Assignment on Intermediate Code Generation

September 25, 2010

Introduction

To build an efficient front end, a proper intermediate representation of source language is one of the most essential steps. We choose Assembly Language as our intermediate form because we have different compilers like tasm, masm, nasm available to compile and execute an assembly program for this back end design. However, you do not have to design the front end. Your first step should be to do some practice on Assembly Language to remember its basic syntax. One of the best approaches is: first write a code with if, while statements and with expressions containing relop (used in conditional expression in while, if), ADDOP, MULOP (as supported by our grammar) and convert it to assembly program and then compile it. From this hopefully, you can easily understand what semantic actions you should associate with each non terminal to convert your source code to assembly code. Let us see an example.

Source Program:

```
var z, x, y : integer;
begin
x := 12;
y := 12;
while x <> 0
do
begin
z := x;
x := y mod x;
y := z;
if y<>4 then
        x:=x+4
end

end
```

Assembly Program: Do it by yourself.

2 Design Approach

Consider Intermediate code generation from parse/syntax tree. As discussed in the class, each non-terminal node corresponding to an expression must have two things. One is an address (a temporary variable) for holding it's value. Another for holding the code to evaluate this value from it's child expression applying the operators. We use SymbolInfo object for each grammar symbol, we must have **two variables** in it to hold those things. We may use the **key** attribute to store the temporary variable name (for identifier use key to hold the lexeme value i,e identifier name, value of ADDOP etc. and the lexeme type information in the SymbolInfo class) and define **code** to hold the produced code. Therefore, you have to add relevant fields in the class SymbolInfo to do your job. You are also required to implement a **symbol table** to hold the SymbolInfo objects.

The class may look like the follows

```
class symbolInfo
{
char* key;
char* type;
char* code;
}
```

Class SymbolTable has an array of type SymbolInfo, an insert function for inserting an object in the array and a lookup function.

For each non terminal that produce an operation as the following rule: (simple-expression --> simple-expression **ADDOP** term) create a new temporary variable to hold the value of it and assign the key attribute with it. Construct the code that produce this operation in assembly and assign the code attribute with it. Your yacc code may look like this:

simple expression: simple-expression **ADDOP** term

```
{
symbolInfo temp;
temp.key = newTemp();
```

```
temp.code = temp.code+ $1.code + $3.code;
temp.code = temp.code + assembly code to do this ADDOP.
$$ = temp;
}
```

assembly code to do this **ADDOP** looks somewhat like this:

```
temp.code=temp.code+"MOV EAX,"+$1.key;
temp.code+="ADD EAX,"+$3. key;
temp.code+="MOV dword"+temp.key+",EAX";
st.insert(temp);
```

**(you can use$1->key or $1->code syntax also!)

For production/rules that have no operation (term --> factor), no temporary is needed. Code for while/if statement is not so easy. There must be some jump to and some label to jump. At statement level, no address information is required and hence no key attribute is used, but only code attribute is required.

3 What To Do ?

You have to write a parser that produce assembly code for the followings:

- Expressions with ADDOP.
- Expressions with MULOP and RELOP
- Statement with if-else.
- Define a function newTemp() that returns a new temporary whenever it is called. Define a function newLabel() that returns a new label on each call.
- Add a new variable code in your symbolInfo.

There might be some other parts that your assembly code must have (Data segment initialization etc). Just print those in your output line from main function of yacc. You may also need some functions in assembly to output a value in console (As we know from our assembly language course). As an example, you may need a function to convert each individual digit of a number into ASCII value and then a function to output that individual digit using interrupt. So define those functions and write them in the output.

4 bonus

The implementation for the while statement and incorporating type information with identifier from the declaration section will be counted as a bonus for you!!!

5 Submission Deadline

A2 and B2: 12th week
A1 and B1: 13th week

6 Grammar to be implemented:

program –>
declarations
compound_statement

declarations–>
declarations **var** identifier_list **:** type**;**

identifier_list–>
id
| identifier_list**, id**

type–> **integer** | **real**

compound_statement →
begin
optional_statements
end

optional_statements →

statement_list
| €

statement_list →
statement
|statement_list**;** statement

statement →
id assignop exprssion
|compound_statement
l **if** expression **then** statement
| **if** expression **then** statement **else** statement
l **while** expression **do** statement

expression →
simple_expression
|simple_expression **relop** simple_expression

simple_expression →
term
|sign term
simple_expression **addop** term

term-->
factor
|term **mulop** factor

factor-->
id
| **num**
|**(**expression**)**
|**not** factor

sign-->
+|**-**