Department of Computer Science
Ashoka University

Programming Language Design and Implementation (PLDI): CS-1319-1

Assignment - 4: `TAC Generator` $\mathcal{F}_{-15}$ *Marks: 100*
Assign Date: *October 18, 2024* Submit Date: *23:55, November 14, 2024*

---

1. *You must submit your assignment using the prefix* `name = FirstName_LastName` *for all files.*
2. *To receive full credit, your program must be correct and your* `.pdf` *file must explain your program adequately.*

---

1 Specification of femtoFORTRAN or $\mathcal{F}_{-15}$

Follow the specification of $\mathcal{F}_{-15}$ as provided in Assignment 2 with the following correction in the semantics of DO loop:

Change: *Iterations continue as long as the index remains smaller than the upper bound*
to: *Iterations continue as long as the index remains smaller than or equal to the upper bound*

2 The 3-Address Code

Use the *3-Address Code* as the *Intermediate Representation* where every 3-Address Code:

- Uses only up to 3 addresses.
- Is represented by a `quad`: opcode, argument 1, argument 2, and result; where argument 2 is optional. In some cases, both arguments are optional.

2.1 Address Types

- *Name*: Source program names (variables) appear as addresses in 3-Address Codes.
- *Constant*: Following types of constants are allowed as deemed addresses:
  - `INTEGER` of 32 bits (4 bytes)
  - `string` as placeholder pointer of 4 bytes
- *Compiler-Generated Temporary*: Create a distinct name each time a temporary is needed.
- *Labels*: Optionally mark positions of 3 address instructions.

2.2 Instruction Types

For Addresses `x`, `y`, `z`, and Label `L`

- *Binary Assignment Instruction*: For a binary (arithmetic) `op` (`+ -`): `x = y op z`
- *Copy Assignment Instruction*: `x = y`
- *Unconditional Jump*: `goto L`
- *Conditional Jump*: For a relational operator `relop` (`<`, `>`, `==`, `!=`, `<=`, `>=`): `if x relop y goto L`.
  - *Branching Control*: For an `IF` branching, we use Conditional Jump to translate: For example,

```
IF (x .LT. y) THEN
    ! THEN CODE
ELSE
    ! ELSE CODE
END IF
```

translates to:

```
IF (x < y) goto L1
    goto L2
L1: ! THEN CODE
    goto L3
L2: ! ELSE CODE
L3: ! CODE AFTER IF
```

- *Loop Control*: For a `DO` loop, we use Conditional and Unconditional Jump to translate: For example,

  ```
  DO i = E1, E2
      ! DO BODY
  END DO
  ```

  translates to:

  ```
      i = E1
  L1: if i <= E2 goto L2
      goto L3
  L2: ! DO BODY
      i = i + 1
      goto L1
  L3: ! CODE AFTER DO
  ```

- *I/O Instructions*:
  - *Input*: `in x` reads a single value into variable `x` from console
  - *Output*: `out x` writes the value of variable / constant `x` to console. Multiple `out`'s continue to write on the same line with single blank separation.
    To start writing on a new line, use `outnew x` or simply `outnew`.
- *End Instructions*: `end` to mark the end of the program.

3 Design of the Translator

1. *Language of Implementation*: You have a choice to code your translator in C or in C++.
2. *Size of Data Types*: To compute the offset and storage mapping of variables, use the following *sizes (in bytes) of types*:

Type	Size	Remarks
`INTEGER`	4	*Variables and constants need to be aligned at addresses divisible by 4*
`string`	...	*Number of characters in the string constant (one byte / character)*
		These need to be aligned at addresses divisible by 4
`char *`	4	*Pointer to string literals*

   Since, using hard-coded sizes for types does not keep the code machine-independent, you may want to use constants for sizes that can be defined at the time of machine-dependent targeting. For example,

   ```
   const unsigned int size_of_int = 4;
   const unsigned int size_of_str_ptr = 4;
   const unsigned int addr_alignment = 4;
   ```

3. *Lexer & Parser*: Use the Flex and Bison specifications (if required you may correct your specifications) you had developed in Assignments 2 and 3 respectively and write semantic actions for translation to 3-address Codes. Note that some grammar rules of your $\mathcal{F}_{-15}$ parser may not have any action or may just have propagate-only actions. Also, some of the lexical tokens may not be used.
4. *Augmentation*: Augment the grammar rules with markers and add new grammar rules as needed for the intended semantic actions. Justify your augmentation decisions within comments of the rules.

5. *Attributes*: Design the attributes for every grammar symbol (terminal as well as non-terminal). List the attributes against symbols (with brief justification) in comment on the top of your Bison specification file. Highlight the inherited attributes, if any.
6. *Symbol Table*: Use symbol table for user-defined variables and compiler-defined temporary. The symbol table should have at least the following fields (may have more).

Name	**Type**	**Category**	**Initial Value**	**Size**	**Offset**
...	...	...	...	...	...

Name is the lexeme or given temporary name of the symbol
Type is the data type of the symbol
Category is used-defined or compiler generated
Initial Value is the initial value, if any, of the symbol
Size is the size of the symbol in bytes
Offset is the number of bytes to the start of the symbol

The following methods may be supported for a Symbol Table:

`lookup(...)`	A method to lookup an `id` (given its name or lexeme) in the Symbol Table. If the `id` exists, the entry is returned, otherwise a new entry is created
`gentemp(...)`	A static method to generate a new temporary, insert it to the Symbol Table, and return a pointer to the entry
`update(...)`	A method to update different fields of an existing entry
`print(...)`	A method to print the Symbol Table in a suitable format. This is needed for debugging

Note:

- The fields and the methods are indicative. You may change their name, functionality and also add other fields and / or methods that you may need.
- It should be easy to extend the Symbol Table as further features are supported and more functionality is added.
- Only one symbol table is needed as there is no nested scope.

7. *String Table*: Use string table for string constants. The string table should have at least the following fields (may have more).

Name	**Value**	**Size**	**Offset**
...	...	...	...

Name is a pseudo name given to the string constant
A string has `string` type
Every string is constant
Value is the value (lexeme) of the string
Size is the # of characters of the string (in bytes)
Offset is the number of bytes to the start of the symbol. This is aligned to 4 bytes

8. Quad*Array*: The array to store the 3-address quad's. Index of a quad in the array is the *address* of the 3-address code. The quad array will have the following fields (having usual meanings)

op	**arg 1**	**arg 2**	**result**
...	...	...	...

Note:

- **arg 1** and / or **arg 2** may be a variable (address) or a constant.
- **result** is variable (address) only.
- **arg 2** may be null.

For example, in the context of `int i = 10; int a[10]; int v = 5;`, and `sizeof(int) = 4`:

```
do i = i - 1;
while (a[i] < v);
```

translates to

```
100: t1 = i - 1
101: i = t1
102: t2 = i * 4
103: t3 = a[t2]
104: if t3 < v goto 100
105:
```

where the quad's are represented as:

Index	op	arg 1	arg 2	result	Code in text
...	...	...	...	...	...
100	-	i	1	t1	`t1 = i - 1`
101	=	t1		i	`i = t1`
102	*	i	4	t2	`t2 = i * 4`
103	=[]	a	t2	t3	`t3 = a[t2]`
104	<	t3	v	100	`if t3 < v goto 100`

The following methods may be supported for Quad Array:

`emit(...)`	A static method to add a (newly generated) quad. Except `op`, one or more of the other parameters may be `null`:
	
	`emit(op, result = null, arg1 = null, arg2 = null):`
	
	No `null`: Binary `op` like `x = y + z`; `if x < y goto L`.
	`arg2 = null`: Unary `op` like `x = -y`; or Copy `x = y`
	`arg1 = null, arg2 = null`: Nullary `op` like `goto L`.
	`result = null, arg1 = null, arg2 = null`: Not an `op` like `end`.

`print(...)`	A method to print the quad array in a suitable format.

Note:

- The fields and the methods are indicative. You may change their name, functionality and also add other fields and / or methods that you may need.

9. *Global Functions*: Following (or similar) global functions and more may be needed to implement the semantic actions:

`makelist(l)` A function to create a new list containing only `l`, an index into the array of quad's, and to return a pointer to the newly created list.
`merge(p1, p2)` A function to concatenate two lists pointed to by `p1` and `p2` and to return a pointer to the concatenated list.
`backpatch(p, l)` A function to insert `l` as the target label for each of the quad's on the list pointed to by `p`.

Naturally, these are indicative and should be adopted as needed. For every function used clearly explain the input, the output, the algorithm, and the purpose with possible use at the top of the function.

4 Sample Translation

$\mathcal{F}_{-15}$ Program

```
! Program: To read a natural number n and
! print if it is even or odd

PROGRAM parity

INTEGER n
INTEGER i
INTEGER p

READ *, n

p = n
DO i = 1, n
    p = p - 2
    IF (p .EQ. 0) THEN
        PRINT *, n, "is even"
    ELSE
        IF (p .EQ. 1) THEN
            PRINT *, n, "is odd"
        END IF
    END IF
END DO

END PROGRAM parity
```

TAC Translation

```
100: in n
101: p = n
102: i = 1
103: t0 = n
104: if i <= t0 goto 106
105: goto 129
106: t1 = p
107: t2 = 2
108: p = t1 - t2
109: t3 = p
110: t4 = 0
111: if t3 == t4 goto 113
112: goto 118
113: t5 = n
114: out t5
115: t6 = t_str1
116: out t6
117: goto 127
118: t7 = p
119: t8 = 1
120: if t7 == t8 goto 122
121: goto 127
122: t9 = n
123: out t9
124: t10 = t_str2
125: out t10
126: goto 127
127: i = i + 1
129: goto 104
129: end
```

ST *Symbol Table for* parity

Name	Type	Category	Initial Value	Size	Offset
n	int	Variable	undefined	4	0
i	int	Variable	undefined	4	4
p	int	Variable	undefined	4	8
t0-t10	int	Temporary	undefined	4	12–52

- sizeof(INTEGER) = 4 as it is 32-bits

StrT *String Table for* parity

Name	Value	Size	Offset
t_str1	"is even"	7	0
t_str2	"is odd"	6	8

- **Size** is the # of characters of the string (in bytes)
- **Offset** is aligned to 4 bytes

5 The Assignment

In this assignment you will write the semantic actions in Bison to translate a $\mathcal{F}_{-15}$ program into an *array of 3-address* quad's, a supporting *symbol table*, and other *auxiliary data structures*. The translation should be machine-independent, yet it has to carry enough information so that you can later target it to a specific architecture (x86 / IA-32 / x86-64).

1. Write a 3-Address Code translator based on the Flex and Bison specifications of $\mathcal{F}_{-15}$. Assume that the input $\mathcal{F}_{-15}$ file is lexically, syntactically, and semantically correct. Hence no error handling and / or recovery is expected.
2. Prepare a Makefile to compile and test the project. Typing `make build` should compile your program and output an executable named `translator`. Use `gcc` to compile C code and `g++` to compile C++ code in your final submission.
3. Name your files as follows:

File	Naming
Flex Specification	`name_A4.l`
Bison Specification	`name_A4.y`
Data Structures Definitions & Global Function Prototypes	`name_A4_translator.h`
Data Structures, Function Implementations & Translator `main()`	`name_A4_translator.(c|cxx)`
Test Input	`name_A4.f15`
Makefile	`name_A4.mak`
Explanations of the translator desing	`name_A4.pdf`

4. Prepare a compressed-archive with the name `name_A4.x` where `x` is one of `tar`, `zip` or `rar`.

6 Credits

1. Explanation of the translator design: **[5 + 10 + 20 + 10 = 45]**
   (a) Augmentation of the grammar
   (b) Attributes of various terminals and non-terminals
   (c) Actions on every production rule
   (d) Auxiliary data structures and functions
2. Write the following test programs in $\mathcal{F}_{-15}$: **[2 + 2 + 3 + 5 + 6 + 7 = 25]**
   (a) Given `m, n, p`, determine if `n` lies within `m - p` and `m + p`.
   (b) Find the sum of odd numbers up to `n > 0`.
   (c) Print the first `n` numbers from the Fibonacci series: `0, 1, 1, 2, 3, 5, ....`
   (d) Print the first `n` numbers from the series `1, 3, 5, 11, 21, 43, 85, 171, ...` defined by:

$$\begin{aligned} s_0 &= 1 \\ s_i - s_{i-1} &= s_{i-1} + 1,\ i \geq 1,\ i\ is\ odd \\ s_i - s_{i-1} &= s_{i-1} - 1,\ i \geq 2,\ i\ is\ even \end{aligned}$$

   (e) Find the digital square root `s > 0` of `n > 0` where

$$s = \lfloor \sqrt{n} \rfloor$$

   (f) Given `n`, print all Pythagorean triples `a <= b <= c <= n` up to `n` where `(a, b, c)` is a Pythagorean triple if

$$a^2 + b^2 = c^2$$

   Note that for $m > n > 0$, (a, b, c) is Pythagorean if

$$\begin{aligned} a &= m^2 - n^2 \\ b &= 2mn \\ c &= m^2 + n^2 \end{aligned}$$

3. Correctness of Translation on every test program for TAC and symbol table: **[(4 + 1) * 6 = 30]**