Department of Computer Science and Engineering
IIT Guwahati

Implementation of Programming Languages Lab: CS 348

Assignment - 5: Machine Independent Code Generator for nanoC *Marks: 100*

In this assignment you will write the semantic actions in Bison to translate a nanoC 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 Scope of Machine-Independent Translation

Assume the following from the different phases to write actions for translation.

1. nanoC: The Lexical Grammar and the Phase Structure Grammar for nanoC as defined in Assignment 3. Follow the specification and the exclusion list carefully.
2. You have a choice to code your translator in C or in C++.
3. To compute the offset and storage mapping of variables, use the following *sizes (in bytes) of types*:

Type	Size	Remarks
`void`	*undefined*	
`char`	1	
`int`	4	*Needs to be aligned at addresses divisible by 4*
`void *`	4	*All pointers have same size*

   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_char = 1;
const unsigned int size_of_int = 4;
const unsigned int size_of_pointer = 4;
```

4. It may also help to support an *implicit* `bool` *(boolean) type* with constants `1` ≡ `true` and `0` ≡ `false`. This type may be inferred for a logical expression or for an `int` expression in logical context.
5. Assume the users cannot define, load, or store variables of `bool` type explicitly. Hence it is not storeable and does not have a size in the static / global allocation table or any activation record.

   For example, in the context of `int x; int y; int f;` expressions like `f = (x < y) || (x == y);` are not allowed while `if ((x < y) || (x == y)) {...} else {...};` or `if (x) {...} else {...};` are allowed.
6. *Function declaration with only parameter type list* are skipped. Hence, `int func(int i, char c);` should be supported while `int func(int, char);` may not be.
7. While *Global* and *Function Scopes* must be supported, *Nested Block Scopes* may be skipped. For example,

```
// ...
int x;              // Global Scope - To be supported
// ...
int func(int a) { // Function Scope - To be supported
    // ...
    int p;
    {               // Nested Block Scope - May be skipped
        // ...
        int p;
        // ...
    }
    // ...
}
```

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.

2.1 Address Types

- *Name*: Source program names appear as addresses in 3-Address Codes.
- *Constant*: Constants of allowed data types are allowed as deemed addresses.
- *Compiler-Generated Temporary*: Create a distinct name each time a temporary is needed.

2.2 Instruction Types

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

- *Binary Assignment Instruction*: For a binary (arithmetic) `op` (`+ - * / %`): `x = y op z`
- *Unary Assignment Instruction*: For a unary (arithmetic / address) operator `op` (`& * + -`): `x = op y`
- *Copy Assignment Instruction*: `x = y`
- *Unconditional Jump*: `goto L`
- *Conditional Jump*:
  - *Value based*: `x` taken as `false` if `x == 0`; as `true`, otherwise

    ```
    if x goto L
    ifFalse x goto L
    ```

  - *Comparison based*: For a relational operator `relop` (`<`, `>`, `==`, `!=`, `<=`, `>=`): `if x relop y goto L`
  - *Control Flow based*: For a logical operator `op` (`&&`, `||`, `!`), we use control flow to translate: For example, `if ((x > y) || (x > z)) { /*THEN CODE*/ } else { /*ELSE CODE*/ }` can be:

    ```
        if x > y goto L1
        if x > z goto L1
        goto L2
    L1: /* THEN CODE */
        goto L3
    L2: /* ELSE CODE */
    L3: /* CODE AFTER if */
    ```

- *Procedure Call*: A procedure call `p(x1, x2, ..., xN)` is coded as (for addresses `p`, `x1`, `x2`, and `xN`):

  ```
  param x1
  param x2
  ...
  param xN
  y = call p, N
  ```

  Note that the number of parameters `N` is not redundant as procedure calls can be nested.
- *Return Value*: Returning a return value and / or assigning it is optional. If there is a return value `v` it is returned from the procedure `p` as: `return v`
- *Indexed Copy Instructions*:

  ```
  x = y[z] /* Read from index */
  x[z] = y /* Write to index */
  ```

- *Address and Pointer Assignment Instructions*:

  ```
  x = &y
  x = *y
  *x = y
  ```

3 Design of the Translator

1. *Lexer & Parser*: Use the Flex and Bison specifications (if required you may correct your specifications) you had developed in Assignments 3 and 4 respectively and write semantic actions for translation to 3-address Codes. Note that some grammar rules of your nanoC parser may not have any action or may just have propagate-only actions. Also, some of the lexical tokens may not be used.

2. *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.

3. *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.

4. *Symbol Table*: Use symbol tables for user-defined (including arrays and pointers) variables, temporary variables and functions.

Name	Type	Initial Value	Size	Offset	Nested Table
...	...	...	...	...	...

For example, for

```
int i;
int a = 17;
int w[10];
int *p;
void func(int i, char c);
char c;
```

the Symbol Tables will look like:

ST(global)[a] *This is the Symbol Table for global symbols*

Name	Type	Initial Value[b]	Size	Offset	Nested Table
i	int[c]	0	4	0	null
a	int	17	4	4	null
w	*array*(10, int)[d]	0	$10 \times 4 = 40$	8	null
p	int*[e]	0	4	48	null
func[f]	int $\times$ char $\to$ void[g]	*not applicable*[h]	0	52	ptr-to-ST(func)
c	char[i]	0	1	52	null

[a]: Used to compute addresses for the static / global table of allocations
[b]: Global variables are initialized to 0 by default
[c]: Needs 4-bytes address alignment for all int
[d]: Needs type-tree for *array*(10, int)
[e]: Needs type-tree for int* = *pointer*(int). Needs 4-bytes address alignment for all ptrs
[f]: Has no entry in static / global table of storage allocation as func is a function
[g]: Needs type-tree for int $\times$ char $\to$ void = *function*(void, int, char)
[h]: A function has no initial value
[i]: Does not need any address alignment for all char

ST(func)[a] *This is the Symbol Table for function* func

Name	Type	Initial Value	Size	Offset	Nested Table
c	char[b]	*undefined*	1	0	null
i	int[c]	*undefined*	4	4	null
retVal	void[d]	*undefined*	0	8	null

[a]: Used to compute addresses for the activation record of func function
[b]: Does not need any address alignment for all char
[c]: Needs 4-bytes address alignment for all int
[d]: Has no entry in activation record for storage allocation as the type is void

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.
- The global symbol table is unique.
- Every function will have a symbol table of its parameters, automatic variables, and compiler-generated temporary. This symbol table will be nested in the global symbol table.
- Symbol definitions within blocks are naturally carried in separate symbol tables. Each such table will be nested in the symbol table of the enclosing scope. This will give rise to an implicit stack of symbol tables (global one being the bottom-most) the while symbols are processed during translation. The search for a symbol starts from the top-most (current) table and goes down the stack up to the global table.
- Since symbol definitions within blocks are not supported, no other nesting of symbol tables is needed.

5. 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;`,

```
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:
```

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. This method has three overloaded forms: `emit(result, arg1, op, arg2)`: `result = arg1 op arg2`. Binary `op`. `emit(result, arg1, op)`: `result = op arg1`. Unary `op`. `emit(result, arg1)`: `result = arg1`. Copy instruction.
`print(...)`	A method to print the quad array in a suitable format.

For example, the above state of the array may be printed (with the symbol information from the Symbol Table/s) as:

```
void main()
{
    int i = 10;
    int a[10];
    int v = 5;
    int t1;
    int t2;
    int t3;

    L100: t1 = i - 1;
    L101: i = t1;
    L102: t2 = i * 4;
    L103: t3 = a[t2];
    L104: if (t3 < v) goto L100;
}
```

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.

6. *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`.
`typecheck(E1, E2)` A function to check if `E1` & `E2` have same types (that is, if `<type_of_E1> = <type_of_E2>`). If not, then to check if they have compatible types (that is, one can be converted to the other), to use an appropriate conversion function `conv<type_of_E1>2<type_of_E2>(E)` or `conv<type_of_E2>2<type_of_E1>(E)` and to make the necessary changes in the Symbol Table entries. If not, that is, they are of incompatible types, to throw an exception during translation.
`conv<type1>2<type2>(E)` A function to convert[a] an expression `E` from its current type `type1` to target type `type2`, to adjust the attributes of `E` accordingly, and finally to generate additional codes, if needed. [a]: This function is called from `typecheck(E1, E2)`. Thus, the conversion is possible.

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 The Assignment

1. Write a 3-Address Code translator based on the Flex and Bison specifications of nanoC. Assume that the input nanoC 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.
3. Prepare test input files `A5_`*group*`.nc` to test the semantic actions and generate the translation output in `A5_`*group*`.out`. You may use the sample nanoC programs from Assignment 3 to test your translator.
4. Name your files as follows:

File	Naming
Flex Specification	`A5_`*group*`.l`
Bison Specification	`A5_`*group*`.y`
Data Structures Definitions & Global Function Prototypes	`A5_`*group*`_translator.h`
Data Structures, Function Implementations & Translator `main()`	`A5_`*group*`_translator.c`
Test Inputs: Number the tests with `<number> = 1, 2, 3, ...`	`A5_`*group*`_test<number>.nc`
Test Outputs: Output of test having `<number>`	`A5_`*group*`_quads<number>.out`

5. Prepare a tar-archive with the name `A4_`*group*`.tar` containing all the files and upload.

5 Credits

Design of Grammar Augmentations: **5**
Explain the augmentations in the production rules in Bison

Design of Attributes: **5**
Explain the attributes in the respective `%token` *and* `%type` *in Bison*

Design and Implementation of Symbol Table & Supporting Data Structures: **10**
Explain the definitions of ST & other Data Structures

Design and Implementation of Quad Array: **5**
Explain with definition of QA

Design and Implementation of Global Functions: **10**
Explain i/p, o/p, algorithm & purpose for every function

Design and Implementation of Semantic Actions:
Explain with every action in Bison
Expression Phase: **15**
Correct handling of operators, type checking & conversions
Declaration Phase: **10**
Handling of variable declarations, function definitions in ST
Statement Phase: **15**
Correct handling of statements
External Definition Phase: **5**
Correct handling of function definitions

Design of Test files and correctness of outputs: **20**
Test at least 5 i/p files covering all rules
Shortcoming and / or bugs, if any, should be highlighted