IR Generation

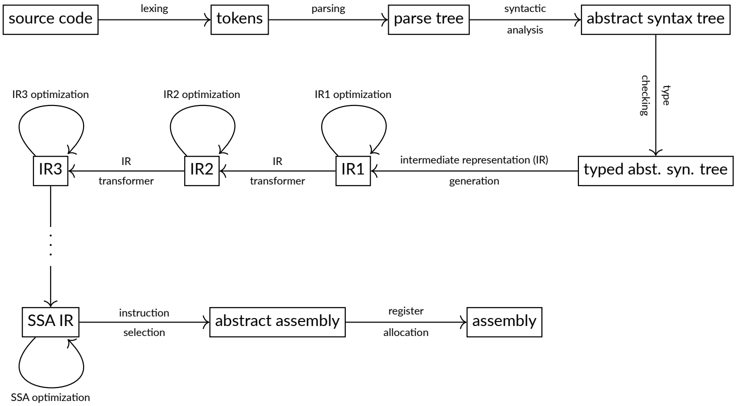
(CSC 3F002 EP) Compilers

Pierre-Yves Strub

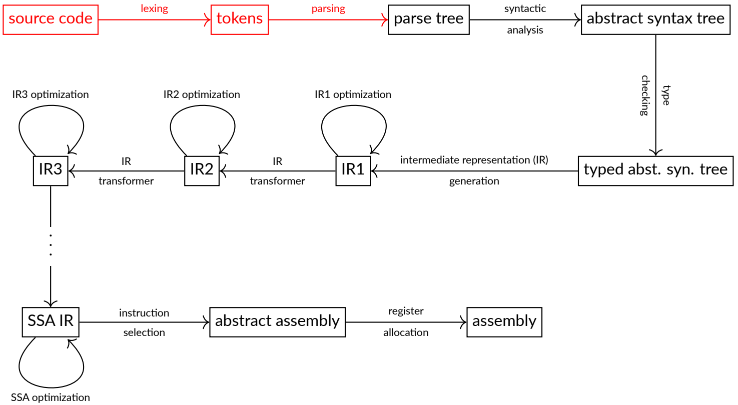
(Slide deck author: **Kaustuv Chaudhuri**)

2025-09-08 // Week 2

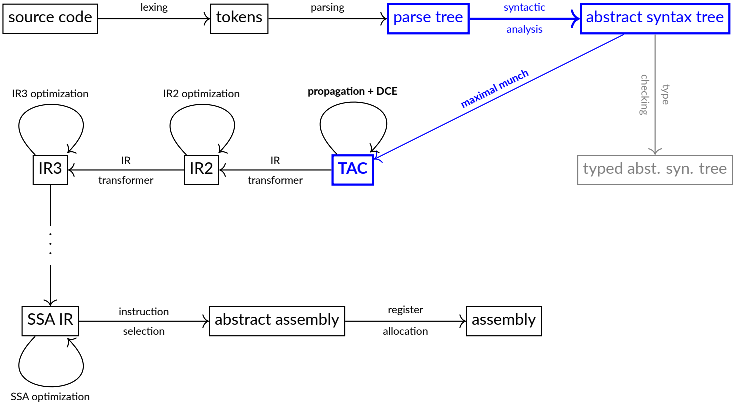
Compiler Stages: Block Diagram



Last lecture



Today



source code
lexing
tokens
parsing
parse tree
syntactic
analysis
abstract syntax tree
type
checking
typed abst. syn. tree
maximal munch
TAC
propagation + DCE
IR
transformer
IR2
IR2 optimization
IR
transformer
IR3
IR3 optimization
SSA IR
SSA optimization
instruction
selection
abstract assembly
register
allocation
assembly

Reminder: straightline BX

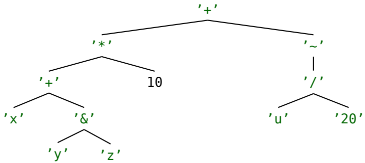
- A simplification of BX with:
  - Only one type of variables, signed 64-bit `int`s
  - No control structures
  - Only one function, `main()`
  - Only I/O: printing `int`s
- BX expressions:
  - Arithmetic (`+`, `-`, `*`, `/`, `%`)
  - Bitwise ops (`~`, `&`, `|`, `^`)
  - Arithmetic shifting (`<<`, `>>`)
- BX statements:
  - Assignment: `x = expression;`
  - Print: `print(expression);`

Reminder: BX Expressions and Assignments

```
x = 10;             // assign immediate

y = 2 * x;          // compute and assign
z = y / 2;          // compute and assign

w = z - x - y;      // compound expression

v = - w - - w;      // - is both binary and unary
                    // parsed as (- w) - (- w), i.e., 0

m = x + y * 2;      // precedence: * binds tighter than +
                    //    so, parsed as x + (y * 2)
n = (x + y) * 2;    // parentheses to force evaluation orders
```

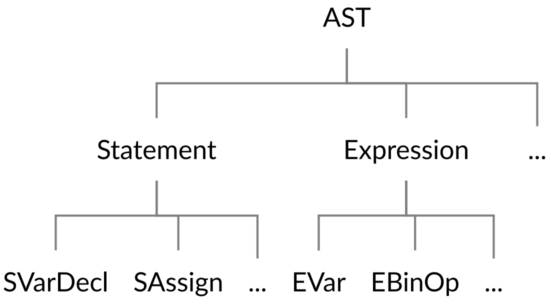
Abstract Syntax Tree

- Programs are represented using an Parse Tree (PST) / Abstract Syntax Tree (AST)
- PST is what is given by the parser. AST is more abstract, can contain extra information (e.g. typing information). The line between PST/AST is thin and sometimes the parser outputs an AST directly.
- For example, ⟨expr⟩s are tree-like with arbitrary depth



- Order of evaluation of an ⟨expr⟩ depends on traversal (preorder, postorder, ...)
- In OOP, PST/AST are defined using a *class hierarchy*

Example of an AST hierarchy



- Each class has several data attributes that store the relevement information. For example, the `SVarDecl` class (Variable Declaration) contains the variable name (a `str`), its type and its initializer (an `Expression`)
- This hierarchy is recursive. For example, the `EBinOp` (Binary Operator Applicaton) contains two `Expression` (the operands).

AST definition in Python

```python
import abc
import dataclasses as dc

@dc.dataclass
class AST(abc.ABC):
  pass

@dc.dataclass
class Statement(AST):
  pass

@dc.dataclass
class Expression(AST):
  pass
```

```python
@dc.dataclass
class SVarDecl(Statement):
  name: str
  type: str
  init: Expression

@dc.dataclass
class SAssignment(Statement):
  lvalue: str
  rvalue: Expression

@dc.dataclass
class EVar(Expression):
  name: str

@dc.dataclass
class EBinOp(Expression):
  binop: str
  left: Expression
  right: Expression
```

AST transformers

There are different ways to work with an AST. The ugly one:

```python
def do_something(ast : AST):
  if isinstance(ast, EVar):
    # Do something
  elif isinstance(ast, EBinOp):
    # Do something else
  # ...
```

- Error prone, hard to read, huge functions, ...
- Pass your way

AST transformers

The OOP way:

```python
@dc.dataclass
class AST(abc.ABC):
  @abc.abstractmethod
  def transformer1(self):
    pass

  @abc.abstractmethod
  def transformer2(self, thearg):
    pass
```

```python
@dc.dataclass
class SAssignment:
  lvalue: str
  rvalue: Expression

  def transformer1(self):
    # Do something (1)

  def transformer2(self, thearg):
    # Do something (2)
```

- Classic representation in OOP, but...
- ...transformers' bodies are spread across multiple classes.
- it is also possible to use the visitor design pattern.

AST transformers

Python has now a notion of pattern matching that plays well with the `dataclasses` module:

```
def transformer1_expr(e : Expression):
  match e:
    case EVar(name):
      # Do something. Can use the ‘name‘ variable
    case EBinOp(_, left, right):
      # Do something. ‘_‘ means that the
      # first data attribute has been
      # ignored (not bound)
```

```
def transformer1_stmt(s : Statement):
  match e:
    case SVarDecl(name = x, init = e):
      # Data attributes can be matched
      # using their names (more robust).
    case SAssign(lvalue, rvalue):
      # Do something.
```

- Close to algebraic pattern matching
- Widely used in functional programming
- Very effective when dealing with an AST

AST Linearization

- Order of evaluation of an $\langle$expr$\rangle$ depends on traversal (preorder, postorder, ...)

- Instructions are shallow –
  operands can be registers, memory addresses, or numbers (depth 0)
- Instructions are sequence-like
- Order of evaluation is explicit – top-to-bottom

- Instruction Generation: going from tree-like to list-like structures

BX (AST) to TAC: Example

BX

```
def main() {
  // variable declarations
  var x = 0 : int;
  var y = 0 : int;

  // straightline code
  x = 10;
  y = 2 * x;
  x = y * y / 2;
  print(9 * x * x + 3 * x - 8);
}
```

→

TAC

```
proc @main:
  %0 = const 0;
  %1 = const 0;
  %0 = const 10;
  %2 = const 2;
  %1 = mul %2, %0;
  %4 = mul %1, %1;
  %7 = const 2;
  %0 = div %4, %7;
  %12 = const 9;
  %11 = mul %12, %0;
  %10 = mul %11, %0;
  %16 = const 3;
  %15 = mul %16, %0;
  %9 = add %10, %15;
  %18 = const 8;
  %8 = sub %9, %18;
  print %8;
```

Three Address Code (TAC)

(our first intermediate representation, IR)

- A (low-level) language that is closer to assembly than BX
- Temporaries:
  - Think: *abstract* register
  - Infinite supply – can use arbitrarily many
  - Anonymous temporaries: `%0`, `%1`, `%42`, … (*we will use these mostly*)
  - Named temporaries: `%x`, `%foo`, …
- Instructions:
  - Unique opcode
  - Zero, one, or two argument temporaries (*with one exception:* `const`)
  - Zero or one result temporary that is written
  - Prototype syntax: $\underbrace{\texttt{\%42 =}}_{\text{result}}\ \underbrace{\texttt{add}}_{\text{opcode}}\ \underbrace{\texttt{\%10, \%3}}_{\text{args}}$

TAC Instructions Reference

(%1, %2 = examples of reads; %0 = example of write; 42 = example of immediate)

Instruction	Description
`%0 = const 42;`	Set temporary `%0` to the value `42`.
`%0 = copy %1;`	Copy the value of `%1` to `%0`.
`%0 = binop %1, %2;`	Compute the value of binary operator `binop` ∈ {`add`, `sub`, `mul`, `div`, `mod`, `and`, `or`, `xor`, `shl`, `shr`} applied to `%1` and `%2` and store in `%0`.
`%0 = unop %1;`	Compute the value of a unary operator `unop` ∈ {`neg`, `not`} applied to `%1` and store in `%0`.
`print %1;`	Print the value of `%1` to the standard output.

Algorithm 1: Top-Down Maximal Munch (TMM)

High-level overview

- Start from the root of the AST and do a postorder (i.e., children before root) traversal
- Allocate fresh temporaries for each child node (if any) and generate instructions (recursively!) to output the value fo that child to that temporary
- Finally, generate the instruction for the root using the temporaries of the children

Algorithm 1: Top-Down Maximal Munch (TMM)

High-level overview

- Start from the root of the AST and do a postorder (i.e., children before root) traversal
- Allocate fresh temporaries for each child node (if any) and generate instructions (recursively!) to output the value fo that child to that temporary
- Finally, generate the instruction for the root using the temporaries of the children

We will now flesh this out.

Top-Down Maximal Munch: Expressions

code (e, x)

- Generate code for the expression e.
- Storing the result in the temporary x.
- Returns: list of instructions.

e	code (e, x)	proviso
42	$[x = \texttt{const}\ 42;]$	—
y	$[x = \texttt{copy}\ y;]$	—
$e_1 + e_2$	code (e_1, y) + code (e_2, z) + $[x = \texttt{add}\ y, z;]$	y, z fresh
- e_1	code (e_1, y) + $[x = \texttt{neg}\ y;]$	y fresh

Top-Down Maximal Munch: Statements

code (s)

- Generate code for the statement s.
- Returns: list of instructions.

s	code (s)	proviso
x = e;	code (e, x)	—
`print`(e);	code (e, x) + [`print` x;]	x fresh

Top-Down Maximal Munch — Example

BX

```
x = 10;
y = 2 * x;
x = y * y / 2;
print(9 * x * x + 3 * x - 8);
```

TMM

TAC

```
%0 = const 10;
%2 = const 2;
%3 = copy %0;
%1 = mul %2, %3;
%5 = copy %1;
%6 = copy %1;
%4 = mul %5, %6;
%7 = const 2;
%0 = div %4, %7;
%12 = const 9;
%13 = copy %0;
%11 = mul %12, %13;
%14 = copy %0;
%10 = mul %11, %14;
%16 = const 3;
%17 = copy %0;
%15 = mul %16, %17;
%9 = add %10, %15;
%18 = const 8;
%8 = sub %9, %18;
print %8;
```

Top-Down Maximal Munch — Example

BX

```
x = 10;
y = 2 * x;
x = y * y / 2;
print(9 * x * x + 3 * x - 8);
```

TMM →

TAC

```
%0 = const 10;
%2 = const 2;
%3 = copy %0;
%1 = mul %2, %3;
%5 = copy %1;
%6 = copy %1;
%4 = mul %5, %6;
%7 = const 2;
%0 = div %4, %7;
%12 = const 9;
%13 = copy %0;
%11 = mul %12, %13;
%14 = copy %0;
%10 = mul %11, %14;
%16 = const 3;
%17 = copy %0;
%15 = mul %16, %17;
%9 = add %10, %15;
%18 = const 8;
%8 = sub %9, %18;
print %8;
```

TMM generates many (often redundant) copies!

Algorithm 2: Bottom-Up Maximal Munch (BMM)

Key idea: make the destination temporary an output instead of an input.

$(x, L) = \text{code}\,(e)$

- Generate code (L) for the expression e.
- Storing the result in the temporary x.
- Returns: both x and L

e	$\text{code}\,(e)$	proviso
42	x, [x = `const 42;`]	x fresh
y	y, [ ]	—
e_1 + e_2	x, L_1 + L_2 + [x = `add` y, z;] where $(y, L_1) = \text{code}\,(e_1)$ and $(z, L_2) = \text{code}\,(e_2)$	x fresh
- e_1	x, L_1 + [x = `neg` y;] where $(y, L_1) = \text{code}\,(e_1)$	x fresh

Bottom-Up Maximal Munch: Statements

$\text{code}\,(s) = L$

- Generate code for the statement s
- Returns: list L of instructions

s	$\text{code}\,(s)$	proviso
$x = e;$	$L + [x = \texttt{copy}\ y;]$ where $(y, L) = \text{code}\,(e)$	—
$\texttt{print}(e);$	$L + [\texttt{print}\ x;]$ where $(x, L) = \text{code}\,(e)$	—

Both Maximal Munches — Example

TAC

```
%0 = const 10;
%2 = const 2;
%3 = copy %0;
%1 = mul %2, %3;
%5 = copy %1;
%6 = copy %1;
%4 = mul %5, %6;
%7 = const 2;
%0 = div %4, %7;
%12 = const 9;
%13 = copy %0;
%11 = mul %12, %13;
%14 = copy %0;
%10 = mul %11, %14;
%16 = const 3;
%17 = copy %0;
%15 = mul %16, %17;
%9 = add %10, %15;
%18 = const 8;
%8 = sub %9, %18;
print %8;
```

← TMM

BX

```
x = 10;
y = 2 * x;
x = y * y / 2;
print(9 * x * x + 3 * x -
      8);
```

BMM →

TAC

```
%0 = const 10;
%1 = copy %0;
%2 = const 2;
%3 = mul %2, %1;
%4 = copy %3;
%5 = mul %4, %4;
%6 = const 2;
%7 = div %5, %6;
%1 = copy %7;
%8 = const 9;
%9 = mul %8, %1;
%10 = mul %9, %1;
%11 = const 3;
%12 = mul %11, %1;
%13 = add %10, %12;
%14 = const 8;
%15 = sub %13, %14;
print %15;
```

Summary of Lecture 1

- The frontend produces an parse/abstract syntax tree (AST)
- Parse/abstract syntax tree are represented using tree-like structures. The concrete representation depends on the programming language
  - Class hierarchy,
  - Algebraic datatypes,
  - ...
- AST is linearized into instructions with maximal munch
  - Top-Down: `copy` on reads
  - Bottom-Up: `copy` on writes