A Compiler for C-imple

Documentation

Zagkas Dimosthenis 4359 cse84359

Andreou Aggelos 4628 cse84628

CONTENTS

INTRO

The main objective of this project is to create a compiler for the programming language C-imple in Python. This documentation follows the order of the implementation and the logic behind the steps taken to completion.

We'll start with a quick showcase of the language C-imple, its quirks and features. We'll move on to analyzing the way we split the implementation into distinct parts and the reasons behind that. For each part, we'll dive deeper into the architecture of the compiler and all the methods we used.

Lastly, we plan on showing some features that we missed or generally failed to implement properly as well as some bugs in the code.

INTRODUCTION TO C-IMPLE

C-IMPLE is a fully functioning programming language, provided to us by the course professor for educational purposes. The structure is similar to the language C from where it borrows basic concepts but of course, it is simplified. Let's proceed to the basics of C-imple.

- The alphabet

C-imple accepts lowercase and uppercase letters of the Latin alphabet (A,…,Z and a,…z).
Numbers, arithmetic symbols and relational operator(0,…,9 ,+ ,- ,* ,/ and <, >, =, <=, >=, <>).
Acceptable is the symbol of insertion (:=) which we'll explain later how it works. Symbols of grouping ([,], (,) , { , }), delimiters (;, ",", :), the full stop symbol (.) as well as the comment symbol (#) are also acceptable.

The keywords of c-imple are the following ('program', 'declare', 'if', 'else', 'while', 'switchcase', 'forcase', 'incase', 'case', 'default', 'not', 'and', 'or', 'function', 'procedure', 'call', 'return', 'in', 'inout', 'input', 'print').

We'll explain the function of each of them in detail.

C-imple only accepts integers with a range of $-(2^{32}-1)$ to $(2^{32}-1)$.

Variables can have a maximum length of 30 characters, starting with a letter and made up from letters and numbers. In case of a variable with a length of over 30 characters, the compiler will throw an error.

White space characters such as tab, space and return are ignored. Same goes for comments.

- Program Layout

Here is an example of how a basic program is constructed

program id
{
declarations
subprograms
statements
}.

Each program begins with the keyword *program* followed by the name of it (id). Inside of the curly brackets the three basic blocks of the program exist.

Declarations: The variable declarations. Those declarations are for the variables of the main program. The subprogram block (i.e., different functions in the main program) can have their own declarations.

Subprograms: Functions inside the main program. They can have their own declarations, parameters, and statements (i.e., commands)

Statements: The commands

- Declarations

C-imple only accepts integers as variables which they must be declared.
Example:

program nameOfProgram
{
declare x;
declare y, z;
subprograms
statements
}.

Notice that at the end of the declaration we use the semicolon, like C.
We'll explain later the intricacies of the semi column in C-imple.

- Operator Priority

  - Multiplication (*, /)
  - Addition (+, -)
  - Relational (=, <, >, <>, <=, >=)
  - Logical (not, and, or, in this exact order)

We'll move on with the explanation of the keywords

- Insertion

Example

```
x := 1;
x := function();
```

Used to assign a value to a variable. This can happen either directly, by assigning an integer to a variable or indirectly, by assigning a function, which returns an integer, to a variable.

- If condition

Example

```
if (condition){
        statements
        }
else{
        statements
    }
```

The else part is optional.

- While loop

Example

while (condition)
statements

This first time the condition is false, the while loop stops

- Switchcase

Example

switchcase
case (condition) statements
case (condition) statements
…
default statements

The statements that follow the first true condition are executed. After that, the compiler continues with the program, ignoring the rest of the switchcase. Incase none of the cases are true, the default statements get executed.

- Forcase

Example

forcase
case (condition) statements
case (condition) statements
…
default statements

Same as switchcase but has a loop after each case. When the default statements get executed, the code continues.

- Incase

Example

incase
case (condition) statements
case (condition) statements
…

For each case, if the condition is true, the statements get executed. At the end of the incase, if at least one of the conditions has been true, the code moves up to the beginning of the incase, re-executing it. Otherwise, the code continues.

- Function Return

Example

function(){
statements
…
return (expression);
}

Used inside function to return a value. We'll explain the role of the expression later.

- Output

Example

print (expression)

Prints at the screen (standard out)

- Input

Example

Input (ID)

Standard input (keyboard). The value will be stored in the variable ID

- Procedure Call

Example

Call functionName (actualParameters)

We'll explain later the actualParameters

- Functions and procedures

Example

```
function ID (formalPars)
{
    declarations
    subprograms
    statements
}

Procedure ID (formalPars)
{
    declarations
    subprograms
    statements
}
```

We'll explain the formal parameters later.

- Parameters

C-imple has two ways to pass parameters into functions and procedures.

With value

Example

function f1 (in x)

Changes to the variable inside the function do not get transmitted to the program that called her.

With reference

Example

function f1 (inout x)

Changes to the variable inside the function do get transmitted to the program that called her.

- Global and local variables

This part is the same as in C. Global variables are accessed by all other subprograms. Local variables are accessible only by the function that created them and by the children of that function.

In case of two variables with the same, one global and one local, the local one gets used.

- Semi column

In C-imple, each statement must end with a semi column. Exception is when that statement is the last one in a block.

Example

```
program example{
    declare x;
    x := 1;
    print (x)
}.
```

Another situation

Example

```
program example {
while (condition){
    if (condition){
        print (x);;
        }
    }
}.
```

Notice that the print command requires two semi columns. That is because there three statements finish, the while, the if and the print. The while command though is the last one the in block of the main program, thus it doesn't require a semi column.

Layout of the architecture

A brief description of the compiler's parts and their work.

- Lexical Analyzer

  Responsible for breaking the input C-imple file into "tokens" which will be passed to the syntax analyzer.
  A token could be a keyword, a variable or even a symbol such as a semi column.

- Syntax Analyzer

  Responsible for checking that the input C-imple file follows the correct syntax. It takes the tokens from the lexical analyzer and determines if they abide by the C-imple rules.

- Symbol Table

  Responsible for storing the data of variables, functions and their parameters in such a way that follows the C-imple rules. The symbol table is used to determine which functions have access to which variables. It is necessary for the production of the final code in assembly.

- Intermediate Code Generator

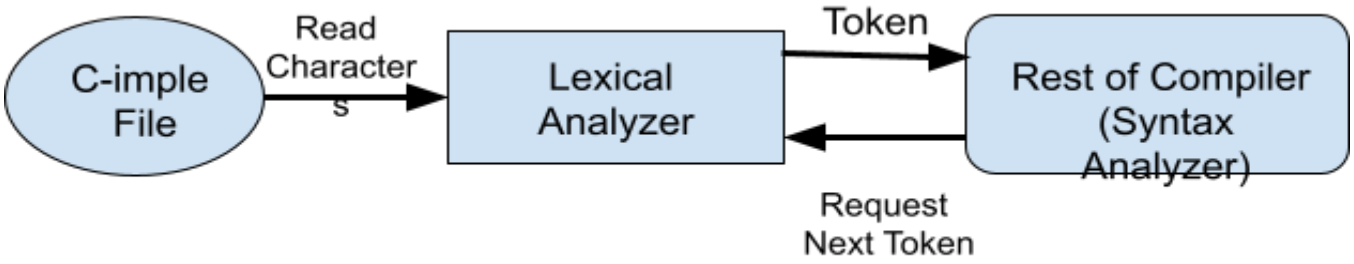
  Responsible for producing the intermediate code which consists of quads. Basically, in the intermediate code, we translate every command in C-imple (and not only commands, but other information too), into quads. Quads are very similar to assembly and with them, we produce the final code.

- Final Code Generator

Responsible for producing the final code in assembly, using the quads from the intermediate code.

Lexical Analyzer

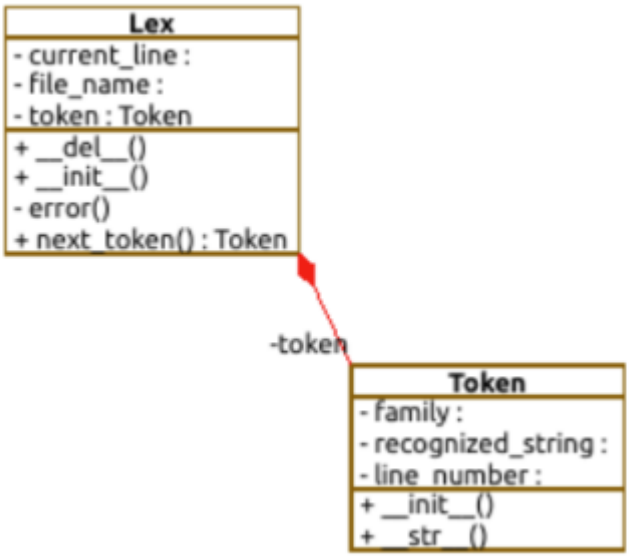
The very first phase of the compiler's creation is the Lexical Analyzer. The Lexical Analyzer takes as an input a C-imple file which contains a program written in C-imple language and reads every character in order to create lexical tokens. At our compiler it is a function called by the next , the syntax Analyzer and returns the next token with a corresponding number. Finally the lexical analyzer recognizes and catches errors at the source text.



The lexical token, will call (as) just token from now on, is a group of characters and is created by reading every character one by one from the input file excluding comments, blanks, tab and new line characters. In our compiler it is an object with 3 parameters: family, recognized string, line number. Grouped characters are saved as a string. We also save the file's line number where the string is recognized. The family is the category that the recognized string belongs to that makes syntax sense. Family categories are the following:

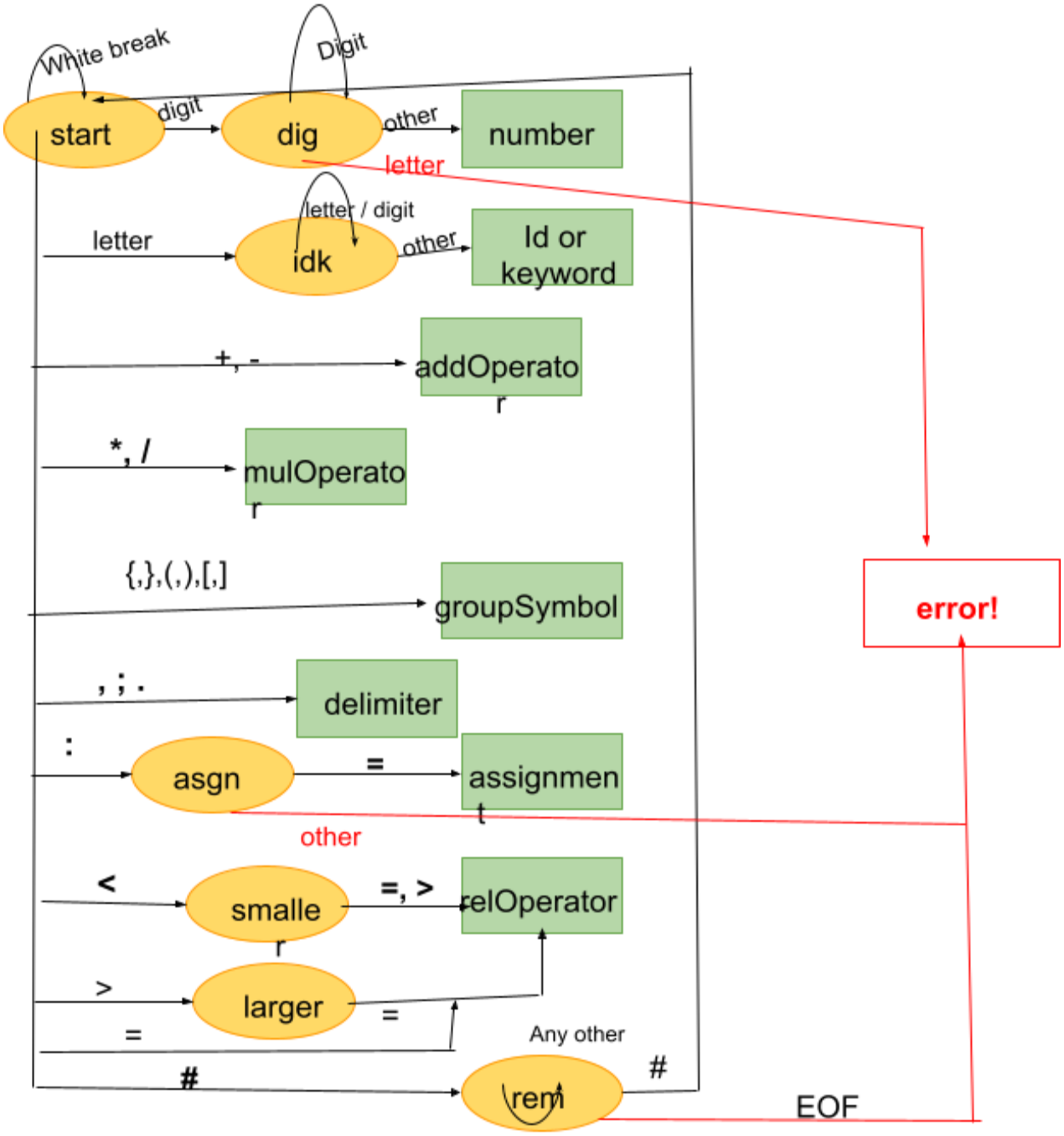
- identifier (ID): a general category to group strings we can't know their function yet, mostly the names of variables, functions, procedures, constants and everything else we need to save the name of.
- keyword: C-imple reserved words, as seen in the previous chapter.
- number: numeric constants
- addOperator: The characters +,-
- mulOperator: The characters *,/
- relOperator: The strings ==, >=, <, <>
- assignment: :=

- delimiter: The characters , . ;
- groupSymbol: `(,),{,},[,]`
- End of file: To recognise when the file scanning is finished.



The lexical analyzer's functions as a finite state machine. When it lands on an accepting state, it saves the token's parameters and saves the current position scanning ready to continue when asked for the next token. When the lexical analyzer lands on a non-acceptable state it stops and returns the invalid character and line of the error with an error message.

We used the following design of the finite state machine to generate the needed tokens and to stop the compiling of the program in case of an error.


White break
Digit
start
digit
dig
other
number
letter
letter / digit
letter
idk
other
Id or keyword
+, -
addOperato
r
*, /
mulOperato
r
{,},(,),[,]
groupSymbol
error!
, ; .
delimiter
:
asgn
=
assignmen
t
other
<
smalle
r
=, >
elOperator
>
larger
=
=
Any other
#
#
rem
EOF

SYNTAX ANALYZER

In the syntax analyzer we need to take each token from the lexical analyzer, basically every “word” of the C-imple program and check if it follows all the grammatical rules of the language.

We’ll proceed by analyzing all the different decisions of the syntax analyzer which are based on C-imple’s grammar.

Every subtitle we show is the token that the lexical analyzer created and the syntax analyzer has to check.

For example:

- Program

  The starting symbol of each program. Followed by its name and a block. Every program ends with a full stop (.)

  ```
  program ->  program ID
                  block
                  .
  ```

  Explanation: The syntax analyzer takes the token “program”. After that it expects the program name (ID), then the block code and finally, it expects to see the full stop symbol.

- Block

  A block consists of declarations, subprograms, and statements

  ```
  block -> {
                  declarations
                  subprograms
                  blockstatements
          }
  ```

Explanation: It takes “block”. It expects the see open curly bracket, declarations (if any), subprograms (if any) and the statements for that block. In the end, it expects to finish with a close curly bracket.

- Declarations

Declaration of variables. Kleene star implies zero or more “declare” statements

declarations -> (declare varlist ;)*

Explanation: It takes “declare”. It expects to see:

- One variable and then semi column
- More than one variable, separated by commas (varlist, we’ll see it later), and then a semi column
- More than one declares

- Varlist

A list of variables following the declaration keyword

varlist -> ID
(, ID)*
| ε

Explanation: After the keyword “declare” the compiler to expects to see one or more variables separated by commas.

- Subprograms

Zero or more subprograms

subprograms -> (subprograms)*

- Subprogram

A subprogram is a function of a procedure, followed by parameters and a block

Subprogram -> function ID (formalparlist)
block
| procedure ID (formalparlist)
block

Explanation: Inside a block, after the curly brackets and after the declarations, if they exist, subprogram can begin. A subprogram is stated either as a function or a procedure. It begins with the corresponding keyword (function/procedure), followed by the name.
Opening brackets are expected for the parameters. After the closing brackets, the block of this subprogram begins, with opening curly brackets and so on.

- Formal parameters list

List of formal parameters. One or more parameters are allowed

formalparlist -> formalparitem
(, formalparitem)*
| ε

- Formal parameter item

A formal parameter. “in”: by value, “inout”: by reference

formalparitem -> in ID
| inout ID

Explanation: For each parameter in a subprogram, the formalparitem is called. It checks if before the id of the parameter the proper mode is depicted, in or inout.

- Statemets

One or more statements. More than one statements should be grouped with curly brackets

statements -> statement;
| {
statement
(; statement)*
}

Explanation: The actual code. Commands ending with semi column (not necessary for the last one in a block).

- Block Statements

Statements considered as block (used in program and subprogram)

blockstatements -> statement
(; statement) *

- Statement

One statement. The actual commands in the code.

statement -> assignStat
| ifStat
| whileSta
| switchcaseStat

| forcaseStat
| incaseStat
| callStat
| returnStat
| inputStat
| printStat
| ε

- Assign Statement

assignStat -> ID := expression

- If statement

ifStat -> if (condition)
statements
elsepart

- Else Part

elsepart -> else
statements
| ε

- While Statement

whileStat -> while (condition)
statements

- Switch Statement

switchStat -> switchcase
(case (condition) statements)*
default statements

- Forcase Statement

forcaseStat -> forcase
(case (condition) statements)*
default statements

- Incase Statement

incaseStat -> incase
(case (condition) statements) *

- Return Statement

returnStat -> return (expression)

- Call Statement

callStat -> call ID (actualparlist)

Explanation: call the function ID with all its parameters

- Print Statement

printStat -> print (expression)

- Input Statement

inputStat -> input (ID)

- Actual Parameters List

List of actual parameters

actualparlist -> actualparitem
(, actualparitem)*
| ε

- Actual Parameter Item

An actual parameter. “in”: by value, “inout”: by reference

actualparitem -> in expression
| inout ID

- Condition

Boolean expression

condition -> boolterm
(or boolterm) *

- Boolean Factor

Factor in Boolean expression

boolfactor -> not [condition]
| [condition]
| expression REL_OP expression

Explanation: A Boolean factor can have three forms.

- A simple condition which is either true or false
- A simple condition which is either true or false, reversed (not)
- A comparison between two numbers (relational operation)

- Expression

Arithmetic expression

expression -> optionalSign term
(ADD_OP term) *

Explanation: The optional Sign in the beginning is necessary for negative numbers.

- Term

Term in arithmetic expression

term -> factor
(MUL_OP factor) *

- Factor

Factor in arithmetic expression

factor -> INTEGER
| (expression)
| ID idtail

- ID Tail

Follows a function or a procedure and describes the parentheses and the parameters

idtail -> (actualparlist)
| ε

- Optional Sign

Symbols “+” and “-“ are optional

optionalSign -> ADD_OP
| ε

- REL_OP, ADD_OP, MUL_OP, INTEGER, ID

REL_OP -> = | <= | >= | > | < | <>
ADD_OP -> + | -
MUL_OP -> * | /
INTEGER -> integers
ID -> a name consisted of the Latin alphabet (up to 30 characters long)

- Token Consumption

The syntax analyzer works on tokens. Each time the analyzer checks that the token follows all the rules, it consumes it and brings in the next token from the lexical analyzer.

INTERMEDIATE CODE GENERATOR

The next stage of the compiler is the Intermediate Code Generator. Here the C-imple code will be converted into an intermediate language as a step before the final code. It creates intermediate code entries and updates the Symbol Table as we will see at the next chapter.

The intermediate Code is a series of quads that include: the operator and three arguments. Each quad has a unique number as a label. After the execution of a quad the next one with the bigger number will be executed, except if something different is specified by this quad.

Quads have the following form:

op, x, y, z

where “op” is the operator, x and y are argument 1 and argument 2

and z is argument 3 (result or dest). The operator can take the value one off: +,-,*,/,:=, jump, any real operator, begin_block, end_ block, halt, out, inp, par .

The arguments x and can be variable names or numbers(numeric constants) and the argument z is the name of a variable.

quad example: 120: +, a , 2 , b Equals: b=a+2

We always have an operator and 3 arguments. In case the action of the operator does not need all 3 arguments we leave them as blanks we mark as “_”.

For example: x := y can be generated as 210: :=, y , _ , x

In case the operator requires more than 3 arguments we can use two or more quads in coordination to achieve the same result. For example the expression d=a * b * c can be generated as:

100: *, a, b, t_1

101: *, t_1, c, d

The Quads that will be generated must be stored in memory. For our compiler we created a class named Quad to represent and manage them. (class Quad (label, op, arg1, arg2, dest)) Different objects of the class will be created along the parser of the input C-imple file and saved at a list(“quads_list”). We’ll use an intermediate support method to achieve the above steps called “genQuad”(def genQuad(`op=None, arg1='_', arg2='_', dest='_'`

)). We will use this list to generate the intermediate file and later with the help of the symbol table (next chapter) the final code.

The operators and there syntax that we'll be using for our C-imple program compiler are:

- **<u>Code blocks:</u>**

  begin_block, "name", _ , _

  end_block, "name", _ , _

  halt, _, _, _

  These quad operations are used to group the commands of different blocks of functions, procedures and the main programm. At the beginning of each block and of the programm there should be the begin_block operator with the block's or program's names a parameter and similarly at the end the end_block operator. At the end of the main program there must be the halt operator before the end block.

  For example: The C-imple Code

  program mainP

  {

  …

  }

  100: begin_block, mainP , _ , _

  101: …

  102: …

  10x: halt, _, _,_

  10x+1: end_block, mainP , _ , _

- Register value:

  :=, source, _ , target

  Registers the **source** value to **target**

  For example the code:

y:=x

will generate:

210: :=,x,_,y

- Numeric operation

  op, operand1, operand2, target

  Where **op** is on of the symbols:+,-,*,/,

  the operand1 and operand2 are the operands on which the arithmetic operation will be performed and the target is where the result will be saved.

  For example the code:

  y:=a+2

  will generate:

  310: + , a , 2 , y

- Jump command

  jump , _ , _ , label

  The jump operator squad will send the control to the command with the specified **label.**

  For example the code

  410: + , a , 2 , y

  411: jump, _ ,_ ,410

  will generate an endless loop, returning the execution flow after the 411 back to 410.

- Conditional Jump Command

  conditional_jump, x , y , label

where the **conditional_jump** is one of the relational operators '=', '<>', '<', '>', '<=', '>=' and if it's true the jump will be executed.

For example the intermediate code:

500: = , value , 0 , 400

501: >, value, 0 , 410

502:<, value, 0 , 420

will jump to 400 quad label if the **value** is 0, to 410 if the **value** is bigger than 0 and to 420 if the **value** is smaller than 0.

- Procedure or function call

  call, name , _ , _

  With the call operation the function or procedure with the **name** will be called**.**

- Parameter pass

  par, name , mode, _

  name is the exact parameter name we want to reference and **mode** the way the parameter is requested. The 3 different modes we use at C-imple are :

  cv: copy the parameter value

  ref: reference directly the parameter

  ret: the return value of a function

  For the ret type we need to make sure that the returned value will be saved in a unique suitable temporary variable. Usually of this style: T_x

  where x is a unique increasing number.

- In, out

  in, x, _, _

  out, x , _, _

When we need to read a parameter value from the keyboard (input(x))

the in operator will save the value to the variable named **x.**

When we need to print a value (print(x))

the out operator will print the value of the variable named **x**

Intermediate Support Functions

As mentioned above in order to generate and manage the Quads needed for the Intermediate Code we created the following functions at our Python Compiler Code:

- genQuad: `def genQuad(op=None, arg1='_', arg2='_', dest='_')`

  Generates the next quad according to the inputs op, the operator , **arg1**, **arg2**, the arguments 1 and 2 , **dest** the argument 3. Also increases by 1 the labels and stores it as label_number parameter.

- nextQuad: `def nextquad():label_number`

  Returns the next quad's label_number.

- newTemp: `def newTemp():new_temp_var`

  Creates and returns a new temporary variable according to the correct pattern T_1, T_2, T_3…

- emptylist: `def emptyList()`

  Creates an empty list of quads labels.

- makeList: `def makeList(label):new_list`

  Creates a new list with only the label of the quad.

- mergeList: `def mergeList(list_a, list_b):list_a + list_b`

  Creates a labels list from the merge of the lists a and b.

- backpatch: `def backpatch(label_list, dest)`

  Scans all the squads labels off the provided list and changes the dest parameter of the Quad to the provided **dest** label value.

- inter_code_file_generator:`def(inter_code_file_gen(input_file_name)`

Generates an intermediate code file based on the quads list, for inspection.

With the use of the above functions at the correct position at the Parser of our compiler we can create the quad's list and use it for the final code generator.

- For the beginning and ending of a block or the main program:

For the example:

program p_name{

}.

We'll use the functions:

genquad("begin_block", 'p_name', '_', '_')

genquad('halt', '_', '_', '_')(if it's the main program)

genquad('end_block', 'p_name', '_', '_')

- For arithmetic expressions:

We'll use the grammatical rules of the C-imple.

- add (rule)

$E \longrightarrow T^{(1)}$ (+ $T^{(2)}$ {p1})∗ {p2}

We create a new temporary variable to store the result

{p1} : n = newTemp()

We generate a new quad object to save the updated result to T2

genQuad('+', $T^{(1)}$.place, $T^{(2)}$.place, n)

the result for P1 is stored at T1 to be used in case there is another T2

$T^{(1)}$.place = n

when there are no more T2 the result is stored at T1

{p2} : E.place = $T^{(1)}$.place

- we follow the rules similarly for the (- ,*, /) expressions
- for the priority of the expressions:

  F ⟶ (E){p1}

  {p1}: F.place = E.place

  after E.place we move to F.place

- for the closing ID symbols :

  F⟶ ID{p1}

  {p1}: F.place = ID.place

  after F.place we move to ID.place

- For BOOLEAN expressions:
  - OR:

    B -> Q1 {P1 } (or {P2 } Q2 {P3 })*

    {P1 }: B.true = Q1 .true B.false = Q1 .false

    {P2 }: backpatch(B.false, nextquad())

    {P3 }: B.true = mergelist(B.true, Q2 .true)

    B.false = Q2 .false

  - AND:

    Q -> R 1 {P1 } (and {P2 } R 2 {P3 })*

    {P1 }: Q.true = R1 .true Q.false = R1 .false

    {P2 }: backpatch(Q.true, nextquad())

    {P3 }: Q.false = mergelist(Q.false, R2 .false)

    Q.true = R2 .true

  - NOT:

R -> (B) {P1 }

{p1} : R.true = B.false

R.false = B.true

Generating the Intermediate code for the C-imple Statements

- loop statement while

  while {p0} (condition) {p1} statements {p2}

```
whileStat():
condQuad = nextQuad()
{p1}:backpatch(condition.true,nextQuad())

{p2} : genQuad('jump','_','_',condQuad)
      backpatch(condition.false,nextQuad())
```

- if statement:

  if (condition) {p1} statements(1) {p2}

  elsePart {p3}

  else statements(2)

  ifStat():

  {p1} : backpatch(condition.true,nextquad())

{p2} : ifList = makeList(nextQuad())

genQuad('jump','_','_','_')

backpatch(condition.false,nextquad())

{p3} : backpatch(ifList,nextquad())

- switchcase statement:

  switchcase

  (case (condition) {p1} statements(1) {p2}) *

  default statements(2){p3}

```
def switchcaseStat():
```

{p0} : exitList = emptyList()

{p1} : backpatch(condition.true,nextQuad())

{p2} : t = makeList(nextQuad)

genQuad('jump','_','_','_')

exitList = mergeList(exitList,t)

backpatch(condition.false,nextQuad())

{p3} : backpatch(exitList,nextQuad())

- forcase statement:

  forcase {p1}

  (case (condition){p2} statements(1)){p3}) ∗

  default statements(2)

```
def forcaseStat():
    {p1} : firstCondQuad = nextQuad()
      {p2} : backpatch(condition.true,nextQuad())
      {p3} : genQuad('jump','_','_','_')
      backpatch(condition.false,nextQuad())
```

- incase statement:

  incase {p1}

  (case (condition) {p2}

  statements(1) {p3}) ∗

  default statements(2)

```
def incaseStat():
    {p1} : flag = newTemp()
      firstCondQuad = nextQuad()
    genQuad(':=',0,_,flag)
{p2} : backpatch(condition.true,nextQuad())
{p3} : genQuad(':=',1,_,flag)
    backpatch(condition.false,nextQuad())
{p4} : genQuad('=',1,_,flag,firstQuad)
```

Generating the Intermediate code for the C-imple Functions and Procedures

During the generation of the functions and procedures we must take notice of the parameters they have as functions and the range of the block they belong to.
For starters we created the function `def returnStat()` to save the value a function or a procedure returns with the return statement.
ret, _ , _ , _
Secondly we have the `def callStat()` function which calls the function by its name only after it has generated the quads for the functions / procedures parameters list.
The function `def actualparitem(self)` recognises and stores the parameter as the following quads depending of there type:

```
    for in:
       genQuad('par',expression,'cv','_')

      for inout:
      genQuad('par',self._token._recognized_string,'ref','_')
```

Symbol table

The symbol table does not follow the conventional definition of a table, with columns and rows. Instead, it is designed as a stack, with nesting levels going deeper and deeper. The reason behind that is to accommodate nesting functions and their access to the data of the parent function.

Let's start with what kind of data we store in the symbol table.

Stored Data

- Variables

    - name
    - datatype
    - offset

    In some programming languages, the type of the variable does not have to be defined. In C-imple the type is of all variables in integer.

    It is necessary to know the position in the memory that the variable is stored. The absolute position is not useful because each time a function or a procedure is called with this variable, the position in the memory changes. Especially in recursive or nested calls, the variable will be stored many times in different positions in the memory.

    That's why we are not interested in the physical position in the memory, but rather, the relative position from a starting point. We'll call that starting position the activation record and we will analyze it in detail later on. In short, it's the space in memory that is given to each function or procedure, in order to store its data. The offset is how far away the variable is store, from the beginning of the activation record.

- Parameter

    - name
    - datatype
    - offset
    - mode

    A parameter is passed into a function or a procedure. It shares many similarities in the information it has with the variable, but their role is completely different and so is the way we handle them.

    A parameter can be passed to a function or procedure with two ways: with value and with reference. Thus, the mode of each parameter can be “in” for value, “ref” for reference and “ret”. The “ret” mode correlates to the return of a function.

- Function

    - name
    - datatype
    - startingQuad
    - framelength
    - formalParameters

    For the datatype, a function can return a value. The type of that value is defined in the datatype of the function. In C-imple, all variables are integers.

    The formalParameters is a list of the parameters passed into the function when it’s called. We’ll analyze it later.

    The startingQuad is necessary, to know in which quad the caller needs to jump in order to start the callee function.

    The framelength is the size of the activation record (in bytes). Its calculation will be explained later.

- Procedure

    - name
    - startingQuad
    - framelength
    - formalParameters

    A procedure shares the same fields in the symbol table with the function, excluding the datatype, since it doesn’t return.

- Formal Parameter

    - name
    - datatype
    - mode

    The order of the parameters passed needs to be preserved to check that the function or the procedure has been called properly. If the order is maintained, then the names of the parameters inside the function can differ from those passed, without any issues.

- Temporary Variables

    - name
    - datatype
    - mode

    Same as the variable. We design it for completion purposes.

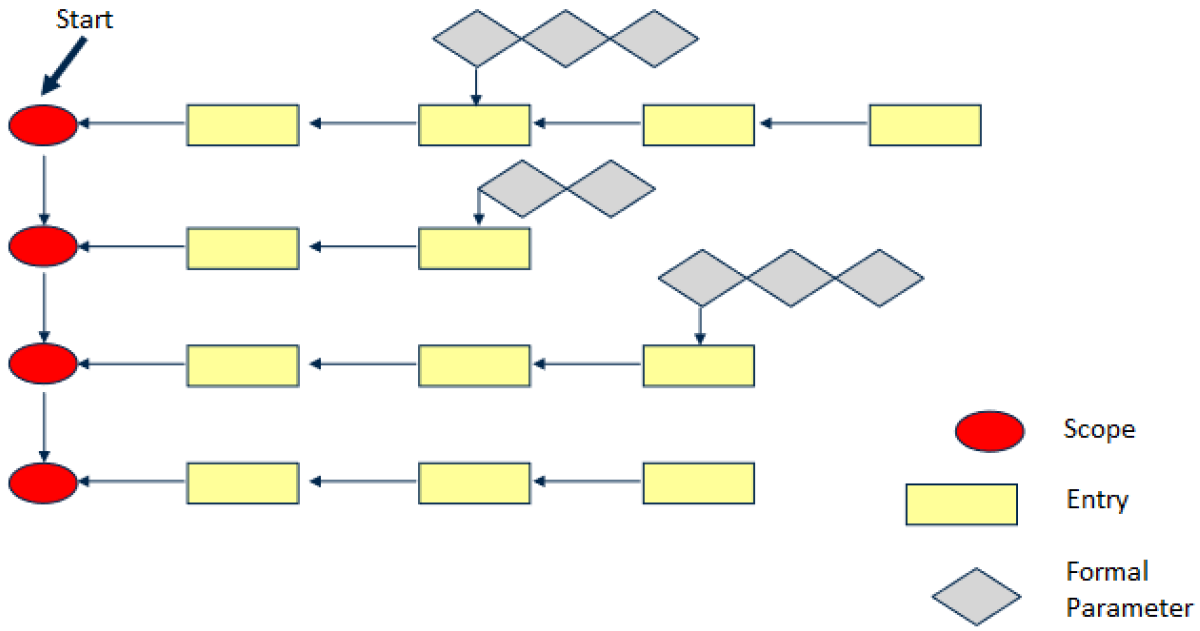
STRUCTURE

The symbol table is made up of scopes. Each scope represents the translation of a function or procedure. At the beginning of a function (or procedure), a new scope gets added and at the completion, that scope gets removed.

For example, two functions, f1 and f2, with the f2 nested in the f1. First the scope for the f1 gets created and inside that one, the f2 scope exists.

With that implementation of the symbol table, functions end up having access to the correct data, according to C-imple's rules.

Example schematic:



Activation Record

The activation record is memory for each function or procedure to store their vital information. It gets created for each function (or procedure) that is about to be called, and it gets destroyed after their completion. In it, the actual parameters of the function (or procedure), the local variables and the temporary variables, get stored.

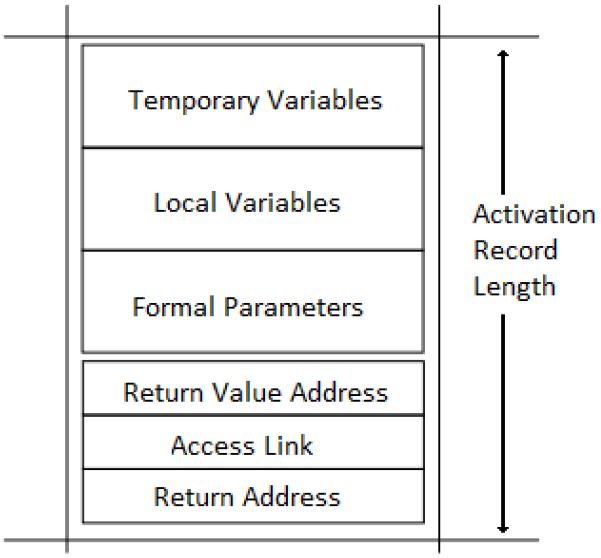
In the first position of the activation record, the return address of the function exists. We will use that information in the final code when we produce the assembly code.

In the second position, the access link gets stored. The access link is a pointer to the variables that don't exist in the activation record of this function, but this function has access to them. Those could be variables from the parent classes, or global ones. Notice that, a function can be nested into a deep scope, meaning a few other classes have been called before her. That function has access to all the variables of all the previous classes.

In the third place of the activation record, the return value address is stored. That is the address of the variable that the function will return the value to. In the case of a procedure, the return value address is empty, because procedures don't return.

Each of those memory slots require 4 bytes, meaning the first 12 bytes of each activation record are always occupied. For every other entry in the activation record, 4 more bytes are required. That is because everything in C-imple is an integer.

Example schematic:



Activation Record Length: It's the size, in bytes, of the activation record. The problem is, the size can be calculated, after we check the number of parameters, variables, and temporary variables of the function (or procedure). That happens at the completion of the intermediate code.

Offset: Is the distance, in bytes, to the beginning of the activation record. If the function (or procedure) only has one local variable, then the offset of that would be 12, because the offset 0 goes to the return address, the offset 4 goes to the access link and the offset 8 goes to the return value address.

METHODS

- Add Variable

  Add a new variable at the last scope, at the last position of the entity list. The entity list is the list of each scope, which contains all the entities in that scope (variables, parameters, functions, procedures)
  We need to check that the variable we are about to add doesn't already exist. This is where the compiler makes sure we don't create two variables

with the same name in a block.
When we add it, we need to calculate the offset of that variable.

- Add Function

  Add new function (procedure). Again, it gets added at the last scope, in the last position of the entities list of that scope.
  Need to check that it doesn't exist already. This is where the compiler makes sure we don't create two functions (or procedures) with the same name.

- Add Parameter

  Add new parameter. It gets added at the last scope, in the last position of the entities list of that scope. Need to update the offset

- Add Temporary Variable

  Same as the add parameter, except the object we add is of temporary variable

- Add Scope

  Add new scope for the main program, a function, or a procedure.

- Remove Scope

  At the completion of a function or a procedure, remove the corresponding scope.

- Update Starting Quad

The starting quad of the function (or procedure) is not available at their creation. When the first quad of that function (or procedure) gets created, then we can update the starting quad to point to that first quad.

- Add Formal Parameter

What is the difference between add parameter and add formal parameter. An example will help clear the confusion.

```
program p1{
  a := 1;
  b := 1;
  c := 1;
  procedure P1 (in x, inout y){
        a := 1;
        function F11(in c){…}
  }
}
```

We notice that the variables a, b and c are defined in the main program, that's why in the symbol table they exist in the scope with nesting level 0.

We also notice the procedure P1 with two parameters, one as in and another as inout. The symbol table stores the P1 in nesting level 0, because it is called from the main program and with it, it stores the information that the P1 has two input parameters with those modes. That last part is done through the add parameter method.

If we pay attention to the nesting level 1 though, we see that those two parameters of the P1 procedure are stored in the symbol table, with their name, their offset, and their input mode. That part is done with the add formal parameter method. The nesting level 1 is the scope of the P1

- Search Entity

  Search all the entities, in all the entities lists, in all the scopes, for the entity asked, by name. The searching begins at the scope with the highest nesting level, meaning the function (or procedure) that was called last, will be searched first.

  If the entity has not been found, call error.

- Print Symbol Table

  Print the symbol table, at the end of each function (or procedure), right before the remove scope method.

Final code generator

The final code is produced from the quads of the intermediate code with the help of the symbol table. Basically, for each quad we produce a series of assembly commands.

The assembly code is design to work on a Risc-V processor. We'll proceed with a brief introduction to the basics of the assembly in Risc-V and then we'll explain the logic behind our implementation of the compiler.

Risk-v assembly

- Registers

  32 registers for integers and 32 for floating point

  - zero: Has the 0.
  - sp: stack pointer. Point to the beginning of the activation record.
  - fp: frame pointer: Point to the beginning of the activation record when it gets created.
  - t0-t6: temporary registers. For every purpose.
  - s1-s11: saved registers. The values on these registers persist between functions.
  - a0-a7: function registers. Used for parameters and return values.
  - ra: return address. Where the program needs to return after calling a jump.
  - pc: program counter. Stores the address of the executed command.
  - gp: global pointer. Points to the start of the global variables, for easier and faster access. Will explain the usage later.
  - tp: thread pointer. Pointer to the data of a thread. Will not be used.

- Load Immediate

li reg, int

load an integer into a register

- Move

  mv reg1, reg2

  Move the value of reg2 to reg1

- Arithmetic Operation
  - add reg, reg1, reg2

    Add to reg the reg1 and the reg2
  - sub reg, reg1, reg2

    Subtract from the reg1 the reg2 and store the result in the reg
  - mul reg, reg1, reg2

    Multiply the reg1 by reg2 and store the value at reg
  - div reg, reg1, reg2

    Divide the reg1 by reg2 and store the value at reg
- addi

  addi target_reg, source_reg, int

  target_reg = source_reg + int

- Load Word

  lw reg1, offset (reg2)

  reg1 = [reg2 + offset]
  reg1 = destination register
  reg2 = base register
  offset = distance from reg2

- Store Word

  sw reg1, offset (reg2)

  [reg2 + offset] = reg1
  reg1 = source register
  reg2 = base register
  offset = distance form reg2

- Branching

  b label

  jumps to label

- Jump with Condition

  - beq reg1,reg2,label # branch if equal
  - bne reg1,reg2,label # branch if not equal

- blt reg1,reg2,label # branch if less than
- bgt reg1,reg2,label # branch if greater than
- ble reg1,reg2,label # branch if less or equal than
- bge reg1,reg2,label # branch if greater or equal than

reg1 , reg2 : the registers to be compared
label : the address to jump to

- Input and Ouput

  For input from the standard input (keyboard) we use the following commands

  ```
  li $a7, 5
  ecall
  ```

  $a0 will have the value we input

  For output to the standard output (display) we use the following commands

  ```
  li $a0, value_to_print
  li $a7, 1
  ecall
  ```

- End of program

  ```
  li $a0, 0     # for return 0 to the operating system
  li $a7, 93
  ecall
  ```

Activation record

At the beginning of the main program, the stack pointer gets placed at the beginning of the activation record of the main program. In the event that a function gets called, the sp needs to move to the beginning of the activation record of the function. When that function gets completed, the stack pointer points back to beginning of the activation record of the main program.

Auxillary Methods

Methods used to either produce assembly code, or move data between registers and memory

- gnlvcode (v)

  Produces assembly code for access in variables that don't exist in the activation record of the current function. As we have mentioned already, a function has access to the variables of all the parent functions that have called her, including the main program.

  Gnlvcode is capable of retrieving global variables but depending on the nesting level of the current function, that can be time consuming, because of amount of backtracking it would require to do, and the amount of assembly code it would have to generate to bring in those variables.
  That's why we use the global pointer to have immediate access to the main program.

  Gnlvcode takes the name of the variable as an argument. The result is:

- if the value of the variable is needed, then its address gets moved to the register t0.

- If the address of the variable is needed, then its memory address gets moved to the register t0

How it works:

- It searches the symbol table for the given name. If it can't find it, it returns an error

- Judging by the nesting level, it determines how many levels it needs to pass in order to reach the activation record that contains the variable

- The first step is to point to the parent. The address of the activation record of the parent is stored in the access link of the current function, in the -4(sp) position (second position).

  lw $t0, -4($sp)

- This process is repeated until we reach the proper nesting level. At that point, the register t0 points at the beginning of the activation record of the proper function
- t0 needs to move offset amount of bytes to reach the point in the activation record where the variable is stored.

All the code:

lw t0,-8(sp)

lw t0,-8(t0)

...

lw t0,-8(t0)

addi t0,t0,offset

- loadvr (v, reg)

  It produces assembly code for reading the value of a variable from the memory and storing it into a register.

  It takes as arguments the name of the variable and the name of the register where it will be stored.

  Loadvr is a big if statement with produces the appropriate code depending on the information about the variable

  If the variable is:

  - Local, or parameter with value, or temporary variable

    lw reg, -offset ($sp)

    Because the variable is in the activation record of the current function, we have all the information available.

  - Parameter with reference

    With reference, only the address of the variable is available. Which means, first we need to load the address into a register and then call the load word on that address

    lw $t0, -offset ($sp)
    lw reg, ($t0)

  - Local variable or parameter with value which belongs to a parent

    In this case, we need the gnlvcode to bring the variable from the activation record of a parent function.

gnlvcode (variable_name)
lw reg, ($t0)

- Parameter with reference which belongs to a parent

  Same as above, with the difference that the gnlvcode will return the address of the variable, not the value. We need to use the load word command to load the value from the address.

  gnlvcode (variable_name)
  lw $t0, ($t0)
  lw reg, ($t0)

- Global Variable

  As we have already mentioned, we could use the gnlvcode to bring the global variables in the current activation record. But that could become costly.

  That's why, we use the global pointer, which always points at the beginning of the beginning of the activation record of the main program.

  lw reg, -offset ($gp)

- Load value

  li reg, integer

- storerv (reg, v)

  Similar in implementation with the loadvr. It just stores instead of loading.

  As arguments, it takes the target register and the name of the variable to be stored.

  If the variable is:

  - Local or parameter with value or temporary variable

    sw reg, -oofset ($sp)

  - Parameter with reference

    lw $t0, -offset ($sp)
    sw reg, ($t0)

  - Local variable or parameter with value which belongs to a parent

    gnlvcode (variable_name)
    sw reg, ($t0)

  - Parameter with reference which belongs to a parent

    gnlvcode (variable_name)
    lw $t0, ($t0)
    lw reg, ($t0)

- Global variable

  sw reg, -offset ($gp)

- Store value

  loadvr (1, $t0)
  storerv ($t0, reg)

GENERATOR

The actual generator is nothing more than a few if statements. Before a scope gets removed, we go through each quad that has been generated at that time, we check what kind of command that quad represents, and we produce the appropriate assembly code.

- Jump to Main Program

  The order of the quads does not represent the order in which the program will be executed. The commands of the main program will be compiled last, but we want them to be executed first. That's why, at the beginning of the assembly code we need to perform a jump to the main label with the main code.

  The first quad has a label number of 0. When we detect that, we produce the following assembly code:

  j Lmain

  The label for the main will be produced later.

- When we detect the block of the main program we produce the following assembly code:

  Lmain:
  ```
      addi sp, sp, framelegth_main
      mv gp, sp
  ```

  Firstly, we need to place the stack pointer at the start of the activation record. At the time of the creation, the sp points at the beginning of the stack which was given by the operating system. We need to move it upwards as many bytes as the activation record of the main program, which is its framelength.

  Secondly, the gp needs to point to the same position, so we copy the sp.

- Begin block

  When a new block begins, a new function gets called. Before we jump to commands of that function, we need to store the address of the command that called her, in order to return after the function has been completed. The following assembly code gets generated:

  ```
  sw $ra, -0( $sp )
  ```

- End block

  When a function is completed, we need to load the address of the command that called her into the ra and then jump to it.

  ```
  lw $ra, -0 ( $sp )
  jr $ra
  ```

- Halt

  When the program ends, we need to generate the appropriate commands in assembly. After the halt quad has been detected produce the following assembly code:

  li $a0, 0
  li $a7, 93
  ecall

- Assignment

  :=, x, _, z # z := x

  loadvr (x, $t0)
  storerv ($t0, z)

  Load the value of x into the temporary register t0 and then store it to the memory which corresponds to the variable z

- Jump

  j Label

- Jump with condition

  We created a table with all the conditions for jump

  cond_jump_fin_code = ['beq', 'bne', 'blt', 'bgt', 'ble', 'bge']
  cond_jump_int_code = [‘==’,’<>’,’<’,’>’,’<=’,’>=’]

  Example:

  cond_jump_int_code , x, y, label
  If the quad has any of these conditions, the following assembly code gets

generated:

```
loadvr ( x, t1)
loadvr (y, t2)
cond_jump_fin_code t1, t2, label
```

- Numeric Operations

  We created these tables for the operations

```
arithmetic_operators = ['+', '-', '*', '/']
ar_operators_fin_code=['add', 'sub', 'mul', 'div']
```

  Example: arithmetic_operator, z, x, y

  Assembly code

```
loadvr (x, t1)
loadvr (y, t2)
ar_operators_fin_code t1, t2, label
storerv(t1, z)
```

- Return

```
loadvr (x, t1)
lw t0, -8(sp)          # take the return value from the activation record
sw t1, (t0)
```

- Input

```
li a7, 5                          # standard command for input
ecall
mv t0, a0                         # move the input to t0
```

```
storerv (t0, dest_variable)        # store the input into the variable
```

- Output

```
loadvr (source_var, t0)            # load the variable into t0
mv a0, t0                          # move t0 to a0, standard out
li a7, 1                           # standard command for output
ecall
```

- Parameter

  The parameters will be placed above the first three entries in the activation record. We will use the frame pointer for easy access.

  addi fp, sp, framelength

  Placing fp at the start of the activation record, according to the stack pointer

  - Parameter with value

    loadvr (variable, t0)
    sw t0, -par_offset (fp)

    Load the value into t0 and then store it in the stack according to the fp.

    par_offset = 12 + (i-1)*4 bytes
    12 bytes for the three places in the activation record as we have discussed and we use the i for each parameter.

- Parameter with reference when the value is stored in the stack

  If we have available the value in the current activation record, we don't need to call the gnlvcode.

  - Local variable or temporary variable or parameter with value

    addi t0, sp, -offset
    sw t0, -par_offset (fp)

  - Local variable or parameter with value which belongs to a parent

    gnlvcode ()
    sw t0, -par_offset(fp)

  - Global Variable

    addi t0, gp, -offset
    sw t0, -par_offset (gp)

- Parameter with reference when the address is stored in the stack

  Similar with the previous one, but we need one extra step to load the value from the address

  - Parameter with reference in the same nesting level

    lw t0, -offset (sp)
    sw t0, -par_offset(fp)

- Parameter with reference from a parent

  gnlcvcode()
  lw t0, (t0)
  sw t0, -par_offset(fp)

- Call

As we mentioned at the Intermediate Code Generator when a function is called a quad of the syntax: ***call, _ , _ , functionName*** is generated. The first thing we do is to check the callers and the functions nesting level to determine their relationship.

- If the nesting level is the same they have the same parent:

  lw t0,-4(sp)

  sw t0,-4(fp)

- If the nesting level is different the caller is the parent of the called fuction:

  sw sp,-4(fp)

- After that we move the stack pointer to the called function

  addi sp,sp,framelength

- We call the function

  jal L_ 'startingQuad'

- after the function is called and we return back to the caller, we return the stack pointer back

  addi sp,sp,-framelength

- last inside the called function at the beginning we saved the return address from the $ra that the jal saved it.

  sw ra,(sp)

- at the end of every function we save its own return address at $ra. By reading the ra we return to the caller.

  lw ra,(sp)

  jr ra

C FILE

Part of this project was the translation of the quads from the intermediate code, into C. This has nothing to do with the compilation of the C-imple program. The C file was requested by the professor for grading purposes, because it's a lot easier to test the functionality of the intermediate code by turning it into C and running it as a C program.

The way we implemented that was using a big for loop. For each quad that we generated in the intermediate code, we went and checked what kind of quad that was (i.e., assignment, addition, etc.). According to that information, we produced code in c, that did the exact same work as the quad in C-imple would do.

With that implementation in mind, we can't translate code from C-imple that has nesting. Meaning, no function and no procedures. According to the professor, that is alright.

Unresolved issues and bugs

During our implementation we stumbled into some issues. Some of them are on the programming side, like bugs, while others are more theoretical, like the functionality of the assembly code.

Here we try to document our issues.

The compiler must be run with python 3

Known bugs:

1) The opened files (test.int, test.symb) cannot be closed in the case of an error. That is because the error method is implemented in the Lex class. In order to close the files, we would need access to them from the Parser class but that is not possible, because those files are defined in the Parser class, which inherits the Lex class.

Possible solution: Create global files. In our implementation, we tried to avoid creating global fields and classes.

2) The ci_var_list is being used to store the variables and later generate the intermediate code file. The problem is that a variable is being stored in the list, each time it is used (even as a parameter). That does not effect the intermediate code in c, it just creates the same declaration many times.

Possible solution: Add if statement before the var_list.append to check if the variable has already been appended.

3) In the assembly file, the commands what should be in the main label end up in their own label, following the main. That causes no issues in the functionality

4) We did not test the functionality of the assembly code using a risk V simulator. All the testing we did was manual.

6) In the storerv auxiliary method for the final code generation, we didn’t implement the load value part.

7) In the final code generator, in the parameter with reference when the value of the parameter is in the stack, we haven’t implemented the global variable.