Intermediate Code Generation

Basic Approach and Application to Assignment and Expressions

Array Expressions

Boolean Expressions

Copyright 2017, Pedro C. Diniz, all rights reserved.
Students enrolled in the Compilers class at the University of Southern California have explicit permission to make copies of these materials for their personal use.
A compiler is a lot of fast stuff followed by some hard problems
- The hard stuff is mostly in Code Generation and Optimization
- For super-scalars, its Allocation & Scheduling that counts
Intermediate Code Generation

- **Direct Translation**
  - Using SDT scheme
  - Parse Tree to Three-Address Instructions
  - Can be done while Parsing in a Single Pass
  - Needs to be able to deal with Syntactic Errors and Recovery

- **Indirect Translation**
  - First validate parsing constructing of AST
  - Uses SDT scheme to build AST
  - Traverse the AST and generate Three Address Instructions

This Lecture

Same but Easier
Three-Address Instructions IR

• High-level Constructs mapped to Three-Address Instructions
  – Register-based IR for Expression Evaluation
  – Infinite Number of Virtual Registers
  – Still Independent of Target Architecture
  – Parameter Passing Discipline either on Stack or via Registers

• Addresses and Instructions
  – Symbolic Names are addresses of the corresponding source-level variable.
  – Various constants, such as numeric and offsets (known at compile time)

• Generic Instruction Format:
  Label: \( x = y \text{ op } z \) or if exp goto L
  – Statements can have Symbolic Labels
  – Compiler inserts Temporary Variables (any variable with t prefix)
  – Type and Conversions dealt in other Phases of the Code Generation
Three-Address Instructions

- Assignments:
  - \( x = y \text{ op } z \) (binary operator)
  - \( x = \text{ op } y \) (unary)
  - \( x = y \) (copy)
  - \( x = y[i] \text{ and } x[i] = y \) (array indexing assignments)
  - \( x = \text{ phi } y \text{ z} \) (Static Single Assignment instruction)

- Memory Operations:
  - \( x = &y; \ x = *y \text{ and } *x = y; \) for assignments via pointer variables.
Three-Address Instructions

- Control Transfer and Function Calls:
  - `goto L` (unconditional);
  - `if (a relop b) goto L` (conditional) where `relop` is a relational operator consistent with the type of the variables `a` and `b`;
  - `y = call p, n` for a function or procedure call instruction to the name or variable `p`
    - `p` might be a variable holding a set of possible symbolic names (a function pointer)
    - the value `n` specifies that before this call there were `n` `putparam` instructions to load the values of the arguments.
    - the `param x` instruction specifies a specific value in reverse order (i.e., the `param` instruction closest to the call is the first argument value.
    - Later we will talk about parameter passing disciplines (Run-Time Env.)
Function Call Example

Source Code

```c
int p(x, z){
    return x+z;
}
```

Three Address Instructions

```c

```

```c
    t1 = a
    t2 = b + 1
    putparam t1
    putparam t2
    y = call p, 2
    p:  getparam z
    getparam x
    t3 = x + z
    return t3
    return
```
Function Call Example

Source Code

```c
int p(x,z){
    return x+z;
}
```

Three Address Instructions

```
t1 = a
putparam t1
t2 = b + 1
putparam t2
y = call p, 2
```

argument evaluation

```
argument passing args on the stack
```

```
getparam z
p: getparam z
t3 = x + z
return t3
```

getting values from the stack
SDT for Three Address Code Generation

• Attributes for the Non-Terminals, E and S
  – Location (in terms of temporary variable) of the value of an expression: E.place. If E.place is t1 it means that the value of E is saved in t1.
  – The Code that Evaluates the Expressions or Statement: E.code
  – Markers for beginning and end of sections of the code S.begin, S.end
    • For simplicity these are symbolic labels.
    • Markers are inherited attributes

• Semantic Actions in Productions of the Grammar
  – Functions to create temporaries newtemp, and labels newlabel
  – Auxiliary functions to enter symbols and lookup types corresponding to declarations in a symbol table.
  – To generate the code we use the 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.
  – Generate code in post-order traversal of the AST
Assignment Statements

\[
S \rightarrow \text{id} = E \quad \{ \ p = \text{lookup}(\text{id}.\text{name}); \\
\quad \quad \text{if} \ (p \neq \text{NULL}) \{ \\
\quad \quad \quad \text{S.code} = \text{gen}(p \ '=' \ E.\text{place}); \\
\quad \quad \} \quad \text{else} \} \\
\quad \text{error;} \\
\quad \text{S.code} = \text{nulllist;} \\
\}
\]

\[
E \rightarrow E_1 + E_2 \quad \{ \ E.\text{place} = \text{newtemp}(); \\
\quad \text{E.code} = \text{append}(E_1.\text{code}, E_2.\text{code}, \\
\quad \quad \text{gen}(E.\text{place} = ' E_1.\text{place} + E_2.\text{place}); \\
\}
\]

\[
E \rightarrow E_1 * E_2 \quad \{ \ E.\text{place} = \text{newtemp}(); \\
\quad \text{E.code} = \text{append}(E_1.\text{code}, E_2.\text{code}, \\
\quad \quad \text{gen}(E.\text{place} = ' E_1.\text{place} * E_2.\text{place}); \\
\}
\]

\[
E \rightarrow - E_1 \quad \{ \ E.\text{place} = \text{newtemp}(); \\
\quad \text{E.code} = \text{append}(E_1.\text{code}, \text{gen}(E.\text{place} = '-' E_1.\text{place})); \}
\]

\[
E \rightarrow (E_1) \quad \{ \ E.\text{place} = E_1.\text{place}; \ E.\text{code} = E_1.\text{code}; \}
\]

\[
E \rightarrow \text{id} \quad \{ \ p = \text{lookup}(\text{id}.\text{name}); \\
\quad \text{if} \ (p \neq \text{NULL}) \\
\quad \quad E.\text{place} = p; \\
\quad \text{else} \\
\quad \quad \text{error;} \\
\quad \text{E.code} = \text{nulllist;} \\
\}
\]
Assignment: Example

\[ x = a \times b + c \times d - e \times f; \]

[Diagram of the expression tree for the equation \( x = a \times b + c \times d - e \times f; \)]
Assignment: Example

\[ x = a \times b + c \times d - e \times f; \]

Production:

\[
E \rightarrow id \quad \{ \ p = \text{lookup}(id.\text{name}); \ \\
\text{if} \ (p \neq \text{NULL}) \ \\
\quad E.\text{place} = p; \ \\
\text{else} \ \\
\quad \text{error}; \ \\
\quad E.\text{code} = \text{null list}; \ \\
\}
\]
Assignment: Example

\[ x = a \times b + c \times d - e \times f; \]

Production:

\[
E \rightarrow id \quad \{ \quad p = \text{lookup}(id.\text{name}); \\
\quad \text{if } (p \neq \text{NULL}) \\
\quad \quad E.\text{place} = p; \\
\quad \text{else} \\
\quad \quad \text{error;} \\
\quad E.\text{code} = \text{null list}; \\
\quad \} 
\]

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

\[ x = a \times b + c \times d - e \times f; \]

Production:

\[
E \rightarrow E_1 \times E_2 \quad \{E.\text{place} = \text{newtemp}(); \\
E.\text{code} = \text{gen}(E.\text{place} = '=' \ E_1.\text{place} \times \ E_2.\text{place});\}
\]
Assignment: Example

\[ x = a \times b + c \times d - e \times f; \]

Production:

\[ E \to E_1 \times E_2 \quad \{ \text{E.place} = \text{newtemp}(); \}
\]

\[ \quad \text{E.code} = \text{gen}(\text{E.place} = '=' \ E_1.\text{place} \times E_2.\text{place}); \}
\]
**Assignment: Example**

**Production:**

\[
x = a \times b + c \times d - e \times f;
\]

\[
S \rightarrow \text{id} = E \quad \{ p = \text{lookup}(\text{id}.\text{name});
\]

\[
\quad \begin{cases}
\text{if } (p \neq \text{NULL})
\quad E.\text{code} = \text{append}(E.\text{code},
\quad \quad \quad \quad \quad \text{gen}(p \text{ ‘=} E.\text{place}));
\quad \quad \quad \quad \quad \text{else}
\quad \quad \quad \quad \quad \text{error};
\end{cases}
\]

place = \text{t5}

\[
E \rightarrow \text{id} = E + E \quad \{ \text{code} = \{ \text{t4} = \text{id} \times \text{id}; \text{t5} = \text{id} + \text{id} \};
\]

place = \text{t4}

\[
E \rightarrow \text{id} = E \times E \quad \{ \text{code} = \{ \text{t1} = \text{id} \times \text{id}; \text{t2} = \text{id} \times \text{id}; \text{t3} = \text{id} + \text{id} \};
\]

place = \text{t2}

\[
E \rightarrow \text{id} = \text{id} \quad \{ \text{code} = \{ \text{t1} = \text{id} \times \text{id}; \text{t2} = \text{id} \times \text{id}; \text{t3} = \text{id} + \text{id} \};
\]

place = \text{t1}

\[
\text{place} = x
\]

place = \text{t3}

\[
\text{place} = \text{t1}
\]

\[
\text{place} = \text{t5}
\]

\[
\text{place} = \text{t4}
\]

\[
\text{place} = \text{t2}
\]

\[
\text{place} = \text{a}
\]

\[
\text{place} = \text{b}
\]

\[
\text{place} = \text{c}
\]

\[
\text{place} = \text{d}
\]

\[
\text{place} = \text{e}
\]

\[
\text{place} = \text{f}
\]

\[
\text{id} = \text{null}
\]

\[
\text{id} = \text{null}
\]

\[
\text{id} = \text{null}
\]

\[
\text{id} = \text{null}
\]

\[
\text{id} = \text{null}
\]

\[
\text{id} = \text{null}
\]

\[
\text{id} = \text{null}
\]
Assignment: Example

\[
x = a \times b + c \times d - e \times f;
\]

\[
s = \text{id} = E + E * (\text{id} \times \text{id} \times \text{id} \times \text{id} \times \text{id} \times \text{id} \times \text{id})
\]

\[
t1 = e \times f;
\]

\[
t2 = c \times d;
\]

\[
t3 = t2 - t1;
\]

\[
t4 = a \times b;
\]

\[
t5 = t4 + t3;
\]

\[
x = t5;
\]
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 temporaries
  – When a temporary is used in an expression decrement counter
  – When a temporary is generated by newtemp increment counter
  – Initialize counter to zero (0)

• Alternatively, can be done as a post-processing pass…
Assignment: Example

\[ x = a \times b + c \times d - e \times f; \]

- Only 2 Temporary Variables and hence only 2 Registers are Needed
Code Generation for Array Accesses

• Questions:
  – What is the Base Type of A?
  – What are the Dimensions of A?
  – What is the “layout” of A?

• Where to Get Answers?
  – Use Symbol Table
  – Check Array Layout

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

\[ t1 = y \times 20 \]
\[ t1 = t1 \times z \]
\[ t2 = \text{baseA} - 84 \]
\[ t3 = 4 \times t1 \]
\[ t4 = t2[t3] \]
\[ X = t4 \]
How does the Compiler Handle $A[i,j]$?

First, we must agree on a Storage Scheme or Layout

**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

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
<td>2,4</td>
<td></td>
</tr>
</tbody>
</table>

These have distinct & different cache behavior

Row-major order

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
<th>2,1</th>
<th>2,2</th>
<th>2,3</th>
<th>2,4</th>
</tr>
</thead>
</table>

Column-major order

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>2,1</th>
<th>1,2</th>
<th>2,2</th>
<th>1,3</th>
<th>2,3</th>
<th>1,4</th>
<th>2,4</th>
</tr>
</thead>
</table>

Indirection vectors

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
<td>2,4</td>
<td></td>
</tr>
</tbody>
</table>
Accessing an Array Element

How to “access” A[i]?

• Computing the Address of a 1-D Array Element:
  – baseA + (i – low) x sizeof(baseType(A))
• In General:
  – base(A) + (i – low) x sizeof(baseType(A))
Computing an Array Address

Computing the Address of \( A[i] \):

- **Computing the Address of a 1-D Array Element:**
  - \( \text{baseA} + (i - \text{low}) \times \text{sizeof}(	ext{baseType}(A)) \)

- **In General:**
  - \( \text{base}(A) + (i - \text{low}) \times \text{sizeof}(	ext{baseType}(A)) \)

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

Defined by the storage class of the array.

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[i1,i2]?

Row-major order, two dimensions
baseA + ((i1 - low1) x (high2 - low2 + 1) + i2 - low2) x sizeof(baseType(A))

Column-major order, two dimensions
baseA + ((i2 - low2) x (high1 - low1 + 1) + i1 - low1) x sizeof(baseType(A))

Indirection vectors, two dimensions
*(A[i1])[i2] — where A[i1] is, itself, a 1-d array reference

This stuff looks expensive!
Lots of implicit +, -, x ops
Optimizing Address Calculation for $A_{[i,j]}$

In row-major order

$$\text{baseA} + (i - \text{low}_1)(\text{high}_2 - \text{low}_2 + 1) \times w + (j - \text{low}_2) \times w$$

Which can be factored into

$$\text{baseA} + i \times (\text{high}_2 - \text{low}_2 + 1) \times w + j \times w$$

$$- (\text{low}_1 \times (\text{high}_2 - \text{low}_2 + 1) \times w) + (\text{low}_2 \times w)$$

If $\text{low}_i$, $\text{high}_i$, and $w$ are known, the last term is a constant

Define $\text{baseA}_0$ as

$$\text{baseA} - (\text{low}_1 \times (\text{high}_2 - \text{low}_2 + 1)) \times w + \text{low}_2 \times w$$

and $\text{len}_2$ as $(\text{high}_2 - \text{low}_2 + 1)$

Then, the address expression becomes

$$\text{baseA}_0 + (i \times \text{len}_2 + j) \times w$$

where $w = \text{sizeof}($baseType$(A))$
Address Calculation for \( A[i_1,i_2,\ldots,i_k] \)

- \( A[i_1,i_2,\ldots,i_k] \)

Addressing generalizes to

\[
((\ldots((i_1n_2+i_2)n_3+i_3)\ldots)n_k+i_k) \times w + \\
+ \text{baseA} - ((\ldots((\text{low}_1n_2+\text{low}_2)n_3+\text{low}_3)\ldots)m_k+\text{low}_k) \times w
\]

where \( n_j = \text{high}_j - \text{low}_j + 1 \) and \( w = \text{sizeof} (\text{baseType}(A)) \)

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
  
  $$\text{offset}_1 = i_1$$
  $$\text{offset}_m = \text{offset}_{m-1} \times n_m + i_m$$

  – At the end multiply by $$w = \text{sizeof(baseType(A))}$$
  – Add the compile-time constant term
  – Keep track of which dimension at each level
  – Use the auxiliary function $$n_m = \text{numElem}(A, m)$$ as the number of elements along the $$m^{th}$$ dimension of $$A$$
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;
  }

x = A[y,z];
SDT for Addressing Arrays Elements

E \rightarrow L \quad \{ \text{if } (L.\text{offset} = \text{NULL}) \text{ then} \\
\quad \quad E.\text{place} = L.\text{place}; \\
\quad \quad \text{else} \\
\quad \quad \quad E.\text{place} = \text{newtemp}; \\
\quad \quad \quad E.\text{code} = \text{gen}(E.\text{place} = L.\text{place}[L.\text{offset}]); \\
\}

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

L \rightarrow id \quad \{ L.\text{place} = id.\text{place}; \\
\quad L.\text{offset} = \text{null}; \\
\}
$x = A[y, z]$;

A is 10 x 20 array with $\text{low}_1 = \text{low}_2 = 1$

$\text{sizeof(baseType(A))} = \text{sizeof(int)} = 4$ bytes

SDT for Addressing Arrays Elements

$x = \text{Elist}$

$\text{Elist} = A[y, z]$
x = A[y, z];

A is 10 x 20 array with low₁ = low₂ = 1
sizeof(baseType(A)) = sizeof(int) = 4 bytes

t₁ = y * 20  // numElem(A, 2) = 20
t₁ = t₁ + z  // E.place is z
t₂ = c  // c = base(A) - (20+1)*4
t₃ = t₁ * 4  // sizeof(int) = 4
t₄ = t₂[t₃]
x = t₄
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 at each reference

Some improvement is possible
• Save lenᵢ and lowᵢ rather than lowᵢ and highᵢ
• Pre-compute the fixed terms in prologue sequence
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
- Contents of dope vector are fixed during each activation

→ Handle them like parameter arrays
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 instructions.
  – Use the same SDT scheme explained for arithmetic operations.

• Control Flow Evaluation (short circuit evaluation - later)
  – 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 symbolic location to be generated
      – laststat (Synthesized) Indicates the last location filled
      – As code is generated down and up the tree attributes are filled with the correct values
Arithmetic Scheme: Grammar and Actions

E → false \hspace{1em} || E.place = newtemp()
\hspace{2em} E.code = {gen(E.place = 0)}
\hspace{2em} E.laststat = E.nextstat + 1

E → true \hspace{1em} || E.place = newtemp()
\hspace{2em} E.code = {gen(E.place = 1)}
\hspace{2em} E.laststat = E.nextstat + 1

E → (E₁) \hspace{1em} || E.place = E₁.place;
\hspace{2em} E.code = E₁.code;
\hspace{2em} E₁.nextstat = E.nextstat
\hspace{2em} E.laststat = E₁.laststat

E → not E₁ \hspace{1em} || E.place = newtemp()
\hspace{2em} E.code = append(E₁.code, gen(E.place = not E₁.place))
\hspace{2em} E₁.nextstat = E.nextstat
\hspace{2em} E.laststat = E₁.laststat + 1
Arithmetic Scheme: Grammar and Actions

\[ E \rightarrow E_1 \text{ or } E_2 \parallel E\.place = \text{newtemp()} \]
\[ E\.code = \text{append}(E_1\.code,E_2\.code,\text{gen}(E\.place = E_1\.place \text{ or } E_2\.place)) \]
\[ E_1\.nextstat = E\.nexstat \]
\[ E_2\.nextstat = E_1\.laststat \]
\[ E\.laststat = E_2\.laststat + 1 \]

\[ E \rightarrow E_1 \text{ and } E_2 \parallel E\.place = \text{newtemp()} \]
\[ E\.code = \text{append}(E_1\.code,E_2\.code,\text{gen}(E\.place = E_1\.place \text{ and } E_2\.place)) \]
\[ E_1\.nextstat = E\.nexstat \]
\[ E_2\.nextstat = E_1\.laststat \]
\[ E\.laststat = E_2\.laststat + 1 \]

\[ E \rightarrow \text{id}_1 \text{ relop } \text{id}_2 \parallel E\.place = \text{newtemp()} \]
\[ E\.code = \text{gen}(\text{if } \text{id}_1\.place \text{ relop } \text{id}_2\.place \text{ goto } E\.nextstat+3) \]
\[ E\.code = \text{append}(E\.code,\text{gen}(E\.place = 0)) \]
\[ E\.code = \text{append}(E\.code,\text{gen}(\text{goto } E\.nextstat+2)) \]
\[ E\.code = \text{append}(E\.code,\text{gen}(E\.place = 1)) \]
\[ E\.laststat = E\.nextstat + 4 \]
Boolean Expressions: Example

\[ a < b \text{ or } c < d \text{ and } e < f \]

00: if \( a < b \) goto 03
01: \( t_1 = 0 \)
02: goto 04
03: \( t_1 = 1 \)
04: if \( c < d \) goto 07
05: \( t_2 = 0 \)
06: goto 08
07: \( t_2 = 1 \)
08: if \( e < f \) goto 11
09: \( t_3 = 0 \)
10: goto 12
11: \( t_3 = 1 \)
12: \( t_4 = t_2 \text{ and } t_3 \)
13: \( t_5 = t_1 \text{ or } t_4 \)
Summary

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