CSE443 Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

Phases of a compiler

Intermediate Representation (IR): specification and generation

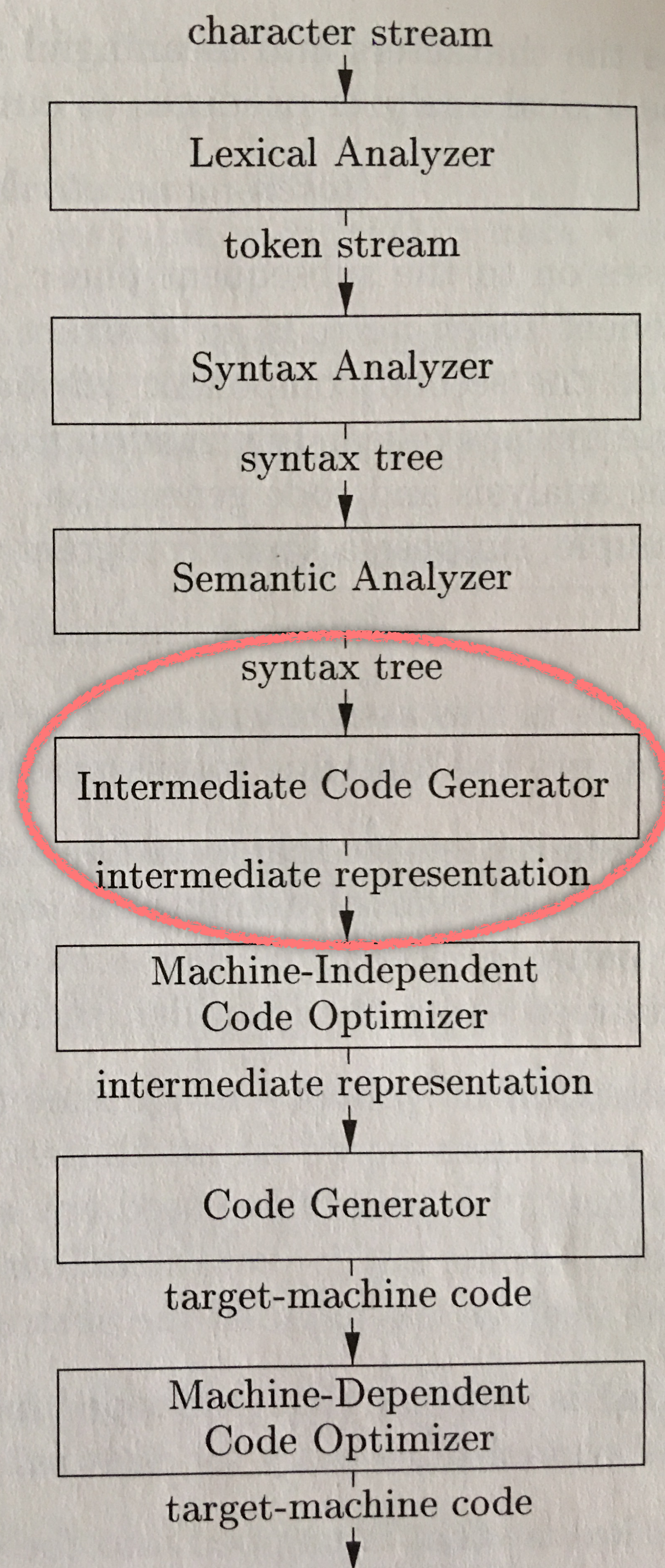


Figure 1.6, page 5 of text

Three-address code

- The DAG does not say anything about how the computation should be carried out.
- For example, there could be one instruction to do this computation:

  x+y*z

  as in,

  $t_1 = x + y * z$

Three-address code

- In three-address code instructions can have no more than one operator on the right of an assignment.
- x+y*z must be broken into two instructions:

$$t_1 = y * z$$
$$t_2 = x + t_1$$

Three address code representation

"Three-address code is a linearized representation of ... a DAG in which explicit names correspond to the interior nodes of the graph." [p. 363]

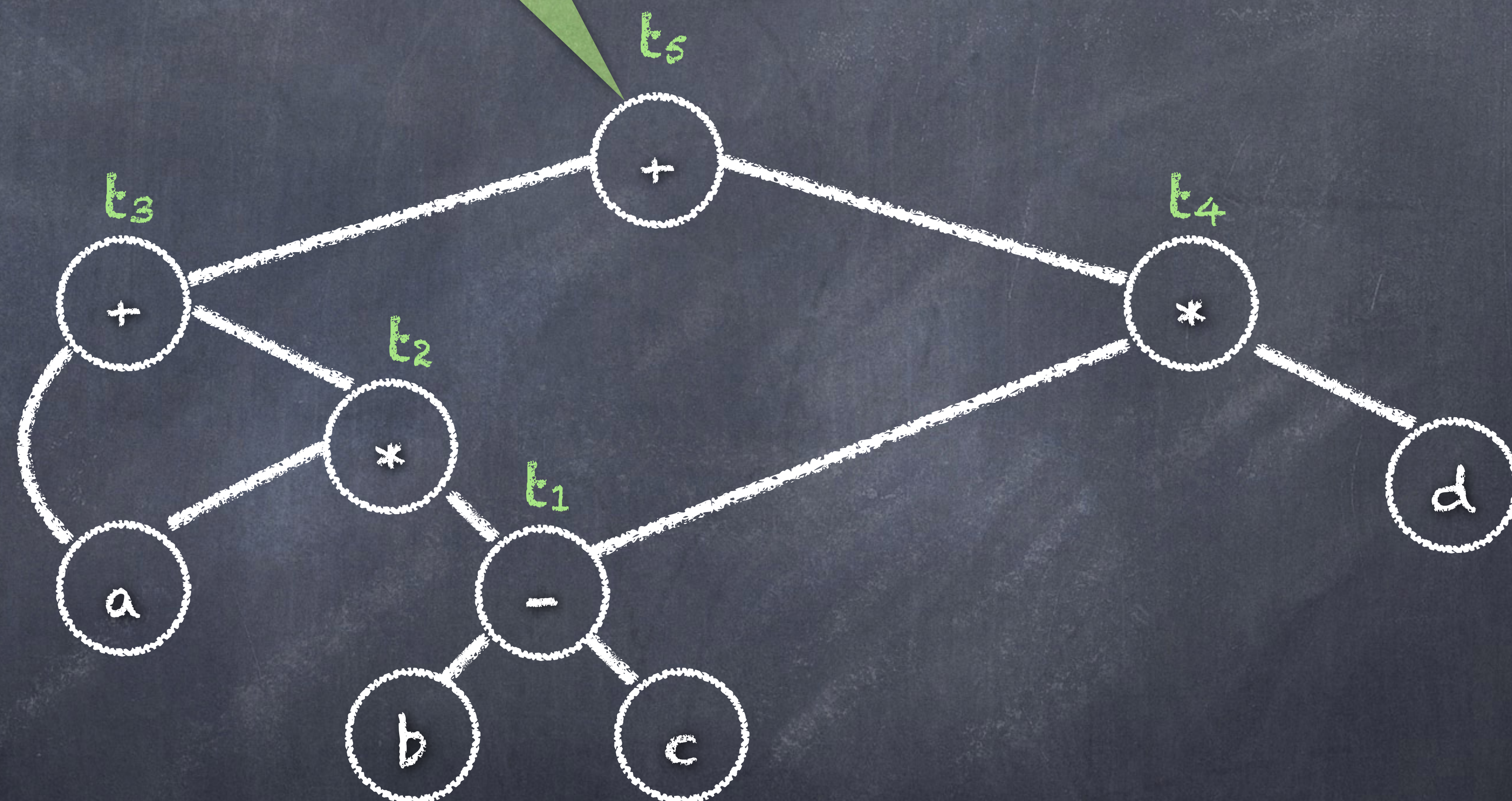
$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$



Three address code instructions (see 6.2.1, pages 364-5)

1. x = y op z
2. x = op y
3. x = y
4. goto L
5. if x goto L / ifFalse x goto L
6. if x relop y goto L
7. function calls:
   - param x
   - call p, n
   - y = call p
   - return y
8. x = y[i] and x[i] = y
9. x = &y, x = *y, *x = y

Three address code instructions (see 6.2.1, pages 364-5)

1. x = y op z
2. x = op y
3. x = y
4. goto L
5. if x goto L / ifFalse x goto L
6. if x relop y goto L
7. function calls:
   - param x
   - call p, n
   - y = call p
   - return y
8. x = y[i] and x[i] = y
9. x = &y, x = *y, *x = y

We'll start with these.

We'll spend significant time on function calls later.

We'll explore these as needed later on.

Representation options

"The description of three-address instructions specifies the components of each type of instruction, but it does not specify the representation of these instructions in a data structure."

[p. 366]

Quadruples

Instructions have four fields:

op, arg1, arg2, result

Example: $t_3 = a + t_2$ is represented as

op	arg1	arg2	result
+	a	t_2	t_3

Example: $t_4 = - c$ is represented as

op	arg1	arg2	result
minus	c		t_4

Variables in representation

Identifiers would be pointers to symbol table entries. Compiler-introduced temporaries can be added to the symbol table.

op	arg1	arg2	result
+	-> entry for a	-> entry for t_2	-> entry for t_3

Triples

Instructions have three fields:

op, arg1, arg2

Example:

$t_2 = \ldots$

$t_3 = a + t_2$

is represented as

line	op	arg1	arg2
5	computation of t_2		
6	+	a	(5)

Indirect triples

Because order matters (due to embedded references instead of explicit variables) it is more challenging to rearrange instructions with triples than with quadruples.

Indirect triples allow for easier reordering (see page 369).

Indirect Triples

Instructions have three fields:

op, arg1, arg2

Example:

$t_2 = \ldots$

$t_3 = a + t_2$

is represented as

Rearranging instructions changes the instruction array contents, but the instructions themselves do not change.

index	instruction	*line*	op	arg1	arg2
72	5	5	computation of t_2		
73	6	6	+	a	((72))

Static Single Assignment (SSA)

an additional constraint on the three address code

"[SSA] is an intermediate representation that facilitates certain code optimizations." [p. 369]

1) Each variable is assigned to exactly once. Occurrences of the same variable are subscripted to make them unique.

$x = r + 1$	$x_1 = r + 1$
$y = s * 2$	$y_1 = s * 2$
$x = 2 * x + y$	$x_2 = 2 * x_1 + y_1$
$y = y + 1$	$y_2 = y_1 + 1$

Static Single Assignment (SSA)

an additional constraint on the three address code

1) Each variable is assigned to exactly once.

2) Need ϕ function to merge split variables:

```
if (e) then { x = a } else { x = b }
y = x
```

With SSA:

if (e) then { $x_1 = a$ } else { $x_2 = b$ }

$y = \phi(x_1 , x_2)$

ϕ function implementation

In $y = \phi(x1,x2)$ simply let x1 and x2 be bound to the same address.

Typical project trajectory

- Sprint 1: char stream -> LEXER -> token stream
- Sprint 2: PARSER builds symbol table, checks for undefined or multiply defined names from token stream.
- Sprint 3: PARSER will also perform type checking and generate intermediate code.

type information

- What information does a type convey?

- How is type information used during compilation?

type information

- What information does a type convey?
  - type indicates size
  - type indicates storage location
    - (a) primitives: either stack or heap
    - (b) records: on heap (via pointer)
    - (c) arrays: on heap (via pointer)
    - (d) functions: code in static, locals on stack
- How is type information used during compilation?

type information

- What information does a type convey?
  - type indicates size
  - type indicates storage location
    - (a) primitives: either stack or heap
    - (b) records: on heap (via pointer)
    - (c) arrays: on heap (via pointer)
    - (d) functions: code in static, locals on stack
- How is type information used during compilation?
  - determines how to lay out records, arrays, invocation records in memory
  - determines how to translate names in program to memory accesses
  - determines which instructions to use to manipulate values in memory

Sizes of types

- int: 32 bits (2's complement)
- real: 64 bits (IEEE 754)
- Boolean: 8 bits (TBD: machine dependent)
- character: 8 bit (ASCII)
- address: 64 bits

Sizes/layouts of values of types

- type string: 1 -> character
  - 4 bytes + length of string * size of character (= 1 byte)
  - # of dimensions is part of type

size of dimension 1 (integer)				(0)	(1)	(2)	(3)	(4)
0	0	0	5	V	A	X	E	S

https://en.wikipedia.org/wiki/VAX

Array layout in memory

Two options:

- row-major
- column-major

Textbook discusses on page 382; row-major and column-major refer to two-dimensional arrays, but can be generalized for arrays with more dimensions.

Row-major array layout

What is the size of an X-dimensional array of type T?

sizes of dimensions (S_i): X*4 bytes

data: $(\Pi_{i \in X} S_i) * \text{sizeOf}(T)$
(plus padding for real to get to proper boundary)

Example shows two-dimensional array (2 rows, 3 columns)

0	
0	size of first dimension
0	
2	
0	
0	size of second dimension
0	
3	
a(0,0)	
a(0,1)	first row
a(0,2)	
a(1,0)	
a(1,1)	second row
a(1,2)	

Column-major array layout

What is the size of an X-dimensional array of type T?

sizes of dimensions (S_i): X*4 bytes

data: $(\prod_{i \in X} S_i)$ * sizeOf(T)

Example shows two-dimensional array (2 rows, 3 columns)

0	size of first dimension
0	
0	
2	
0	size of second dimension
0	
0	
3	
a(0,0)	first col
a(1,0)	
a(0,1)	second col
a(1,1)	
a(0,2)	third col
a(1,2)	

Variables and memory

- Variables have names in our high level programs
- Names don't exist at runtime
- Variables are allocated space in a block of memory
    - local variables have space in a stack frame (a.k.a. invocation record)
    - array cells and record members have space in heap-allocated block of memory

Variables and memory

- Every use of a variable is translated into an address by the compiler…

  …but not an absolute address - we have no idea where in memory things will be loaded!

- For every allocated block of memory there is a base/reference address.
- Variables housed within each block have a location in the block that is relative to the base/reference address.

Variables and memory

- The relative address is expressed as an offset from the base/reference address.
- The offset is determined by
  - where other variables in the block are located,
  - how much space is needed to hold the variable's type of value, and
  - whether or not we need to align the starting address on a specific boundary.

Arrays

base address and offset (0) for size of first dimension

What is the size of a multi-dimensional array of type T?

sizes of dimensions (S_i): X*4 bytes

data: $(\Pi_{i \in X} S_i) * \mathrm{sizeOf}(T)$

assume sizeOf(T) is 1

address for size of second dimension (and offset 4)

address for a(0,0): offset 8

address for a(0,1): offset 9

address for a(0,2): offset 10

address for a(1,0): offset 11

etc.

0	size of first dimension
0	
0	
2	
0	size of second dimension
0	
0	
3	
a(0,0)	first row
a(0,1)	
a(0,2)	
a(1,0)	second row
a(1,1)	
a(1,2)	

Scopes

dblocks (6.3.5 and 6.3.6)

records (in separate symbol table), sequence of declarations at start of sblock

dblock → '['
{ Stack.push(offset); }
declaration-list ']'
{ offset=Stack.pop(); }

Since declarations must be gathered together at the start of an sblock, and cannot themselves be directly nested: keep running offset, but remember old offset when entering embedded scope.

---

offset = 0 offset = 4 offset = 8

{ [integer : x , y]

push offset = 8 onto stack

offset = 8 offset = 16 offset = 24

{ [real : x , z] … … }

pop offset = 8 from stack

push offset = 8 onto stack

offset = 8 offset = 9 offset = 10

{ [Boolean : y ; character : z]

… …

}

pop offset = 8 from stack

}