TEMPORARY NAME

Compiler Construction only Final Project

Jeroen Brinkman - s1366491 & Tom Velthausz – s1330187

Inhoud

Summary

Our language is called TempName and the file extension is .tmp. All .tmp files are compiled into an iloc program. Those iloc programs are runnable using the iloc simulator.

The TempName language supports variables of the following data types: Char, Integer, Bool and String. Besides variables constant values of those types are also supported.

Various operations can be done on variables and constants: Add, subtract, divide, modulo, print, read and the negation operation. Table \ref shows what operations are applicable to what type.

Type\Operation	+	-	/	*	%	!	and,or	<>, ==	<, >, <=, >=	read	print
Bool	*n*	*n*	*n*	*n*	*n*	*y*	*y*	*y*	*n*	*y*	*y*
Char	*y**	*n*	*n*	*n*	*n*	*n*	*n*	*y*	*n*	*y*	*y*
String	*y**	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*y*
Integer	*y*	*y*	*y*	*y*	*y*	*n*	*n*	*y*	*y*	*y*	*y*

* as a concat after a String (first argument must be a String).

Every statement is considered an expression with a return type. If the return type is void there is no return value, otherwise there exists a return value. Only multi statement prints, multi statement reads, declarations and while loops are voids.

Expression blocks are surrounded by brackets and return the type and value of the last expression in the block. Every block is a contained scope, closing a scope makes all variables declared in that block inaccessible.

Parentheses can be used to adjust the priority in which expressions are evaluated. The default priority can be found in table \ref.

Priority	Operators
1	(unary) -, !
2	*, /, %
3	+,-
4	<, <=, >=, >, ==, <>
5	and

6	or

Problems and Solutions

During the implementation of the program we encountered several problems. The major problems and their solutions will be discussed here.

ConcurrentModificationException in the memorymanager

While testing programs that declared a string variable and assigned it a value in the same line the MemoryManager started throwing ConcurrentModificationException. This was rather strange as we did not implement any concurrency into the MemoryManager.

Eventually we figured out that while iterating over an ArrayList we both removed an item and started a new iteration to add a new item without breaking the first iteration. Adding a break statement to the first iteration before adding the new element solved the problem. See below for a pseudocode example of the error generating implementation.

```
For(int i: ints){
        if(i = condition){
                Remove i;
                for(int j: ints){
                        if(j condition){
                                Add some int;
                        }
                }
        }
}
```

Character and String prints

We decided to use the standard iloc OpCode.cout for printing our characters and strings on the standard output. However instead of printing characters/strings correctly, the ASCII value of the various characters was printed. We initially solved this solution by editing the iloc library files, but after a discussion with Arend Rensink we decided to update iloc to the newest version instead.

String inputs

Because we allocate memory during code generation we were unable to deal with the variable length of strings inserted during runtime, because we were unable to guess how much memory should be allocated. We considered using a fixed length for these kinds of string but decided against it as the “overflow length” at the end of a fixed length string would contain a bunch of nonsense.

Another solution we considered were giving the user the option to define the length of such an input string. At the same time, we considered changing our memory allocation, using a pointer register in iloc. However, there was simply not enough time to implement those two solutions.

Detailed Language Description

The default expressions used to do integer arithmetic are the same as the java and c languages, because of this similarity we have decided not to elaborate as much on those kinds of expressions.

Program

A program written in the TempName language should always be saved as .tmp file. A program should always consist of a Start statement followed by a expression. Usually this expression is an blockexpression, see the relevant section for more information.

An example program could look like this

Start print(1)

This program would print the integer 1 on the standard output. Comments should be surrounded by #.

#This is a comment#

Whitespace and comments are disregarded by the parser and can be used as the user sees fit.

Code generation

A OpCode.nop is generated with the label “program”.

The code that is generated by the expressions follows this label directly.

Block expression

A block expression is a list of expressions which are executed in order. A block statement is always surrounded by brackets. Every expression in a block statement should be followed by a semicolon. Below is an example program using a block expression. This program prints the integers 1 and 2 on separate lines in the standard output.

```
Start {
        print(1);
        print(2);
}
```

Code generation

The code for a block statement is simply the code for all its members concatenated together. The various expressions are not separated by any labels or other barriers.

At the end of the block statement however there is some additional code for the return value of the entire block. As said before the return value of the entire block is the return value of the last expression in the block. Do note that this value is returned whether it is used or not, this results in some “useless” code at the end of every program using blockstatements.

Declaration Expression and Assignment Expression

Variables have to be declared before they can be assigned a value. To do so a declaration expression can be used. There are two different declarations in the following block expression.

```
{
Integer i := 0;
Integer j;
}
```

A variable can be declared without assigning a value and it can be declared while assigning a value. If you decide to directly assign a value the type of the value should be equal to the type of the variable that is declared.

Declarations follow this schema:

<Type> <ID> (:= <Expression>)?

Declarations have the return type void this means they cannot be used in combination with other expressions. Because of this the example below will not compile!

Integer a := 1 + Integer b:= 8;

Assignment expressions assign a different value to an already declared variable. Assignment operations on variables that are not yet in scope will generate compilation errors. Since assignments return the value that is assigned they can be chained together. See the example program below which will print the value 4 twice.

```
Start{
        Integer i := 1;
        Integer j := 2;
        i := j := 3 + i;
        print(i,j);
}
```

It is important to note that variable identifiers are case sensitive and should always start with a letter.

Assignments follow this schema:

<ID> := <Expression>

Code generation

A declaration without assignment does not generate any iloc code. It simply reserves a memory address for future use. However if there is a value to be assigned there will be iloc code that loads the value into a register and then stores the value at the correct offset in memory. There are of course small differences in assigning the value of another variable and assigning a constant value. In the former case the value has to be loaded from memory and in the later it has to be stored directly into memory.

If we assign a value to a string it will read and store every character one by one. This can result in an (almost) excessive amount of loads and stores close to each other.

An assignment expression generates the exact same iloc code.

If Then Else Expression

An if then else expression works very similar to other languages, except for the fact that it is regarded as an expression and not as a statement. Because of both the Then block and the Else block should have the same type. The if block should be of the type Bool as this is the condition. With this in mind a program with an If Then Else could look like this:

```
Start{
        Integer i;
        Integer j;
        read(j);
        If j > 5 then {
                i := 5;
        }else{
                i := j;
        };
}
```

Important to note is that both assignments have the type integer. This means that the return type of the if then else expression is integer as well. Secondly the else block is optional. Thirdly the entire if then else expression is followed by a semicolon, because every expression in a block should be followed by a semicolon. The general layout of the if then else is:

IF <Bool expression> THEN <Expression> (ELSE <Expression>)?

Where both <Expression>'s have the same type.

Code generation

The iloc code generated firstly visits the boolean conditional expression. It uses the return value of the expression for a conditional break. The true value of the conditional break will always point to the label of the then statement. If an else statement exists the false value of the cbr will point to the label of this statement, otherwise the false value will point to the endiflabel.

After the conditional break an empty statement will be inserted with a label, this is the start of the then block. After this label the then expression will be visited and generated. At the end of the then expression a OpCode.jumpI will be emitted pointing to the end iflabel.

Once this is done the compiler will check for the existence of an else statement. If the else exists it will insert an empty statement with the else label. After this label the else is visited and its code is generated.

The last thing that happens is the insertion of the endif label, which marks the end of the if expression.

While expression

While expressions always have the return type void and do not have a return value. While expressions follow this schema:

WHILE <Expression> DO <Expression>.

The first expression should always have the return type boolean. The return type of the second expression does not matter. The while loop will execute the expression after the DO keyword on repeat until the first expression no longer returns true. If the first expression is false from the start the body will never be executed. An example program which counts to ten using a while loop can be found below.

```
Start{
	Integer i := 0;
	While i <= 10 Do{
		print(i);
		i := i +1;
	};
}
```

Code generation

The while code generation starts with an empty statement that has the wbegin label. After that label the boolean expression is visited and generated. The return value of this expression is then used to create a conditional branch which jumps to the wend label when false or wbody label when true. After the branch the body label is inserted, followed by visiting the body expression. Directly after the body a jumpI is inserted back to the start of the while. The last statement is an empty statement with the wend label.

Print and Read expressions

For input and output we have implemented a print and read statement.

The print statement uses one or more expressions as its arguments. Those expression do not need to have the same type, but if there is more than one expression the return type of the print becomes void. If there is only one type the return type and value will be those of the only expression argument. All arguments should be separated by commas.

The read statement reads user input from the command line and stores it in the variable that is specified in the read. A read is always a void no matter the amount of or the types of the arguments. Again all arguments should be separated by commas. It is important that the ID's that are used for the read are in scope. All arguments should be ID's.

Read is sadly not supported for strings because of complications with our memory management.

An example program using reads and writes (reads two values and prints them):

```
Start{
	Integer i;
	Integer j;
	read(i,j);
	print(i,j);
}
```

Code generation

Print

The Print visits all expressions and uses OpCode.out and OpCode.cout to print those return values on the standard output.

Read

The read uses OpCode.in and OpCode.cin to let the user give their input. Afterwards it stores the input at the correct point in memory using the identifiers in its argument list.

Arithmetic expressions

TempName supports basic arithmetic, table \ref elaborates which operator can be used with what types. The operator priority can be found in \ref table.

Description of the Software

TempNameCompiler.java

The main class of the zip file, this class can be runned in two ways:

Using the main of the compiler, with the filepath to a .tmp file and a run mode as arguments.

Using the “TempNameCompiler.instance” to call upon the compiler. This method has been used in all the test classes.

If the program is runned by method 1, the compiler creates a new instance and compiles using the filename. It parses the file and returns the ParseTree resulting from the parser. The ParseTree is used as input for the checker, and this results in a Result. The ParseTree, together with the just generated Result, are then given to the generator to generate the iloc Program and returns it. This Program is prettyPrinted and then passed on to the simulator, which tries to run the program. If input is needed, the program prints the variable name with a question mark and you can provide the input which is needed.

The run modes are simple, type “ RUN” after the filename to let it compile and run it through the simulator or “ FILE” to transfer the generated iloc code to a .iloc file.

TypeKind.java

TypeKind is the enum which contains the different types used. Integer: INT, Boolean: BOOL, Character: CHAR, String: STRING and for the empty: VOID.

Result.java

This class contains the results of the TypeChecker, which consists of three ParseTreeProperties: the entries, which are the associations from a parse tree node to the flow graph entry, the types, which are the associations from an expression/type/assignment node to the corresponding (inferred) type and lastly the read types, which contain the association from a read node with the corresponding types of the arguments.

Type.java

The Superclass for the type subclasses, which contains the method overwrites to get the size (the size to be used as offset in the memory) of every type an the toString() function.

TypeChecker.java

This class is the implementation of the listener generated by the grammar. The ParseTree generated by the basic TempNameParser is the input for the main check() method. This ParsTree will be traversed and for every exit of an expression/operation/type an entry and a type is added to the result as a parsetree property. This is done for every expression, except for the while expression, as the result of this expression is always a Type.VOID, the decleration expression, which is also always a Type.VOID and lastly the program itself.

SymbolTable.java

The class which is used to store/check the variables in the different scopes.

A single instance of the SymbolTable is used in the checker. This instance can be used to open/close a scope, which means that a Hashmap containing the variables with their types is pusht on/popt off the stack respectively.

Generator.java

The class which extends the visitor generated by the grammar. This class' main method "generate()" generates the iloc Program from the Result of the TypeChecker in combination with the ParseTree from the parser. It creates a new ParsetreeProperty labels to give labels to the iloc code (for if/else/while purposes) and adds a MemoryManager before it begins visiting the nodes. For all visit methods it visits the children in the necessary order and checks if extra id's should be loaded if the sub expression is a variable name. The visit returns this id in three cases: the decleration, the assignment and the visitIdExpr. The latter is always entered via the former two, and thus the ID is thrown up. The reason this ID has to be thrown up to the upper lying visits is for acquiring the value with the correct variable name from the memory, as the comparison of nodes results in a different/no location.

MemoryManager.java

The class that keeps track of memory allocation, using layered scope we are capable of freeing memory that is no longer used for further use. It keeps an abstract model of the memory during the generator execution. The generator is also responsible for opening and closing scopes. The MemoryManager has a RegisterManager for register management.

RegisterManager.java

Keeps track of which registers are available and which are in use. Capable of giving out registers and freeing them when no longer used. The MemoryManager is the only class that should communicate with the register manager.

The testfiles

CheckerTest.java, GeneratorTest.java, StringTest.java and ProgramsTest.java are the testclasses used to test the respective classes/programs. In the next part will be explained what all the tests contain and on what parts they are tested for.

Test Plan and Results

The testing, according to the assignment, should consist of the checks in the:

1. Lexical syntax (e.g. spelling errors).
2. Context-free syntax (e.g. language-construct errors).
3. Context constraints (e.g. declaration, scope and type errors).
4. Semantics (e.g. run-time errors).

The checks for these four syntaxes have been included in the zip file. The folder src/files contains all the files used in the tests. For all test programs the first two points (spelling errors or wrong syntax) are being filter out right away and will produce an error looking like:

```
line 7:5 - no viable alternative at input 'a1337'
```

```
line 4:8 - no viable alternative at input 'boolenb'
```

```
line 1:10 - mismatched input 'ba' expecting {
```

For a spelling error, or:

```
Errors:
line 7:3 - token recognition error at: '= '
```

For a syntax fault.

These two usually come pared as an incorrect token causes spelling issues. But a spelling error in a keyword causes the program to shut down before it reaches the compiler. This means that no other errors will be found/printed after this error. This ensures that only a program which will be parsed correctly will go to the typechecker/compiler.

In the src/files/TypeCheckerTestFiles folder are the tests used in the TypeChecker/TypeCheckerTest. These test check the correct declarations, types and scopes of the testfiles (point 3).

```
Line 11:1 - Illegal declaration, 'ad' already in scope.
Line 14:17 - Expected type 'Boolean' but found 'Integer'
```

The folder contains the following files:

- Basic.tmp, which should not return errors and checks correct type declarations.
- Err0.tmp, which returns errors. This checks for the errors returned by incorrect type declarations.
- Ops.tmp, which should not return errors and checks the typechecking in the operations (multiplication/division and modulo, addition and subtraction, prefixes, boolean operations and lastly, boolean comparisons).
- Err1.tmp, which returns errors. This checks for the errors returned by incorrect types used in operations.
- Exprs.tmp, which should not return errors and checks the typechecking in the expressions (If, While, Read, Print, Parentheses and Blocks).
- Err2.tmp, which returns errors. This checks for the errors returned by incorrect types used in the expressions.
- Err3.tmp, this returns an error. This is an incorrect program causing a spelling error and thus not reaching the compiler.

The last point (run-time errors) are not specially intercepted, but are handled as it usually is when Java handles these errors. These errors will be found while running the program, as a different input can also cause e.g. a division by zero error and is not handled by our generator.

In the src/files/GeneratorTestFiles folder are the tests used in the Generator/GeneratorTest. These test check correct output from various programs and inputs.

The folder contains the following files:

- arithmeticMult.tmp, which checks for correct output by using the multiplication, division and modulo operations.
- arithmeticPlus.tmp, which checks for correct output by using the addition, subtraction and the unary operations.
- assDecl.tmp, which checks for correct output by using the assignment and declaration expressions.
- compBool.tmp, which checks for correct output by using the “and/or” and comparison operations.

- ifWhile.tmp, which checks for correct output by using the if and while expressions, for the if this also checks for correct assignment of the result of the if statement.
- Program.tmp, which checks if one expression in the program executes normally.
- readPrint.tmp, which checks for correct output by using the read and print expressions, for these two this also checks for correct assignment of the result of the read/print statement respectively.

In the src/files folder are two basic programs: A program that calculates the greatest common divisor of two inputs and a program that calculates the Fibonacci number of one input.

The fibonacci program is included in the appendix, together with the generated iloc code and some sample input/outputs.

Conclusions

After doing all the homework exercises from the lab sessions, a lot of small things that seem simple are broached. But when they have to be combined, a lot of different things are joining the fray and keeping these parts all manageable is a difficult task to do if you build a complete programming language from scratch. The symbol table, the memory/register management and the different errors that need to be handled with are some of the things we had to think really hard about as to how we would implement these. The simplest would be to do it in a similar matter as the homework exercises, but this came apparent that it would not be so easy to combine with our language and reducing chaos with those parts underling.

The only regrets we have or maybe things that could have been better are the strings. It works, but it causes for a lot of loads in the iloc code. Also we should have done the Memory Management different.

Appendix

Antlr grammar:

```
grammar TempName;

program
      : START expr EOF
      ;


/** Target of an assignment. */
target
    : ID                #idTarget
    ;


/** Expression. */
expr: type ID (ASS expr)?                          #declExpr
      | target (ASS expr)                          #assExpr
    | IF expr THEN expr (ELSE expr)?               #ifExpr
    | WHILE expr DO expr                           #whileExpr
    | READ LPAR ID (COMMA ID)* RPAR                #readExpr
    | PRINT LPAR expr (COMMA expr)* RPAR           #printExpr
    | prfOp expr                                   #prfExpr
    | expr multOp expr                             #multExpr
    | expr plusOp expr                             #plusExpr
    | expr compOp expr                             #compExpr
    | expr boolOp expr                             #boolExpr
    | LPAR expr RPAR                               #parExpr
    | LBRACE (expr SEMI)* expr SEMI RBRACE         #blockExpr
    | ID                                           #idExpr
    | NUM                                          #numExpr
    | TRUE                                         #trueExpr
    | FALSE                                        #falseExpr
    | CHR                                          #charExpr
    | STR                                          #stringExpr
    ;

/** Prefix operator. */
prfOp: MINUS | NOT;

/** Multiplicative operator. */
multOp: STAR | SLASH | MODULO;

/** Additive operator. */
plusOp: PLUS | MINUS;

/** Boolean operator. */
//TODO appart?
boolOp: AND | OR;

/** Comparison operator. */
compOp: LE | LT | GE | GT | EQ | NE;

/** Data type. */
type: INTEGER  #intType
    | BOOLEAN  #boolType
    | CHAR     #charType
    | STRING   #stringType
    ;
```

```
// Keywords
AND:      A N D;
BOOLEAN: B O O L E A N ;
CHAR:     C H A R ;
INTEGER: I N T E G E R ;
DO:       D O ;
ELSE:     E L S E ;
FALSE:    F A L S E ;
IF:       I F ;
THEN:     T H E N ;
NOT:      N O T ;
OR:       O R ;
OUT:      O U T ;
PRINT:        P R I N T;
READ:     R E A D ;
START:        S T A R T ;
STRING:   S T R I N G ;
TRUE:     T R U E ;
WHILE:    W H I L E ;

ASS:     ':=';
COLON:   ':';
COMMA:   ',';
DOT:     '.';
DQUOTE:  '"';
EQ:      '==';
GE:      '>=';
GT:      '>';
HTAG: '#';
LE:      '<=';
LBRACE:  '{';
LPAR:    '(';
LT:      '<';
MINUS:   '-';
MODULO:      '%';
NE:      '<>';
PLUS:    '+';
QUOTE:   '\'';
RBRACE:  '}';
RPAR:    ')';
SEMI:    ';';
SLASH:   '/';
STAR:    '*';

// Content-bearing token types
ID: LETTER (LETTER | DIGIT)*;
NUM: DIGIT (DIGIT)*;
STR: DQUOTE .*? DQUOTE;
CHR: QUOTE LETTER QUOTE;

fragment LETTER: [a-zA-Z];
fragment DIGIT: [0-9];

// Skipped token types
WS: [ \t\r\n]+ -> skip;
COMMENT: HTAG .*? HTAG -> skip;

fragment A: [aA];
fragment B: [bB];
fragment C: [cC];
fragment D: [dD];
```

```
fragment E: [eE];
fragment F: [fF];
fragment G: [gG];
fragment H: [hH];
fragment I: [iI];
fragment J: [jJ];
fragment K: [kK];
fragment L: [lL];
fragment M: [mM];
fragment N: [nN];
fragment O: [oO];
fragment P: [pP];
fragment Q: [qQ];
fragment R: [rR];
fragment S: [sS];
fragment T: [tT];
fragment U: [uU];
fragment V: [vV];
fragment W: [wW];
fragment X: [xX];
fragment Y: [yY];
fragment Z: [zZ];
```

Antlr listener:

```
package checker;

import grammar.TempNameBaseListener;
import grammar.TempNameParser;
import grammar.TempNameParser.ExprContext;
import grammar.TempNameParser.*;

import java.util.ArrayList;
import java.util.List;

import org.antlr.v4.runtime.ParserRuleContext;
import org.antlr.v4.runtime.Token;
import org.antlr.v4.runtime.misc.NotNull;
import org.antlr.v4.runtime.tree.ParseTree;
import org.antlr.v4.runtime.tree.ParseTreeWalker;

/** Class to type check and calculate flow entries and variable offsets. */
public class TypeChecker extends TempNameBaseListener {
	/** Result of the latest call of {@link #check}. */
	private Result result;
	/** List of errors collected in the latest call of {@link #check}. */
	private List<String> errors;
	/** Variable scope for the latest call of {@link #check}. */
	private SymbolTable sT;

	/**
	 * Runs this checker on a given parse tree, and returns the checker result.
	 *
	 * @throws ParseException
	 *		if an error was found during checking.
	 */
```

```
	public Result check(ParseTree tree) throws ParseException {
		this.result = new Result();
		this.errors = new ArrayList<>();
		new ParseTreeWalker().walk(this, tree);
		if (hasErrors()) {
			throw new ParseException(getErrors());
		}
		return this.result;
	}

	/** Indicates if any errors were encountered in this tree listener. */
	public boolean hasErrors() {
		return !getErrors().isEmpty();
	}

	/** Returns the list of errors collected in this tree listener. */
	public List<String> getErrors() {
		return this.errors;
	}

	/**
	 * Checks the inferred type of a given parse tree, and adds an error if it
	 * does not correspond to the expected type.
	 */
	private void checkType(ParserRuleContext node, Type expected) {
		Type actual = getType(node);
		if (actual == null) {
			throw new IllegalArgumentException("Missing inferred type of " +
node.getText());
		}
		if (!actual.equals(expected)) {
			addError(node, "Expected type '%s' but found '%s'", expected, actual);
		}
	}

	/**
	 * Checks the inferred type of a given parse tree, and adds an error if it
	 * does correspond to the expected type.
	 */
	private void checkNotType(ParserRuleContext node, Type expected) {
		Type actual = getType(node);
		if (actual == null) {
			throw new IllegalArgumentException("Missing inferred type of " +
node.getText());
		}
		if (actual.equals(expected)) {
			addError(node, "Illegal type '%s' not allowed", expected);
		}
	}

	/**
	 * Records an error at a given parse tree node.
```

```
 *
 * @param ctx
 *          the parse tree node at which the error occurred
 * @param message
 *          the error message
 * @param args
 *          arguments for the message, see {@link String#format}
 */
public void addError(ParserRuleContext node, String message, Object... args) {
        addError(node.getStart(), message, args);
}

/**
 * Records an error at a given token.
 *
 * @param token
 *          the token at which the error occurred
 * @param message
 *          the error message
 * @param args
 *          arguments for the message, see {@link String#format}
 */
private void addError(Token token, String message, Object... args) {
        int line = token.getLine();
        int column = token.getCharPositionInLine();
        message = String.format(message, args);
        message = String.format("Line %d:%d - %s", line, column, message);
        this.errors.add(message);
}

/** Convenience method to add a type to the result. */
private void setType(ParseTree node, Type type) {
        this.result.setType(node, type);
}

/** Returns the type of a given expression or type node. */
private Type getType(ParseTree node) {
        return this.result.getType(node);
}

/** Convenience method to add a flow graph entry to the result. */
private void setEntry(ParseTree node, ParserRuleContext entry) {
        if (entry == null) {
                throw new IllegalArgumentException("Null flow graph entry");
        }
        this.result.setEntry(node, entry);
}

/** Returns the flow graph entry of a given expression or statement. */
private ParserRuleContext entry(ParseTree node) {
        return this.result.getEntry(node);
}
```

```
@Override
public void enterProgram(@NotNull TempNameParser.ProgramContext ctx) {
	sT = new SymbolTable();
	sT.openScope();
}

@Override
public void exitProgram(@NotNull TempNameParser.ProgramContext ctx) {
	sT.closeScope();
}

@Override
public void enterBlockExpr(BlockExprContext ctx) {
	sT.openScope();
}

@Override
public void exitTrueExpr(TrueExprContext ctx) {
	setType(ctx, Type.BOOL);
	setEntry(ctx, ctx);
}

@Override
public void exitBoolType(BoolTypeContext ctx) {
	setType(ctx, Type.BOOL);
}

@Override
public void exitStringExpr(StringExprContext ctx) {
	setType(ctx, Type.STRING);
	setEntry(ctx, ctx);
}

@Override
public void exitParExpr(ParExprContext ctx) {
	setType(ctx, getType(ctx.expr()));
	setEntry(ctx, entry(ctx.expr()));
}

@Override
public void exitCompExpr(CompExprContext ctx) {
	if (ctx.compOp().getText().equals("<>") || ctx.compOp().getText().equals("==")) {
		this.checkNotType(ctx.expr(1), Type.STRING);
		checkType(ctx.expr(1), getType(ctx.expr(0)));
	} else {
		checkType(ctx.expr(0), Type.INT);
		checkType(ctx.expr(1), Type.INT);
	}
	setType(ctx, Type.BOOL);
	setEntry(ctx, entry(ctx.expr(0)));
}
```

```
@Override
public void exitStringType(StringTypeContext ctx) {
	setType(ctx, Type.STRING);
}

@Override
public void exitIfExpr(IfExprContext ctx) {
	checkType(ctx.expr(0), Type.BOOL);
	setEntry(ctx, entry(ctx.expr(0)));
	if (ctx.expr(2) != null) {
		checkType(ctx.expr(1), getType(ctx.expr(2)));
	}
	setType(ctx, getType(ctx.expr(1)));
}

@Override
public void exitBlockExpr(BlockExprContext ctx) {
	ExprContext last = ctx.expr(ctx.expr().size() - 1);
	setType(ctx, getType(last));
	if (ctx.expr().size() > 0) {
		boolean set = false;
		int i =0;
		while(!set && i< ctx.expr().size()){
			if(entry(ctx.expr(i))!= null){
				setEntry(ctx, entry(ctx.expr(i)));
				set =true;
			}
			i++;
		}

	}
	sT.closeScope();
}

@Override
public void exitFalseExpr(FalseExprContext ctx) {
	setType(ctx, Type.BOOL);
	setEntry(ctx, ctx);
}

@Override
public void exitPrintExpr(PrintExprContext ctx) {
	for (ExprContext e : ctx.expr()) {
		checkNotType(e, Type.VOID);
	}
	if (ctx.expr().size() == 1) {
		setType(ctx, getType(ctx.expr(0)));
		setEntry(ctx, entry(ctx.expr(0)));
	} else {
		setEntry(ctx, entry(ctx.expr(0)));
		setType(ctx, Type.VOID);
```

```
		}
	}

	@Override
	public void exitCharType(CharTypeContext ctx) {
		setType(ctx, Type.CHAR);
	}

	@Override
	public void exitIdTarget(IdTargetContext ctx) {
		final String id = ctx.ID().getText();
		if (!sT.contains(id)) {
			addError(ctx, "Variable '%s' not declared", id);
			setEntry(ctx, ctx);
			setType(ctx, Type.VOID);
		} else {
			setType(ctx, sT.getType(id));
			setEntry(ctx, ctx);
			// setOffset(ctx, sT.offset(id));
		}
	}

	@Override
	public void exitCharExpr(CharExprContext ctx) {
		setType(ctx, Type.CHAR);
		setEntry(ctx, ctx);
	}

	@Override
	public void exitIntType(IntTypeContext ctx) {
		setType(ctx, Type.INT);
	}

	@Override
	public void exitReadExpr(ReadExprContext ctx) {
		Type[] tps = new Type[ctx.ID().size()];
		for (int i = 0; i < ctx.ID().size(); i++) {
			String targetString = ctx.ID(i).toString();
			if (!sT.contains(targetString)) {
				errors.add("Attempting to read for a undeclared variable '" +
ctx.ID().toString() + "' !");
			}
			tps[i] = sT.getType(ctx.ID(i).getText());
			if(tps[i].equals(Type.STRING)){
				errors.add("String reads are not supported, at : " + ctx.getText());
			}
		}

		result.setReadTypes(ctx, tps);
		if (ctx.ID().size() > 1) {
			setType(ctx, Type.VOID);
			setEntry(ctx, ctx);
```

```
			} else {
				setType(ctx, sT.getType(ctx.ID(0).getText()));
				setEntry(ctx, ctx);
			}

		}

		@Override
		public void exitMultExpr(MultExprContext ctx) {
			checkType(ctx.expr(0), Type.INT);
			checkType(ctx.expr(1), Type.INT);
			setType(ctx, Type.INT);
			setEntry(ctx, entry(ctx.expr(0)));
		}

		@Override
		public void exitNumExpr(NumExprContext ctx) {
			setType(ctx, Type.INT);
			setEntry(ctx, ctx);
		}

		@Override
		public void exitPlusExpr(PlusExprContext ctx) {
			if (getType(ctx.expr(0)).equals(Type.STRING)) {
				if (getType(ctx.expr(0)).equals(Type.STRING) ||
getType(ctx.expr(0)).equals(Type.CHAR)) {
					setType(ctx, Type.STRING);
				}else{
					addError(ctx, "Illegal type second argument, STRING or CHAR
expected.");
					setType(ctx, Type.VOID);
				}
			} else if (getType(ctx.expr(0)).equals(Type.INT)) {
				checkType(ctx.expr(1), Type.INT);
				setType(ctx, Type.INT);
			} else {
				addError(ctx, "Illegal type first argument, STRING or INT expected.");
				setType(ctx, Type.VOID);
			}
			setEntry(ctx, entry(ctx.expr(0)));
		}

		@Override
		public void exitDeclExpr(DeclExprContext ctx) {
			if (ctx.expr() != null) {
				checkType(ctx.expr(), getType(ctx.type()));
				setEntry(ctx, entry(ctx.expr()));
				if (!sT.add(ctx.ID().getText(), getType(ctx.type()))) {
					addError(ctx, "Illegal declaration, '%s' already in scope.",
ctx.ID().getText());
				} else {
```

```
                                setType(ctx.ID(), getType(ctx.type()));
                                setType(ctx, getType(ctx.type()));
                                return;
                        }
                } else {
                        if (!sT.add(ctx.ID().getText(), getType(ctx.type()))) {
                                addError(ctx, "Illegal declaration, '$s' already in scope.",
ctx.ID().getText());

                        }
                        // TODO entry?
                }
                setType(ctx, Type.VOID);
        }

        @Override
        public void exitWhileExpr(WhileExprContext ctx) {
                checkType(ctx.expr(0), Type.BOOL);
                setEntry(ctx, entry(ctx.expr(0)));
                setType(ctx, Type.VOID);
        }

        @Override
        public void exitAssExpr(AssExprContext ctx) {
                checkType(ctx.expr(), getType(ctx.target()));
                setType(ctx, getType(ctx.target()));
                setEntry(ctx, entry(ctx.expr()));
        }

        @Override
        public void exitPrfExpr(PrfExprContext ctx) {
                Type type;
                if (ctx.prfOp().MINUS() != null) {
                        type = Type.INT;
                } else {
                        type = Type.BOOL;
                }
                checkType(ctx.expr(), type);
                setType(ctx, type);
                setEntry(ctx, entry(ctx.expr()));
        }

        @Override
        public void exitBoolExpr(BoolExprContext ctx) {
                checkType(ctx.expr(0), Type.BOOL);
                checkType(ctx.expr(1), Type.BOOL);
                setType(ctx, Type.BOOL);
                setEntry(ctx, entry(ctx.expr(0)));
        }

        @Override
        public void exitIdExpr(IdExprContext ctx) {
```

```
                String id = ctx.ID().getText();
                Type type = this.sT.getType(id);
                if (type == null) {
                        addError(ctx, "Variable '%s' not declared", id);
                        setType(ctx, Type.VOID);
                } else {
                        setType(ctx, type);
                        // setOffset(ctx, this.scope.offset(id));
                        setEntry(ctx, ctx);
                }
        }

}
```

Antlr visitor:

```
package generator;

import org.antlr.v4.runtime.ParserRuleContext;
import org.antlr.v4.runtime.Token;
import org.antlr.v4.runtime.tree.ParseTree;
import org.antlr.v4.runtime.tree.ParseTreeProperty;

import iloc.eval.Machine;
import iloc.model.*;
import iloc.Simulator;
import checker.Result;
import checker.Type;
import grammar.TempNameBaseVisitor;
import grammar.TempNameParser.*;

public class Generator extends TempNameBaseVisitor<String> {
        /** The representation of the boolean value <code>false</code>. */
        public final static Num FALSE_VALUE = new Num(Simulator.FALSE);
        /** The representation of the boolean value <code>true</code>. */
        public final static Num TRUE_VALUE = new Num(Simulator.TRUE);

        /** The base register. */
        private Reg arp = new Reg("r_arp");
        /** The outcome of the checker phase. */
        private Result checkResult;
        /** Association of statement nodes to labels. */
        private ParseTreeProperty<Label> labels;
        /** The program being built. */
        private Program prog;
        /** The memory manager of this generator */
        private MemoryManager mM;

        /**
         * Generates ILOC code for a given parse tree, given a pre-computed checker
         * result.
         */
        public Program generate(ParseTree tree, Result checkResult) {
```

```
		this.prog = new Program();
		this.checkResult = checkResult;
		this.labels = new ParseTreeProperty<>();
		this.mM = new MemoryManager();
		tree.accept(this);
		return this.prog;
	}

	/**
	 * Helpfunction for TypedStore stores a string
	 */
	private void storeString(ParseTree from, ParseTree to, boolean closeScope, String fromid,
String toid) {
		// SETUP
		int[] stringData = mM.getSizeAndOffset(from, fromid);
		if (closeScope) {
			mM.closeScope();
		}
		int parentoff = mM.getOffset(to, stringData[0], toid);
		Reg helpreg = new Reg(mM.getConstReg());

		// MOVE ALL CHARS
		for (int i = 0; i < stringData[0]; i = i + Machine.DEFAULT_CHAR_SIZE) {
			emit(OpCode.cloadAI, arp, new Num(i + stringData[1]), helpreg);
			emit(OpCode.cstoreAI, helpreg, arp, new Num(i + parentoff));
		}
	}

	/**
	 * Store the result of a child ctx as the result of the parent ctx (another
	 * ctx). IF an id is know it will store the value of the id as the result of
	 * the parent.
	 *
	 * Closes the scope if necesary
	 *
	 * @requires from != null
	 * @requires to != null
	 */
	private void typedStore(ParseTree from, ParseTree to, boolean closeScope, String id) {
		if (id != null) {
			// hoeft volgens mij niet
			System.out.println("yup, het moest dus wel");
		} else if (mM.hasMemory(from)) {
			if (checkResult.getType(from).equals(Type.CHAR)) {
				emit(OpCode.cloadAI, arp, offset(from), reg(to));
				if (closeScope)
					mM.closeScope();
				emit(OpCode.cstoreAI, reg(to), arp, offset(to));
			} else if (checkResult.getType(from).equals(Type.STRING)) {
				this.storeString(from, to, closeScope, null, id);
			} else {
				emit(OpCode.loadAI, arp, offset(from), reg(to));
```

```
                        if (closeScope)
                                mM.closeScope();
                        emit(OpCode.storeAI, reg(to), arp, offset(to));
                }
        } else {
                if (checkResult.getType(from).equals(Type.CHAR)) {
                        if (closeScope)
                                mM.closeScope();
                        emit(OpCode.cstoreAI, reg(from), arp, offset(to));
                } else if (checkResult.getType(from).equals(Type.STRING)) {
                        System.out.println("this state should not happen");
                } else {
                        mM.closeScope();
                        emit(OpCode.storeAI, reg(from), arp, offset(to));
                }
        }
}

/**
 * Loads an id from memory for use by the given node
 *
 * @requires id != null
 * @requires ctx != null
 */
private void typedLoad(ParseTree ctx, String id) {
        Type type = checkResult.getType(ctx);
        if (type.equals(Type.CHAR)) {
                emit(OpCode.cloadAI, arp, offset(ctx, id), reg(ctx));
        } else if (type.equals(Type.STRING)) {
                // Strings cant be loaded into a single register
        } else {
                emit(OpCode.loadAI, arp, offset(ctx, id), reg(ctx));
        }
}

/**
 * Improved visit method. Visits the node and if the result of the node was
 * stored to an id loads that value into the register of the node.
 *
 * @requires ctx != null
 * @return
 */
private String visitH(ParseTree ctx) {
        String id0 = visit(ctx);
        if (id0 != null && mM.hasMemory(ctx)) {
                typedLoad(ctx, id0);
        }
        return id0;
}

/**
 * Generates the necesary iloc code to return the result of one node and
```

```
 * load it as input into the other node. If an id is known it will use that
 * instead of the node.
 *
 * @requires from != null
 * @requires to != null
 */
private void returnResult(ParseTree from, ParseTree to, String fromid, String toid) {
	Type type = checkResult.getType(from);
	if (mM.hasReg(from) || fromid != null || mM.hasMemory(from)) {
		if (type.equals(Type.CHAR)) {
			emit(OpCode.cstoreAI, reg(from), arp, offset(to, toid));
		} else if (type.equals(Type.STRING)) {
			storeString(from, to, false, fromid, toid);
		} else {
			emit(OpCode.storeAI, reg(from), arp, offset(to, toid));
		}
	} else {
		// No return needed
	}
}

/**
 * Constructs an operation from the parameters and adds it to the program
 * under construction.
 */
private Op emit(Label label, OpCode opCode, Operand... args) {
	Op result = new Op(label, opCode, args);
	this.prog.addInstr(result);
	return result;
}

/**
 * Constructs an operation from the parameters and adds it to the program
 * under construction.
 */
private Op emit(OpCode opCode, Operand... args) {
	return emit((Label) null, opCode, args);
}

/**
 * Looks up the label for a given parse tree node, creating it if none has
 * been created before. The label is actually constructed from the entry
 * node in the flow graph, as stored in the checker result.
 */
private Label label(ParserRuleContext node) {
	Label result = this.labels.get(node);
	if (result == null) {
		ParserRuleContext entry = this.checkResult.getEntry(node);
		result = createLabel(entry, "n");
		this.labels.put(node, result);
	}
	return result;
```

```
	}

	/** Creates a label for a given parse tree node and prefix. */
	private Label createLabel(ParserRuleContext node, String prefix) {
		Token token = node.getStart();
		int line = token.getLine();
		int column = token.getCharPositionInLine();
		String result = prefix + "_" + line + "_" + column;
		return new Label(result);
	}

	/**
	 * If the id is not null it will look up the memory offset of this id, or
	 * create a new one if it doesnt exist.
	 *
	 * If the id is null it will look up the memory offset of the return value
	 * of the node, or create a new one if it doesnt exist.
	 *
	 * @requires node != null
	 * @ensures result>0
	 */
	private Num offset(ParseTree node, String id) {
		int size = 0;
		if (checkResult.getType(node).equals(Type.INT) ||
checkResult.getType(node).equals(Type.BOOL)) {
			size = Machine.INT_SIZE;
		} else if (checkResult.getType(node).equals(Type.CHAR)) {
			size = Machine.DEFAULT_CHAR_SIZE;
		}
		Num offset = new Num(mM.getOffset(node, size, id));
		return offset;
	}

	/**
	 * @see offset(node, null);
	 */
	private Num offset(ParseTree node) {
		return offset(node, null);
	}

	/**
	 * Gives the reg that belongs to this node, reserves a new reg if the node
	 * does not have a reg.
	 *
	 * @requires node != null
	 * @ensure result != null
	 */
	private Reg reg(ParseTree node) {
		Reg reg;
		// IF the value is ONLY stored in memory, it should be loaded into the
		// register before use
```

```
		if (mM.hasMemory(node) && !mM.hasReg(node)) {
			reg = new Reg(mM.getNodeReg(node));
			Type type = checkResult.getType(node);
			if (type.equals(Type.CHAR)) {
				emit(OpCode.cloadAI, arp, offset(node), reg);
			} else if (type.equals(Type.STRING)) {
				// Cannot preload for strings

			} else {
				emit(OpCode.loadAI, arp, offset(node), reg);
			}

		} else {
			reg = new Reg(mM.getNodeReg(node));
		}

		return reg;
	}

	// -----------Program-----------

	@Override
	public String visitProgram(ProgramContext ctx) {
		emit(new Label("Program"), OpCode.nop);
		mM.openScope();
		visit(ctx.expr());
		mM.closeScope();
		return null;
	}

	// -----------Expression-----------

	@Override
	public String visitParExpr(ParExprContext ctx) {
		visitH(ctx.expr());
		emit(OpCode.i2i, reg(ctx.expr()), reg(ctx));
		return null;
	}

	@Override
	public String visitCompExpr(CompExprContext ctx) {
		String id1 = visitH(ctx.expr(0));
		String id2 = visitH(ctx.expr(1));
		if (id1 != null) {
			typedLoad(ctx.expr(0), id1);
		}
		if (id2 != null) {
			typedLoad(ctx.expr(1), id2);
		}
		String compOp = ctx.compOp().getText();
		switch (compOp) {
		case ("<="):
```

```
                emit(OpCode.cmp_LE, reg(ctx.expr(0)), reg(ctx.expr(1)), reg(ctx));
                break;
        case ("<"):
                emit(OpCode.cmp_LT, reg(ctx.expr(0)), reg(ctx.expr(1)), reg(ctx));
                break;
        case ("<>"):
                emit(OpCode.cmp_NE, reg(ctx.expr(0)), reg(ctx.expr(1)), reg(ctx));
                break;
        case (">="):
                emit(OpCode.cmp_GE, reg(ctx.expr(0)), reg(ctx.expr(1)), reg(ctx));
                break;
        case (">"):
                emit(OpCode.cmp_GT, reg(ctx.expr(0)), reg(ctx.expr(1)), reg(ctx));
                break;
        case ("=="):
                emit(OpCode.cmp_EQ, reg(ctx.expr(0)), reg(ctx.expr(1)), reg(ctx));
                break;
        }

        return null;
}

@Override
public String visitIfExpr(IfExprContext ctx) {
        visitH(ctx.expr(0));
        Label endIf = createLabel(ctx, "endIf");

        boolean elseExists = ctx.expr(2) != null;
        Label elsez = elseExists ? label(ctx.expr(2)) : endIf;
        emit(OpCode.cbr, reg(ctx.expr(0)), label(ctx.expr(1)), elsez);
        emit(label(ctx.expr(1)), OpCode.nop);
        String id = visitH(ctx.expr(1));
        if (id != null) {
                typedLoad(ctx.expr(1), id);
        }
        returnResult(ctx.expr(1), ctx, id, id);
        emit(OpCode.jumpI, endIf);
        if (elseExists) {
                emit(elsez, OpCode.nop);
                id = visitH(ctx.expr(2));
                if (id != null) {
                        typedLoad(ctx.expr(1), id);
                }
                returnResult(ctx.expr(2), ctx, id, id);
        }
        emit(endIf, OpCode.nop);
        return null;
}

@Override
public String visitBlockExpr(BlockExprContext ctx) {
        int last = ctx.expr().size() - 1;
```

```
                mM.openScope();
                for (int i = 0; i < ctx.expr().size() - 1; i++) {
                        visitH(ctx.expr(i));
                }
                String id = visitH(ctx.expr(last));
                if (id != null) {
                        typedLoad(ctx.expr(last), id);
                }
                typedStore(ctx.expr(last), ctx, true, id);
                return null;
        }

        @Override
        public String visitPrintExpr(PrintExprContext ctx) {
                Type type;
                if (ctx.expr().size() > 1) {
                        for (int i = 0; i < ctx.expr().size(); i++) {
                                String id = visitH(ctx.expr(i));

                                if (id != null) {
                                        typedLoad(ctx.expr(i), id);
                                }
                                type = checkResult.getType(ctx.expr(i));
                                if (type.equals(Type.CHAR)) {
                                        emit(OpCode.loadI, new Num(1), reg(ctx));
                                        emit(OpCode.cpush, reg(ctx.expr(i)));
                                        emit(OpCode.push, reg(ctx));
                                        emit(OpCode.cout, new Str(ctx.expr(i).getText() + ": "));
                                } else if (type.equals(Type.STRING)) {
                                        int[] stringData = mM.getSizeAndOffset(ctx.expr(i), id);
                                        // Push chars
                                        for (int j = stringData[0]; j > 0; j--) {
                                                emit(OpCode.cloadAI, arp, new Num(stringData[1] + j
* Machine.DEFAULT_CHAR_SIZE - 1), reg(ctx));

                                                emit(OpCode.cpush, reg(ctx));
                                        }
                                        // Push size
                                        emit(OpCode.loadI, new Num(stringData[0]), reg(ctx));
                                        emit(OpCode.push, reg(ctx));

                                        emit(OpCode.cout, new Str(ctx.expr(i).getText() + ": "));
                                } else {
                                        emit(OpCode.out, new Str(ctx.expr(i).getText() + ": "),
reg(ctx.expr(i)));
                                        storeString(ctx.expr(0), ctx, false, id, null);
                                }
                        }
                } else {
                        String id = visitH(ctx.expr(0));
                        if (id != null) {
                                typedLoad(ctx.expr(0), id);
```

```
                        }
                        type = checkResult.getType(ctx.expr(0));
                        if (type.equals(Type.CHAR)) {
                                emit(OpCode.loadI, new Num(1), reg(ctx));
                                emit(OpCode.cpush, reg(ctx.expr(0)));
                                emit(OpCode.push, reg(ctx));
                                emit(OpCode.cout, new Str(ctx.expr(0).getText() + ": "));
                                returnResult(ctx.expr(0), ctx, null, null);
                        } else if (type.equals(Type.STRING)) {
                                int[] stringData = mM.getSizeAndOffset(ctx.expr(0), id);
                                // Push chars
                                for (int j = stringData[0]; j > 0; j--) {
                                        emit(OpCode.cloadAI, arp, new Num(stringData[1] + j *
Machine.DEFAULT_CHAR_SIZE - 1), reg(ctx));

                                        emit(OpCode.cpush, reg(ctx));
                                }
                                // Push size
                                emit(OpCode.loadI, new Num(stringData[0]), reg(ctx));
                                emit(OpCode.push, reg(ctx));

                                emit(OpCode.cout, new Str(ctx.expr(0).getText() + ": "));

                                storeString(ctx.expr(0), ctx, false, id, null);
                        } else {
                                emit(OpCode.out, new Str(ctx.expr(0).getText() + ": "),
reg(ctx.expr(0)));
                                returnResult(ctx.expr(0), ctx, null, null);
                        }
                }
                return null;
        }

        @Override
        public String visitReadExpr(ReadExprContext ctx) {
                Type[] types = checkResult.getReadTypes(ctx);
                for (int i = 0; i < ctx.ID().size(); i++) {
                        if (types[i].equals(Type.CHAR)) {
                                emit(OpCode.cin, new Str(ctx.ID(i).getText() + "? : "));
                                emit(OpCode.pop, reg(ctx));
                                emit(OpCode.cpop, reg(ctx));
                                emit(OpCode.cstoreAI, reg(ctx), arp, offset(ctx, ctx.ID(i).getText()));
                        } else if (types[i].equals(Type.STRING)) {
                                // Not supported by our memory/registry manager, too much
                                // work to edit it in properly
                        } else {
                                emit(OpCode.in, new Str(ctx.ID(i).getText() + "? : "), reg(ctx));
                                emit(OpCode.storeAI, reg(ctx), arp, offset(ctx, ctx.ID(i).getText()));
                        }
                }

                if (ctx.ID().size() == 1) {
```

```
                return ctx.ID(0).getText();
        }
        return null;
}

@Override
public String visitMultExpr(MultExprContext ctx) {
        String id1 = visitH(ctx.expr(0));
        String id2 = visitH(ctx.expr(1));
        if (id1 != null) {
                typedLoad(ctx.expr(0), id1);
        }
        if (id2 != null) {
                typedLoad(ctx.expr(1), id2);
        }
        if (ctx.multOp().getText().equals("*")) {
                // Times
                emit(OpCode.mult, reg(ctx.expr(0)), reg(ctx.expr(1)), reg(ctx));
        } else if (ctx.multOp().getText().equals("/")) {
                // Division
                emit(OpCode.div, reg(ctx.expr(0)), reg(ctx.expr(1)), reg(ctx));
        } else {
                // Modulo
                String str1 = mM.getConstReg();
                Reg r1 = new Reg(str1);
                emit(OpCode.div, reg(ctx.expr(0)), reg(ctx.expr(1)), r1);
                emit(OpCode.mult, r1, reg(ctx.expr(1)), r1);
                emit(OpCode.sub, reg(ctx.expr(0)), r1, reg(ctx));
        }
        return null;
}

@Override
public String visitPlusExpr(PlusExprContext ctx) {
        String id1 = visitH(ctx.expr(0));
        String id2 = visitH(ctx.expr(1));
        if (id1 != null) {
                typedLoad(ctx.expr(0), id1);
        }
        if (id2 != null) {
                typedLoad(ctx.expr(1), id2);
        }
        if (checkResult.getType(ctx).equals(Type.INT)) {
                if (ctx.plusOp().getText().equals("+")) {
                        emit(OpCode.add, reg(ctx.expr(0)), reg(ctx.expr(1)), reg(ctx));
                } else {
                        emit(OpCode.sub, reg(ctx.expr(0)), reg(ctx.expr(1)), reg(ctx));
                }
        } else {
                // concat string+string or string+char
                int[] stringData1 = mM.getSizeAndOffset(ctx.expr(0), id1);
                int[] stringData2 = mM.getSizeAndOffset(ctx.expr(1), id2);
```

```
                        int offset = mM.getOffset(ctx, stringData1[0] + stringData2[0], null);
                        for (int i = 0; i < stringData1[0]; i += Machine.DEFAULT_CHAR_SIZE, offset +=
Machine.DEFAULT_CHAR_SIZE) {
                                emit(OpCode.cloadAI, arp, new Num(stringData1[1] + i), reg(ctx));
                                emit(OpCode.cstoreAI, reg(ctx), arp, new Num(offset));
                        }
                        for (int i = 0; i < stringData2[0]; i += Machine.DEFAULT_CHAR_SIZE, offset +=
Machine.DEFAULT_CHAR_SIZE) {
                                emit(OpCode.cloadAI, arp, new Num(stringData2[1] + i), reg(ctx));
                                emit(OpCode.cstoreAI, reg(ctx), arp, new Num(offset));
                        }
                }
                return null;
        }

        @Override
        public String visitPrfExpr(PrfExprContext ctx) {
                String id = visitH(ctx.expr());
                if (id != null) {
                        typedLoad(ctx.expr(), id);
                }

                if (ctx.prfOp().getText().equals("-")) {
                        emit(OpCode.rsubI, reg(ctx.expr()), new Num(0), reg(ctx));
                } else {
                        emit(OpCode.addI, reg(ctx.expr()), new Num(1), reg(ctx));
                        emit(OpCode.rsubI, reg(ctx), new Num(0), reg(ctx));
                }
                return null;
        }

        @Override
        public String visitDeclExpr(DeclExprContext ctx) {
                if (ctx.expr() != null) {
                        visitH(ctx.expr());
                        Type type = checkResult.getType(ctx);
                        if (type.equals(Type.CHAR)) {
                                emit(OpCode.cstoreAI, reg(ctx.expr()), arp, offset(ctx.ID(),
ctx.ID().getText()));
                        } else if (type.equals(Type.STRING)) {
                                storeString(ctx.expr(), ctx, false, null, ctx.ID().getText());
                        } else {
                                emit(OpCode.storeAI, reg(ctx.expr()), arp, offset(ctx.ID(),
ctx.ID().getText()));
                        }

                }
                return ctx.ID().getText();
        }

        @Override
        public String visitWhileExpr(WhileExprContext ctx) {
```

```
                emit(label(ctx), OpCode.nop);
                visitH(ctx.expr(0));
                Label endLabel = createLabel(ctx, "end");
                emit(OpCode.cbr, reg(ctx.expr(0)), label(ctx.expr(1)), endLabel);
                emit(label(ctx.expr(1)), OpCode.nop);
                visitH(ctx.expr(1));
                emit(OpCode.jumpI, label(ctx));
                emit(endLabel, OpCode.nop);
                return null;
        }

        @Override
        public String visitAssExpr(AssExprContext ctx) {
                String id = visitH(ctx.expr());
                if (id != null) {
                        typedLoad(ctx.expr(), id);
                }
                if (checkResult.getType(ctx).equals(Type.CHAR)) {
                        emit(OpCode.cstoreAI, reg(ctx.expr()), this.arp, offset(ctx.target(),
ctx.target().getText()));
                } else if (checkResult.getType(ctx).equals(Type.STRING)) {
                        storeString(ctx.expr(), ctx, false, id, ctx.target().getText());
                } else {
                        emit(OpCode.storeAI, reg(ctx.expr()), this.arp, offset(ctx.target(),
ctx.target().getText()));
                }
                return ctx.target().getText();
        }

        @Override
        public String visitBoolExpr(BoolExprContext ctx) {
                String id1 = visitH(ctx.expr(0));
                String id2 = visitH(ctx.expr(1));
                if (id1 != null) {
                        typedLoad(ctx.expr(0), id1);
                }
                if (id2 != null) {
                        typedLoad(ctx.expr(1), id2);
                }
                if (ctx.boolOp().getText().contains("o") || ctx.boolOp().getText().contains("O")) {
                        emit(OpCode.or, reg(ctx.expr(0)), reg(ctx.expr(1)), reg(ctx));
                } else {
                        emit(OpCode.and, reg(ctx.expr(0)), reg(ctx.expr(1)), reg(ctx));

                }
                return null;
        }

        // -----------Terminal expressions-----------

        @Override
        public String visitIdExpr(IdExprContext ctx) {
```

```
            return ctx.ID().getText();
        }

        @Override
        public String visitNumExpr(NumExprContext ctx) {
            emit(OpCode.loadI, new Num(Integer.parseInt(ctx.NUM().getText())), reg(ctx));
            return null;
        }

        @Override
        public String visitCharExpr(CharExprContext ctx) {
            int chara = (int) ctx.CHR().getText().charAt(1);
            emit(OpCode.loadI, new Num(chara), reg(ctx));
            emit(OpCode.i2c, reg(ctx), reg(ctx));
            return null;
        }

        @Override
        public String visitStringExpr(StringExprContext ctx) {
            String str = ctx.STR().getText();
            str = str.substring(1, str.length() - 1);
            int offset = mM.getOffset(ctx, Machine.DEFAULT_CHAR_SIZE * str.length(), null);
            for (int i = 0; i < str.length(); i++) {
                int chara = (int) str.charAt(i);
                emit(OpCode.loadI, new Num(chara), reg(ctx));
                emit(OpCode.i2c, reg(ctx), reg(ctx));
                emit(OpCode.cstoreAI, reg(ctx), arp, new Num(offset + i *
Machine.DEFAULT_CHAR_SIZE));
            }
            return null;
        }

        @Override
        public String visitTrueExpr(TrueExprContext ctx) {
            emit(OpCode.loadI, TRUE_VALUE, reg(ctx));
            return null;
        }

        @Override
        public String visitFalseExpr(FalseExprContext ctx) {
            emit(OpCode.loadI, FALSE_VALUE, reg(ctx));
            return null;
        }
}
```