exhaustive set of tuples of type (a, b), duplicates removed.

#

`#` is a unary operator returning the number of elements in a set.

3.5 Language Syntax

3.5.1 Structure of a Setup Program

Every SETUP progam is a sequence of zero or more function definitions and/or variable declarations. Variable declarations outside of function bodies are considered global in scope and may not be assigned a value when declared.

$$
\begin{aligned}
&\textit{program} \rightarrow \epsilon \\
&\textit{program} \rightarrow \textit{funcdef program} \\
&\textit{program} \rightarrow \textit{vdecl program}
\end{aligned}
$$

Execution (of non-empty programs) begins by calling the `main` function which returns an `int`. User-defined functions must be defined before the `main` function. Nested function definitions are not permitted.

3.5.2 Expressions

Terminals

expr → *float* | *int* | *string* | *bool* | *id*

Tuples

expr → *tuple*
tuple → `(` *expr-list* `)`
expr-list → *expr* | *expr*`,` *expr-list*

Sets

expr → *set*
set → `{` *expr-list* `}`
set → `{` *int* `...` *int* `}`
set → `{` *expr* | *source-list* `}`
source-list → *id* `in` *expr* | *id* `in` *expr, source-list*

Operations

expr → *id* `=` *expr*
expr → *expr binop expr*
expr → *unop expr*

3.5.3 Functions

Function definitions begin with the keyword `function` followed by a valid identifier (see 3.1.3), a possibly empty list of formals, a return type specified using the keyword `return` and a curly-brace enclosed list of statements.

funcdef → `function` *id* `[` *formals-list* `]` `returns` *typespec* `{` *stmt-list* `}`
formals-list → ϵ | *formals-tail*
formals-tail → *formal* | *formal formals-tail*
formal → *typespec id*
typespec → `int` | `set` | `float` | `string` | `tuple` | `bool`

Function definitions specify the expression to be returned using a return statement:

`return` *expr*`;`

Functions are called by specifying an *id* and a list of arguments enclosed in parentheses:

funcall → *id*`(` *arglist* `)`
arglist → ϵ | *exprlist*
exprlist → *expr* | *expr* `,` *exprlist*

3.5.4 Statements

A statement list is simply a sequence of one or more statements:

stmt-list → *stmt* | *stmt stmt-list*

with the following valid statements:

Variable Declaration and Assignment

stmt → *typespec id*`;`
stmt → *typespec id* `=` *expr*`;`

Control Statements

stmt → `if(` *expr* `) then {` *stmt-list* `} else {` *stmt-list* `}`
stmt → `while (` *expr* `) {` *stmt-list* `}`
stmt → `return (` *expr* `);`

Definitions

Print Statement

SETUP provides a method for nice printing to the standard output:

stmt → `Print (` *expr* `);`

Expressions

A statement may also be an expression:

$$stmt \rightarrow expr;$$

3.6 Scope Rules

There exists a global scope containing global variables and function definitions. Each function body and set-builder expression defines a local scope. Variable identifiers without type definitions are treated as references; local scope is first searched, followed by successive enclosing scopes. Variable identifiers with type definitions are treated as new declarations, and mask all variable identifiers in enclosing scopes with the same identifier.

Sourcelist variables exist in the left side of set-bulider notation:

{expr pipe sourcelist}

A simple example of set-builder scope:

```
set A1 = {1,2};
set A2 = {3,4};
set A = {A1,A2};

/* result: B = {(1,2),(2,1),(3,4),(4,3)} */
set B = {(x,y) | a in A, x in a, y in a-{x}};
```

A richer example, using nested scopes:

```
set A1 = {1,2};
set A2 = {3,4};
set A = {A1,A2};

/* result: B = {4,6,8,10} */
set B = {x*2 | a in A, x in {y+1 | y in a } };

          /*Pseudocode:
        foreach a in A
            foreach y in a
                x = y+1
                B.add(x*2)
            next y
        next a
           */
```

An example of variable shadowing:

```
int x = 2;

/* result: A = {4,16} */
set A = {x*x | x in {y | y in {x,x*x} } };
x; /* result: x=2 */

      /*Pseudocode
        x = 2
        temp t = {x, x*x}
        foreach y in t
            x=y;
            A.add(x*x)
        next y
       */
```

Chapter 4

Project Plan

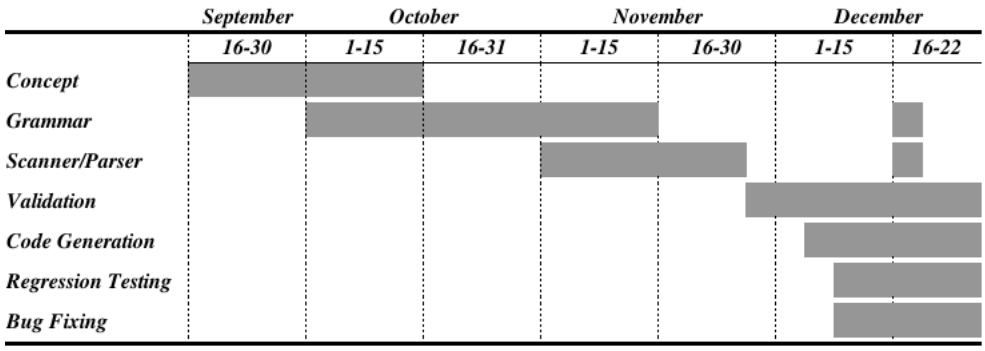
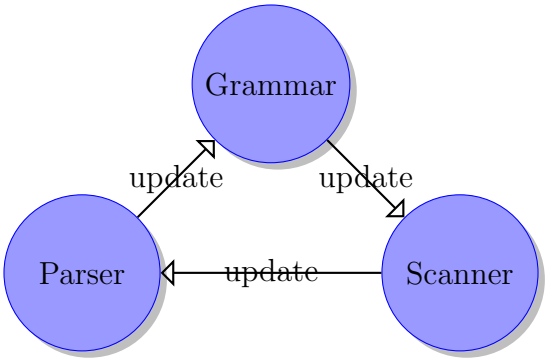


Figure 4.1: Timeline for Setup Team

September. Our team held weekly meetings to discuss different potential ideas for a language. By the end of the month, we had decided on a language that would manipulate sets. We then set to work developing a rough draft for a working grammar and began drafting our reference manual.

October. In conjunction with writing the first draft of our LRM, we began work on the scanner and parser. As we built the parser, we found several areas where our grammar needed to be "improved." The following graphic illustrates nicely how we spent October:



November. We completed work on our parser in November and began working on validation (static semantic analysis) to perform type checking. By the end of the month we were at the point where we could start generating code.

December. Writing code to generate valid C++ turned out to be more difficult than we anticipated, largely due to deep typing of nested tuples, and we continued to struggle with the Set-Builder construction mechanism. We implemented a set class abstraction in C++ to handle some of the peculiarities of arbitrary set and tuple containers. Throughout December, we generated code and tested for bugs.

Chapter 5

Compiler Architecture

The compiler consists of 5 principal components: Scanner, Parser, Validator, Code Generator and Set Class. The Scanner receives input in the form of plain text files with `.su` suffix and the compiler generates valid cpp code.

The Code Generator uses the concrete parse tree generated from the Parser to output valid C++ code. A polymorphic class hierarchy was implemented in C++ (Set Class component) to implement complicated Tuple and Set types.

The following lays out who worked on the various component implementations:

Component	**Team Members**
Scanner	Adam
Parser	Adam, Ian
Validation	Bill
Code Generator	Andrew, Bill, Ian
Set Class	Bill, Andrew
Regression Suite	Ian, Bill, Andrew

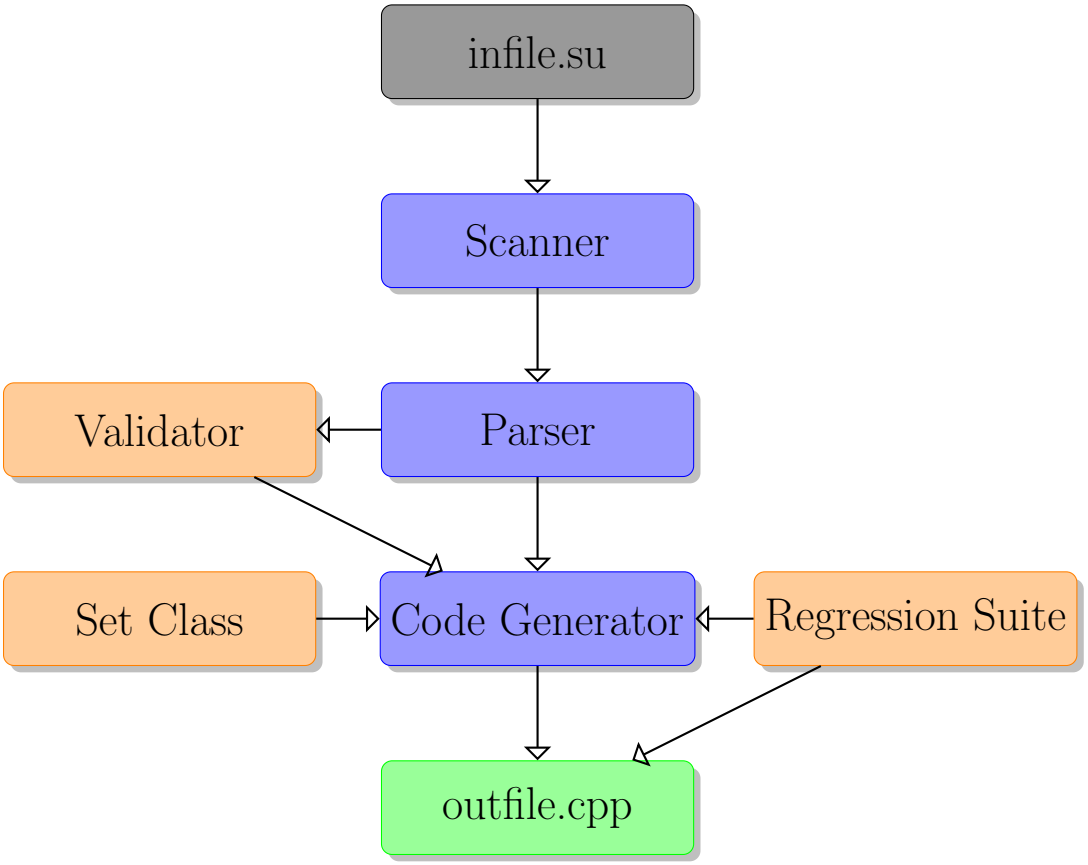


Figure 5.1: Overview of Setup Compiler

Chapter 6

Test Plan

Chapter 7

Lessons Learned

Chapter 8

Appendix