CS3300 - Compiler Design (July - Nov 2024)
(Assignment - 3)

Due **Oct 27 [23:59]** on moodle

Instructor: Prof. Rupesh Nasre

Description

In this assignment, Your task is to translate a Program written in MiniC (a Subset of C language) to x86 assembly code using Flex-Bison/Lex-Yacc. Use A2 code to help you generate the TAC code for the input MiniC program. You are expected to first translate the input program to TAC and then translate it to x86 assembly.

1 Problem Statement

Write one (or two) Flex-Bison / Lex-Yacc programs to parse the input program written in MiniC and generate the output code in x86 Assembly. The exact specification of MiniC is given below. The requirement of the output code is that when run, the output of the MiniC code (using gcc) and the output of the generated Assembly code (using assembler) must be exactly same. The input will be a syntactically and semantically valid MiniC Program. A zip file `minic.zip` containing three files `miniC.y, miniC.l` and `Makefile` are attached with this assignment to help you parse the input MiniC code. `SampleTCs.zip` is also attached for reference.

2 MiniC Grammar

In the Grammar Specification, We follow the following conventions:

- **A?** indicates that non-terminal A is optional, It may or may not be present.
- **A*** means that non-terminal A may repeat for any number of times (including 0).
- <**IDENTIFIER**> refers to any valid identifier in C
- <**INTEGER_LITERAL**> is a constant integer literal in C
- <**STRING_LITERAL**> refers to a potentially formatted string in C (enclosed in quotes).
- <**CONSTANT_CHAR_LITERAL**> refers to a constant character literal in C (E.g: $'A','\$' \cdots$ etc)

MiniC doesn't support exponentiation operator (∗∗). There is no for-loop as well.

2.1 Grammar for MiniC

```
Program                               ::=  "#include < stdio.h > " (VariableDeclaration)*
                                           (FunctionDefinition)*  Main
VariableDeclaration                   ::=  IntDeclaration
                                        |  CharArrayDeclaration
IntDeclaration                        ::=  "int"  Identifier  ";"
CharArrayDeclaration                  ::=  "char"  Identifier  "["  IntegerLiteral  "]"  ";"
FunctionDefinition                    ::=  "int" Identifier "(" (FormalParameterList)? ")" FunctionBody
FunctionBody                          ::=  "{" (VariableDeclaration)* (Statement)*  "return" Expression ";" "}
FormalParameterList                   ::=  FormalParameter  ( FormalParameterRest )*
FormalParameter                       ::=  IntParameter
                                        |  CharArrayParameter
IntParameter                          ::=  "int"  Identifier
CharArrayParameter                    ::=  "char"  Identifier  "["  "]"
FormalParameterRest                   ::=  ","  FormalParameter
Main                                  ::=  "int"  "main"  "("  ")"  MainBody
MainBody                              ::=  "{"  (VariableDeclaration)*  (Statement)*  "return"  "0"  ";"  "}"
Statement                             ::=  AssignmentStatement
                                        |  CharArrayIndexAssignmentStatement
                                        |  IfStatement
                                        |  WhileStatement
                                        |  FunctionCallStatement
                                        |  PrintStatement
AssignmentStatement                   ::=  Identifier  "="  Expression  ";"
CharArrayIndexAssignmentStatement     ::=  Identifier "[" Expression "]" "=" ConstantChar ";"
                                        |  Identifier "[" Expression "]" "=" Identifier "[" Expression  "]"";"
PrintStatement                        ::=  "printf"  "("  FormattedStringWithQuotes  (PrintfArgument)*
                                           ")"  ";"
PrintfArgument                        ::=  ","  Expression
IfStatement                           ::=  IfthenElseStatement
                                        |  IfthenStatement
IfthenStatement                       ::=  "if"  "("  BooleanExpression  ")"  "{"  (Statement)*  "}"
IfthenElseStatement                   ::=  "if"  "("  BooleanExpression  ")"  "{"  (Statement)*  "}"  "else"
                                           "{"  (Statement)*  "}"
```

```
WhileStatement                 ::= ”while” ”(” BooleanExpression ”)” ”{” (Statement)* ”}”
FunctionCallStatement          ::= Identifier ”(” (ArgumentList)? ”)” ”;”
ArgumentList                   ::= Argument (ArgumentRest)*
Argument                       ::= Expression
ArgumentRest                   ::= ”,” Argument
CharArray                      ::= Identifier
BooleanExpression              ::= BooleanOrExpressionList
BooleanOrExpressionList        ::= BooleanAndExpressionList
                                |  BooleanAndExpressionList "||" BooleanOrExpressionList
BooleanAndExpressionList       ::= BooleanPrimaryExpression
                                |  BooleanPrimaryExpression "&&" BooleanAndExpressionList
BooleanPrimaryExpression       ::= Expression RelationOperator Expression
                                |  ”(” BooleanExpression ”)”
                                |  ”!” ”(” BooleanExpression ”)”
RelationOperator               ::= ”<=”
                                |  ”==”
                                |  ”>=”
                                |  ”!=”
                                |  ”>”
                                |  ”<”
Expression                     ::= ReturnOfFunctionCall
                                |  BinaryOperation
                                |  PrimaryExpression
PrimaryExpression              ::= ”(” Expression ”)”
                                |  Identifier
                                |  IntegerLiteral
ReturnOfFunctionCall           ::= Identifier ”(” (ArgumentList)? ”)”
BinaryOperation                ::= PrimaryExpression BinaryOperator PrimaryExpression
BinaryOperator                 ::= ”+”
                                |  ”-”
                                |  ”/”
                                |  ”*”
IntegerLiteral                 ::= <INTEGER_LITERAL>
Identifier                     ::= <IDENTIFIER>
FormattedStringWithQuotes      ::= <STRING_LITERAL>
ConstantChar                   ::= <CONSTANT_CHAR_LITERAL>
```

2.2 Detailed Description of MiniC

Observe that MiniC is just a subset of C language created for this assignment. In fact MiniC is a minor modification of the input program in A2. Here is a detailed description of MiniC Language(Highlighted parts refer to modifications from A2):

- There is only one header, namely `#include<stdio.h>` followed by zero or more global declarations that are followed by zero or more definitions of int-returning functions followed by the main function. (`int main()` is always present)
- Every function will have declarations (if any) only at the start of the function. Any function may call another function only if it is defined above. Every function has a single return statement which returns an integer expression. All control paths will lead to this statement.
- MiniC only supports `int` and `char[]` data types. Functions, if they take any arguments, can be of both `int` and `char[]` types. `char[]` type is only used for printing. (No Expressions)
- Statements in MiniC can be either an Assignment, Char Array Assignment, If Statement, While Loop, A function call or Print statement. Minic supports Unary plus and minus .
- The body of If-statements and While-Loops will always be enclosed in curly brackets.
- RHS of any char array index assignment will be a valid C `const char` literal (Like 'A') or another char array index. RHS of any Integer Variable Assignment can be any complex expression defined by the grammar using the given operators.
- Operator precedence is same as that of C language.There is no expression with mixing of `int` and `char[]` types. Every Expression will be well nested with brackets.
- Conditions will only be present in If-Statements and While-Loop. Brackets and Logical NOT (!) has higher precedence than Logical AND (`&&`). Logical OR (`||`) has the least precedence.
- Relational operators are supported only for integer expressions. No Char Array comparison or index comparison is supported. There will be no need for type casting.
- MiniC follows short-circuit evaluation. In case of multiple ORs, The evaluation is done from left to right until a boolean which evaluates to 1 is found. (**For e.g:** Let $b1 = 0, b2 = 0, b3 = 1, b4 = 1$. Now in the boolean expression `(b1 || b2 || b3 || b4)`, the evaluation stops at $b3$ and $b4$ is not evaluated). A similar extension is true in case of Logical AND. **(Note that the above example is illustrative and not the exact syntax)**.
- Whenever a variable is accessed, the most local scope is used. This becomes relevant when there are nested function calls. A detailed example is illustrated in one of the sample TCs.
- Every Function name will be guaranteed to be unique. Global variables can be modified inside functions . Variable names might be the same across functions (Similar to A2).
- MiniC supports both C like inline and multiline comments. (`//` and `/* */`)

3 x86 Assembly

We will be using the x86 Assembly syntax used by the GAS (GNU Assembler) for the 32 bit machines.

3.1 Registers

There are 2 types of registers available in 32 bit machines

- **General Purpose Registers**: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
- **Special Registers**: 6 Segment registers, and EFLAGS register. EFLAGS is a 32 bit register which has different flags inside it, modified by various instructions.

Note: You may not have to use all the registers for this assignment.

3.2 Instruction Set for the Output x86

Given the MiniC subset, the following 32-bit x86 instructions should suffice:

- Data Movement: `movl`, `leal`.
- Arithmetic: `addl`, `subl`, `imull`, `idivl`.
- Logical: `andl`, `orl`, `notl`.
- Control flow: `call`, `ret`
- Jumps: `je`, `jne`, `jg`, `jl`, `jge`, `jle`, `jmp`
- Comparison: `cmpl`
- Stack: `pushl`, `popl`
- Data (Global): `.data`, `.bss`, `.text`
- Syscall/Library instructions: `printf()` call from the `stdio.h` library
- Comments are similar to python, hashtag (#) followed by text.
- All the instructions with suffix `l` are for 32-bit values (4 bytes, usually ints). Allowed suffixes are
  - `b` (byte = 8) (for char),
  - `w` (word = 16),
  - `l` (long = 32),
  - `q` (quad = 64) (used for 64 bit addresses in x86-64 arch).

Note: You may not require other suffixes for this assignment.

3.3 Globals

3.3.1 Declaration

Globals are defined in the `.bss` section of the assembly code. `.space <space>` (4 byes for int) is used to allocate space for it.

```
int x;
```

is converted to

```
    .bss
x:  .space 4
```

3.3.2 Usage

Globals declared can be used using the label it is declared with.

```
    .globl main
main:
    ...
    movl $5, x
    movl x, %eax
    ...
```

3.3.3 Const strings

The `char []` declared globally or the format string used in the printf function, is processed as a global const string by the assembly in the `.data` section. The example is as follows.

```
    .data
fmt:    .asciz "Sum is %d\n"
```

For global `char []`, the characters can be mutated by the following instruction.

```
    .bss
mystring:   .space 20
    ...
    .text
    ...
    movb $'B', mystring
    movb $'y', mystring+1
    movb $'e', mystring+2
    ...
```

3.4 Local Variables

3.4.1 Integers

Space required by the local variables are allocated in the function's prologue. The variable locations are accessed using (negative) offsets from `%ebp`. The following example illustrates this.

```
    .globl main
main:
    pushl %ebp
    movl %esp, %ebp
    sub $N, %esp         # Allocate space for local variables

    movl  $5, -4(%ebp)   # x = 5
    movl  $10, -8(%ebp)  # y = 10
    ...

    movl %ebp, %esp
    popl %ebp
    # leave              # leave instr is same as mov %ebp,%esp; pop %ebp
    ret
```

3.4.2 Character Arrays

Allocate space in stack for the character array. Use movb for individual character (byte) assignments or copy, and make sure you terminate it with a null character.

```
    char mystring[10];
    mystring[0] = 'H';
    mystring[1] = 'E';
    mystring[2] = 'L';
    mystring[3] = 'L';
    mystring[4] = 'O';
    mystring[5] = '\0';

    readstring(mystring);   //int readstring (char[] string);
```

The corresponding assembly code is given below:

```
    subl $20, %esp      #Space allocated in the function prologue

    # Properly initialize ’message’ with "Hello"
    movb    $’H’, -20(%ebp)
    movb    $’e’, -19(%ebp)
    movb    $’l’, -18(%ebp)
    movb    $’l’, -17(%ebp)
    movb    $’o’, -16(%ebp)
    movb    $0, -15(%ebp)       # Null terminator for the string

    leal    -20(%ebp),%eax      # Push address of ’message’
    pushl   %eax
    call    printMessage        # Call printMessage(message)
```

3.5 Conditionals and Loops

If-else and While statements can be translated to assembly similar to the 3AC, using jumps and labels. Labels are declared similar to the function decls (just without the `.globl`). Example:

```
if (x>y){
    x = y + 5;
}
else{
    y = x + 5;
}
```

Equivalently in assembly is

```
        movl    -12(%ebp), %eax       #   (ebp-12) contains x
        cmpl    -16(%ebp), %eax       #   (ebp-16) contains y
        jle     .L4
        movl    -16(%ebp), %eax
        addl    $5, %eax
        movl    %eax, -12(%ebp)
        jmp     .L5
.L4:
        movl    -12(%ebp), %eax
        addl    $5, %eax
        movl    %eax, -16(%ebp)
.L5:
```

Support short-circuiting, in the same way as in assignment 2. While loop can also be structured similarly using conditional and unconditional jumps.

3.6 Function Declaration

Specify `.globl <func_name>` before the start of the function, and start the function with a label (preferably of the same name as the function). If it is the first function declared, start preceed it with a `.text` section, which highlights the start of the code. The below MiniC function

```
int add (int x, int y)
{
    ...
    return ...;
}
```

can be converted to x86 as:

```
    .text
    .globl add
add:
    ...                  #prologue
    # Use 8(%ebp) for first arg
    # Use 12(%ebp) for second arg
    # ...   # 4(%ebp) stores the return address
    ...                  #epilogue
```

3.6.1 Function Prologue and Epilogue

You have to allocate stack space for a function when called (for the its own local variables, return address and other operations during function call). The stack space allocated has to be cleaned up on return of a function. The following example illustrates this.

```
    .text
    .globl add
add:
    pushl %ebp         # Save base pointer
    movl %esp, %ebp    # Set new base pointer
    sub $N, %esp       # N Bytes space for local variable
    ...
    movl %ebp, %esp    # Restore the stack pointer
    popl %ebp          # Restore the caller's base pointer
    # leave            # leave instr is same as mov %ebp,%esp; pop %ebp
    ret                # Return to caller
```

3.7 Function calls (and stack management)

In the 32-bit x86 architecture function calls follow the cdecl (C declaration) calling convention. The summary is

- The args for the function are passed using the stack, which are pushed in the right to left order before the function is called.
- The function call is done using `call <func_label_name>` statement.
- The callee retrieves these args from the stack for it's execution.
- The return value is stored in `eax` register.
- The caller cleans up the stack space used by the args.

1) Integer arguments

```
    # Call add(x, y)
    pushl -8(%ebp)              # Push y
    pushl -4(%ebp)              # Push x
    call add                    # Call add function
    addl  $8, %esp              # Clean up args from stack
```

2) Arguments which are char[] (using const strings)

```
    pushl   -12(%ebp)           # Push second arg for printf
    pushl   $fmt                # Push address of fmt as first arg
                                # fmt = "The value is %d\n"
    call    printf              # Call printf
    addl    $8, %esp            # Clean up the stack (2 arg)
```

4 Recommended Approach and Guidelines

4.1 Generating TAC

- Inorder to approach this assignment, It is highly recommended to first parse the MiniC code and generate the TAC code.
- Note that **it is not mandatory to use the given starter codes**. You can modify your own submission of A2 to parse MiniC code.
- Also note that it is not necessary to **completely** change the grammar rules of .y file of A2 to the grammar rules of MiniC given, The purpose of complete specification is to guarantee that every input program will definitely satisfy the grammar specification given. Slightly modifying the existing .y file of A2 would suffice.

4.2 Converting TAC to Assembly

Once we have generated the TAC code, we try to substitute statements of TAC to sequence of x86 instructions having same functionality. The complete instruction set of x86 is very vast and **not all** instructions are required. Identify the specific x86 instructions corresponding to TAC statements generated from MiniC.

- Some of the challenges in translation to assembly are **Mapping of variables to stack locations** and **Function Calls**.
- Notice that there is only a small finite set of registers available, but TAC contains unbounded number of variables and temporaries. So, we need to store the variables and temporaries in the stack memory and access/modify them at their specific stack locations.
- In Function call, a new scope is created where there might be definitions and usage of variables having same name in the parent scope. But once the function call is complete, the old values must be preserved. To implement this, before a function call is executed, The old values are stored in stack memory, and restored back once the call is complete. (See sample testcases).
- We also need to evaluate and store the parameters in the stack for the function to use. Fix a stack frame structure to decide where the number of arguments, old register values, parameter values, return address etc need to be stored. Use the same structure to access the parameters while executing the function call and restore the original stack content after function call is complete.
- The responsibility of these tasks is divided between the caller (where the function is called) and the callee (the function being called). Fix on some division and follow it throughout the assignment.

5 Submission Rules

- Create a new directory and rename it to your roll number, in the format CSXXBXXX (e.g., CS22B001).
- The directory must **only** contain a makefile and a maximum of two sets of Flex-Bison/Lex-Yacc files (one set of files converting MiniC to x86 is also allowed).
- Files must be named as **tac.y**, **tac.l** (For TAC generation) and **a3.y**, **a3.l** (For x86 generation)
- Other files (.yy.c, .out, .exe, $\cdots$ etc) **must not** be present.
- The final executable (which converts .c to .s) must be named **a.out**.
- Make sure that your code works with the tester bash script (We will provide this in few days). The same script will be used for evaluation (With different Test Cases :)).

- Zip the directory using the following command:
  `zip CSXXBXXX.zip -r CSXXBXXX`
- Unzipping CSXXBXXX.zip must give a folder CSXXBXXX containing only source files. Please ensure this by downloading your submission in moodle and unzipping after submission.
- Failure to adhere to Submission Guidelines might lead to tester script failure (0 marks). To avoid this, please double check your submission.

6 FAQs

1. **Why should I do this assignment?**

   Doing this assignment will help you understand and gain some hands on experience on using the high level structure of TAC to generate low level assembly code. On a side note, This assignment has a weightage of 8 percent of course total which is for **15** credits.

2. **How long should I devote for this assignment?**

   In terms of difficulty, it is not more difficult than the previous assignment, if not easier. Try to write code incrementally and test it at each step. Try to create your own testcases.

3. **What should I do when I have queries regarding the assignment?**

   Please post your queries in moodle discussion page. So that it is uniform and may help others who share the same query. Please refrain from DMs and personal mails.

4. **The assignment seems daunting. The problem statement has lot of pages. Is it doable?**

   The problem statement is lengthy to describe the input format and expected approach clearly. As you can see, most specifications have already been mentioned in previous assignments.

5. **My generated assembly code has lot of redundant label jumps compared to the expected output. In general my code is very different from expected output. Is it fine?**

   Yes. It is completely fine as long as it implements the exact semantic evaluation of MiniC code. In other words, when run, both of the codes must produce the same output.

6. **Since the input is a C file, Can I simply use a de-assembler to get the assembly code directly? Can I hard-code the given example TCs?**

   Absolutely Not. If we catch anyone using any kind of in-built assembly converter (using gcc -S) or de-assembler (like *objdump*) either in Makefile or in the flex-bison files (disguised as system call or in any form), He/She will be awarded **0** marks. Any kind of unfair means will be reported to Prof. Rupesh. No hard-coding of example TCs is allowed.

7. **Is the input grammar for A2 and A3 the same?**

   No. MiniC is an *almost* proper subset of the input grammar for A2. The major differences are highlighted in yellow . There are some very minor differences as well. The exact grammar specification of MiniC is given so that there is no confusion on what could be valid input.

8. **Will there be any unpredictable behavior in the input code? Like out of bound access or use of uninitialized variables?**

   No. We will only have well-formed, valid MiniC input code.

9. **How many testcases will be used for evaluation?**

   There will be **20** public testcases (released shortly), and **20** private testcases (released after you get your marks). All testcases are of equal weightage.

10. **How to test if the output assembly file is syntactically and semantically correct?**

    You can login in the following DCF machines having IP addresses: **10.21.225.111** and **10.21.225.112.** (using ssh via iitmwifi/ Institute LAN) and test your output code with these commands:

    ```
            $ gcc -m32 output.s -o output
            $ ./output
    ```

11. **What all constant character literals should I support?**

    You need to support all single character literals (’a’,’b’,’#’) and some escaped characters as well(’\0’,’\n’,’\\’). Check the lexer file of the starter code on what all characters to support.

12. **Will there be variable names in the input file which are keywords in either TAC or x86 assembly?**

    No, we will make sure that the variable names in miniC input are not keywords in either miniC, the TAC from A2 or the x86 assembly code to be generated. Any issues in this regard can be brought up to the TAs, and suitable modifications will be done.

13. **Should I do any kind of register allocation for this assignment?**

    No, You can use the stack space to store the values of **all** the variables/temporaries and use/modify them by loading/storing using registers whenever necessary.

14. **Should I keep checking the problem statement frequently for updates?**

Yes. Over time, we may compile some of your classmates' important doubts in this FAQ section for your quick reference. There may be minor changes to the problem statement, but they will be highlighted appropriately, and a proper announcement will also be given in Moodle.