CS4XX INTRODUTION TO COMPILER THEORY

WEEK 10

Reading:

Chapter 7 and Chapter 8 from Principles of Compiler Design, Alfred V. Aho & Jeffrey D. Ullman

Objectives:

1. To understand the concepts of Run-Time environments(Finish the previous part)
2. To learn the concepts of Intermediate Code Generations
3. To use the concepts learned in Syntax-directed translations

Concepts:

1.Language facilities for dynamic storage allocation--1/2 hr
2. Dynamic storage allocation techniques---1/2 hr
3. Intermediate languages--1 hr
4. Declarations--1 hr

Outlines:

1. Language facilities for dynamic storage allocation
    a. Explicit and Implicit allocation of memory
    b. Garbage Collection
2. Dynamic Storage allocation Techniques
3. Intermediate language
    a. Graphical representations - Syntax Trees & DAG
    b. Postfix notation
    c. Three Address Code
4. Declarations
    a. Translation scheme for declarations in a procedure
    b. Keeping track of scope
    c. Operations supporting nested STs
    d. Translation scheme for nested procedures
    e. Adding ST lookups to assignments

CS 4xx: Week 10 – Lecture Notes

1. Language Facilities for dynamic storage allocation

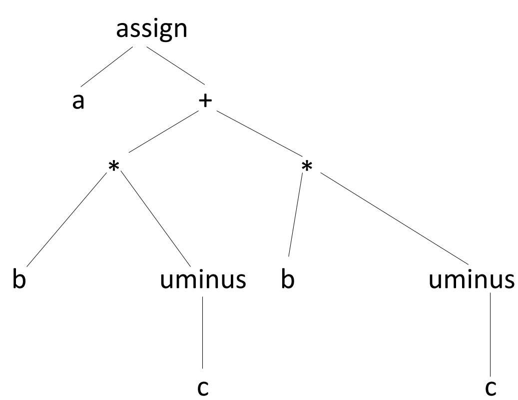
- Explicit and Implicit allocation of memory to variables
  - Most languages support dynamic allocation of memory.
  - Pascal supports new(p) and dispose(p) for pointer types.
  - C provides malloc() and free() in the standard library.
  - C++ provides the new and free operators.
  - These are all examples of EXPLICIT allocation.
  - Other languages like Python and Lisp have IMPLICIT allocation.
- Garbage – Finding variables that are not referred by the program any more
  - In languages with explicit deallocation, the programmer must be careful to free every dynamically allocated variable, or GARBAGE will accumulate.
  - Garbage is dynamically allocated memory that is no longer accessible because no pointers are pointing to it.
  - In some languages with implicit deallocation, GARBAGE COLLECTION is occasionally necessary.
  - Other languages with implicit deallocation carefully track references to allocated memory and automatically free memory when nobody refers to it any longer

2. Dynamic storage allocation techniques

- Explicit allocation of fixed sized blocks of memory with contiguous block of memory available for usage by the program
- Explicit allocation of variable sized blocks of memory where the storage can be fragmented
- Implicit deallocation of blocks of memory that are not used any more by using reference counts and marking techniques
- We assume the heap is an initially empty block of memory.
- As memory is allocated and deallocated, fragmentation occurs.
- For allocation, we must find a HOLE large enough to hold the requested memory.
- For deallocation, we must merge adjacent holes to prevent further fragmentation.

3. Intermediate Language

- Intermediate codes are machine independent codes, but they are close to machine instructions.
- Example assignment statement a: = b * -c + b * -c

- Syntax trees
  - A syntax tree depicts the natural hierarchical structure of a source program. A DAG (Directed Acyclic Graph) gives the same information but in a more compact way because common sub-expressions are identified.
  - 

- **Postfix notation**
  - Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the in which a node appears immediately after its children.
  - a b c uminus * b c uminus * + assign
  - postfix is convenient because it can run on an abstract STACK MACHINE
- **Three Address Code**
  - A more common representation is THREE-ADDRESS CODE (3AC)
  - 3AC is close to assembly language, making machine code generation easier.
  - 3AC has statements of the form

    x := y op z
  - To get an expression like x + y * z, we introduce TEMPORARIES:

    t1 := y * z

    t2 := x + t1
  - 3AC is easy to generate from syntax trees. We associate a temporary with each interior tree node.
  - Types of 3AC statements
    - Assignment statements of the form x := y op, where op is a binary arithmetic or logical operation.
    - Assignment statements of the form x := op Y, where op is a unary operator, such as unary minus, logical negation
    - Copy statements of the form x := y, which assigns the value of y to x.
    - Unconditional statements goto L, which means the statement with label L is the next to be executed.
    - Conditional jumps, such as if x relop y goto L, where relop is a relational operator (<, =, >=, etc) and L is a label. (If the condition x relop y is true, the statement with label L will be executed next.)
    - Statements param x and call p, n for procedure calls, and return y, where y represents the (optional) returned value. The typical usage:

      param x1
      param x2
      …
      param xn
      call p, n
    - Index assignments of the form x := y[i] and x[i] := y. The first sets x to the value in the location i memory units beyond location y. The second sets the content of the location i unit beyond x to the value of y.

4. Declarations

- When we encounter declarations, we need to lay out storage for the declared variables.
- For every local name in a procedure, we create a ST entry containing:

  – The type of the name
  – How much storage the name requires
  – A relative offset from the beginning of the static data area or beginning of the activation record.
- For intermediate code generation, we try not to worry about machine-specific issues like word alignment.
- To keep track of the current offset into the static data area or the AR, the compiler maintains a global variable, OFFSET.
- OFFSET is initialized to 0 when we begin compiling.
- After each declaration, OFFSET is incremented by the size of the declared variable.

- **Translation scheme for declarations in a procedure**

```
P ->                          { offset := 0 }
        D
D ->    D ; D
D -> id : T                   { enter( id.name, T.type, offset );
                                offset := offset + T.width }
T -> integer                  { T.type := integer; T.width := 4 }
T -> real                     { T.type := real; T.width := 8 }
T -> array [ num ] of T1      { T.type := array( num.val, T1.type );
                                T.width := num.val * T1.width }
T -> ^ T1                     { T.type := pointer( T1.type );
                                T.width := 4 }
```

- **Keeping track of scope**
  - When nested procedures or blocks are entered, we need to suspend processing declarations in the enclosing scope.
  - Let's change the grammar:

    P -> D

    D -> D ; D | id : T | proc id ; D ; S
  - Suppose we have a separate ST for each procedure.
  - When we enter a procedure declaration, we create a new ST.
  - The new ST points back to the ST of the enclosing procedure.
  - The name of the procedure is a local for the enclosing procedure.
  - Example: Fig. 8.12 in the text

- **Operations supporting nested STs**
  - **mktable(previous)** creates a new symbol table pointing to previous, and returns a pointer to the new table.

    enter(table,name,type,offset) creates a new entry for name in symbol table table with the given type and offset.

    addwidth(table,width) records the width of ALL the entries in table.

    enterproc(table,name,newtable) creates a new entry for procedure name in ST table, and links it to newtable.
- **Translation scheme for nested procedures**
  - ```
    P -> M D                      { addwidth(top(tblptr), top(offset));
                                    pop(tblptr); pop(offset) }
    M -> ϵ                        { t := mktable(nil);
                                    push(t,tblptr); push(0,offset); }
    D -> D1 ; D2
    D -> proc id ; N D1 ; S       { t := top(tblptr);
                                    addwidth(t,top(offset));
                                    pop(tblptr); pop(offset);
                                    enterproc(top(tblptr),id.name,t) }
    D -> id : T                   { enter(top(tblptr),id.name,T.type,top(offset));
                                    top(offset) := top(offset)+T.width }
    N -> ϵ                        { t := mktable( top( tblptr ));
                                    push(t,tblptr); push(0,offset) }
    ```
    ```

- **Adding ST lookups to assignments**
  - Let’s attach our assignment grammar to the procedure declarations grammar.

S -> id := E	{ p := lookup(id.name); if p != nil then emit(p ‘:=‘ E.place) else error }
E -> E1 + E2	{ E.place := newtemp(); emit(E.place ‘:=‘ E1.place ‘+’ E2.place) }
E -> E1 * E2	{ E.place := newtemp(); emit(E.place ‘:=‘ E1.place ‘*’ E2.place) }
E -> - E1	{ E.place := newtemp(); emit(E.place ‘:=‘ ‘uminus’ E1.place) }
E -> (E1)	{ E.place := E1.place }
E -> id	{ p := lookup(id.name); if p != nil then E.place := p else error }

(Continue next week on Intermediate Code Generation)