TD5 - Tuesday, October 12

1 Syntax-Directed Code Generation

Exercise 1 (TP)

Code generation involves the generation of the target representation (object code) from the annotated parse tree (or Abstract Syntactic Tree, AST) produced by syntactic and semantic analysis. When the code is generated directly, the object code is produced directly from the syntactic tree, while in indirect code generation, the code generator produces an intermediate representation. Code generation can also be distinguished by how it is integrated into the syntactic analysis :
– A single pass compiler performs semantic actions as syntactic rules are applied, and these semantic actions generate the code (either assembler directly, or the intermediate code).
– A multiple pass compiler separates syntactic analysis and code generation. The parse tree is produced in its entirety, and this is the input to the code generation phase.

The rest of this exercise assumes direct code generation, and a single pass compiler, generating code via semantic actions.

Your task is to implement a simple one pass compiler.

The source language is a small Pascal-like language, having only integer variables and no nested functions.

Ex.
```
var x,y
   function fact(n)
   var tmp
   begin
      if n=0 then tmp:=1;
      else tmp:= n*fact(n-1);
      fact:=temp
   end
begin
   y:=2;
   x:=fact(y)
end
```

The object code is the x86 Assembler, with 4 general purpose registers : AX, BX, CX, DX.

Stack : BP = base pointer
SP = stack pointer
The stack pointer is decreased for a PUSH operation.
Convention : AX stores the intermediary results ; Accumulator register.

Instructions :

– MOV reg,mem|| reg,reg|| reg, integer
– ADD reg1,reg2 : reg1=reg1+reg2
– MUL reg : DX :AX=AX*reg

– PUSH reg : put reg on stack
– POP reg : pop reg from stack

How code is generated ?

There are several ways to generate code from the syntax tree. In this exercise, we will assume it is done via the semantic actions connected to the syntax rules. As seen in course, for generating the code, the following conventions are made :

1. Declarations

– We maintain an environment variable-> memory address
– We store the variables on the stack

2. Expressions

– the results are stored in AX (accumulation register)

3. Instructions

– We generate labels

4. Procedures - we store on the stack the parameters and the local variables.

2 Symbol Tables

Exercise 2 In static-typing programming languages, variables need to be declared before they are used. The declaration provides the data type of the variable.
– E.g., int a ; float b ; string c ;

Most typically, declaration is valid for the scope in which it occurs :

– Function Scoping : each variable is defined anywhere in the function in which it is defined, after the point of definition

– Block Scoping : variables are only valid within the block of code in which they are defined, e.g

```
prog xxx {int a; float b}
{ int c;
   { int b;
   c = a + b;
   }
   return float(c) / b
}
```

In the symbol tables we store the information about identifiers (name,kind, type, address, etc.) Give two ways of representing scope information in the symbol table.

In the following, consider that one symbol table is used for each scope and a unique hash table is used to encode the entries. Consider a one pass compiler for a blocked structured language. Answer the following questions :

– At the point indicated in the code, what identifiers appear in the symbol table situated highest in the stack ?

```
{ int a; float c;
   {
```

```
        int b, c;
        a = b * 5;
    }
    {
        <====== here
        int d;
        d= a * c;
    }
}
```

– Given the following program, which of the following sentences are true ?
- In line 3 we have a semantic error for an undeclared variable.
- The type of variable a in line 3 is int.
- There is no semantic error in the program.
- The type of variable c in line 3 is int.

```
1.{ int a, c;
       ...
2.    { float a, b;
          ...
3.        b = c+a;
          ...
4.    }
5.    a = a + b;
       ...
6. }
```

Given the following program, show the hierarchy of symbol tables and what they contain. How many sematic error are catched and how is this done ?

```
int x;
void f(int m) {
   float x, y;
   ...
   { int i, j; x = 1; }
   { int x; l: i = 2; }
}
int g(int n) {
   bool t;
   x = 3;
}
```

Exercise 3 (TP) Type checkers :

Implement a small type checker for the language specified in Exercise 1, that is based on syntax directed definitions and can detect assignment errors, operator usage errors, function call errors.