F29LP: ASSIGNMENT PART 2

OVERVIEW

Your task is to develop a compiler for the language given below which we call FUNC. The compiler should produce MIPS code that can be emulated using the MARS MIPS emulator. You are expected to use the labs throughout the course to work on it.

The assignment is split into two parts:

- **Part 1:** The front end should be completed by Thursday of week 8 (additional lab slot). This is weighted at 30/80 of the mark.
- **Part 2:** The back end should be completed by Thursday of week 11 (additional lab slot). This is weighted at 50/80 of the mark.

SUBMISSION DETAILS

Your solution will be **marked in lab**. You can get it marked any time until the deadline. In addition, **you have to submit a zip archive of your source code on Vision before each deadline.** This will not be marked in itself, but is part of the submission so you are required to submit it. There are two reasons you have to submit your source code:

- In the past there have been cases of plagiarism, and this will be used to check that the work has not been copied from others. See plagiarism details below.
- As this will count towards your final degree, the external examiner will want to see some solutions.

More details are found at the end of the document.

THE SOURCE LANGUAGE

The FUNC language consists of a set of functions and has the following syntax:

```
<program> ::= <functions>
<functions> ::= <function>; [<functions >]
<function> ::= function <name>([<args>])[returns (<arg>)]
               [is <defs>]
               begin <commands> end <name>
<args> ::= <arg> [,<args>]
<arg> ::= <name> : Integer
<defs> ::= <def>; [<defs>]
<def> ::= <name> : <type>
<type> ::= Integer | Array of size <number>
<commands> ::= <command>; [<commands>]
<command> ::= <assign> | <if> | <while> | read <name> | write <expr>
<assign> ::= <name>[ [ <expr> ] ] := <expr>
<if> ::= if <condexpr> then <commands> [else <commands>] end if
<for> ::= while <condexpr> loop <commands> end loop
<condexpr> ::= <bop>(<exprs>)
<bop> ::= Less | LessEq | Eq | NEq
<exprs> ::= <expr> [,<exprs>]
<expr> ::= <name>[(<exprs>) | [ <expr> ] ] | <number>
```

`<number>` is a natural number
`<name>` is any string starting with character followed by characters or numbers (that is disjoint from the keywords)

In addition, you can assume the following:

- Each program should have a function called **`Main`** with no arguments and no return value
- All other functions should have a return value.
    - Note that only integers are supported for arguments and return values.
- You should support the following built-in functions, that accepts two integers and return an integer:
    - **`Plus`**, which adds the arguments
    - **`Times`**, which multiplies the arguments
    - **`Minus`**, which subtracts the arguments
    - **`Divide`**, which divides the arguments.
- All the Boolean operators (`<bop> ::=` **`Less`** | **`LessEq`** | **`Eq`** | **`NEq)`** are binary
- The **`read`** command assumes that the given variable is an integer

The following example illustrates a valid FUNC program:

```
function Inc(x : Integer) returns (xx : Integer)
begin
  xx := Plus(x,1);
end Inc;

function MyTimes(x : Integer,y : Integer) returns (res : Integer) is
  i : Integer;
begin
  res := 0;
  i := 0;
  while LessEqual(i,y) loop
    res := Plus(x,res);
    i := Inc(i);
  end loop;
  write y;
end MyTimes;

function Main() is
  a : Integer;
  b : Integer;
  x : Integer;
  y : Integer;
begin
  read a;
  read b;
  x := MyTimes(a,b);
  y := Times(a,b);
  if Eq(a,b)
   then write 1;
   else write 0;
  end if;
end Main;
```

More examples will be made available in due course.

PART 1: FRONT END [DEADLINE: WEEK 8; 30/80]

Your task is to implement the front end. To complete this part you need to understand the contents of the first 4 lectures, which will be completed in week 7. Your task is to produce:

- A FLEX file with a suitable token representation (15/80)
- A recursive descent parser generating an AST with a suitable representation of the nodes (15/80)

In week 8 there will be an additional lab on Thursday at 10:15-11:15. This is your final chance to get this part marked.

THE TARGET LANGUAGE

From the AST in part 1 your task is to develop the back-end that should generate MIPS code which can be executed in the MARS simulator. In the labs you can run MARS with the command:

```
$ mars
```

If you want to run it on your own computer then you can download it from this address:

http://courses.missouristate.edu/KenVollmar/MARS/

When the compiler is executed with a given FUNC program

```
myprogram.fun
```

then it should generate a file

```
myprogram.asm
```

which can be opened and emulated in MARS.

Similar to the SIMP language from the lectures you should use "common sense" understanding of the semantics of most constructs. Here are some further requirements and assumptions you can make beyond those already discussed in part 1:

- You can assume that there are sufficient registers, so all variables can be in the registers.
- You can ignore arrays for the basic solution (see next section), so it only needs to support integers.
- There is no global state.
- The code in the `Main()` method should be executed (using the `main:` label in MIPS)
    - All other functions can only be reached by calling them from `Main` (possibly indirectly via other function calls)
- Use the `$s0-$s7` registers to store variables of the `Main()` function.
- Use the `$t0-$t7` registers for the other methods.
- Use `$a0-$a3` to store arguments. You can assume that there are not more than 4 arguments
- Use `$v0` to store the return value. Note that there are no return statements in the language. The variable declared in the **`returns`** field should be returned in `$v0`.
- Registers should be allocated in the order variables are introduced, starting with `$s0/$t0`
- You can use `$t8` and `$t9` for intermediate computations as we have done for SIMP.

Further requirements and assumptions are detailed for each relevant part below, and may appear on Vision as the course progresses.

PART 2: BACK END [DEADLINE: WEEK 11; 50/80]

BASIC SOLUTION WITH NO FUNCTION [30/80]

The basic solution should support the `Main()` function with all its features, except the ability to call other user defined functions (you still need to support the built-in: `Plus, Minus, Times, Divide`). For partial solutions, a suitable number of marks will be deducted depending on the problem and/or which features are missing.

SUPPORT FOR BASIC FUNCTIONS [10/80]

To get full marks for this part, you need to extend the basic solution with function calls. You do not need to support nested and recursive calls. However, you need to implement a suitable register based protocol where `$a0-$a3` should be used for arguments and $v0 for the result. You can assume max 4 arguments. Remember, you should use the `$t0-$t7` registers for local variables (except for `Main` where `$s0-$s7` are used).

ADDITIONAL FEATURES [10/80]

For the final part you can choose between the listed additional features below. Each of them are weighted at 10, however partial to solution to two of them will be accepted. Further details for this part may be announced during the course. If you would like to implement other features than listed here instead, then please discuss this with the course leader.

SUPPORT FOR NESTED AND RECURSIVE FUNCTIONS

Extend functions to allow recursive and nested calls. Here you need to develop a stack *frame*. As all variables are in registers, (when required) you will need to push a stack frame with all caller-save registers before the call is made and pop them back to the correct registers when control is returned to the caller.

SUPPORT FOR ARRAYS

Add support for arrays. Arrays should be stored in the static area and only used within the `Main()` function. Note that the syntax of FUNC does not allow passing arrays as arguments and returning arrays. You should support the syntax detailed in the `SOURCE LANGUAGE' section of this document. The following example illustrates use of an array:

```
function Main () is
 a : Array of size 2;
begin
  a[0] := 1;
  a[1] := Plus(a[0],1);
end Main;
```

OPTIMISATIONS

Implement the following optimisations:

- Unnecessary PUSH and POP operation
- Unnecessary movement from registers
- At least 3 algebraic laws

FURTHER DETAILS

IMPLEMENTATION LANGUAGE

In the lecture notes we have been using C as an implementation language. You are free to use a different language, as long as this is approved by the course leader. However, note that you are unlikely to get as much help if you chose to use a language different from C. One requirement of the language is that it has support for a lexical analyser compare to FLEX. At least the following languages support a FLEX variant:

- Java: jLex, JFlex (may be others too)
- ML: ml-lex
- Haskel: alex (again, there may be others)

SUBMISSION OF SOURCE CODE ON VISION

In addition to the demos during the labs, the source code needs to be submitted for each part with the same deadline specified. This should be submitted on Vision under `Assessment'. Late submissions may be penalised.

LAB AND LECTURE PLAN

- Week 6: You should be able to do the FLEX file and parser.
- Week 7: We have covered **all the materials in the lectures to complete part 1**.
- **Week 8:** Demo of **part 1** (we will also use Thursday morning lab slot). You should be able to start with the code generator for part 2 this week.
- Week 9: We have covered **all the materials in the lectures to complete part 2**.
- Week 10: Special industrial lab.
- **Week 11:** Demo of **part 2** (we will also use Thursday morning lab slot).

PLAGIARISM

You are being assessed on individual work. You should make sure you are familiar with the university plagiarism regulations

http://www.hw.ac.uk/students/doc/plagiarismguide.pdf

You should not use illegal software or copyrighted materials. You should acknowledge the source of material you have adapted, as well as help you have received from other students and staff.

LATE SUBMISSION & EXTENSION OF COURSEWORK

Students who have serious concerns about meeting submission dates for coursework should consult the Course Leader as soon as possible. Any extension to the submission deadline must be approved by the Course Leader, and the reason for the extension will be recorded. Applications for extensions made after the due submission date will not normally be approved.

For late submissions/demos the Course Leader will apply a penalty mark deduction on a case-by-case basis as an alternative time slot for the demo session needs to be found.