1# Expressive: A language for text register programs.

Santiago Cerón Uribe

A Thesis presented for the degree of
Systems and Computer Engineer

Advisor: Silvia Takahashi
Departamento de Ingeniería de Sistemas y Computación
Universidad de los Andes
Bogotá, Colombia

March 1, 2016

$1\#$ Expressive: A language for text register programs.

Santiago Cerón Uribe

Submitted for the degree of Systems and Computer Engineer
March 1, 2016

Abstract

Inspired in Lawrence Moss's $1\#$ language for text register programs [5], this work documents the theoretical background, design, implementation and usage of $1\#$ Expressive ($1\#X$), a language that provides a friendly syntax and compiles to $1\#$, facilitating the writing and understanding of $1\#$ programs and hence aiding in the study of theory of recursion and self replication through text register machines. In the process, we explain concepts about text register machines, domain specific languages, and their development using the Xtext framework.

Acknowledgements

Contents

List of Figures

Chapter 1

Preliminaries

In this chapter we introduce the project and its objectives, as well as give the basic theoretical notions concerning $1\#$ and $1\#X$, which include concepts on register machines and Domain Specific Languages (DSLs).

1.1 Introduction

In [5], Lawrence Moss proposed the minimalistic $1\#$ language to study theory of recursion using text register programs. Though the language meets its purpose, as it is seen in the paper, its syntax makes programs very cryptic, difficult to write and read. For this reason, we propose the creation of an expressive version of this language, which we will call $1\#X$ ($1\#$ Expressive). $1\#X$ is a language that compiles into $1\#$, but provides a much more readable syntax, which facilitates the process of writing and understanding text register programs by allowing the use of intuitive constructs and promoting consistent code reusability via function definitions, in the likes of high level programming languages. Alongside its design, a full implementation of an editor and compiler was done using Xtext. We end up with a toolset aimed to enhance and complement $1\#$'s labor in the study of text register programs in the context of recursion theory.

Complete instructions on how to get started with the developed $1\#X$ tools can be found in Appendix B

1.2 Outline

Chapter 1 introduces the project and basic theoretical notions concerning $1\#X$.
Chapter 2 introduces the $1\#$ language, in a format followed in Chapter 3 to introduce the $1\#X$ language. Chapter 3 also shows some program examples written in both $1\#$ and $1\#X$, for comparison. Further examples in other $1\#$-like languages can be found in Appendix A
Chapter 4 delves into design and implementation details of $1\#X$.
In Chapter 5 we show more examples of programs written in $1\#X$, this time related to theory of recursion.

1.3 Objectives

The objective is to design and implement a language that emulates $1\#$, doing so with a friendlier syntax. The result, $1\#X$, should be equivalent to $1\#$, and as we will show, programs can be converted from $1\#X$ to $1\#$ and viceversa. This with the following purposes:

1. To facilitate the writing and understanding of $1\#$ programs, aiding with its purpose of studying the theory of recursion.
2. To research the process of designing and implementing a DSL.
3. To gain expertise of DSL development using Xtext.

1.4 Text register machines

1.4.1 Register machines

In [5], Moss discusses many variants of the register machine model of computation that was initially proposed by Shepherdson and Sturgis in [8]. The concept of *unlimited*

register machines is introduced. The provided here is not the same as Moss's, but it is evidently equivalent.

Definition 1 (Unlimited Register Machines) Given a set of symbols (or alphabet) $\mathcal{A}$, a *register* acts a queue of symbols. An unlimited register machine, or *URM*, is a set of registers $\{r_1, r_2, ..., r_n\}$ with instructions:

- $W(i, a)$ which "writes", or enqueues a in r_i.
- $D(i)$ which "deletes" from r_i, popping the head of r_i, provided that it is not empty.
- $M(i, a, k)$ which "moves", polling r_i. If a is in the head of r_i, then the program moves k lines. Otherwise, it moves to the next instruction.

Notice that although only a finite set of registers is allowed, there is no bound on its size.

1.4.2 Computable partial functions: Executing register machines

We now formalize the "execution" of register machines. Though Moss does it specifically for $1\#$ in [5], his definition can be easily generalized to programs that operate over registers.

Definition 2 (Φ_P) Let P be a program that operates over registers (for example, a URM). For every $n \in \mathbb{N}$ we define a function $\Phi_P : (\mathcal{A}^*)^n \mapsto \mathcal{A}^*$ such that

$$\Phi_P(x_1, ..., x_n) = y$$

if the instructions of P are executed (or P is run) starting with x_i in r_i for $i = 1, ..., n$ and the other registers empty, then eventually the machine comes to a halt (this concept will be defined formally in terms of $1\#$, but for now the intuitive meaning of a program halting should suffice), with y in r_1 and the rest of the registers empty. If P doesn't halt, then Φ_P is not defined for $\langle x_1, ..., x_n \rangle$.

Equivalence of programs

Definition 3 (Program equivalence)

Given two programs P and Q that operate over registers over some alphabet $\mathcal{A}$, we say that P and Q are equivalent, and note it $P \simeq Q$, if for every $n \in \mathbb{N}$ and every tuple $\langle x_1, ..., x_n \rangle$ either:

- $\Phi_P(x_1, ..., x_n) = \Phi_Q(x_1, ..., x_n)$.
- Both Φ_P and Φ_Q are not defined for $\langle x_1, ..., x_n \rangle$.

1.5 Domain specific languages

As the name indicates it, domain specific languages are programming languages intended to work in a specific context, and are not meant to solve general problems. SQL and Ant are both examples of well known DSLs. Since DSLs are designed with specific purposes, problems in the context of a DSL will be more easily solved using the DSL instead of a general purpose language [3]. Indeed, $1\#$ (and hence $1\#X$) are DSLs. They are not meant to solve every programming problem, but instead were created with the specific purpose of studying theory of recursion through the Text Register Machine model of computation.

In [3], Lorenzo Bettini gives a good outline of the main concepts of designing and implementing a DSL. What's presented here is just a very brief summary of explanations by Voelter in *DSL engineering: Designing, implementing and using domain-specific languages* [11] (for design), and Lam et. al in *Compilers: Principles, Techniques and Tools* [4].

1.5.1 Designing the DSL

The first step is, of course, designing the language. For this, the domain should be clearly delimited, as well as the abstractions from that domain that the language will cover. Aspects that should be taken into account are:

1. **Expressivity**: Typically, a DSL should be more expressive than a general purpose language. A highly expressive language will result in shorter, easier to read code. However, higher expressivity makes compilation and learning the language more difficult.

2. **Coverage**: Refers to how well the DSL covers its domain. It could be thought of as a ratio of the number of programs in the domain expressable by the language, over the number of programs in the domain.

3. **Semantics**: Denote the behavior of a program, both before execution (in terms of constraints) and during execution.

4. **Concrete syntax**: Determines what the actual language is and the notations used to express programs. The syntax should promote the *Writability*, *Readability*, *Learnability* and *Effectiveness* of the language.

Grammar One way to express the concrete syntax of a language, as well as to have a general idea of the general structure for programs, is by defining a grammar in some standard notation (for example BNF). This is the first step that was taken to implement $1\#X$, as will be seen in later sections.

1.5.2 Lexical Analysis

When compiling a program, the first step is to break the text down into tokens, which are single atomic elements of the language. This procedure of grouping characters into tokens is also known as *lexical analysis* and it is done by the *lexer*. Blanks and other characters defined as white spaces are discarded by the lexer. Take for example the following statement:

$$x = 1 + y$$

The lexer would identify the tokens:

1. x as a token of type ⟨**id**.⟩

2. = as a token of type ⟨=⟩ (The character is the same token type.)

3. 1 as a token of type ⟨**int**⟩.

4. $+$ as a token of type ⟨$+$⟩.

5. y as a token of type ⟨**id**⟩.

1.5.3 Syntactic Analysis

Now that we have our text "tokenized", we now need to check that they follow the syntactic structure expected by the language, making them a valid statement in the program. In our example, this would follow the structure of an assignment on our language. The syntactic analyzer, or *Parser*, builds a tree with a structure that follows the one indicated in the grammar of the language. [9]. In our example, this would look like:

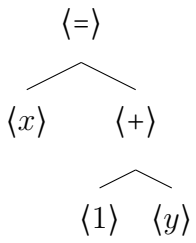


Figure 1.1: An example of a syntactic tree.

1.5.4 Semantic Analysis

Takes the information from the syntax tree and checks for semantic consistency. For example, in a language with types, it does *type checking*, where the compiler checks that each operator has matching operands. After this, it "translates" the program to another language to be executed. It may do so in more than one step, using one or more intermediate languages to do this translation.

Chapter 2

The $1\#$ Language

This Chapter introduces the $1\#$ language, enunciates its relation with Unlimited Text Register Machines, and describes extensions that have been done to $1\#$ by other authors.

2.1 The $1\#$ instruction set and syntax

$1\#$ is a minimalistic language that can be used to describe URMs over the alphabet $\mathcal{A} = \{1, \#\}$. It has four basic instructions: *Write, Forward, Back* and *Cases*. What's special about $1\#$ is that the descriptions of the programs use the same alphabet as the inputs/outputs, so all the instructions of $1\#$ are expressed in terms of 1s and #s. This language is presented by Moss in the main reference [5].

2.1.1 The Write instruction

The *Write* command adds a $1\#$ symbol to a specified register. The command:

$$1^n\#$$

Adds a 1 to register n, which we refer to as Rn, (so $11\#$ adds a 1 to $R2$). In a similar way,

$$1^n\#\#$$

Adds a $\#$ to Rn. After a *Write* instruction, the ext instruction in the program is executed.

2.1.2 Control transfer: The Forward and Back instructions

Transfers control to some other instruction. The command

$$1^n\#\#\#$$

Tells the program to jump (or transfer) *Forward* to the nth instruction after the current instruction. Similarly, the command

$$1^n\#\#\#\#$$

Tells the program to transfer *Back* to the nth instruction after the current one. For example, the program

```
1#       ;Add 1 to 1
111###   ;Forward 3
1##      ;Add # to 1
11###    ;Forward 2
11####   ;Back 2
```

./Examples/SimpleOsExample.os

finishes its execution (or as we will call it, *halts*), writing $1\#$ to the first register as a result.

Loops The transfer instructions allow us to define loops, so a program may never halt. Consider the simple example of going back and forward indefinitely.

```
1###   ;Forward 1
1####  ;Back 1
```

./Examples/SillyLoop.os

Halting and stopping improperly A program may or may not stop (as seen in the above example). Even if it stops, a program may not do it properly.

Definition 4 (Halting) We say a program stops properly, or ***halts***, if at some point in its execution, control is transfered to the point JUST AFTER the last instruction of the program.

Definition 5 (Stopping improperly) If at some point, control is transfered before the first instruction, of to any other point after the last one, then the program ***stopped improperly***.

For example,

$$1\#\#\# \text{ (Forward 1)}$$

halts, while both

$$11\#\#\# \text{ (Forward 2)}$$

and

$$1\#\#\#\# \text{ (Back 1)}$$

stop improperly.

2.1.3 The Cases instruction

Finally, we have an instruction that reads from registers. This is the *Cases* instruction. It reads and removes the first character from a specified register, and does something different depending on what it reads, either the register is empty, or it reads 1 or a $\#$. In $1\#$, a *Cases* instruction is made up of four instructions. The first one specifies the register that should be read, with syntax

$$1^n\#\#\#\#\#$$

if an empty register is read, then the program skips 1 instruction. If a 1 is read, the program skips 2 instructions, and finally if a $\#$ is read, 3 instructions are skipped.

$$1^n\#\#\#\#\#$$

Case empty

Case 1

Case $\#$

The usage of this instruction, in conjunction with the rest of the set will be clearer in further examples.

2.1.4 Comments

Though not officially part of the language, $1\#$ compilers ignore all characters that do not belong to $\mathcal{A}$, and ignore everything in a line written after a semicolon (so we can use them as line comments).

2.2 $1\#$ programs as URMs: Equivalence

From now on, unless otherwise stated, $\mathcal{A}$ refers to the alphabet $\{1, \#\}$.
We expand here on the brief (but very clear) connection that [5] explains between $1\#$ and URMs. We prove that every $1\#$ program describes a URM, and that every URM over $\mathcal{A} = \{1, \#\}$ has an equivalent $1\#$ program. More formally talking about equivalence, we want to see that:

1. For every $1\#$ program P, there exists a URM Q over $\mathcal{A}$ such that for every $n \in \mathbb{N}$ and tuple $\langle x_1, ..., x_n \rangle \in (\mathcal{A}^*)^n$,

$$\Phi_P(x_1, ..., x_n) = \Phi_Q(x_1, ..., x_n)$$

2. For every URM Q over $\mathcal{A}$, there exists a $1\#$ program P such that for every $n \in \mathbb{N}$ and tuple $\langle x_1, ..., x_n \rangle \in (\mathcal{A}^*)^n$,

$$\Phi_P(x_1, ..., x_n) = \Phi_Q(x_1, ..., x_n)$$

Roughly speaking, we define two programs to be equivalent if, when executed with the same inputs, give the same outputs.

In [8], a variant to the instruction set of URMs is introduced. Given a URM over an alphabet $\mathcal{B} = \{a_1, ..., a_m\}$, consider the Scan and Delete instruction:

> $Scd(n)[k_1, ..., k_m]$: Scan register r_n. If r_n is empty, jump to the next instruction. If the first symbol of r_n is a_i, then jump k_i instructions.

Shepherdson and Sturgis prove in Appendix C that the original Move and Delete functions can be replaced by the Scan and Delete instruction, (so the Move and Delete instructions can be emulated with the Scan and Delete, and vise versa). With that result, the following two propositions now become apparent.

Proposition 2.2.1 ($1\#$ programs are URMs) Every $1\#$ program has an equivalent URM.

Proof. First, notice that every $1\#$ program uses a finite number of registers, since there can only be finite instructions, and each instruction uses at most one register. Now, we need to translate $1\#$ instructions to URM instructions. Let P be a $1\#$ program.

1. **Write**: A $1^n\#$ instruction is translated to $W(n,1)$, while $1^n\#\#$ is translated to $W(n,\#)$.

2. **Moves**: We need to transfer control without affecting the state of the registers. Since only finite registers are used by a $1\#$ program, there exists a register n that remains empty during the execution of P. We use this register to simulate the move. A $1^k\#\#\#$ instruction (Forward k) becomes:

$$W(n,1)$$
$$Scd(n)[k,k]$$

   We use the auxiliary register n to write a 1 that immediately after gets scanned and deleted, making the program move the required number of instructions. Since register n is not used during the execution of P, we can assume that the register is empty before calling the instruction. Also note that right after the instruction, Rn is emptied out again.
   Similarly, a $1^k\#\#\#\#$ instruction (Back k) becomes:

$$W(n,1)$$
$$Scd(n)[-k-1,-k-1]$$

   Notice that in this case we have to add a -1 to take into account the W instruction that was placed.

3. **Cases** With the Scd instruction, a $1^k\#\#\#\#\#$ (Cases on Rk) instruction is easily translated into:

$$Scd(k)[2,3]$$

□

Proposition 2.2.2 Every URM over the alphabet $\mathcal{A}$ has an equivalent $1\#$ program.

Proof. It suffices to show that every instruction of a URM has an equivalent program in $1\#$. Using the Scan and Delete description of URMs, this is very easy to do:

1. **Write**: This one is very straightforward. A $W(n,1)$ is translated to $1^n\#$. Similarly, $W(n,\#) \simeq 1^n\#\#$.

2. **Scan and delete**: Consider the instruction $Scd(n)[k_1,k_2]$. Suppose that $k_1, k_2 > 0$. The negative case is resolved similarly. We want an instruction that pops from Rn, moving forward 1 instruction if Rn is empty, k_1 instructions if a 1 is popped and k_2 instructions if a $\#$ is popped. Doing

$$1^n\#\#\#\#\# \text{ (Cases on } n\text{)}$$
$$111\#\#\# \text{ (Forward } 3\text{)}$$
$$1^{k_1+1}\#\#\# \text{ (Forward } k_1+1\text{)}$$
$$1^{k_2}\#\#\# \text{ (Forward } k_2\text{)}$$

simulates it. If Rn is empty, then the cases instruction directs to a *Forward 3* instruction that transfers control to the next instruction after the cases block. If 1 is popped, then the *Forward* k_1+1 instruction is called, which skips the $\#$ case (with the $+1$), and then moves k_1 instructions after the cases block. If $\#$ is popped, then the *Forward* k_2 instruction is called, moving k_2 instructions after the cases block. Dealing with negative moves is done similarly using the *Back* instruction and taking into account the extra lines used in the Cases block.

□

2.3 Previous Work

Some work has already been done to extend, or somehow facilitate the process of writing $1\#$ programs. Though there are many enthusiasts who implement their own $1\#$ compiler, there are two tools that actually modify/extend the language, both published in [1]:

2.3.1 The $1\#$ *Sanity* parser

This parser allows writing in shorthand language, which is then parsed to normal $1\#$ code. So, instead of writing $1\#\#$ to add a $\#$ to $R1$, we can write *push*$\#1$. Each $1\#$ instruction has an equivalent one in the *Sanity* parser. The *Sanity* parser also allows labeling lines, which then can be accessed via a *goto* statement. We will use this idea in $1\#X$. Though it provides some easier usage of $1\#$, the Sanity parser is just a bit more than a textual representation of $1\#$ instructions, and programs written in it still have the problem of jumping forward and back when doing cases statements, which is responsible (in a good part) of the unreadability of $1\#$ programs.

2.3.2 The $1\#$ flowchart: *Asylum*

Asylum is a flowchart editor for $1\#$ programs. It has Cases statements and Add instructions. Control in the programs is transferred by creating links between nodes containing these statements. With the flowchart editor, creating programs is very intuitive, and its graphic nature makes them very readable. However, since it is a graphic editor, writing long programs in the flowchart is impractical. It allows importing $1\#$ code to be included within a flowchart, however, it does not create the flowchart visualization of the code (which would be a really nice feature). Of course, it can also create $1\#$ code from a given flowchart.

Chapter 3

The $1\#X$ language

This Chapter introduces the $1\#X$ instruction set, and gives some examples of programs written in $1\#X$, comparing them with their $1\#$ counterparts.

3.1 The $1\#X$ instruction set and syntax

Since $1\#X$ is intended to be an expressive version of $1\#$, the instruction set is very similar. Recall that we are treating registers like character queues, so adding is done to the right (end) and removing is done from the left (beginning). For a better understanding on the structure of a $1\#X$ program, refer to its grammar in section 4.3

3.1.1 $1\#X$ programs

$1\#X$ programs have the following structure:

Function declarations

Main program body

First, all functions must be declared. Then comes the program body as a sequence of the commands explained in this section. Note that in this part, no unknown values are allowed, so all function calls must be made with actual values. (No references to "variable" parameters are allowed.)

3.1.2 Write

Just like in $1\#$, there is an instruction that writes, or adds characters, to a specified register. In $1\#X$, more than one character can be written. The instruction

add n *word*

adds *word* to register n, where *word* is a string of 1s and $\#$s.

3.1.3 Control transfer: Labels and gotos

To transfer control from one instruction to another, labels are used. Every $1\#X$ command can be labeled:

label: *instruction*

The **goto** instruction transfers control to a labeled instruction. The loop in section 2.1.2 can be written in $1\#X$ in the following way:

```
1 one: goto two
2 two: goto one
```

./Examples/SillyLoop.osx

The exit instruction $1\#$ programs can *stop improperly*. Notice that if (so far) the only way to transfer control on a $1\#X$ program is by going to an existing label, this should never happen. For this reason, an extra instruction was added,

exit

which makes the program stop improperly immediately, no matter where it is called.

Label scopes A label scope is a set of commands such that if one of them has label L, another command in the scope can transfer control to that command via a **goto** a instruction. For $1\#X$, the main program body forms one label scope. The body of each function declaration is a label scope. The next section explains the *Cases*

instructions. The reader will notice here a difference between the scoping in $1\#X$ and in languages like Java. In Java, variables can be declared to be used within a loop, for example. In $1\#X$, labels are more global, meaning that labels declared inside a case in a *Cases* statement can be accessed from outside the statement, and vice versa. This follows more closely the nature of $1\#$, where using control transfer instructions, one can go from *any* part of the program to *any* other.

3.1.4 Cases

Recall from section 2.1.3 that $1\#$ cases begin by specifying a register, and then, in order, what happens if empty, 1 or $\#$ are read. Cases in $1\#X$ work in a similar fashion:

```
1 //Cases on regNumber
2 pop regNumber
3   case
4   //Instructions for empty case
5   case
6   //Instructions for Case 1
7   case
8   //Instructions for Case #
9 end
```

./Examples/CaseUsage.osx

In each case, there can be any number of instructions that will be executed sequentially. After these instructions are executed, control is transfered to the *Rest of the program* part (just as in a Java/C++ **if** statement). This makes Cases programming significantly easier in $1\#X$ than it is in $1\#$, as we will see in the Examples section 3.2.

3.1.5 Functions

Purely as syntactic sugar, functions can be defined, an later called, in a $1\#X$ program. They can receive parameters as well.

The $1\#X$ types There are two types in $1\#X$.

- **int** refers to positive integers.
- **string** refers to non-empty words made of 1s and #s.

Functions are declared, and then can be called from the main program body, or from other functions, by assigning values to each of their parameters, just like in high level languages.

```
/**
* This is a function declaration in OSX
*/
FunctionDeclaration(int intParameter, string stringParameter)
begin
  //Some instructions using the parameters would be added here.
end

//We then call the function from the program's main body.
FunctionDeclaration(2,'1##1#')
/*Parameters used are intParameter = 2,
*     stringParameter = '1##1#'*/
```

./Examples/FunctionUsage.osx

Their usage will become clearer in the Examples section 3.2.

Observation 1 (Recursion) For the moment, no recursive calls are allowed between functions. For further discussion on this topic, please refer to Observation 3 of this document.

Observation 2 (Parameters) Even though parameters are allowed, all values for parameters in function calls should be known at **compile time**, so they are not to be confused with variables.

3.2 Examples in $1\#X$ and $1\#$

Before moving on with the topic, we present some examples of $1\#$ and $1\#X$ programs, based on the examples presented on Moss's course on Theory of Recursion using

$1\#$ [6]

3.2.1 The *Move* program

Move$_{2,1}$

We start with a program that moves the contents from $R2$ to $R1$. The idea is simple: Read from $R2$ until empty. Write the character read to $R1$. Of course, there are many ways to implement this. One simple way of doing it is presented.

Example 1 (*Move*$_{2,1}$ in $1\#$)

```
11#####         ;Cases on R2
111111###       ;Case empty (Move Forward 6 to end)
111###          ;Case 1 (Move Forward 3 to
                ;     Case 1 implementation)
1##             ;Case # (Write # to 1)
1111####        ;Back 4 (To Cases statement for loop)
1##             ;Case 1 implementation (Write 1 to 1)
111111####      ;Back 6 (To Cases statement for loop)
```

./Examples/Move21.os

Example 2 (*Move*$_{2,1}$ in $1\#X$)

```
loop: pop 2
  case
    //If R2 is empty, do nothing.
  case
    //If a 1 is read, add 1 to R1 and loop.
    add 1 '1'
    goto loop
  case
    //If a # is read, add # to R1 and loop.
    add 1 '#'
    goto loop
end
```

./Examples/Move21.osx

Though a little longer in terms of lines (7 in $1\#$ vs 9 in $1\#X$), we see that programming in $1\#X$ is a bit easier, since we don't have to be jumping around the program (we do, but the syntax hides it from us) to do the cases statement.

Move$_{s,d}$

If we need now to Move the contents of some source register Rs to a destination Rd, for example, from $R1$ to $R2$, in $1\#$ we have no option but to write another program from scratch. Though it will not be much different from *Move*$_{2,1}$, it will differ in the number of ones on some of the instructions, but the most code recycling we can do is copy pasting and changing appropriately. Consider the $1\#$ code for a program that first does *Move*$_{2,1}$ and then *Move*$_{1,2}$. It would take a while analyzing the code before figuring out what it does. In $1\#X$, however, we can define a *Move* function with parameters s and d. By reading the code it will now be very clear what the program will do.

Example 3 (*Move*$_{s,d}$ in $1\#X$)

```
1  /**
2   * Empties the contents of source, copying them to the end of
      dest.
3   */
4  Move(int source, int dest)begin
5    loop: pop source
6      case
7      case
8        add dest '1'
9        goto loop
10     case
11       add dest '#'
12       goto loop
13     end
14 end
15
16 Move(2,1)
17 Move(1,2)
```

./Examples/MoveFn.osx

The function is first defined, and then it can be used in the main program body (which comes after all the function declarations)

3.2.2 The *Copy* program

We now want a program that *copies* the contents of some register to another register, but now, at the end of the execution, the contents of the source register should be the same as when it started. The implementation is simple, using a Cases statement like in the *Move* function, but instead of writing characters only to the destination register, we also add them to a temporary register, which we assume starts being empty, so that later we can move the contents from the temporary register to the original one. We note this function as $Copy_{s,d,t}$, which means that we will copy the content of Rs to Rd using Rt as the temporary register.

Example 4 ($Copy_{1,2,3}$ in $1\#$)

```
1#####       ;Cases on R1
11111111###  ;Case Empty, we are done reading R1 and now need
             ; to move contents back from R3.
1111###      ;Case 1, go Forward 3 to Case 1 implementation
11##         ;Case #, add # to R2 and R3
111##        ;
11111####    ;Back 5 (To Cases statement for loop)
11111####    ;Back 5 (To Cases statement for loop)
11#          ;Case 1 Implementation, add 1 to R2 and R3
111#         ;
11111111####;Back 8 (To Cases statement for loop)
111#####     ;Cases on R3, Move(3,1)
111111###    ;Case Empty, we are done, Forward 6 to end
111###       ;Case 1, go Forward 3 to Case 1 implementation6
1##          ;Case #, add # to R1
1111####     ;Back 4 (To Cases statement for loop)
1#           ;Case 1 Implementation, add 1 to R1
```

```
11111####    ;Back 6 (To Cases statement for loop)
```

./Examples/Copy123.os

In $1\#X$, we can again define a function that takes s, d, t as parameters.

Example 5 (*Copy*$_{s,d,t}$ in $1\#X$)

```
/**
 * Copies the content of source at the end of dest.
 * Pre: Temp is empty, as is used as a temporal register.
 */
Copy(int source, int dest, int temp)begin
  loop: pop source
    case
      //Case empty, finished reading RSource. Now we need to
   Move.
    case
      add dest "1"
      add temp "1"
      goto loop
    case
      add dest "#"
      add temp "#"
      goto loop
    end

  Move(temp,source)
end

Copy(1,2,3)
```

./Examples/CopyFn.osx

It's worth noting that the previous example assumes that the *Move* function has already been declared earlier in the program, it is not shown in the code for space reasons. With this example, we see that functions can be called from other functions using parameters.

3.3 Translation from $1\#$ to $1\#X$

A requirement for $1\#X$ is to be equivalent to $1\#$. We prove now one (easy) half of this equivalence:

Proposition 3.3.1 ($1\#$ is as powerful as $1\#X$) For every $1\#$ program P, there exists a $1\#X$ program Q such that $P \simeq Q$.

Proof. We just need to translate each $1\#$ instruction to a $1\#X$ equivalent. But this is very straightforward:

1. **Write**: The translation of Write commands from $1\#$ to $1\#X$ is direct.
   $1^n\# \mapsto$**add** n '1'.

2. **Control transfer:** The *Forward* and *Back* instructions are probably the most difficult to translate. The easiest way to go about this is to label every instruction in the $1\#X$ translation with its line number. Leave a placeholder translation for $1\#$ control transfers, each one will take a single line. Having every instruction labeled with its line number, control transfers are now simple *Goto*s. A control transfer to somewhere outside the program is translated to an *exit* instruction.

3. **Cases**: Translation of Cases commands is also direct.

```
1 1#####
2 ;Case empty
3 ;Case 1
4 ;Case #
```

   Is translated to:

```
1 pop 1
2 case ;Case empty translation
3 case ;Case 1 translation
4 case ;Case # translation
5 end
```

□

Chapter 4

Implementing $1\#X$

4.1 General Architecture

We present below a general architecture for the $1\#X$ development tools implementation. Each of the components will be explained in detail throughout this chapter.

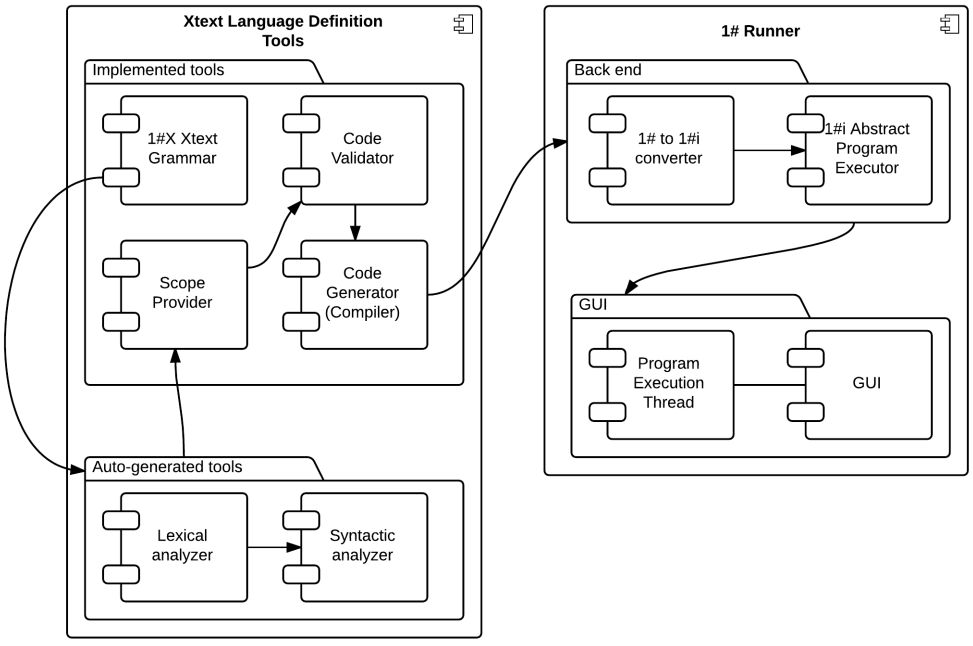


Figure 4.1: $1\#X$ component diagram

4.2 Xtext

Xtext is an open source framework for developing programming languages [2]. It provides a handful of powerful features such as:

1. A *lexer* and *parser* for a language, given its grammar.
2. A text editor for our language with syntax highlighting, bracket matching, code formatting, code validation, amongst many other features.
3. Tools to implement and modify the functionalities mentioned above.

Just by providing a *lexer* and *parser*, Xtext makes DSL development a lot less tedious (compared to implementing them manually). Xtext provides many mechanisms to configure different aspects of the language. These mechanisms will be mentioned in more detail in later sections, when the implementation of $1\#X$ is explained. From code editing to compilation is done using tools provided by Xtext.
It is worth noting that the decision of using Xtext derives from the experience of Alejandro Sotelo and his (and my) advisor Silvia Takahashi in the development of *GOLD*, a DSL for manipulating graphs and other data structures [9].

4.3 $1\#X$ grammar, lexical and syntactic analysis

As a first step, we define a grammar that provides the general structure of every $1\#X$ program. This is one of the most important components of our DSL, since it allows us to identify if a sequence of text characters is indeed a $1\#X$ program. More so, Xtext provides a lexer and parser given the $1\#X$ grammar.

4.3.1 EBNF grammar

First we write the grammar in EBNF notation. The full description of the notation can be found in the original article [7], while a good summary is provided in Appendix A of [9]. However, the way it is used to describe context-free grammars should be straightforward.

```
1  //***********************
2  // Terminal Symbols
3  //***********************
4  LETTER ::= ('a'..'z')|('A'..'Z')
5  DIGIT  ::= ('0'..'9')
6  NUMBER ::= ('1'..'9') DIGIT*
7  OS     ::= (1|#)+
8  ID     ::= (LETTER|DIGIT|OS|'_')+
9  WS     ::= // Whitespaces
10        (' '|'\t'|'\r'|'\n')+
11 Âă
12 //***********************
13 // Primitive Functions
14 //***********************
15 Write   ::= 'add' (NUMBER|ID) (OS|ID)
16 Case    ::= 'pop' (NUMBER|ID) 'case' Command* 'case' Command* '
      case' Command* 'end'
17 Goto    ::= 'goto' ID
18 Call    ::=  ID '(' ((NUMBER|OS|ID) (',' (NUMBER|OS|ID))*)? ')'
19 Exit    ::= 'exit'
20 Âă
21 //***********************
22 // Commands
23 //***********************
24 Command ::= (ID ':')? (Add|Case|Goto|Call|Exit)
25 Âă
26 //***********************
27 // Functions
28 //***********************
29 FuncHeader ::= ID '(' (('int'|'string') ID) ? (',' (('int'|'
      string') ID))* ')'
30 FuncDec    ::= FuncHeader 'begin' Command* 'end'
31 Âă
32 //***********************
33 // 1# Program
34 //***********************
35 OSProgram ::= FuncDec* Command*
```

osx_grammar.ebnf

4.3.2 XText grammar

The EBNF grammar is then implemented in XText. It is modified using XText mechanisms that make code validation and generation easier. The grammar, as it is, could use factoring to improve performance in the lexical and syntactic analysis. However, our way of presenting it increases its readability, while in practice noticing that the performance is not affected. Full documentation on implementing grammars in XText can be found in [2]. One of the most significant features is the ability to declare references in the grammar.

References Notice the definition for the *Goto* command:

'goto' label=[Command]

The label is not a *Command* per se, it should be an *ID*, as the EBNF grammar suggests. The square brackets represent that the label should be a **reference** to a *Command*. Production Rules are referenced via their *name* attribute. Indeed, looking at the definition of a *Command*, we see that its *name* is an *ID*. So, in order to be considered a valid statement, the *Goto* statement needs to follow the production rule

'goto' ID

and there exists a *Command* with such *ID* as its name. As an example, in Java, a program will show errors if there is an attempt to call a method that does not exist. Method calls expect a **reference** to an existing method.

```
1 grammar ceronsantiago.OneSharpExpressive hidden (WS,ML_COMMENT,
    SL_COMMENT)
2
3 import "http://www.eclipse.org/emf/2002/Ecore" as ecore
4
5 generate oneSharpExpressive "http://www.ceronsantiago.org/
    OneSharpExpressive"
```

```
6
7  //TODO: Factorize
8  //-------------------------------
9  // START SYMBOL - 1#X Program
10 //-------------------------------
11 Program:
12     functions+=FunctionDeclaration*
13     body=CommandBlock;
14 //-------------------------------
15 // TERMINAL FRAGMENTS
16 //-------------------------------
17 terminal fragment LETTER: //Latin alphabet
18     'A'..'Z'|'a'..'z'|'_';
19 terminal fragment DIGIT:
20     '0'..'9'
21 ;
22 //-------------------------------
23 // TERMINAL RULES
24 //-------------------------------
25 terminal INT returns ecore::EInt:
26     ('1'..'9') DIGIT*
27 ;
28 terminal ID: //A valid ID for a function, label, parameter, etc.
29     LETTER+ (LETTER|DIGIT|'#')*
30 ;
31 terminal OS:
32     ('\"' ('1'|'#')+ '\"') | ('\'' ('1'|'#')+ '\'')
33 ;
34 terminal WS:
35     (' '|'\t'|'\r'|'\n')+
36 ;
37
38 terminal ML_COMMENT : '/*' -> '*/';
39 terminal SL_COMMENT : ('//'|';') !('\n'|'\r')* ('\r'? '\n')?;
40
41 //-------------------------------
42 //Variables
```

```
43 //--------------------------------
44
45 /*
46  * A parameter for a register number. Can be either a number or
      a reference to an IntParam.
47  */
48 RegisterParam:
49     ((param=[IntParamDec])|value=INT)
50 ;
51
52 /*
53  * A parameter for a 1# word. Can be either a word (OS) or a
      reference to a StringParam.
54  */
55 WordParam:
56     ((param=[StringParamDec])|value=OS)
57 ;
58
59 /*
60  * A value that can be used as a parameter when calling a
      function.
61  * Can be either an int, a string, or a reference to another
      parameter.
62  */
63 CallParam:
64     ((intVal=INT|stringVal=OS|param=[ParamDec]))
65 ;
66
67
68 //--------------------------------
69 // PRIMITIVE FUNCTIONS
70 //--------------------------------
71
72 /* Adds a word to a register */
73 Write:
74     'add' register=RegisterParam word=WordParam
75 ;
```

```
76 /* The cases statement. */
77 NormalCase:
78     'pop' register=RegisterParam 'case' caseEmpty=CommandBlock '
       case' caseOne=CommandBlock 'case' caseSharp=CommandBlock '
       end'
79 ;
80
81 /* The goto statement. */
82 Goto:
83     'goto' label=[Command]
84 ;
85
86 /* Calling a function that was declared previously.
87  * References to a FunctionDeclaration.
88  */
89 Call:
90     function=[FunctionDeclaration] '(' (params+=CallParam (','
       params+=CallParam)*)? ')'
91 ;
92
93 /* Stop improperly.
94  * Curly brackets to force Exit object creation for easier
       compiling.
95  */
96 Exit:
97     {Exit} 'exit'
98 ;
99 //---------------------------------
100 // Commands
101 //---------------------------------
102 /*
103 *   One of the basic commands of 1#X, maybe accompanied by a
       label, which, if present
104 *   will be the way to reference the command.
105 */
106 Command:
107     (name=ID ':')? command=UnlabeledCommand
```

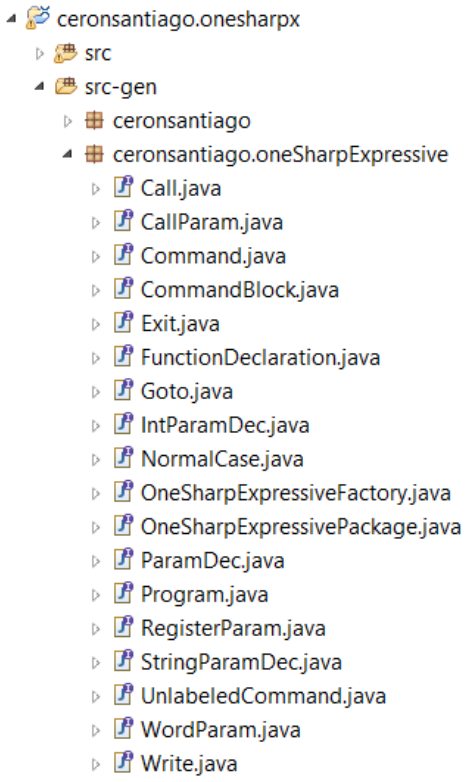
```
108 ;
109
110 /*
111 *  One of the basic commands of 1#X
112 */
113 UnlabeledCommand:
114     (Write|NormalCase|Goto|Call|Exit)
115 ;
116
117 /*
118  * A list of commands.
119  */
120 CommandBlock:
121     {CommandBlock} (commands+=Command)*
122 ;
123
124 //---------------------------------
125 // Functions
126 //---------------------------------
127
128 /* Declaration of an int parameter. */
129 IntParamDec:
130     'int' name=ID
131 ;
132 /* Declaration of a string parameter. */
133 StringParamDec:
134     'string' name=ID
135 ;
136 /* Declaration of a parameter. */
137 ParamDec:
138      IntParamDec|StringParamDec
139 ;
140 /* Declaration of a function.
141  * It is referenced by its name.
142  */
143 FunctionDeclaration:
144     name=ID '(' (params+=ParamDec (',' params+=ParamDec)*)? ')'
```

```
    'begin' body=CommandBlock 'end'
;
```

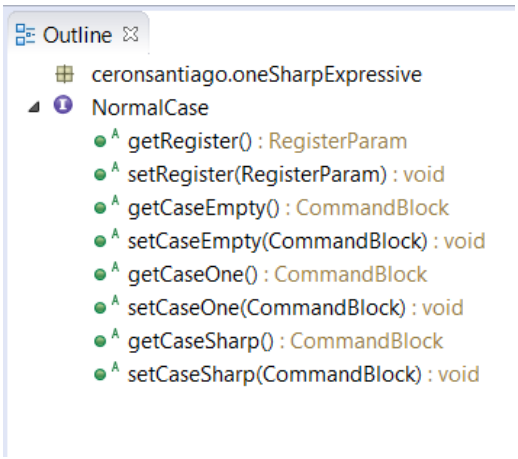
OneSharpExpressive.xtext

4.3.3 The XText generated model

Notice that, in the XText grammar, we also named the different components of the Production Rules. For example, a *Goto* statement is made up of the 'goto' word, and a *label*. When the grammar has been declared, classes are automatically generated for each production rule, generating the model of the language. Each class has the components of the production rule as attributes. For example the *Cases* statement has a *CommandBlock* for each of the possible cases. This model allows us to traverse the syntax tree with ease.



(a) Classes generated.

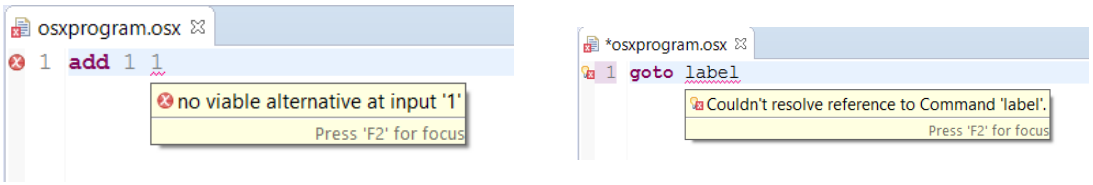


(b) The *NormalCase* class outline.

Figure 4.2: XText model generation from grammar

4.3.4 Lexical and Syntactic analysis

Given the grammar, XText provides an editor with syntax highlighting and basic auto completion. It also includes lexical and syntactic analysis, detecting some errors in the programs written.



(a) Wrong type error.

(b) Reference error.

Figure 4.3: Errors detected by the XText parser.

The parser will also generate the syntax tree for a program. For example, for the program:

```
1  label: add 1 '1'
2  goto label
```

XText will provide a *Program* object, with a *CommandBlock* attribute, representing its *body*, that contains two *Command* objects, one *Write* and one *Goto*. Usage of this syntax tree will be discussed in section 4.5

4.4 Code validation: Scoping and Validating

4.4.1 Scoping

There are some programs that, though syntactically valid, are not valid $1\#X$ programs. Consider the following program:

```
1  Func() begin
2    label: add 1 '1'
3  end
4
5  goto label
```

Indeed, it follows the grammar, and the *Goto* statement references. However, this statement makes little sense. How would we get inside a function without calling it? What should happen after the rest of the function is executed? As we mention in section 3.1.3, *Goto* statements should reference labels within a function declaration or within the main program body. The scope refers to the set of visible elements from a given reference site (for example, a *Goto* statement). [11]. XText provides tools to define these kind of scopes, so that the behavior from the editor is the expected:

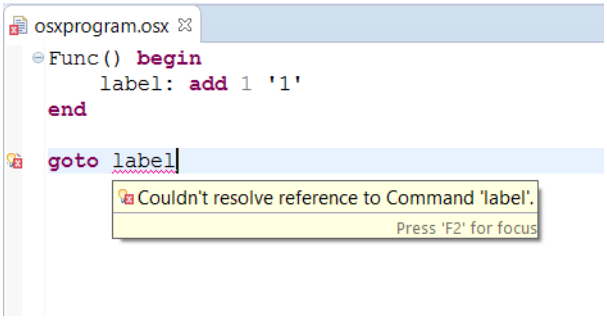


Figure 4.4: Scoping error.

Using the XText scoping tools we provide scopes for:

1. Labels and *Goto* statements.
2. Use of **int** and **string** parameters. (Only should be used within the same function they are declared).

4.4.2 Validation

Even after scoping correctly, there are some statements that we would like to consider as "incorrect". Consider the following:

```
1 label: goto label
```

This not only has no direct equivalent in 1#, it makes no sense to have it. If we really wished to loop infinitely, we could use the silly loop from 2.1.2. And though

this issue could be resolved by removing the label from the scope, this is more a problem of *constraints* than a problem of *scopes* [11]. Therefore, we use XText tools for code validation. The result is the following:

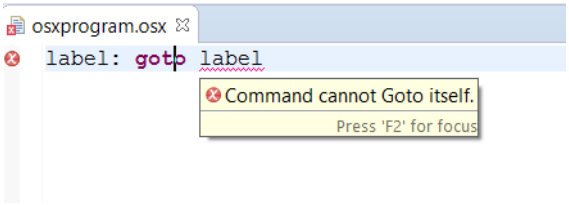


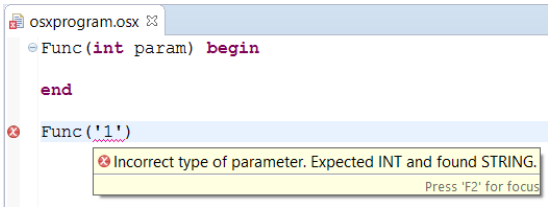
Figure 4.5: Constraint error.

Validation is specially useful for checking that function calls are done with the right types of parameters. The following code, for example, is syntactically valid and meets all scoping constraints:

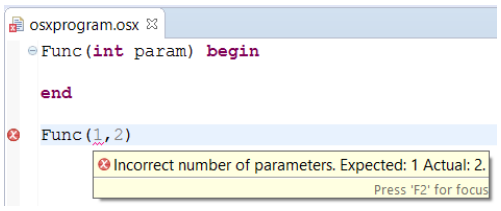
```
1 Func(int iparam)begin
2   write iparam '1'
3 end
4 Func('1')
5 Func(1,2)
```

The first function call attempts to do so with the wrong type of parameter, while the second one uses the wrong number of parameters. Validation is also used to prevent recursive function calls, since these can't be resolved at compile time:

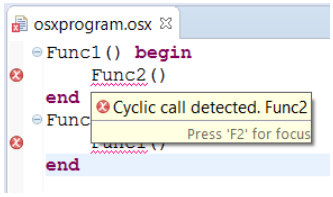
```
1 Func1() begin
2 Func2()
3 end
4
5 Func2() begin
6 Func1()
7 end
8
9 Func1()
```



(a) Wrong type error.



(b) Wrong number of parameters error.



(c) Cyclic call error.

Figure 4.6: Errors detected by validation.

With XText, we can identify these as bad code.

Validation is also used to assure that labels, parameter names and function names are unique within their scopes.

4.5 Code generation: Compiling $1\#X$

Once the editor is set up, it is time to compile $1\#X$ programs. XText provides a basis to implement code generation from the syntax tree. We start with an object of type *Program*, which contains all the components of the program written by the user. The job is to figure out how each of the components should translate to $1\#$. Formally, using the definitions from section 1.4.2, we will prove that:

Proposition 4.5.1 ($1\#X$ is as strong as $1\#$) For every $1\#X$ program P, there exists a $1\#$ program P' such that $P' \simeq P$.

General idea The idea is to implement methods:

```
1 int compile(int start, Command cmd);
2 int compile(int start, CommandBlock block);
3 int compile(int start, UnlabeledCommand cmd);
```

Every basic instruction on $1\#X$ extends *UnlabeledCommand*. The methods receive as parameter the line in the $1\#$ program where they will start writing the translation for the component, and return the first line in the $1\#$ program that goes after the $1\#$ translation of the specified component.

4.5.1 Resolving labels and parameters

Label Contexts

A map is kept indicating, for a label, its corresponding instruction (line) number in the $1\#$ translation. There is one map for each label scope. It is worth noting that every time a function is called, a new map is created. This will be more clear when we explain how function calls are compiled in section 4.5.8. We will call each of these maps a **Label Context**.

Parameter Contexts

A map is also kept to resolve, given a parameter's name, its actual value. We also keep a mapping from parameter names to values. Recall that all parameter values should be known at compile time. However, over the execution, a function may be called multiple times, each time its parameters taking different values. For this reason, more than one map is necessary. We will call these maps **Parameter Contexts**. They are kept in a stack that follows the function call stack [10].

4.5.2 Compiling command blocks

Compiling command blocks reduces to compiling, sequentially, each of the commands that make it up. The main responsibility of this method is to keep the correct label context updated.

4.5.3 Compiling *Command*

A *Command* is simply a simple instruction (an *UnlabeledCommand*) maybe accompanied by a label. If there is a label, a new entry is added to the current label

context, mapping the label to the corresponding line number of the command in the $1\#$ translation. After the map is updated, the contained *UnlabeledCommand* is compiled.

4.5.4 Compiling *Write*

This is the easiest instruction to compile, since there is a direct translation between a *Write* instruction in $1\#X$ and in $1\#$. We may need to access the current Parameter Context to resolve the register number or the word to be written, in case they were indicated by parameter names.

4.5.5 Compiling *Goto*

Since a mapping is kept between labels and line numbers, a *Goto* statement is translated by determining the current line number, and the line number which the label maps to. With these two numbers, we can do a control transfer instruction in $1\#$.

4.5.6 Compiling *Exit*

The exit instruction means that the program should halt improperly. We use the current line number, and do a *Back* in $1\#$ which we are sure points to an instruction before the first one. This will cause the $1\#$ program to halt improperly as well.

4.5.7 Compiling *Cases*

As it is done when writing $1\#$ programs, the cases statement points to another place in the program where the actual case is implemented. Compiling the *Cases* instruction in $1\#X$ is a matter of keeping track of where the implementations for each case will go in the $1\#$ program. At the end of each case implementation, a transfer instruction is added to transfer control to the end of the complete cases statement. For each case, we call the method that compiles *Command Blocks*. If the register number is indicated by a parameter, we resolve this using the current (top) Parameter Context.

4.5.8 Compiling function *Calls*

For each *Call* statement, a new Parameter Context is added to the top of the stack with the parameter values indicated by the call. After that, the *compile(int,CommandBlock)* method is called. This means that each time a function is called, it is compiled. This could be done more efficiently, but this has not proved to cause a performance problem yet.

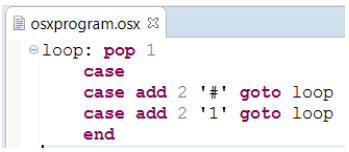
Observation 3 (Recursive Calls) Because of the way we compile functions, where each time they are called, their $1\#$ code is generated again and added to the program, we cannot allow recursive function calls, or cyclic calls in functions, since this would not always allow us to, at compile time, to provide a bound on the length of the program, and the code generator could end up in an infinite loop, writing the same function definition over and over again. This could be fixed by compiling function calls in a different way, like having a single block of code for each function. This on its own has many (technical) complications, including:

- Since functions may contain parameters, these cannot be hard coded in the resulting $1\#$ program, like it is done with the current compilation method. Therefore, we would need to find a way of having functions read the parameters they need, and operate accordingly.
- Functions may be called more than once during the execution of a program. Calling the function itself is not a problem, since the start of its definition block is static and therefore can be resolved at compile time. However, when the function is executed, the program must transfer control back to the place where the function was called. So function definition blocks should have a way of resolving, at RUNTIME, where the program should continue executing after the function call is completed.

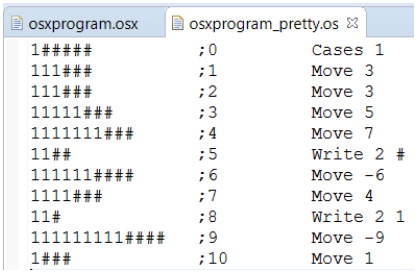
For the reasons mentioned above, we decided not to forgo with the technical difficulties of allowing recursive function calls in $1\#X$, as we believe it to be out of scope, following the objectives stated in Section 1.3

4.5.9 Compilation result

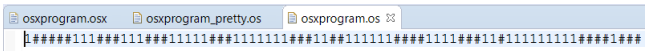
The compilation process takes a $1\#X$ program and transforms it into a list of $1\#$ instructions. This list is then written out to a *.os* file. Another file, with the suffix *_pretty.os* is created, which is a commented version of the $1\#$ code. A third file is written, with *.osi* extension. This corresponds to the $1\#X$ program translated to an intermediate language, $1\#i$, explained in section 4.6.1.



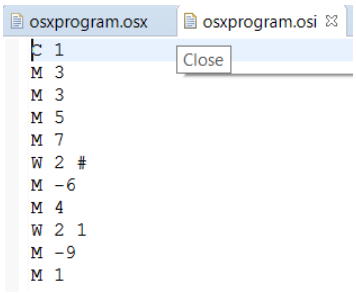
(a) Original $1\#X$ program.



(b) $1\#$ pretty generated code.



(c) $1\#$ generated code.



(d) $1\#i$ generated code.

Figure 4.7: Compilation results.

4.5.10 Generated code vs. Original Code

The way code is translated from $1\#X$ to $1\#$, and the lack of optimizations in this process, make translations to $1\#$ longer than their equivalent original counterparts. Let's take a look again at the $Move_{2,1}$ program provided in [6]:

```
1 11#####        ;Cases on R2
2 111111###      ;Case empty (Move Forward 6 to end)
3 111###         ;Case 1 (Move Forward 3 to
4                ;     Case 1 implementation)
5 1##            ;Case # (Write # to 1)
6 1111####       ;Back 4 (To Cases statement for loop)
```

```
1##             ;Case 1 implementation (Write 1 to 1)
111111####      ;Back 6 (To Cases statement for loop)
```

./Examples/Move21.os

Writing the same program in $1\#X$ and compiling it to $1\#$ yields:

```
11#####          ;0          Cases 2
111###           ;1          Move 3
111###           ;2          Move 3
11111###         ;3          Move 5
1111111###       ;4          Move 7
1#               ;5          Write 1 1
111111####       ;6          Move -6
1111###          ;7          Move 4
1##              ;8          Write 1 #
111111111####    ;9          Move -9
1###             ;10         Move 1
```

./Examples/Move21_gen.os

For this simple program the difference is 4 lines, the generated code is around 157% longer. Although in here it does not seem like much, there are some examples in which this becomes more evident. (See section 5.3).
So, there is a tradeoff when using $1\#X$ and translating to $1\#$, instead of writing programs directly in $1\#$.

4.6 Running $1\#X$ programs

$1\#X$ programs are never run directly. Instead, they are converted to $1\#$ programs. [6] provides several interpreters for $1\#$. We implemented another one, called the *OsRunner*. Complete instructions on its usage can be found in Appendix B.3.

4.6.1 $1\#i$: The intermediate language

The Java interpreter does not run $1\#$ programs directly. Instead, it converts them to an intermediate language, $1\#i$. Though we will not explain its syntax in detail,

it can be easily inferred from the examples presented in Appendix A, and it can be compared to the shorthand $1\#$ syntax of the *Sanity* parser described in section 2.3.1. The idea is that the $1\#i$ commands are much easier to read and parse than the commands made from 1s and $\#$s. When the *OsRunner* receives a $1\#$ program, it will automatically convert it to $1\#i$. However, it can receive $1\#i$ programs directly, which is why the $1\#X$ compiler generates $.osi$ files as well.

4.6.2 Executing $1\#i$ programs

Having a list of $1\#i$ commands, it is very easy to execute. We just need to keep the current instruction number that is being executed. Every time a *Write* instruction appears, the indicated character is written to the indicated register. Cases instructions just change the instruction number depending on the contents of the indicated register (as it would be done in $1\#$, if the register is empty move ahead one instruction, if it has a 1, pop it and move ahead two instructions, and move ahead three instructions if a $\#$ is popped). *Move* instructions move to the indicated instruction. Execution finishes if the current instruction number does not correspond to an instruction in the $1\#i$ program (so it is either less than 0, or greater than the program length).

Chapter 5

Studying Theory of Recursion with $1\#X$

We now shift focus from the implementation of $1\#X$, to its actual usage as a learning tool for Theory of Recursion, which is one of the main objectives of the original $1\#$. We will use the formal notation defined in section 1.4.2 to describe some programs studied in Lawrence Moss's *Theory of Recursion* course, which uses $1\#$ [6] [5].

5.1 The *Write* program

Definition 6 [The *Write* program] *Write* is a program such that for every $x \in \mathcal{A}^*$,

$$\Phi_{Write}(x) = y \text{ such that}$$
$$\Phi_y() = x$$

It is not difficult to come up with such program, for every 1 read, write a *write* 1 instruction, and the same for every $\#$ read. We need to use an auxiliary register to write the result, and then move the result to $R1$.

Example 6 (*Write* in $1\#$)

```
1 1#####          ;Cases on R1
2 111111111###    ;Empty. Forward 9 to Move(2,1)
3 11111###        ;One. Forward 5 to Case 1 Impl.
```

```
11#            ;Sharp. Write '1##' to R2. Add '1' to R2.
11##           ;              Add '#' to R2.
11##           ;              Add '#' to R2.
111111####     ;Back 6 to Cases statement.
11#            ;Case 1 Impl. Write '1#' to R1. Add '1' to R1.
11##           ;               Add '#' to R1.
111111111#### ;Back 9 to Cases statement.
11#####        ;Move(2,1). Cases on 2
111111###      ;Empty. Forward 6 to End.
111###         ;One. Forward 3 to Case 1 Impl.
1##            ;Sharp. Write '#' to R1.
1111####       ;Back 4 to Cases on 2.
1#             ;Case 1 Impl. Write '1' to R1.
111111####     ;Back 6 to Cases on 2.
```

./Examples/Write.os

Example 7 (*Write* in $1\#X$)

```
/**
 * Writes the Write program for the initial content of SRC in
   the SRC register.
 * Pre: TEMP is empty.
 * Pos: TEMP is empty.
 */
Write(int src, int temp)begin
  loop: pop src
    case
      Move(temp,src)
    case
      add temp "1#"
      goto loop
    case
      add temp "1##"
      goto loop
  end
end

```

```
19 Write(1,2)
```

./Examples/Write.osx

We define the *Write* program as a function in $1\#X$. Notice that to meet Definition 6, parameter *src* should be 1, and parameter *temp* should be different than 1. Of course, the $1\#X$ code is more readable. It is also shorter (in lines) than the $1\#$ code, not counting commentaries.

5.2 The *Diag* program

Definition 7 [The *Diag* program] *Diag* is a program such that for every $x \in \mathcal{A}^*$,

$$\Phi_{Diag}(x) = \Phi_{Write}(x) + x$$

Where + corresponds to string concatenation.

For now, it is not very clear what the *Diag* program does. However, it will be important for self-replication. We now present the (undocumented) version of the $1\#$ code. The idea is to read $R1$, writing the results of *Write* to $R3$ and copying $R1$ to $R2$. When we finish reading $R1$, we move $R3$ to $R1$ and then $R2$ to $R1$.

Example 8 [*Diag* in $1\#$]

```
1 1#####
2 1111111111###
3 111111###
4 11##
5 111#
6 111##
7 111##
8 1111111####
9 11#
10 111#
11 111##
12 1111####
13 111#####
```

```
111111###
111###
1##
1111####
1#
11####
11#####
111111###
111###
1##
1111####
1#
11####
```

./Examples/Diag.os

This program really shows how much easier it is to write these programs in $1\#X$. We take the description of Definition 7 and come up with the algorithm:

Example 9 [*Diag* in $1\#X$]

```
/**
 * Executes the Diag() program for the contents of src register,
    using temp1 and temp2 as temporal
 * registers.
 * Pre: temp1 and temp2 are empty.
 * Pos: The result of Diag(src) are on the src register, temp1
   and temp2 are empty.
 */
Diag(int src, int temp1, int temp2) begin
  Copy(src,temp1,temp2) //Copy the contents of Rsrc to Rtemp1.
  Write(src,temp2)    //Do the Write program for Rsrc.
  Move(temp1,src)     //Move the contents of Rtemp1 to Rsrc.
end

Diag(1,2,3)
```

./Examples/Diag.osx

5.3 The *Self* program

Definition 8 [The *Self* program] *Self* is a $1\#$ program such that:

$$\Phi_{Self}() = Self$$

Self is a self writing program.

Note that *Self* cannot be defined for $1\#X$, since the alphabet used in $1\#X$ programs is not the same alphabet used in registers. We can, however, use $1\#X$ to get to a *Self* program. Note that there may be many programs in $1\#$. This is also the case with the other programs defined, so we should not talk about THE *Program* but about A *Program*.

Proposition 5.3.1 (A way of obtaining *Self*) Let D be a *Diag* program in $1\#$. Let S be a $1\#$ program such that:

$$S = \Phi_D(D)$$

Then S is a *Self* program.

Proof. From Definition 7 we have that:

$$\begin{aligned}\Phi_S() &= \Phi_{\Phi_D(D)}() \\ &= \Phi_{\Phi_{Write}(D)+D}() \\ &= \Phi_D(\Phi_{Write(D)}()) \\ &= \Phi_D(D) \\ &= S\end{aligned}$$

□

Just for amusement, we present two different *Self* programs.

Example 10 (*Self* program obtained using the $1\#$ *Diag* program from Example 8)

```
1# 1## 1## 1##1##1##1#1#1#1#1#1#1#1#1#1#1#1##1##
1##1#1#1#1#1#1#1##1##1##1#1#1##1##1#1#1#1##1#
```

```
1#1#1##1##1#1#1#1##1##1#1#1#1#1#1#1#1##1##1##
1##1#1#1##1#1#1#1##1#1#1#1##1##1#1#1#1#1#1#1#
1#1#1#1#1##1##1##1##1#1#1#1##1##1##1##1##1#1#1#
1#1#1#1##1##1##1#1#1#1##1##1##1#1##1##1#1#1#1#
1##1##1##1##1#1##1#1#1#1#1#1#1##1##1##1##1#1#1##
1##1##1##1##1#1#1#1#1#1#1#1##1##1#1#1#1#1##1##1##
1#1##1##1#1#1#1#1##1##1##1##1#1##1#1#1#1#1#1#1##
1##1##1##1#####1111111111##111111##11##111#111##111##
1111111#####11#111#111##1111111111####111#####111111###111###
1##1111#####1#111111#####11#####111111###111###1##1111#####1#
111111####
```

./Examples/self.os

Example 11 (*Self* program obtained using the $1\#X$ *Diag* program from Example 9 and then

```
1#1##1##1##1##1##1#1#1#1##1##1##1#1#1#1##1##1##1#
1#1#1#1#1#1##1##1##1#1#1#1#1#1#1#1#1#1##1##1##1#1#
1##1#1#1#1##1#1#1#1#1#1#1#1##1##1##1##1#1#1#1#1#1##
1##1##1#1#1##1##1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1##
1##1##1#1#1##1##1##1#1#1#1#1#1##1##1##1##1#1#1#1##1##
1##1#1#1#1##1##1##1#1#1#1#1#1#1##1##1##1#1#1#1#1##1##
1##1#1#1#1##1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1##1##
1##1##1##1#1#1#1##1##1##1#1#1#1#1#1#1#1#1#1#1#1#1#1##1##
1##1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1##
1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#
1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1##
1##1#1#1#1##1#1#1#1##1##1#1#1#1##1##1#1#1#1#1#1#1#1#1#
1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1##
1##1##1##1#1#1#1##1##1##1#1#1#1##1##1#1#1#1#1#1##1##1##
1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#1#
1#1##1##1##1#1##1##1#1#1#1#1#1#1#1#1#1#1##1##1##1#1##1##
1#####111###111###111111###111111111###11#111#1111111####
```

```
11111###11##111##1111111111####1###111#####111###111###
11111###1111111###1#111111####1111###1##1111111111####1###
1#####111###1111111111111###1111111111111111111###111#####111###
111###11111###1111111###1#111111####1111###1##1111111111####
1###1111111111###111#111##1111111111111111111####11111###111#
111##111##1111111111111111111111####1###11#####111###111###
11111###1111111###1#111111####1111###1##1111111111####1###
```

./Examples/selfOsx.os

Notice that the *Self* program acquired via $1\#X$ is almost twice as long! This is because our implementation of *Diag* is different, and the auto generated $1\#$ code is very non-optimized, in general being longer than its native $1\#$ counterpart! However, both programs are self writing programs, since they come from *Diag* programs. This leads us to a small fun conclusion:

Proposition 5.3.2 (Length of *Self* programs) There is no upper bound for the length of *Self* programs.

Proof. The proposition follows from the fact that there is no upper bound for the length of *Diag* programs. Consider this function:

```
Useless() begin
  Move(1,2)
  Move(2,1)
end

```

It is indeed a useless function, provided that $R2$ is empty. Now, the program D':

```
  Diag(1,2,3)
  Useless()

```

Is still a *Diag* program, but its translation to $1\#$ (which we will call [D'])has 22 more instructions (the length of the translation of *Useless*) than the original *Diag*

(which has 60 instructions). Doing $S' = \Phi_{D'}([D'])$ gives us another self writing program that is gonna be longer than $S = \Phi_D([D])$. Defining D'', D''' in a similar way by adding more Useless calls allow us to make arbitrarily large *Self* programs. Each *Useless* call adds about 200 lines to the resulting *Self* program.

□

Chapter 6

Conclusions

6.1 Conclusion

We developped an expressive version of $1\#$, $1\#X$, which we proved in this document to be equivalent. It meets the objectives stated in section 1.3. In the examples section and appendices, as well as in section 5, we have seen how the syntax of $1\#X$ makes programs a lot easier to write and understand, their contents and functionality being a lot clearer than their $1\#$ counterparts. As such, it builds on $1\#$'s objective of studying the Theory of Recursion, working well as a DSL for its specified domain.

In the process of designing and implementing $1\#X$, we have visited key references like [11] and [3], using their insights to produce a better final product. With the aid of these, we were able to use Xtext successfully to develop $1\#X$, from specifying its grammar to implementing a code generator that compiles $1\#X$ programs to $1\#$. This is another case where Xtext proves to be an excellent tool that allows us to effectively implement languages for a great number of domains.

6.2 Future work

There are many improvements that can be made to $1\#X$, in terms of both design and implementation. As of now $1\#X$ is just an expressive version of $1\#$, mirroring its instruction set and providing an equivalent syntax. However, there exist more

instructions that can be achieved with text register machines that are not direct derivates of the $1\#$ instructions. Such is the case of the *Move* instruction that we talked about in section 1, when defining Unlimited Text Register Machines. $1\#X$ could be extended to cover some of these alternative instructions. Cyclic function calls, discussed in Observation 3, could also be considered in this category.

In terms of implementation, there are more features that could be added to the $1\#X$ editor, like auto-completion, code assisting and quickfixing. All of these features are provided by Xtext and can be quickly and naturally added to the existing language definition.

Bibliography

[1] Tools for 1# by three planets software.

[2] Heiko Behrens, Michael Clay, Sven Efftinge, Moritz Eysholdt, Peter Friese, Jan Köhnlein, Knut Wannheden, and Sebastian Zarnekow. Xtext user guide. *Dostupné z WWW: http://www. eclipse. org/Xtext/documentation/1_0_1/xtext. html*, 2008.

[3] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. PACKT Publishing, 2013.

[4] Monica Lam, Ravi Sethi, JD Ullman, and AV Aho. Compilers: Principles, techniques and tools, 2006.

[5] Lawrence S Moss. Recursion theorems and self-replication via text register machine programs. *Bulletin of the EATCS*, 89:171–182, 2006.

[6] Lawrence S Moss. 1#: a text register machine introduction to computability theory, 2009.

[7] Richard E Pattis. Ebnf: A notation to describe syntax. *While developing a manuscript for a textbook on the Ada programming language in the late 1980s, I wrote a chapter on EBNF*, 1980.

[8] John C Shepherdson and Howard E Sturgis. Computability of recursive functions. *Journal of the ACM (JACM)*, 10(2):217–255, 1963.

[9] Alejandro Sotelo Arévalo. Gold 3: Un lenguaje de programación imperativo para la manipulación de grafos y estructuras de datos. 2012.

[10] Robert D Tennent. *Principles of programming languages*. Prentice Hall PTR, 1981.

[11] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. *DSL engineering: Designing, implementing and using domain-specific languages*. dslbook. org, 2013.

Appendix A

Examples in $1\#$-like languages

A.1 The $Move_{2,1}$ program

Example 12 (The $Move_{2,1}$ program in $1\#i$)

```
C 2      ;Cases on R2
M 6      ;Case empty (Move Forward 6 to end)
M 3      ;Case 1 (Move Forward 3 to Case 1 implementation)
W 1 #    ;Case # (Write # to 1)
M -4     ;Back 4 (To Cases statement for loop)
W 1 1    ;Case 1 implementation (Write 1 to 1)
M -6     ;Back 6 (To Cases statement for loop)
```

./Examples/Move21.osi

Example 13 (The $Move_{2,1}$ program in *Sanity*)

```
case 2
jump 6
jump 3
push# 1
jumpb 4
push1 1
jumpb 6
goto end
label end
```

./Examples/Move21.sanity

Appendix B

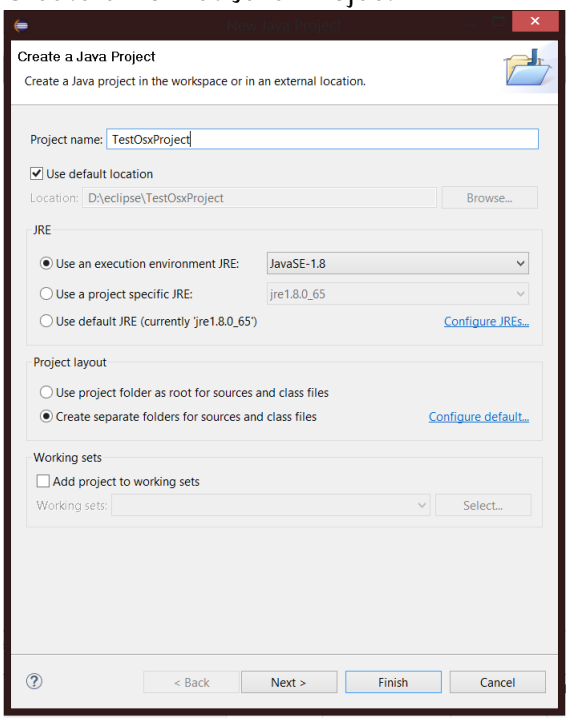
Installing the $1\#X$ environment

The most straightforward way to get started with $1\#X$ is by downloading the *Eclipse* version included in `https://goo.gl/hKfdLW`

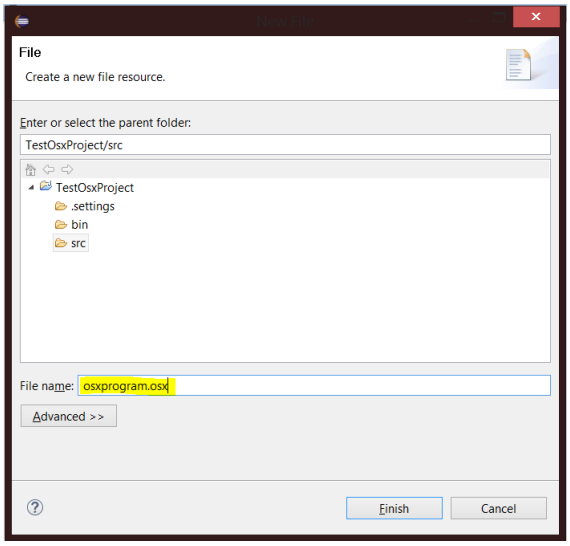
B.1 Creating an $1\#X$ project

Once inside the *Eclipse OSX* version:

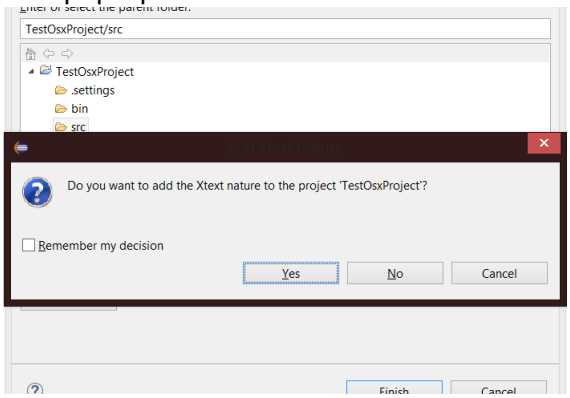
1. Create a normal Java Project.



2. Create a plain file. Make sure to name it with a *.osx* extension.

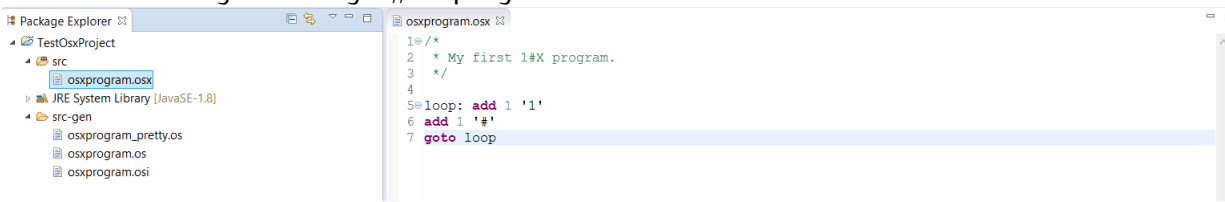


3. Eclipse should now identify automatically that this is an *osx* project, showing this pop up.



You should select 'Yes'.

4. You can now begin writing $1\#X$ programs.



B.2 Alternate: Manual install

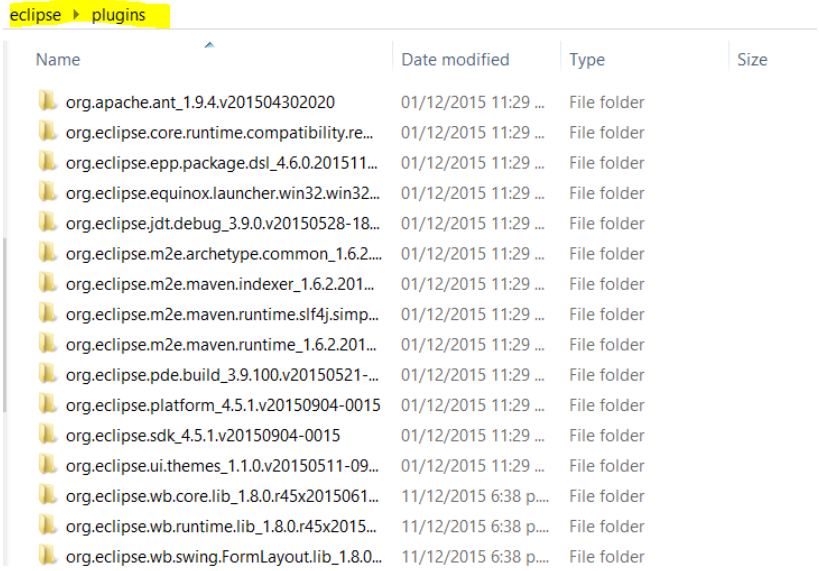
If you don't want to get a new version of Eclipse just for $1\#X$, or use an operating system different from Windows, follow these steps:

1. Start with an Eclipse distribution with Xtext installed, following the instructions

here `https://eclipse.org/Xtext/`.

2. Download the *plugin* directory from the download link given above.

3. Paste the contents of the *plugin* directory into the *plugins* directory of your Eclipse distribution:

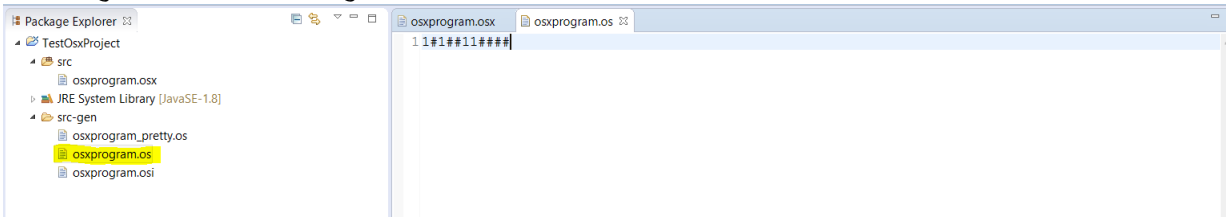


4. Restart Eclipse.

B.3 Compiling and running $1\#X$ programs

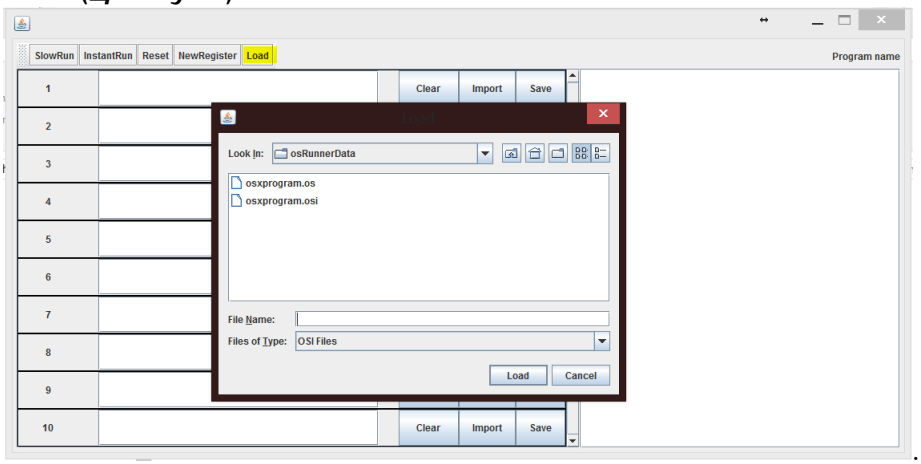
As explained before, $1\#X$ programs compile to $1\#$ programs. We have included a $1\#$ interpreter written in Java, as a Runnable JAR, available here `https://goo.gl/hKfdLW`

1. When a correct $1\#X$ program is written, it is automatically compiled to $1\#$, creating a file in the *src-gen* folder.

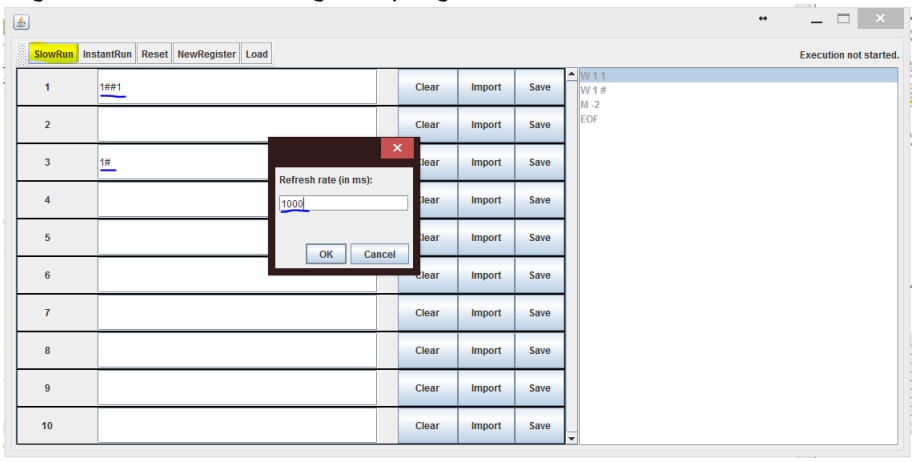


2. Open the *OsRunner* JAR and click *Load* to load a $1\#$ program. The runner will accept either the $1\#$ or the $1\#i$ (*.os* and *.osi* extensions) code created by

the $1\#X$ compiler. **Note that it currently does not accept the decorated $1\#$ code (*_pretty.os*)**.

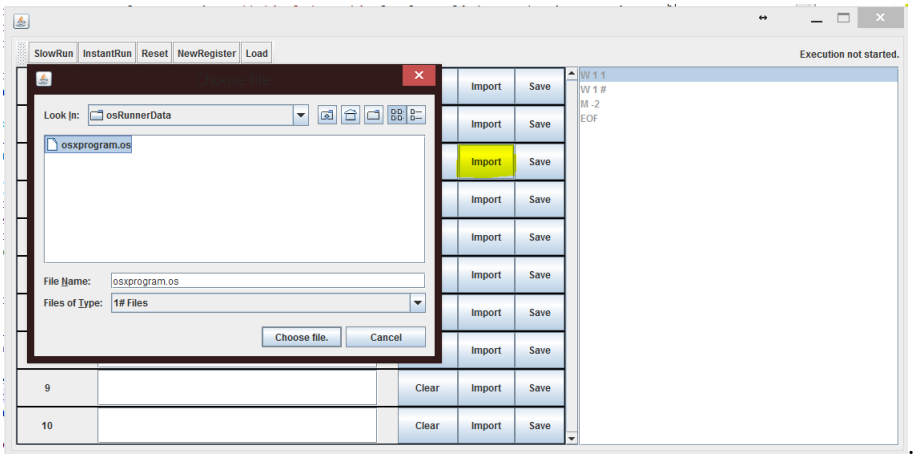


3. Once loaded, there are two ways to run the $1\#$ program. First there is the *Slow Run*, which allows you to define a *refresh rate* t. The program then will run one instruction every t milliseconds. You can also edit the contents of the registers before running the program.



The second way of running a program is by using the *InstantRun*, which runs the code instantly and presents the result.

4. You can import $1\#$ strings into the registers by clicking the *Import* button.



5. You can also save the contents of a specific register. If these correspond to a $1\#$ program, there is an option to save a $1\#i$ version alongside the $1\#$ code. **Make sure to name the file with a *.os* extension if you are saving a $1\#$ program.**