Computer Science & Engineering Department
I. I. T. Kharagpur

Compilers Laboratory: CS39003
3rd Year CSE: 5th Semester

Assignment – 5: Machine Independent Code Generator for tinyC *Marks: 100*
Assign Date: *7th October, 2014* Submit Date: *23:55, 20th October, 2014*

1 Preamble – tinyC

The Lexical Grammar (Assignment 3) and the Phase Structure Grammar (Assignment 4) for tinyC have already been defined as subsets of the C language specification from the International Standard **ISO/IEC 9899:1999 (E)**.

In this assignment you will write the semantic actions in yacc to translate a tinyC 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).

2 Scope of Machine-Independent Translation

Focus on the following from the different phases to write actions for translation.

2.1 Expression Phase

Support all arithmetic, shift, relational, bit, logical (boolean), and assignment expressions excluding:

1. **sizeof** operator
2. Comma (**,**) operator
3. Compound assignment operators

   ```
   *= /= %= += -= <<= >>= &= ^= |=
   ```

   Support only simple assignment operator (=)
4. Structure component expression

2.2 Declarations Phase

Support for declarations should be provided as follows:

1. Simple variable, pointer, array, and function declarations should be supported. For example, the following would be translated:

   ```
   double d = 2.3;
   int i, w[10];
   int a = 4, *p, b;
   void func(int i, double d);
   char c;
   ```

2. Consider only **void**, **char**, **int**, and **double** *type-specifiers*. As specified in C, **char** and **int** are to be taken as signed.

   For computation of offset and storage mapping of variables, assume the following sizes[1] (in bytes) of types:

[1] Using hard-coded sizes for types does not keep the code machine-independent. Hence you may want to use constants like size_of_char, size_of_int, size_of_double, and size_of_pointer for sizes that can be defined at the time of machine-dependent targeting.

Type	Size	Remarks
void	*undefined*	
char	1	
int	4	
double	8	
void *	4	*All pointers have same size*

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. Note that the users cannot define, load, or store variables of **bool** type explicitly, hence it is not storable and does not have a size.

3. Initialization of arrays may be skipped.

4. *storage-class-specifier*, *enum-specifier*, *type-qualifier*, and *function-specifier* may be skipped.

5. Function declaration with only parameter type list may be skipped. Hence,

```
void func(int i, double d);
```

should be supported while

```
void func(int, double);
```

may not be.

2.3 Statement Phase

Support all statements excluding:

1. Declarations within *compound-statement* (block).

2. Declaration within **for**.

3. All Labelled statements (*labeled-statement*).

4. **switch** in *selection-statement*.

5. All Jump statements (*jump-statement*) except **return**.

2.4 External Definitions Phase

Support function definitions and skip external declarations.

3 The 3-Address Code

Use the 3-Address Code specification as discussed in the class. For easy reference the same is reproduced here. Every 3-Address Code:

- Uses only up to 3 addresses.
- Is represented by a `quad` comprising – opcode, argument 1, argument 2, and result; where argument 2 is optional.

3.1 Address Types

- *Name*: Source program names appear as addresses in 3-Address Codes.
- *Constant*: Many different types and their (implicit) conversions are allowed as deemed addresses.
- *Compiler-Generated Temporary*: Create a distinct name each time a temporary is needed – good for optimization.

3.2 Instruction Types

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

- *Binary Assignment Instruction*: For a binary `op` (including arithmetic, shift, relational, bit, or logical operators):

  ```
  x = y op z
  ```

- *Unary Assignment Instruction*: For a unary operator `op` (including unary minus or plus, logical negation, bit, and conversion operators):

  ```
  x = op y
  ```

- *Copy Assignment Instruction*:

  ```
  x = y
  ```

- *Unconditional Jump*:

  ```
  goto L
  ```

- *Conditional Jump*:
  - *Value-based*:

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

  - *Comparison-based*: For a relational operator `op` (including $<$, $>$, $==$, $!=$, $\leq$, $\geq$):

    ```
    if x relop y goto L
    ```

- *Procedure Call*: A procedure call `p(x1, x2, ..., xN)` having `N` ≥ 0 parameters is coded as (for addresses `p`, `x1`, `x2`, and `xN`):

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

  Note that `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]
  x[z] = y
  ```

- *Address and Pointer Assignment Instructions*:

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

4 Design of the Translator

Lexer & Parser Use the flex and yacc specifications[2] you had developed in Assignments 3 and 4 respectively and write semantic actions for translating the subset of tinyC as specified in Section 2. Note that many grammar rules of your tinyC parser may not have any action or may just have propagate-only actions. Also, some of the lexical tokens may not be used.

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.

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 yacc specification file. Highlight the inherited attributes, if any.

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

```
double d = 2.3;
int i, w[10];
int a = 4, *p, b;
void func(int i, double d);
char c;
```

the Symbol Tables will look like:

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

Name	Type	Initial Value	Size	Offset	Nested Table
`d`	**double**	`2.3`	8	0	`null`
`i`	**int**	`null`	4	8	`null`
`w`	*array*(10, **int**)	`null`	40	12	`null`
`a`	**int**	`4`	4	52	`null`
`p`	*ptr*(**int**)	`null`	4	56	`null`
`b`	**int**	`null`	4	60	`null`
`func`	*function*	`null`	0	64	`ptr-to-ST(func)`
`c`	**char**	`null`	1	64	`null`

ST(func) *This is the Symbol Table for function* `func`

Name	Type	Initial Value	Size	Offset	Nested Table
`i`	**int**	`null`	4	0	`null`
`d`	**double**	`null`	8	4	`null`
`retVal`	**void**	`null`	0	0	`null`

The Symbol Tables may support the following methods:

`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.

[2]You may correct your specification/s if you need.

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 functionalities are added.
- The global symbol table is unique.
- Every function will have a symbol table of its parameters and automatic variables.
- Since symbol definitions within blocks are not supported, no other nesting of symbol tables is needed.

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, if

```
int i = 10, a[10], 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
```

the `quad`'s are represented as:

Index	**op**	**arg 1**	**arg 2**	**result**
...	...	...	...	...
100	−	i	1	t1
101	=	t1		i
102	*	i	4	t2
103	=[]	a	t2	t3
104	<	t3	v	100

The `Quad` Array may support the following methods:

`emit(...)`	An overloaded static method to add a (newly generated) `quad` of the form: `result = arg1 op arg2` where `op` usually is a binary operator. If `arg2` is missing, `op` is unary. If `op` also is missing, this is a 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) 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.

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

makelist(i) A global function to create a new list containing only i, an index into the array of quad's, and to return a pointer to the newly created list.
merge(p1, p2) A global function to concatenate two lists pointed to by p1 and p2 and to return a pointer to the concatenated list.
backpatch(p, i) A global function to insert i as the target label for each of the quad's on the list pointed to by p.
typecheck(E1, E2) A global function to check if E1 & E2 have same types (that is, if <type_of_E1> = <type_of_E2>(E)). 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 global 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]It is assumed that this function is called from typecheck(E1, E2) and hence the conversion is a 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.

5 The Assignment

1. Write a 3-Address Code translator based on the flex and yacc specifications of tinyC. Assume that the input tinyC 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 ass5_*roll*_test<number>.c to test the semantic actions and generate the translation output in ass5_*roll*_quads<number>.out.

4. Name your files as follows:

File	Naming
Flex Specification	ass5_*roll*.l
Yacc Specification	ass5_*roll*.y
Data Structures (Class Definitions) & Global Function Prototypes	ass5_*roll*_translator.h
Data Structures, Function Implementations & Translator `main()`	ass5_*roll*_translator.cxx
Test Inputs	ass5_*roll*_test<number>.c
Test Outputs	ass5_*roll*_quads<number>.out

5. Prepare a tar-archive with the name ass5_*roll*.tar containing all the files and upload to Intinno.

6 Credits

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

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

Design and Implementation of Symbol Table &
Supporting Data Structures: **10**
Explain with class definition of ST & other Data Structures

Design and Implementation of Quad Array: **5**
Explain with class 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 yacc
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