CS39003 Compilers Laboratory, Autumn 2024–2025
Assignment No: 5
Date: 14-Oct-2024

Intermediate Code Generation

This assignment deals with a simulation of a computing environment to run a sequence of three-address instructions. We use the same expression grammar as in Assignment 3. The grammar is reproduced below.

PROGRAM	⟶	STMT PROGRAM | STMT
STMT	⟶	SETSTMT | EXPRSTMT
SETSTMT	⟶	**(set** ID NUM **)** | **(set** ID ID **)** | **(set** ID EXPR **)**
EXPRSTMT	⟶	EXPR
EXPR	⟶	**(** OP ARG ARG **)**
OP	⟶	**+** | **–** | ***** | **/** | **%** | ******
ARG	⟶	**ID** | **NUM** | EXPR

All operators are assumed to be binary and to work on signed integer operands, and ** is the exponentiation operator.

This time, you do not prepare any parse tree or expression tree. Instead, the parser will output a file `intcode.c` (should be a C file) consisting of a sequence of three-address instructions. The main( ) function of `intcode.c` simulates the working of a computer. It starts by declaring two arrays: `R[]` and `MEM[]`. `R[]` is capable of storing exactly 12 integers, and simulates the registers available in the CPU. `MEM[]`, on the other hand, is a model of the main memory, and is capable of storing a large number (like 65536) of integers. The registers `R[0]` and `R[1]` are used for fetching memory-resident operands (and sometimes for storing the results of arithmetic operations). The other registers `R[2]`, `R[3]`, . . . , `R[11]` are used to store the temporaries. If more than ten temporaries are needed during some expression evaluation, then `MEM[]` is used to store them.

After the declarations of the two arrays, `intcode.c` stores the three-address instructions. Assume that our CPU cannot directly work on a memory-resident operand, so such an operand must be first fetched to a register. Likewise, the result of an arithmetic expression can only be stored in a register, and subsequently moved to memory if required. Consider the set statement:

```
(set c (+ a b))
```

Assuming that a, b, c reside at memory locations 5, 8, and 6 respectively, the 3-address code for the set statement can be:

```
R[0] = MEM[5];
R[1] = MEM[8];
R[2] = R[0] + R[1];
MEM[6] = R[2];
```

A standalone expression like

```
(* (+ a 6) (- 15 b))
```

can have intermediate code as follows.

```
R[0] = MEM[5];
R[2] = R[0] + 6;
R[0] = MEM[8];
R[3] = 15 - R[0];
R[2] = R[2] * R[3];
```

Exponentiation is not available as a basic operator in C/C++. Moreover, you need to check the correctness of your intermediate code. For these, assume that we have the following three-address instructions.

`pwr(arg1,arg2)`	Compute `arg1 ^ arg2`
`mprn(MEM,idx)`	Print `MEM[idx]`
`eprn(R,idx)`	Print `R[idx]`

Implement these functions in a C file `aux.c`, and #include this file from `intcode.c`.

Assume that each temporary is used only once. Therefore if some register stores a temporary, then that register can be reused immediately after that temporary is used. This is illustrated in an earlier example:

```
R[2] = R[2] + R[3];
```

However, if all the registers `R[2]` through `R[11]` already store temporaries, and we need to store another temporary, then we compute the new temporary in R[0], and move it to the next available `MEM` location.

```
R[0] = MEM[18];
R[0] = R[7] / R[0];
MEM[31] = R[0];
```

Later, when that temporary is to be used, that MEM entry is again fetched to `R[0]` or `R[1]` as illustrated below. Assume that at that time R[9] is available to store a new temporary.

```
R[0] = MEM[25];
R[1] = MEM[31];
R[9] = R[0] – R[1];
```

After this, we may reuse `MEM[31]`. For simplicity, you do not have to do that because `MEM[]` is a large array. Note also that a memory-resident temporary is not brought to a register as soon as some register is free to accommodate a new temporary. It is fetched only when it is used. In the last example, we assumed that some register like `R[9]` was free. If not, `R[0]` will store the result for sending to the next available memory location.

A sample input file and the corresponding intermediate-code file `intcode.c` are supplied at the end of this document. A larger sample will be provided as an external file in the assignment website.

Files you should prepare

`expr.l`	This is the lex file that is supposed to return the tokens from the input. In our example, the tokens are: **set** (Keyword) **ID** (Follow the same naming conventions as in C) **NUM** (signed integers) **+ – * / % **** (Arithmetic operators) **( )** (Punctuation)
`expr.y`	This is the yacc/bison file. Do not implement any function in this file. Maintain a symbol table for the ID's. For each ID, the symbol table will store the following two items. The name of the variable (a string) The offset of the variable in `MEM[]` You **<u>do not store</u>** the value of the variables in the symbol table. Note that the symbol table does not exist when the program `intcode.c` runs. So the output intermediate code cannot access variables by names. The offsets of the variables in `MEM[]` are explicitly specified in the intermediate code. For example, in the sample at the end of the document, the input file talks about the variables `a`, `b`, `c`, and so on. The intermediate-code file talks about `MEM[0]`, `MEM[1]`, `MEM[2]`, and so on. In reality, variables of different types and widths are stored in the memory, and their offsets are usually specified in bytes (or words). For simplicity, this assignment deals only with integer variables, and so `MEM[]` is modeled as an int array. You may use the index in this array as the offset of a variable. Note finally that you may need to store temporaries in the memory. You may adopt some special naming convention for the temporaries like `$1`, `$2`, `$3`, . . . , so that these names do not clash with the names of user-defined variables. The symbol table should store these special names for memory-resident temporaries.
`intcodegen.c` / `intcodegen.cpp`	In this file, implement all functions needed for parsing. In particular, these functions will incrementally write the three-address instructions to the output file `intcode.c`. The file `intcodegen.c` should have a main() function. When it is compiled and run, the file `intcode.c` will be generated.

	The register and memory arrays do not exist for the parsing code `intcodegen.c`(pp). These arrays exist in the intermediate-code file `intcode.c`. The parsing code should only refer to indices in these arrays. The actual data storage and movement in these arrays happen when `intcode.c` will run.
aux.c	In this file, implement the three functions pwr(), mprn(), and eprn() as explained earlier.
sample.txt	You may use the sample file at the end of this document. A larger sample file checking all the functionalities can be downloaded from the assignment website.
makefile	Write a makefile with the usual targets: all Run yacc and flex, and compile `intcodegen.c` to an executable file `icgen`. run Run `icgen`. This will generate `intcode.c`. Compile this to an executable file `ic`. Run `ic`. clean Remove all files created by make all and make run (including `intcode.c`, `icgen`, and `ic`).

Pack the above six files in a zip/tar/tgz archive, and submit that single file.

A small sample

Input file `sample.txt`

```
(set a 5)
(set b a)
(set c (+ (* a b) (+ a b)))
(+ (* a a)
   (** a b)
)
(set a 2)
(set b 9)
(/ (+ (** a 2) (** b 2))
   (+ (* a b) -1)
)
(set tmp b)
(set b (- (* 5 b) a))
(set a tmp)
(/ (+ (** a 2) (** b 2))
   (+ (* a b) -1)
)
( +
  4
  ( +
    ( *
      ( +
        ( *
          14
          (+ 6 (+ 12 (* 2 16)))
        )
        9
      )
      ( *
        (+ (* 11 ( + 3 21)) (* 15 (* 8 13)))
        19
      )
    )
    ( %
      (+
         (+ 1 (* 10 (+ 7 17)))
         20
      )
      (* 5 18)
    )
  )
)
```

Parser output `intcode.c`

```
#include <stdio.h>
#include <stdlib.h>
#include "aux.c"

int main ( )
{
   int R[12];
   int MEM[65536];

   MEM[0] = 5;
   mprn(MEM,0);
   R[0] = MEM[0];
   MEM[1] = R[0];
   mprn(MEM,1);
   R[0] = MEM[0];
   R[1] = MEM[1];
   R[2] = R[0] * R[1];
   R[0] = MEM[0];
   R[1] = MEM[1];
   R[3] = R[0] + R[1];
   R[2] = R[2] + R[3];
   MEM[2] = R[2];
   mprn(MEM,2);
   R[0] = MEM[0];
   R[1] = MEM[0];
   R[2] = R[0] * R[1];
   R[0] = MEM[0];
   R[1] = MEM[1];
   R[3] = pwr(R[0],R[1]);
   R[2] = R[2] + R[3];
   eprn(R,2);
   MEM[0] = 2;
   mprn(MEM,0);
   MEM[1] = 9;
   mprn(MEM,1);
   R[0] = MEM[0];
   R[2] = pwr(R[0],2);
   R[0] = MEM[1];
   R[3] = pwr(R[0],2);
   R[2] = R[2] + R[3];
   R[0] = MEM[0];
   R[1] = MEM[1];
   R[3] = R[0] * R[1];
   R[3] = R[3] + -1;
   R[2] = R[2] / R[3];
   eprn(R,2);
   R[0] = MEM[1];
   MEM[3] = R[0];
   mprn(MEM,3);
   R[0] = MEM[1];
   R[2] = 5 * R[0];
   R[0] = MEM[0];
   R[2] = R[2] - R[0];
   MEM[1] = R[2];
   mprn(MEM,1);
   R[0] = MEM[3];
   MEM[0] = R[0];
   mprn(MEM,0);
   R[0] = MEM[0];
   R[2] = pwr(R[0],2);
   R[0] = MEM[1];
   R[3] = pwr(R[0],2);
   R[2] = R[2] + R[3];
```

Output of `intcode.c`

```
+++ MEM[0] set to 5
+++ MEM[1] set to 5
+++ MEM[2] set to 35
+++ Standalone expression evaluates to 3150
+++ MEM[0] set to 2
+++ MEM[1] set to 9
+++ Standalone expression evaluates to 5
+++ MEM[3] set to 9
+++ MEM[1] set to 43
+++ MEM[0] set to 9
+++ Standalone expression evaluates to 5
+++ Standalone expression evaluates to 24571189
```

	<pre> R[0] = MEM[0];
	   R[1] = MEM[1];
	   R[3] = R[0] * R[1];
	   R[3] = R[3] + -1;
	   R[2] = R[2] / R[3];
	   eprn(R,2);
	   R[2] = 2 * 16;
	   R[2] = 12 + R[2];
	   R[2] = 6 + R[2];
	   R[2] = 14 * R[2];
	   R[2] = R[2] + 9;
	   R[3] = 3 + 21;
	   R[3] = 11 * R[3];
	   R[4] = 8 * 13;
	   R[4] = 15 * R[4];
	   R[3] = R[3] + R[4];
	   R[3] = R[3] * 19;
	   R[2] = R[2] * R[3];
	   R[3] = 7 + 17;
	   R[3] = 10 * R[3];
	   R[3] = 1 + R[3];
	   R[3] = R[3] + 20;
	   R[4] = 5 * 18;
	   R[3] = R[3] % R[4];
	   R[2] = R[2] + R[3];
	   R[2] = 4 + R[2];
	   eprn(R,2);
	
	   exit(0);
	}</pre>	