Compiler Construction

Lecture 9 – Semantic Analysis and Intermediate Code Generation

Intermediate Code Generation

- The semantic processing procedures need to create some intermediate representation of the program.
- One such representation is called ***quadruples*** or three-address codes.
  - Quadruples contain an assembler-level operator and three operands which do not depend on any particular machine architecture.
  - These can include:
    - assignment (for scalar values or array elements)
    - the arithmetic operations
    - conditional and unconditional **goto**s and their labels
    - identifying arguments and calling procedures

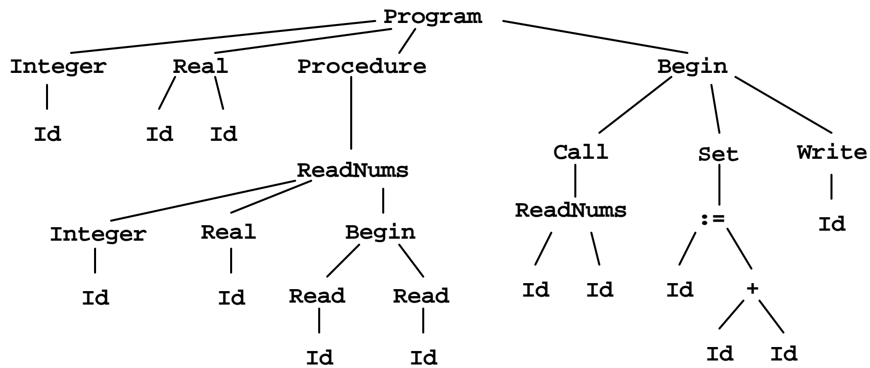
Quadruples

- Quadruples consist of four fields: one operation and up to three operands.
- Quadruples also known as three address codes.
- Quadruples can be implemented using an enumerated type to represent the operations and pointers to symbol table entries to represent the operands.

Sample Program in JASON

```
PROGRAM Sample;
  INTEGER i;
  REAL  x, y;
  PROCEDURE ReadNums(INTEGER i; REAL y);
    BEGIN
      READ i;
      READ y
    END;
  BEGIN
    CALL ReadNums(i, y);
    SET x := i+y;
    WRITE x
  END.
```

AST for `Sample`



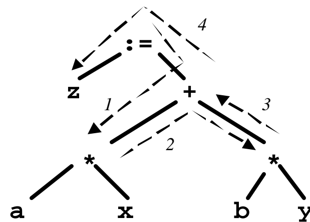
Quadruples for `Sample`

```
arg i
arg y
call ReadNums
t1 := float(i)
t2 := t1 + y
x := t2
arg x
call  Write
return
```

```
ReadNums:
      param i
      param y
      arg i
      call Read
      arg y
      call Read
      return
```

Generating Quadruples for Expressions

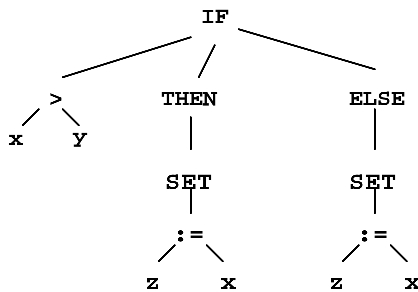
```
SET z := a * x + b * y
```




```
t1 := a * x
t2 := b * y
t3 := t1 + t2
z := t3
```

Generating Quadruples for **IF-THEN-ELSE-ENDIF**

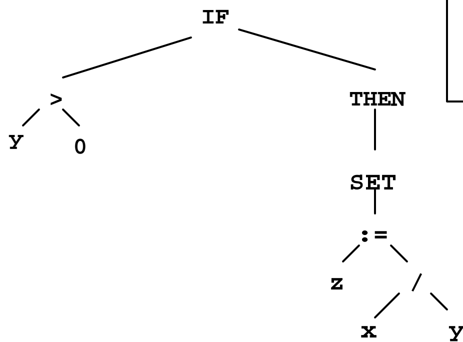
```
IF x > y THEN SET z := x
         ELSE SET z := y
ENDIF;
```




```
      if x <= y goto L1
      z := x
      goto L2
L1:
      z := y
L2:
```

Generating Quadruples for `IF-THEN-ENDIF`

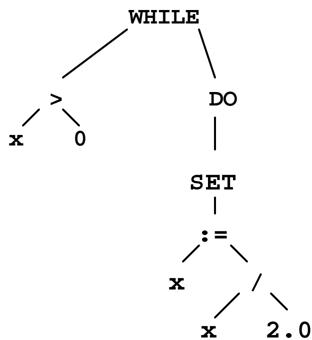
```
IF y > 0 THEN SET z := x/y
ENDIF;
```




```
        if y <= 0 goto L3
        z := x/y
L3:
```

Generating Quadruples for `WHILE-DO`

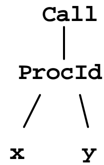
```
WHILE x > 0 THEN SET x := x/2.0
ENDWHILE;
```




```
L4:
        if x <= 0 goto L5
        x := x/2.0
        goto L4
L5:
```

Generating Quadruples for `Call` ProcId(x, y)

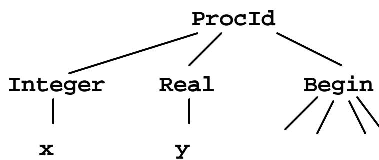
```
CALL ProcId(x, y);
```




```
arg x
arg y
call ProcId
```

Generating Quadruples for A Procedure

```
PROCEDURE ProcId(INTEGER x; REAL y);
    BEGIN … END;
```




```
ProcId:
      param x
      param y
      … …
      return
```

Semantic Actions in a Recursive-Descent Parser

```
void  ProcHeader(void)
{
      /*  Header ::= program identifier ; */
      if  (thistoken != tokprogram)
            error("Expected reserved word "
                  "\"program\"", linenum);

      thistoken = gettoken(ifp, &tabindex);
      if  (thistoken != tokidentifier)
            error("Expected identifier",
                        linenum);
```

```
      /* Set the identifier's attributes as
                        program  */
      installdatatype(tabindex, stprogram,
                  dtprogram);

      /*  Initialize the record thisproc
            to indicate the main program  */
      thisproc = initprocentry(tabindex);
      thistoken = gettoken(ifp, &tabindex);
      if  (thistoken != toksemicolon)
            error("Expected \";\"", linenum);
      thistoken = gettoken(ifp, &tabindex);
}
```

Semantic Actions and Recursively Parsing Expressions

```
struct addressrec   ProcExpression(void)
{
       struct addressrec   op1, op2, op3;
       enum optype   optor;

       /* Expression ::= Term RestOfExpression */
       /* The expression's address */
       op1 = ProcTerm();
       /* Get the operator and the second operand */
       ProcRestOfExpression(&optor, &op2);
```

Semantic Actions and Recursively Parsing Expressions

```
       /*  If the operator is not null, generate the
                          necessary instructions */
       op3 = (optor != opnull)?
              GenArithQuad(op1, optor, op2): op1;
       return(op3);
}
```

Changing The Parsing Algorithm TO Accommodate Semantic Actions

Processing context -free expressions requires the use of a stack. The Parsing algorithm uses a stack:

Place the start symbol in a node and push it onto the stack.

Fetch a token

REPEAT

Pop a node from the stack

IF it contains a terminal, match it to the current token (no match indicates a parsing error) and fetch another token

ELSE IF it contains a nonterminal, look it up in the production table using the nonterminal and the current token. Place the variables in REVERSE order on the stack

ELSE IF it contains an action, call the appropriate procedure to perform the action

UNTIL the stack is empty

The Parser Driver

```
/*
 * Parse() - This procedure checks the production table
   to
 *     make certain that there is a production for
 *     which this nonterminal can be expanded that
 *     begins with this token.  If there isn't, this
 *     is a fatal syntactic error; the compiler will
 *     terminate execution.
 *
 *     Then it pushes its right sentential form on
 *     the stack after linking them to their next
 *     node.
 */
```

```
void   parse(void)
{
   int i, lines = 0;

   initparsestack();
   parsetree = getparsenode(Nonterm, NTProgram);
   parsepush(parsetree);
   do  {
        /*
         *  Look up the production in the production
         *  table.  If not there, terminate with an
         *  error message.
         */
        thisnode = parsepop();
```

```
        switch (thisnode -> TermOrNonterm)      {
          case Term:
               /* If it's a terminal, match it to the
                * lookahead and get a new lookahead token
                */
               if (matchtoken(thistoken,
                             thisnode -> ParseItem))
                      thisnode -> symtabentry = tabindex;
               else   {
                      putchar('\"');
                      printlexeme(thisnode -> ParseItem);
                      printf("\" expected instead of \"");
                      printlexeme(thistoken);
                      error("\"", linenum);
               }
               lasttoken = thistoken;
               oldtabindex = tabindex;
               thistoken = gettoken(ifp, &tabindex);
               break;
```

```
        case Nonterm:
            /* Expand the nonterminal and push the
             * items on the right hand side in reverse
             * order                        */
            processnonterm(thisnode);
            if (thisnode -> ParseItem == NTHeader)
                AddProgName();
            break;

        case Action:
          /*  Perform the appropriate semantic action */
            processaction(thisnode -> ParseItem);
            break;
        default:
            error("Encountered invalid production"
                  " item", linenum);
      }
   }  while(!parseempty());
}
```

```
/*
 * ProcessAction() -    The driver for the
 *                      semantic actions.  It
 *                      calls the appropriate
 *                      function for the
 *               necessary semantic action.
 */
void  processaction(int actiontype)
{
  switch(actiontype)    {
    case AcAddProgName: AddProgName(); break;
    case AcPushReal:    PushReal(); break;
    case AcPushInt:     PushInt(); break;
      … … …          default:
      error("Invalid action", linenum);
        }
}
```

```
/*
 * GenRead() -    Generate the intermediate code
 * for Read statements
 */
void  GenRead(void)
{
  struct addressrec x;

  /*
   *  Generate two instructions: an arg
   *  instruction to pass the identifier as the
   *  one argument and a call of the Read
   *  procedure
   */
```

```
  if (!isvalidtype(oldtabindex))    {
      printlexeme(oldtabindex);
      error(" is an invalid type", linenum);
  }

  x = setaddress(opnvar, oldtabindex);
  nextop = genquad(opread, x, no_op, no_op);
}
```

```
/*
 * CalcTerm() -   Calculate a term in a
 *     expression being parsed.  The factor and
 *     the rest of the term are on the semantic
 *     actions stack along with the operator,
 *     which is the middle item on the top of the
 *     stack.
 */
 void CalcTerm(void)
{
   enum tokentype  tokoptor, tokopnd1, tokopnd2;
   enum optype     optor;
   struct addressrec     op1, op2, op3;
```

```
   /*  Pop the two operands and the operator */
   tokopnd2 = actionpop();
   tokoptor = actionpop();
   tokopnd1 = actionpop();

   /* Convert the operator's token into the
       matching operation */
   switch(tokoptor)      {
     case tokstar:       optor = opmult; break;
     case tokslash:      optor = opdiv;    break;
     default:            printlexeme(tokoptor);
             error(" is an invalid operator",
                   linenum);
   }
```

```
  /*
   * Convert the operand's symbol table entry
   *  into an address and generate the
   * intermediate code
   */
  op2 = getaddress(tokopnd1);
  op3 = getaddress(tokopnd2);

  op1 = GenArithQuad(op2, optor, op3);
  actionpush(op1.opnd);
}
```

```
/*
 * GenArithQuad() -     Generate an arithmetic
 *                      quadruple instruction
 */
struct addressrec GenArithQuad
      (struct addressrec op2,
      enum optype optor, struct addressrec op3)
{
  struct addressrec     op1, op4;
  enum optype           otype;
  /*
   *  If the two operands match, get a temporary
   *  variable of matching type.  If not, convert
   *  the integer variable to real and get a real
   *  temporary variable.
   */
```

```
if (data_class(op2.opnd) == dtinteger)
    if (data_class(op3.opnd) == dtinteger)
          op1 = gettempvar(dtinteger);
    else  {
    /*  Convert the integer operand to real */
          op4 = op2;
          op2 = gettempvar(dtreal);
          nextop = genquad(opfunc, op2, func,
                                op4);
          op1 = gettempvar(dtreal);
    }
```

```
  else if (data_class(op3.opnd) == dtinteger)   {
      op4 = op3;
      /*  Convert the integer operand to real */
      op3 = gettempvar(dtreal);
      nextop = genquad(opfunc, op3, func, op4);
      op1 = gettempvar(dtreal);
  }
  else
      op1 = gettempvar(dtreal);
  nextop = genquad(optor, op1, op2, op3);
  return(op1);
}
```