Compilers

by
Marwa Yusuf

Lecture 10
Mon. 26-4-2021

Chapter 6 (6.3 to 6.5.2)

Intermediate Code Generation

Types’ Applications

1) **Type Checking**: insures operands match operators’ needs (ex: &&).
2) **Translation Applications**: ex: storage needed for a name, address within array, type conversions, right version or arithmetic operator.

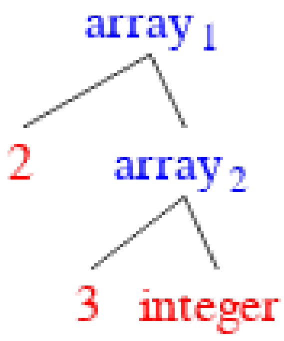
Note: Actual storage at runtime, relative addresses at compile time.

Type Expressions

- **Represent type structure.**

1) Basic type.
2) Applying type constructor on a type expression.

- Depend on the language.
- Ex: int[2][3] : *array*(2, *array*(3, *integer*))



Type Expressions Definition

What is considered a type expression:

- A basic type: boolean, integer, char, void (lack of value).
- A type name. (**typedef, classes**).
- Applying ***array*** type constructor to a number and a type expression.
- Applying ***record*** type constructor to field names and their types.
- Applying type constructor $\rightarrow$ for function types ($s \rightarrow t$).
- Cartesian Product of two type expressions, like function parameters ($s \times t$) (left associative, higher precedence than $\rightarrow$).
- May contain variables whose values are type expressions.

Can be represented by a graph.

Type Equivalence

- Type checking: “**if** 2 type expressions are equal **then** return a certain type **else** error”.
- Potential ambiguities: names given more than once. (a name stands for itself or it is an abbreviation?)
- 2 type expressions are equivalent iff either:
  - They are the same basic type.
  - They are formed by applying the same constructor to structurally equivalent types.
  - One is a type name that denotes the other.

Declarations

- Grammar declaring one name at a time:

$D \rightarrow T\ \textbf{id}\ ;\ D \mid \varepsilon$

$T \rightarrow B\ C \mid \textbf{record}\ \text{`\{`}\ D\ \text{`\}'}$

$B \rightarrow \textbf{int} \mid \textbf{float}$

$C \rightarrow \varepsilon \mid [\ \textbf{num}\ ]\ C$

Storage Layout for Local Names

- Type determines amount of runtime storage needed for a name. Relative address can be set at compile time. Type and relative address are stored in symbol table.
- Varying length data (strings) and data whose size is determined at runtime (dynamic arrays) are handled using fixed sized pointers.

Storage Layout for Local Names

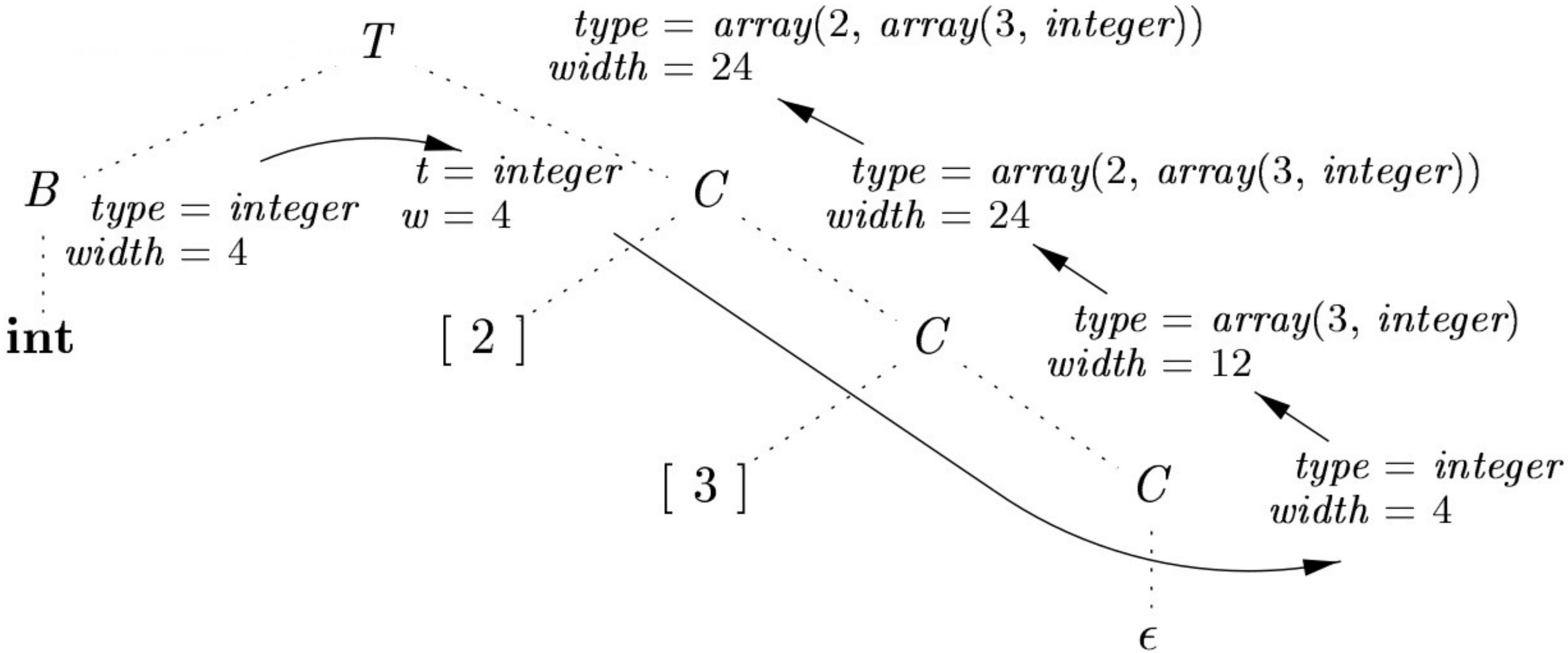
- Width of type: # of storage units needed:
  - Basics: integral # of bytes.
  - Aggregates: allocated in one contiguous block of bytes.

$T \to B$	$\{ t = B.type;\ w = B.width; \}$
$\quad C$	$\{T.type = C.type;\ T.width = C.width; \}$
$B \to \textbf{int}$	$\{ B.type = integer;\ B.width = 4; \}$
$B \to \textbf{float}$	$\{ B.type = float;\ B.width = 8; \}$
$C \to \varepsilon$	$\{ C.type = t;\ C.width = w;$
$C \to [\ \textbf{num}\ ] C_1$	$\{ C.type = array(\textbf{num}.value,\ C_1.type);$ $C.width = \textbf{num}.value \times C_1.width; \}$

~

Example

- `int [2][3]`



Sequences of Declaratios

$P \rightarrow \quad \{ \ offset = 0; \}$

$\quad D$

$D \rightarrow T \ \textbf{id} \ ; \quad \{ \ top.put(\textbf{id}.lexeme, \ T.type, \ offset); \ offset = offset + T.width; \}$

$\quad D_1$

$D \rightarrow \varepsilon$

- **Note:** $\boldsymbol{P \rightarrow \{ \ offset = 0; \} \ D}$
- **Using marker non-terminals:**

$P \rightarrow M \ D$

$M \rightarrow \varepsilon \quad \{ \ offset = 0; \}$

Fields in Records and Classes

- For record, add $T \rightarrow$ **record** '{' D '}'
  - Unique field names within a record.
  - Offset for a field name relative to record's data area.
- Ex:
```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
x = p.x + q.x;
```
- Use a symbol table for record type (for types and relative addresses).
- Record type : *record*(t): record (type constructor), t (symbol table object)

Fields in Records and Classes

Add to the prev. grammar:

$T \rightarrow$ **record** '{' $\{ \, Env.push(top);\ top = \textbf{new}\ Env();$
$Stack.push(offset);\ offset = 0;\ \}$

D '}' $\{ T.type = record(top);\ T.width = offset;$
$top = Env.pop();\ offset =$
$Stack.pop();\ \}$

Translation of Expressions

- Expression like `a + b * c` will be translated to a series of single operator instructions.
- Array reference like `A[i][j]` will be translated to a series of instructions to calculate array element address.

Operations within Expressions

- SDD for generating three address code for expressions
- Ex: `a = b + -c`
- `t`$_1$ `= minus c`

  `t`$_2$ `= b + t`$_1$

  `a = t`$_2$

PRODUCTION	**SEMANTIC RULES**
$S \rightarrow$ **id** $= E\ ;$	S.code = E.code || gen(top.get(**id**.lexeme) '=' E.addr)
$E \rightarrow E_1 + E_2$	$E.addr$ = **new** $Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid$ $gen(E.addr$ '=' $E_1.addr$ '+' $E_2.addr)$
$\mid\ - E_1$	$E.addr$ = **new** $Temp()$ $E.code = E_1.code \mid\mid$ $gen(E.addr$ '=' '**minus**' $E_1.addr)$
$\mid\ (E_1)$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid$ **id**	$E.addr = top.get($**id**$.lexeme)$ $E.code$ = ''

Incremental Translation

- Translation scheme for generating three address code for expressions
- Ex: `a = b + -c`

PRODUCTION	CODE FRAGMENTS
$S \rightarrow \textbf{id} = E\ ;$	$\{gen(\ top.get(\textbf{id}.lexeme)\ \text{'='}\ E.addr);\}$
$E \rightarrow E_1 + E_2$	$\{E.addr = \textbf{new}\ Temp();$ gen$(E.addr\ \text{'='}\ E_1.addr\ \text{'+'}\ E_2.addr);\}$
$\mid\ \text{-}\ E_1$	$\{E.addr = \textbf{new}\ Temp();$ $gen(E.addr\ \text{'='}\ \text{'}\textbf{minus}\text{'}\ E_1.addr);\}$
$\mid\ (\ E_1\ )$	$\{E.addr = E_1.addr;\}$
$\mid\ \textbf{id}$	$\{E.addr = top.get(\textbf{id}.lexeme);\}$

- t_1 = minus c

  $t_2 = b + t_1$

  $a = t_2$

Addressing Array Elements

- Array elements stored in a block of consecutive locations.
- Address of element i is: $\boldsymbol{base + i * w}$
- For $A[i_1][i_2]$: $\boldsymbol{base + i_1 * w_1 + i_2 * w_2}$
- k dim: $\boldsymbol{base + i_1 * w_1 + i_2 * w_2 + \ldots + i_k * w_k}$
- Or: $\boldsymbol{base + (i_1 * n_2 + i_2) * w}$

  $\boldsymbol{base + ((\ldots(i_1 * n_2 + i_2)*n_3 + i_3)\ldots)*n_k + i_k) * w}$
- If array starts at *low*:

  $\boldsymbol{base + (i - low) * w}$

  $\boldsymbol{c + i * w}$ where $c = base - low * w$ (calculated at compile time).
- Pre-calculations for multi-dimensional array elements also.
- In case of dynamic size array, no pre-calculations.

Addressing Array Elements

- The prev. calculations are for row-major layout. (C and Java).
- There is also column-major layout (Fortran).

A[1, 1]	A[1, 1]
A[1, 2]	A[2, 1]
A[1, 3]	A[1, 2]
A[2, 1]	A[2, 2]
A[2, 2]	A[1, 3]
A[2, 3]	A[2, 3]

Translation of Array References

PRODUCTION	CODE FRAGMENTS
$S \rightarrow$ **id** $= E$;	{ *gen*(*top.get*(**id**.*lexeme*) '=' *E.addr*); }
| $L = E$;	{ gen(*L.addr.base* '[' *L.addr* ']' '=' *E.addr*); }
$E \rightarrow E_1 + E_2$	{ *E.addr* = **new** *Temp*(); *gen*(*E.addr* '=' E_1.*addr* '+' E_2.*addr*); }
| **id**	{ *E.addr* = *top.get*(**id**.*lexeme*); }
| L	{ *E.addr* = **new** *Temp*(); *gen*(*E.addr* '=' *L.addr.base* '[' *L.addr* ']'); }
$L \rightarrow$ **id** [E]	{ *L.array* = *top.get*(**id**.*lexeme*); *L.type* = *L.array.type.elem;* *L.addr* = **new** *Temp*(); *gen*(*L.addr* '=' *E.addr* '*' *L.type.width*); }

| L_1 [E]	{ *L.array* = L_1.*array*;
	L.type = L_1.*type.elem*;
	t = **new** *Temp*();
	L.addr = **new** *Temp*();
	gen(*t* '=' *E.addr* '*' *L.type.width*);
	gen(*L.addr* '=' *L1.addr* '+' *t*); }

Example

- `c + a[i][j]`
- (let `a = int[2][3]`)

- t_1 = i * 12
  t_2 = j * 4
  t_3 = t_1 + t_2
  t_4 = a [t_3]
  t_5 = c + t_4