MARIA TSIGKOU : AM 4191

TRIANTAFULLIW DOUMANI : AM 4052

CIMPLE COMPILER

ABOUT CIMPLE

Cimple is a simple educational programming language based on C.

This language supports popular programming commands and functions like:

- While
- If - Else
- Switchcase
- Return
- Print
- Input
- Call

.

Also includes the new commands :
Forcase and **Incase** .

Cimple also supports functions and procedures, pass by reference and by value ,recursive calls and declaration of nested functions(Not supported by C).

However Cimple does not support basic programming tools like:

- For loops
- Strings
- Real numbers
- Arrays.

These omissions have been made for educational purposes in order to simplify the procedure construction of the compiler.

Lastly, The compiler will produce, as the final language, the assembly language of the MIPS processor.Using a MIPS emulator we will be able to execute the assembly code generated by the compiler.

Cimple files end in .ci

The Cimple alphabet

- Letters(A,...,Z and a,...,z)
- Digits (0,...,9)
- Arithmetic operations (+, -, *, /)
- Relational operators (<, >, =, <=, >=, <>)
- Assignment symbol(:=)
- Separators (;, “,” , :)
- Grouping symbols([,], (,) , { , })
- Program termination symbol (.)
- Comment separation(#)

The priority of the operators from the largest to the smallest is:

- *, /
- +, -
- =, <, >, <>, <=, >=
- not
- and
- or

The symbols [,] are used in logical representations such as the symbols (,) in numbers representations.
The numbers take values from –(2^32 − 1) to (2^32 − 1).
Language identifiers (ID) are strings consisting of letters and numbers, but starting with a letter.
ID with more than 30 characters is considered incorrect.
The white characters (tab, space, return) are ignored.

RESERVED WORDS :

• program	print
• if	while
• switchcase	incase
• not	or
• function	call
• input	case default
• declare	return in
• else	inout
• forcase	procedure
• and	

These words cannot be used as variables.

PROGRAM FORMAT:

```
program ID
        Declarations
        Subprograms
        Statements
```

DECLARATIONS

declare ID (,ID) ;*
We are allowed to have more of one consecutive uses of declare

SUBPROGRAMS

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

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

FormalPars is the list of standard parameters

By reference: ***inout***
By value: ***in***

STATEMENTS

- **Assignment**

  ID := expression
  Used to assign the value of a variable or a constant, or an expression in a variable.

- **If - Else**

```
if ( condition )
    statements 1
[else
    statements 2 ]
```

- **while**

```
while ( condition )
    statements
```

- **switchcase**

The switchcase repetition structure checks the conditions after the cases.
Just one of these are found to be true, then the statements1 are executed (following the condition).
The program then continues **out** of switchcase.
If none of the cases apply, then the control goes to default and statements2 are executed.
Then the program continues .

```
switchcase
        (case ( condition ) statements 1 ) ∗
        default statements 2
```

- **forcase**

The forcase repetition structure checks the conditions after the cases.
Just one of these are found to be true, then the statements1 are executed (following the condition).
The program then goes to the **beginning** of forcase.
If none of the cases apply, then the control goes to default and statements2 are executed.
Then the program continues .

```
forcase
        (case ( condition ) statements 1 ) ∗
        default statements 2
```

- **Incase**

The incase repetition structure checks the conditions after the cases, examining them in order.
For each for which the corresponding condition applies, the statements are executed(following the condition).
All conditions will be examined in turn and will be executed all statements whose conditions apply.
After all the cases are examined, the control goes outside the incase structure if none of the statements have been executed, or goes to the beginning of the incase, if even one of the statements has been executed.

```
incase
            (case ( condition ) statements 1 ) ∗
            default statements 2
```

- **return**

  return (expression)

- **print**

  print (expression)

- **input**

  input (ID)

  Asks the user to enter a value through the keyboard. The input value will be given to the variable ID.

- **call**

  call funtionName (actualParameters)

Scope rules

Global are the variables declared in the main program and are accessible to all..
Local are the variables declared in a function or process and are only accessible through that function or process.

Grammar of Cimple

```
<program>          :      program ID <block> .

<block>            :      <declarations> <subprograms> <statements>

<declarations>     :      ( declare <varlist> ; ) ∗
```

```
<varlist>          :     ID ( , ID )∗
                   |     ε

<subprograms>      :     ( <subprogram> )∗

<subprogram>       :     function ID ( <formalparlist> ) <block>
                   |     procedure ID ( <formalparlist> ) <block>

<formalparlist>    :     <formalparitem> ( , <formalparitem> )∗
                   |     ε

<formalparitem>    :      in ID
                   |      inout ID

<statements>       :     <statement> ;
                   |     { <statement> ( ; <statement> )∗ }

<statement>        :     <assignStat>
                   |     <ifStat>
                   |     <whileStat>
                   |     <switchcaseStat>
                   |     <forcaseStat>
                   |     <incaseStat>
                   |     <callStat>
                   |     <returnStat>
                   |     <inputStat>
                   |     <printStat>
                   |     ε

<assignStat >      :     ID := <expression>

<ifStat>           :     if ( <condition> ) <statements> <elsepart>

<elsepart>         :      else <statements>
                   |      ε

<whileStat>  :     while ( <condition> ) <statements>

<switchcaseStat>:  switchcase ( case ( <condition>) <statements> )∗
                         default <statements>
```

```
<forcaseStat>       :     forcase ( case ( <condition>) <statements> )∗
                          default <statements>

<incaseStat>        :      incase ( case ( <condition>) <statements> )∗

<returnStat>        :     return( <expression> )

<callStat>          :     call ID( <actualparlist> )

<printStat>  :      print( <expression> )

<inputStat>  :      input( ID )

<actualparlist>     :     <actualparitem> ( , <actualparitem> )∗
                    |     ε

<actualparitem>     :     in <expression>
                    |     inout ID

<condition> :       <boolterm> ( or <boolterm> )∗

<boolterm>  :       <boolfactor> ( and <boolfactor> )∗

<boolfactor>        :     not [ <condition>]
                    |     [ <condition>]
                    |     <expression> <REL_OP> <expression>

<expression>        :     <optionalSign> <term> ( <ADD_OP> <term> )∗

<term>              :     <factor> ( <MUL_OP> <factor> )∗

<factor>            :     INTEGER
                    |     ( <expression> )
                    |     ID <idtail>

<idtail>            :     ( <actualparlist> )
                    |     ε

<optionalSign>      :      <ADD_OP>
                    |     ε

<REL_OP>            :     = | <= | >= | > | < | <> ;
```

<ADD_OP>	:	+ | -
<MUL_OP>	:	* | /
INTEGER	:	[0-9]+
ID	:	[a-zA-Z][a-zA-Z0-9]*

Example of Cimple

```
program countDigits
declare x, count;

# main #
{
        input(x);
        count := 0;
        while (x>0)
        {
                x := x/10;
                count := count+1;
        };
        print(count);
}.
```

SYNTAX ANALYSIS

During this process ,we check if the program's structure belongs or not to Cimple language. This is achieved using recursion descent and based on Grammar LL(1).

Specifically, for each one of the grammar rules we implement the corresponding function. Each function when it meets non-terminal sign ,it calls the corresponding function and when it meets terminal sign ,it checks if this specific sign matches with the vocabulary's sign. In case we have a match, syntax analysis continues. Otherwise, the compilation terminates with error message. Finally, in case the last word of the program is recognized ,then the compilation has been successfully completed.

Example of the first rule

<program> : program ID <block> .

```
function program():
   lex()
   if(tokenString == "program"):
      lex()
      if(tokenType == "idtk"):
         block()
      else:
         print("Syntax Error line: " + str(line) + "\nProgram name expected")
         exit()
   else:
      print ("Syntax Error line: " + str(line)+ "\nThe keyword ' program' expected")
      exit()
```

..second rule

<block> : <declarations> <subprograms> <statements>

```
function block():
   lex()
   declarations()
   subprograms()
   lex()
   while(1):
      statements()
      if(tokenString == "}"):
         lex()
   print("\nSTATEMENTS PASS\nexit with : " + tokenString)
```

Respectively, we work with each of the other rules

INTERMEDIATE CODE

The Intermediate Code is a set of quads that are composed by one operator and three performers(?τελουμενα?)(i.e. +,a,b,t_1). The quads are numbered. Each one, has a unique number in front of it that characterizes it. Once the execution of a quad is completed, the quad to be executed is the one with the next largest number, unless the quad has just been executed indicate something different(i.e.jump,_,_,100).

We have the following categories:

1. **op, x, y, z**
   - In which "op" can be: +, -, *, /
   - The performers "x, y" can be: variable names or numerical constants
   - The performer "z" can be: variable name
   - The op operator is applied to the x and y variants and the result is placed in the performed z.

   - For example:
     - +, a, b, c   corresponds to: *c = a + b*
     - /, a, b, c   corresponds to: *c = a / b*
2. **:=, x, _, z**
   - In which "x" can be: variable name or numerical constant
   - The performer "z" can be: variable name
   - The value of "x" is assigned to "z" variable

   - For example:
     - *:=, x, _, z*   corresponds to: z := x
     - The x *:= y – 2* corresponds to: 100: -, y, 2, T_0
       101: :=, T_0, _, x

3. **Jump, _, _, z**
   - jump without terms in the quad with number "z".

4. **relop, x, y, z**
   - In which relop can be: =, >, <, <>, >=, <=
   - Jump in the quad with number "z" if "x relop y" is true.

- For example:
    - 100: =, a, 4, 120
    - 110: jump, _, _, 140
    - 120: :=, 1, _, b
    - 130: jump 150
    - 140: :=, 2, _, b
    - 150: …

  **b = 1, if (a = 4) and b = 2, if(a !=4)*

5.

- "begin_block, name, _,_"
  Beginning of a subprogram or program named "name"
- "end_block, name, _, _"
  End of a subprogram or program named "name"
- "halt, _, _, _"
  Program termination
- For example the program:

```
program ex()
{
        Function p1()
        {
                Y := Y – 1
        }
}.
```

Corresponds to:

```
100:  begin_block_p1, _, _
101:  -, Y, 1, T_0
102:  :=, T_0, _, Y
103:  halt, _, _, _
104:  end_block, p1, _, _
```

Also, we have quads that concern functions&procedures:

6. Par, x, m, _

- In which "x" is: function parameter
- And "m" is: way of passing
    - CV: "by value"
    - REF: "by reference"
    - RET: "return value of a function"

7. **Call, name, _, _**
    - Call a function named “name”

8. **Ret, x, _, _**
    - return value of a function

9. **For example:**

    - X := foo(in a, inout b) corresponds to:

```
100: par, a, cv, _
110: par, b, ref, _
120: par, T_1, ret, _
130: call, foo, _, _
```

For the intermediate code implementation we used some helpful functions:

- **nextquad():** returns the number of the next quad to be produced

- **genquad(op, x, y, z):** creates the next quad (op, x, y, z).Specifically, creates a list of 5 places(new_quad). In the first position, places the number of the next quad(= label) ,calling the “nextquad()” function and in the other four positions ,places “op, z, y , z” . Also, it puts this 5-places list in a global list named quad_list, which keeps the lists of all quads we need for the intermediate code.

- **newtemp():**
    - creates and returns a new temporary variable
    - temporary variables are of the form T_1, T_2, T_3…

- **emptylist():** creates an blank list of quads labels

- **makelist(x):** creates a list of quads labels containing only “x”

- **merge(list1, list2):** creates a list of quads labels by merging list1, list2

- **backpatch(list, z):**
  - the list “list” consists of pointers in quads whose last variant is not completed
  - backpatch visits these quads one by one and completes them with label “z”

We use the functions above in the functions of Syntax analysis code.
For example a Cimple program:

```
while(x > 1)
        {
                sum := sum + x;

        };
        print(sum);
```

The corresponding intermediate code:

```
[1, '>', 'x', '1', 3]
[2, 'jump', '_', '_', 6]
[3, '+', 'sum', 'x', 'T_0']
[4, ':=', 'sum', '_', 'T_0']
[5, 'jump', '_', '_', 1]
[6, 'out', 'sum', '_', '_']             ??????????????????
```

SEMANTIC ANALYSIS

In semantic analysis, we have to check if each function has at least one “return” statement and there is no statement in procedures. To achieve this ,we created a function called *check_function_return_statement().* There is a global variable named “ret” that increases every time the function *returnStat* is called. So, when a function in Cimple code reaches the end, we check ,from the return value of check_function_return_statement(), how many “returns” it contains and we print an error message in case there is none. We work accordingly in case we have a procedure with the only difference that we print an error message in case there is at least a “return” statement.

So, *check_function_return_statement()*:

- Returns 0, if there’s no "return” statement in function or procedure
- or, returns 1 in any other case.

SYMBOL TABLE

Symbol Table consists of Scopes, Entities and Arguments which contain:

- **Scopes**

  ```
  class Scope:
      entity = []
      offset = 12
      def __init__(self, nestingLevel):
  ```

  - List of entities
  - Nesting Level
  - An auxiliary variable named “offset” ,initialized to zero value, which increases by four with each new variable added to Scope(??total??distance from the top of the stack).
- **Entity**

  ```
  class Entities:

      arguments = []
      def __init__(self, name, value, parMode, offset, type, startQuad,
  framelength):
  ```

  - Variable
    - Name
    - Type
    - Offset(distance from the beginning of the record activation)

- Function
  - Name
  - Type
  - startQuad(label of the first quad of its code function)
  - list of argument(parameter list)
  - framelength(activation record length)
- Paremeter
  - Name
  - parMode(way of passing)
  - offset(distance from the beginning of the record activation)

- Temporary variable
  - Name
  - Offset(distance from the beginning of the record activation)

- **Argument**
  class Arguments:

  ```
      def __init__(self, parMode, varType):
      self.parMode = parMode
      self.varType = varType
  ```

  - parMode(way of passing)
  - type (type of variable)

We have three classes(Entities,Scope,Arguments) with the above fields and one global list named "scopes" which contains Scopes.
Every entity ,depending on its type, it is of the form:

- **type Var:**
  - ent = Entities(name,None,None,scopes[-1].offset, "Var",None,None)
- **type TempVar**
  - ent = Entities(name,None,None,scopes[1].offset,"tempVar",None,None)
- **type Function**
  - ent = Entities(name,None,None,None,"Function",None,None)
- **type Par**
  - ent = Entities(name,None,parMode,scopes[-1].offset, "Par",None,None)

Also, for the creation of Symbol Table we've implemented some functions:

- **addNewScope():**
  - adds a new Scope in the global list "scopes"
- **addNewVar(name):**
  - with parameter the variable name, creates an object type of "Entities" type of "Var" with the respective fields*(name,None,None,scopes[-1].offset, "Var",None,None)*. In the "offset" field it increases it by four and finally puts the specific entity in the last Scope of "scopes" list.
- **addNewTempVar(name):**
  - with parameter the temporary variable name, creates an object type of "Entities" type of "tempVar" with the respective fields*(name,None,None,scopes[1].offset,"tempVar",None,None)*. In the "offset" field it increases it by four and finally puts the specific entity in the last Scope of "scopes" list.
- **addNewPar(name,parMode):**
  - with parameter the parameter's name and its way of passing, creates an object type of "Entities" type of "Par", " with the respective fields*(name,None,parMode,scopes[-1].offset, "Par",None,None)* and puts the specific entity in the last Scope of "scopes" list.
  -
- **addNewFunction(name):**
  - with parameter the temporary variable name, creates an object type of "Entities" type of "function" with the respective fields*(name,None,None,None,"Function",None,None)* and puts the specific entity in the last Scope of "scopes" list.

The functions above, are called in Syntax Analysis' functions body.

- Initially, the creation of Scopes is done first in the function *"program()"* (here the main Scope is created) and secondly all remaining Scopes are created in the *"subprogram()"* function .
- Creation of entities:
  - "var" type entities are created when we declare the variables ,after we've checked whether there is a variable with the same name. Specifically, in the function *"declarations()"*. (line:261)
  - "tempVar" type entities are created at the moment a new temporary variable in the function "newtemp()".(line:145)
  - "function" entities are created when a new function is declared in *"subprogram()"* function. (line:301)
  - "par" entities are created by definition parameters in *"actualparitem()"*.(line:760,line:771)

EXAMPLES

EXAMPLE 1

Cimple program:

```
program absolutevalue

        declare x;

        function absvalue(in x)
        {
                forcase
                        case (x >= 0) x := x;
                        case (x < 0) x := - x;
                        default;
                return(x);

        }
        # m a i n #
        {
                input(x);
                print(absvalue(in x));
        }.
```

Intermediate Code:

```
[0, 'begin_block', 'absvalue', '_', '_']
[1, 'par', 'x', 'CV', '_']
[2, '>=', 'x', '0', 4]
[3, 'jump', '_', '_', 6]
[4, ':=', 'x', '_', 'x']
[5, 'jump', '_', '_', 2]
[6, '<', 'x', '0', 8]
[7, 'jump', '_', '_', 10]
[8, ':=', 'x', '_', 'x']
[9, 'jump', '_', '_', 2]
[10, 'retv', 'x', '_', '_']
[11, 'end_block', 'absvalue', '_', '_']
[12, 'begin_block', 'absolutevalue', '_', '_']
[13, 'inp', 'x', '_', '_']
[14, 'par', 'x', 'CV', '_']
```

```
[15, 'par', 'T_0', 'RET', '_']
[16, 'call', '_', '_', 'absvalue']
[17, 'out', 'T_0', '_', '_']
[18, 'halt', '_', '_', '_']
[19, 'end_block', 'absolutevalue', '_', '_']
```

Symbol Table:

```
x   Var   12
absvalue   Function   2
x   Par   12   in
T_0   tempVar   16
```

**the symbol table is printed in terminal*

EXAMPLE 2

Cimple program:

```
program factorial
        # d e c l a r a t i o n s #
        declare x,y;
        declare i,fact;
        # m a i n #
        {
                input(x);
                fact := 1;
                i := 1;
                while (i<=x)
                {
                        fact := fact*i;
                        i := i+1;
                };
                print(fact);
        }.
```

Intermediate Code:

```
[0, 'begin_block', 'factorial', '_', '_']
[1, 'inp', 'x', '_', '_']
[2, ':=', 'fact', '_', '1']
[3, ':=', 'i', '_', '1']
[4, '<=', 'i', 'x', 6]
```

```
[5, 'jump', '_', '_', 11]
[6, '*', 'fact', 'i', 'T_0']
[7, ':=', 'fact', '_', 'T_0']
[8, '+', 'i', '1', 'T_1']
[9, ':=', 'i', '_', 'T_1']
[10, 'jump', '_', '_', 4]
[11, 'out', 'fact', '_', '_']
[12, 'halt', '_', '_', '_']
[13, 'end_block', 'factorial', '_', '_']
```

Symbol Table:

```
x   Var   12
y   Var   16
i   Var   20
fact   Var   24
T_0   tempVar   28
T_1   tempVar   32
```

**the symbol table is printed in terminal*

EXAMPLE 2

Cimple program:

```
program InCase

                declare x,k,t,y;
                declare count,mult;

        # main #
        {
                incase
                        case(k>1000) t := t*30;
                        case (k<500) t := t-100;

                 ;
                while ( count <= y)
                {
                        mult :=mult * count;
                        count := count + 10;
                };
                print(mult);
        }.
```

Intermediate Code:

```
[0, 'begin_block', 'InCase', '_', '_']
[1, '>', 'k', '1000', 3]
[2, 'jump', '_', '_', 5]
[3, '*', 't', '30', 'T_1']
[4, ':=', 't', '_', 'T_1']
[5, '<', 'k', '500', 7]
[6, 'jump', '_', '_', 9]
[7, '-', 't', '100', 'T_2']
[8, ':=', 't', '_', 'T_2']
[9, '=', 'T_0', '0', 1]
[10, '<=', 'count', 'y', 12]
[11, 'jump', '_', '_', 17]
[12, '*', 'mult', 'count', 'T_3']
[13, ':=', 'mult', '_', 'T_3']
[14, '+', 'count', '10', 'T_4']
[15, ':=', 'count', '_', 'T_4']
[16, 'jump', '_', '_', 10]
[17, 'out', 'mult', '_', '_']
[18, 'halt', '_', '_', '_']
[19, 'end_block', 'InCase', '_', '_']
```

Symbol Table:

```
x   Var   12
k   Var   16
t   Var   20
y   Var   24
count   Var   28
mult   Var   32
T_0   tempVar   36
T_1   tempVar   40
T_2   tempVar   44
T_3   tempVar   48
T_4   tempVar   52
```

**the symbol table is printed in terminal*