Intermediate Code Generation

(using a Syntax-Directed Translation Scheme)

Assignment and Expressions
Array Expressions
Boolean Expressions

SDT to Three Address Code

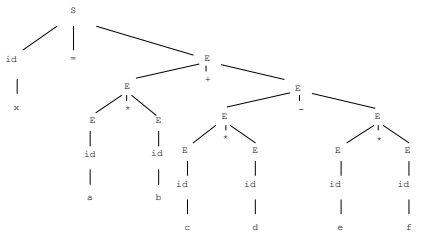
- Attributes for the Non-Terminals, E and S
  - Location (in terms of temporary variable) of the value of an expression: E.place
  - The Code that Evaluates the Expressions or Statement: E.code
  - Markers for beginning and end of sections of the code S.begin, S.end
- Semantic Actions in Productions of the Grammar
  - Functions to create temporaries `newtemp`, and labels `newlabel`
  - Use Auxiliary functions to enter symbols and consult types corresponding to declarations in aside data structure that can be built as the code is being parsed - a *symbol table*.
  - To generate the code we use the emit function `gen` which creates a list of instructions to be emitted later and can generate symbolic labels corresponding to next instruction of a list.
  - Use of `append` function on lists of instructions.
  - Synthesized and Inherited Attributes

Assignment Statements

```
S → id = E    { p = lookup(id.name);
                if (p != NULL)
                  S.code = gen(p '=' E.place);
                else
                  error;
                S.code = nulllist;
              }
E → E1 + E2   {E.place = newtemp();
               E.code = append(E1.code,E2.code,gen(E.place '=' E1.place '+' E2.place); }
E → E1 * E2   {E.place = newtemp();
               E.code = append(E1.code,E2.code,gen(E.place '=' E1.place '*' E2.place); }
E → - E1      {E.place = newtemp();
               E.code = append(E1.code,gen(E.place '=' '-' E1.place)); }
E → (E1)      {E.place = E1.place; E.code = E1.code; }
E → id        {p = lookup(id.name);
               if (p != NULL)
                 E.place = p;
               else
                 error;
               E.code = nulllist;
              }
```

Assignment: Example

```
x = a * b + c * d - e * f;
```

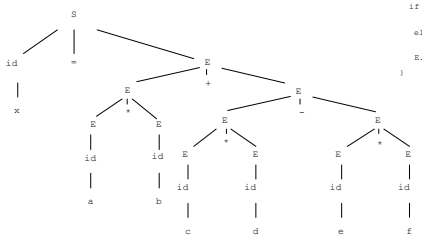


Assignment: Example

```
x = a * b + c * d - e * f;
```

Production:

```
E → id      { p = lookup(id.name);
              if (p != NULL)
                 E.place = p;
              else
                 error;
              E.code = null list;
            }
```

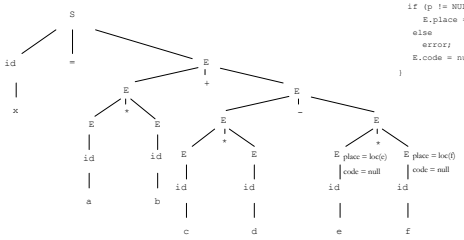


Assignment: Example

```
x = a * b + c * d - e * f;
```

Production:

```
E → id      { p = lookup(id.name);
              if (p != NULL)
                 E.place = p;
              else
                 error;
              E.code = null list;
            }
```

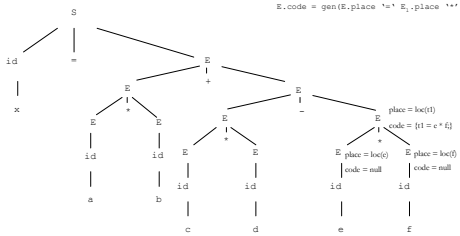


Assignment: Example

```
x = a * b + c * d - e * f;
```

Production:

```
E → E1 * E2   {E.place = newtemp();
               E.code = gen(E.place '=' E1.place '*' E2.place);}
```

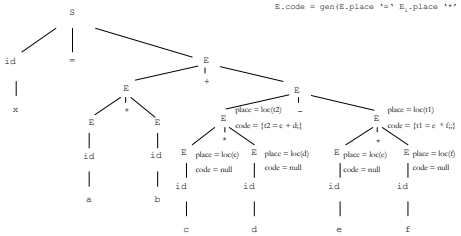


Assignment: Example

```
x = a * b + c * d - e * f;
```

Production:

```
E → E1 * E2   {E.place = newtemp();
               E.code = gen(E.place '=' E1.place '*' E2.place);}
```

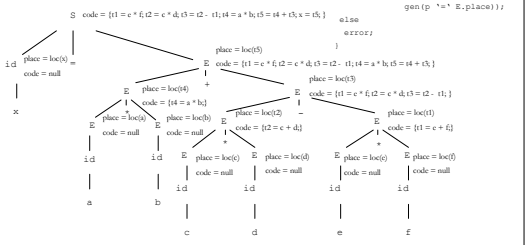


Assignment: Example

```
x = a * b + c * d - e * f;
```

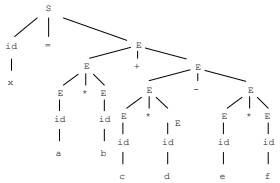
Production:

```
S → id = E   { p = lookup(id.name);
               if (p != NULL)
                 E.code = append(E.code,
                                 gen(p '=' E.place));
               else
                 error;
             }
```



Assignment: Example

```
x = a * b + c * d - e * f;
```



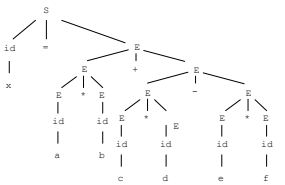
```
t1 = e * f;
t2 = c * d;
t3 = t2 - t1;
t4 = a * b;
t5 = t4 + t3;
x = t5;
```

Reusing Temporary Variables

- Temporary Variables
  - Short lived
  - Used for Evaluation of Expressions
  - Clutter the Symbol Table
- Change the `newtemp` Function
  - Keep track of when a value created in a temporary is used
  - Use a counter to keep track of the number of active temps
  - When a temporary is used in an expression decrement counter
  - When a temporary is generated by `newtemp` increment counter
  - Initialize counter to zero

Assignment: Example

```
x = a * b + c * d - e * f;
```

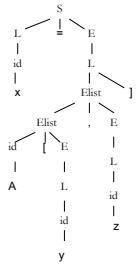


```
                        // c = 0
t1 = e * f;             // c = 1
t2 = c * d;             // c = 2
t1 = t2 - t1;           // c = 1
t2 = a * b;             // c = 2
t1 = t2 + t1;           // c = 1
x = t1;                 // c = 0
```

- Only 2 Temporary Variables and hence only 2 registers are Needed

Code Generation for Array Accesses

```
...
x = A[y,z];
...
```



- Questions:
  - What is the Base Type of A?
  - What are the Dimensions of A?
- Answers:
  - Use Symbol Table
  - Check Array Layout

?

```
t1 = y * 20
t1 = t1 * z
t2 = baseA - 84
t3 = 4 * t1
t4 = t2[t3]
X  = t4
```

How does the Compiler Handle A[i,j] ?

First, must agree on a storage scheme

Row-major order (most languages)
- Lay out as a sequence of consecutive rows
- Rightmost subscript varies fastest
- A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]

Column-major order (Fortran)
- Lay out as a sequence of columns
- Leftmost subscript varies fastest
- A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3]

Indirection vectors (Java)
- Vector of pointers to pointers to … to values
- Takes much more space, trades indirection for arithmetic
- Not amenable to analysis

Laying Out Arrays

The Concept

A

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4

These have distinct & different cache behavior

Row-major order

A

1,1	1,2	1,3	1,4	2,1	2,2	2,3	2,4

Column-major order

A

1,1	2,1	1,2	2,2	1,3	2,3	1,4	2,4

Indirection vectors

A

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4

Computing an Array Address

A[i]

- baseA + (i – low) x sizeof(baseType(A))
- In general: base(A) + (i – low) x sizeof(baseType(A))

Computing an Array Address

A[i]

- baseA + (i – low) x sizeof(baseType(A))
- In general: base(A) + (i – low) x sizeof(baseType(A))

Almost always a power of 2, known at compile-time and use a shift for speed

int A[1:10] where low is 1; Make low 0 for faster access (saves a –) as in the C language

Computing an Array Address

A[i]

- baseA + (i – low) x sizeof(baseType(A))
- In general: base(A) + (i – low) x sizeof(baseType(A))

What about $A[i_1,i_2]$?

This stuff looks expensive! Lots of implicit +, -, x ops

Row-major order, two dimensions

$baseA + ((i_1 - low_1) \times (high_2 - low_2 + 1) + i_2 - low_2) \times sizeof(baseType(A))$

Column-major order, two dimensions

$baseA + ((i_2 - low_2) \times (high_1 - low_1 + 1) + i_1 - low_1) \times sizeof(baseType(A))$

Indirection vectors, two dimensions

$*(A[i_1])[i_2]$ — where $A[i_1]$ is, itself, a 1-d array reference

Optimizing Address Calculation for A[i,j]

where w = sizeof(baseType(A))

In row-major order

$baseA + (i - low_1)(high_2 - low_2 + 1) \times w + (j - low_2) \times w$

Which can be factored into

$baseA + i \times (high_2 - low_2 + 1) \times w + j \times w - (low_1 \times (high_2 - low_2 + 1) \times w) + (low_2 \times w)$

If low_i, $high_i$, and w are known, the last term is a constant

Define $baseA_0$ as

$baseA - (low_1 \times (high_2 - low_2 + 1)) \times w + low_2 \times w$

and len_2 as $(high_2 - low_2 + 1)$

Then, the address expression becomes

$baseA_0 + (i \times len_2 + j) \times w$

Compile-time constants

Address Calculation for A[$i_1,i_2,\ldots,i_k$]

- $A[i_1,i_2,\ldots,i_k]$

Addressing generalizes to

$((\ldots((i_1 n_2 + i_2) n_3 + i_3)\ldots)n_k + i_k) \times w +$
$+baseA - ((\ldots((low_1 n_2 + low_2) n_3 + low_3)\ldots)m_k + low_k) \times w$

where $n_i = high_i - low_i + 1$ and w = sizeof(baseType(A))

Compile-time constants

First term can be computed using the recurrence

$e_1 = i_1$

$e_m = e_{m-1} \times n_m + i_m$

at the end multiply by w and add compile-time constant term

SDT for Addressing Arrays Elements

- Three Attributes
  - place: just the name or base address of the array
  - offset: the index value into the array
  - ndim: the number of dimensions
- Use the recurrence to compute offset

  $offset_1 = i_1$

  $offset_m = offset_{m-1} \times n_m + i_m$
  - At the end multiply by w = sizeof(baseType(A))
  - Add the compile-time constant term
  - Keep track of which dimension at each level
  - Use the auxiliary function n_m = numElem(A,m)

SDT for Addressing Arrays Elements

```
L → Elist ]          {
          L.place = newtemp();
          L.offset = newtemp();
          code1 = gen(L.place '=' constTerm(Elist.array));
          code2 = gen(L.offset '=' Elist.place * sizeof(baseType(Elist.array)));
          L.code = append(Elist.code,code1,code2);
          }

Elist → Elist1 , E {
          t = newtemp();
          m = Elist1.ndim + 1;
          code1 = gen(t = Elist1.place * numElem(Elist1.array,m));
          code2 = gen(t = t + E.place);
          Elist.array = Elist1.array;
          Elist.place = t;
          Elist.ndim = m;
          Elist.code = append(Elist1.code,E.code,code1,code2);
          }

Elist → id [ E  {
              Elist.array = id.place;
              Elist.place = E.place;
              Elist.ndim = 1;
              EList.code = E.code;
          }
```

SDT for Addressing Arrays Elements

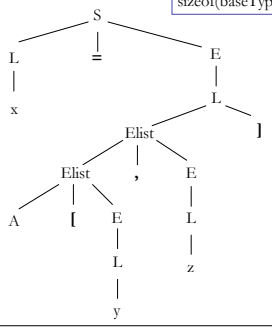
```
E → L     { if (L.offset = NULL) then
              E.place = L.place;
            else
              E.place = newtemp;
              E.code = gen(E.place = L.place[L.offset]);
          }

S → L = E   { if L.offset = NULL then
              E.code = gen(L.place = E.place);
            else
              S.code = append(E.code,gen(L.place[L.offset] = E.place);
          }

L → id    { L.place = id.place;
            L.offset = null;
          }
```

SDT for Addressing Arrays Elements

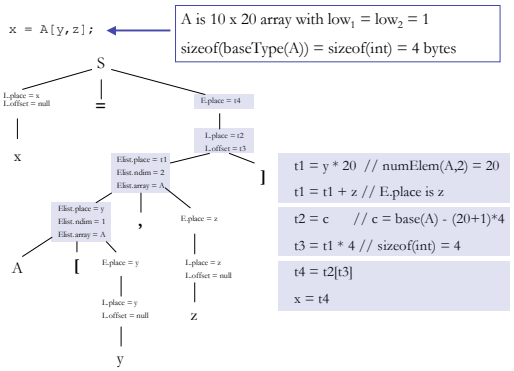
```
x = A[y,z];
```

A is 10 x 20 array with $low_1 = low_2 = 1$

sizeof(baseType(A)) = sizeof(int) = 4 bytes

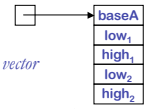


SDT for Addressing Arrays Elements



Array References

What about arrays as actual parameters?



Whole arrays, as call-by-reference parameters

- Need dimension information ⇒ build a *dope vector*
- Store the values in the calling sequence
- Pass the address of the dope vector in the parameter slot
- Generate complete address polynomial at each reference

Some improvement is possible

- Save len_i and low_i rather than low_i and $high_i$
- Pre-compute the fixed terms in prologue sequence

What about call-by-value?

- Most c-b-v languages pass arrays by reference
- This is a language design issue

Array References

What about A[12] as an actual parameter?

If corresponding parameter is a scalar, it's easy

- Pass the address or value, as needed
- Must know about both formal & actual parameter
- Language definition must force this interpretation

What is corresponding parameter is an array?

- Must know about both formal & actual parameter
- Meaning must be well-defined and understood
- Cross-procedural checking of conformability

⇒ Again, we're treading on language design issues

Array References

What about variable-sized arrays?

Local arrays dimensioned by actual parameters

- Same set of problems as parameter arrays
- Requires dope vectors (or equivalent)
  - dope vector at fixed offset in activation record

  ⇒ Different access costs for textually similar references

This presents a lot of opportunity for a good optimizer

- Common sub-expressions in the address polynomial
- Contents of dope vector are fixed during each activation
- Should be able to recover much of the lost ground

⇒ Handle them like parameter arrays

Array Address Calculations in a Loop

```
DO J = 1, N
   A[I,J] = A[I,J] + B[I,J]
END DO
```

- Naïve: Perform the address calculation twice

```
DO J = 1, N
   R1 = baseA0 + (J x len1 + I ) x floatsize
   R2 = baseB0 + (J x len1 + I ) x floatsize
   MEM(R1) = MEM(R1) + MEM(R2)
END DO
```

Array Address Calculations in a Loop

```
DO J = 1, N
   A[I,J] = A[I,J] + B[I,J]
END DO
```

- Sophisticated: Move common calculations out of loop

```
R1 = I x floatsize
c = len1 x floatsize   ! Compile-time constant
R2 = baseA0 + R1
R3 = baseB0 + R1
DO J = 1, N
   a = J x c
   R4 = R2 + a
   R5 = R3 + a
   MEM(R4) = MEM(R4) + MEM(R5)
END DO
```

Array Address Calculations in a Loop

```
DO J = 1, N
   A[I,J] = A[I,J] + B[I,J]
END DO
```

- Very sophisticated: Convert multiply to add (Operator Strength Reduction)

```
R1 = I x floatsize
c = len1 x floatsize  ! Compile-time constant
R2 = baseA0 + R1 ;  R3 = baseB0 + R1
DO J = 1, N
   R2 = R2 + c
   R3 = R3 + c
   MEM(R2) = MEM(R2) + MEM(R3)
END DO
```

SDT Scheme for Boolean Expressions

- Two Basic Code Generation Flavors
  - Use boolean **and**, **or** and **not** instructions (like arithmetic)
  - Control-flow (or positional code) defines **true** or **false** of predicate
- Arithmetic Evaluation
  - Simpler to generate code as just eagerly evaluate the expression
  - Associate '1' or '0' with outcome of predicates and combine with logic instr.
  - Use the same SDT scheme explained for arithmetic operations.
- Control Flow Evaluation (**short circuit evaluation**)
  - More efficient in many cases
  - Complications
    - Need to Know Address to Jump To in Some Cases
    - Solution: Two Additional Attributes
      - `nextstat` (Inherited) Indicates the next location to be generated
      - `laststat` (Synthesized) Indicates the last location filled
      - As code is generated the attributes are filled with the correct value

Arithmetic Scheme: Grammar and Actions

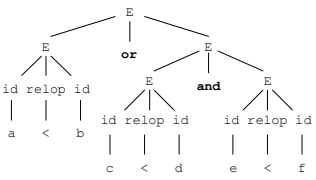
```
E → false     ‖ E.place = newtemp()
                E.code = {gen(E.place = 0)}
                E.laststat = E.nextstat + 1

E → true      ‖ E.place = newtemp()
                E.code = {gen(E.place = 1)}
                E.laststat = E.nextstat + 1

E → (E1)      ‖ E.place = E1.place;
                E.code = E1.code;
                E1.nextstat = E.nextstat
                E.laststat = E1.laststat

E → not E1    ‖ E.place = newtemp()
                E.code = append(E1.code,gen(E.place = not E1.place))
                E1.nextstat = E.nextstat
                E.laststat = E1.laststat + 1
```

Arithmetic Scheme: Grammar and Actions

```
E → E1 or E2  ‖ E.place = newtemp()
                E.code = append(E1.code,E2.code,gen(E.place = E1.place or E2.place)
                E1.nextstat = E.nexstat
                E2.nextstat = E1.laststat
                E.laststat = E2.laststat + 1

E → E1 and E2 ‖ E.place = newtemp()
                E.code = append(E1.code,E2.code,gen(E.place = E1.place and E2.place)
                E1.nextstat = E.nexstat
                E2.nextstat = E1.laststat
                E.laststat = E2.laststat + 1

E → id1 relop id2 ‖ E.place = newtemp()
                    E.code = gen(if id1.place relop id2.place goto E.nextstat+3)
                    E.code = append(E.code,gen(E.place = 0))
                    E.code = append(E.code,gen(goto E.nextstat+2))
                    E.code = append(E.code,gen(E.place = 1))
                    E.laststat = E.nextstat + 4
```

Boolean Expressions: Example

a < b or c < d and e < f




```
00: if a < b goto 03
01: t1 = 0
02: goto 04
03: t1 = 1
04: if c < d goto 07
05: t2 = 0
06: goto 08
07: t2 = 1
08: if e < f goto 11
09: t3 = 0
10: goto 12
11: t3 = 1
12: t4 = t2 and t3
13: t5 = t1 or t4
```

Summary

- Intermediate Code Generation
  - Using Syntax-Directed Translation Schemes
  - Expressions and Assignments
  - Array Expressions and Array References
  - Boolean Expressions (Arithmetic Scheme)