YSC4230: Programming Language Design and Implementation

Week 5: LLVMlite and Lexing

Ilya Sergey

ilya.sergey@yale-nus.edu.sg

Announcements

- HW3: LLVMlite
  - Will be available on Canvas and GitHub on Saturday.
  - Due: Tuesday, 28 September 2020 at 23:59:59

Representing Data Types

Working with Arrays

Arrays

```
void foo() {
  char buf[27];

  buf[0] = 'a';
  buf[1] = 'b';
  ...
  buf[25] = 'z';
  buf[26] = 0;
}
```

```
void foo() {
  char buf[27];

  *(buf) = 'a';
  *(buf+1) = 'b';
  ...
  *(buf+25) = 'z';
  *(buf+26) = 0;
}
```

- Space is allocated on the stack for buf.
  - Note, without the ability to allocated stack space dynamically (C's alloca function) need to know size of buf at compile time…
- buf[i] is really just
  (base_of_array) + i * elt_size

Multi-Dimensional Arrays

- In C, int M[4][3] yields an array with 4 rows and 3 columns.
- Laid out in *row-major* order:

M[0][0]	M[0][1]	M[0][2]	M[1][0]	M[1][1]	M[1][2]	M[2][0]	…

- In Fortran, arrays are laid out in *column major order*.

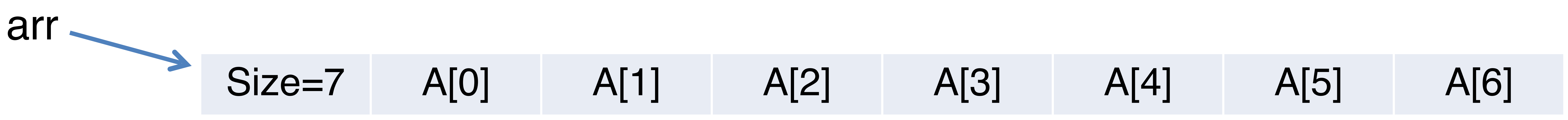
M[0][0]	M[1][0]	M[2][0]	M[3][0]	M[0][1]	M[1][1]	M[2][1]	…

- In ML and Java, there are no multi-dimensional arrays:
  - (int array) array is represented as an array of pointers to arrays of ints.
- Why is knowing these memory layout strategies important?

Array Bounds Checks

- Safe languages (e.g. Java, C#, ML but not C, C++) check array indices to ensure that they're in bounds.
  - Compiler generates code to test that the computed offset is legal

- Needs to know the size of the array… where to store it?
  - One answer: Store the size *before* the array contents.



- Other possibilities:
  - Pascal: only permit statically known array sizes (very unwieldy in practice)
  - What about multi-dimensional arrays?

Array Bounds Checks (Implementation)

- Example: Assume %rax holds the base pointer (arr) and %ecx holds the array index i. To read a value from the array arr[i]:

```
        movq -8(%rax) %rdx                 // load size into rdx
        cmpq %rdx %rcx                     // compare index to bound
        j l __ok                           // jump if 0 <= i < size
        callq __err_oob                    // test failed, call the error handler
__ok:
        movq (%rax, %rcx, 8) dest          // do the load from the array access
```

- Clearly more expensive: adds move, comparison & jump
  - More memory traffic
  - Hardware can improve performance: executing instructions in parallel, branch prediction
- These overheads are particularly bad in an inner loop
- Compiler optimisations can help remove the overhead
  - e.g. In a for loop, if bound on index is known, only do the test once

C-style Strings

- A string constant "foo" is represented as global data:

  _string42: 102 111 111 0

- C uses null-terminated strings
- Strings are usually placed in the *text* segment so they are *read only*.
  - allows all copies of the same string to be shared.

- Rookie mistake (in C): write to a string constant.

```
char *p = "foo”;
p[0] = 'b’;
```

  Attempting to modify the string literal is *undefined behaviour.*

- Instead, must allocate space on the heap:

```
char *p = (char *)malloc(4 * sizeof(char));
strncpy(p, “foo”, 4);   /* include the null byte */
p[0] = 'b’;
```

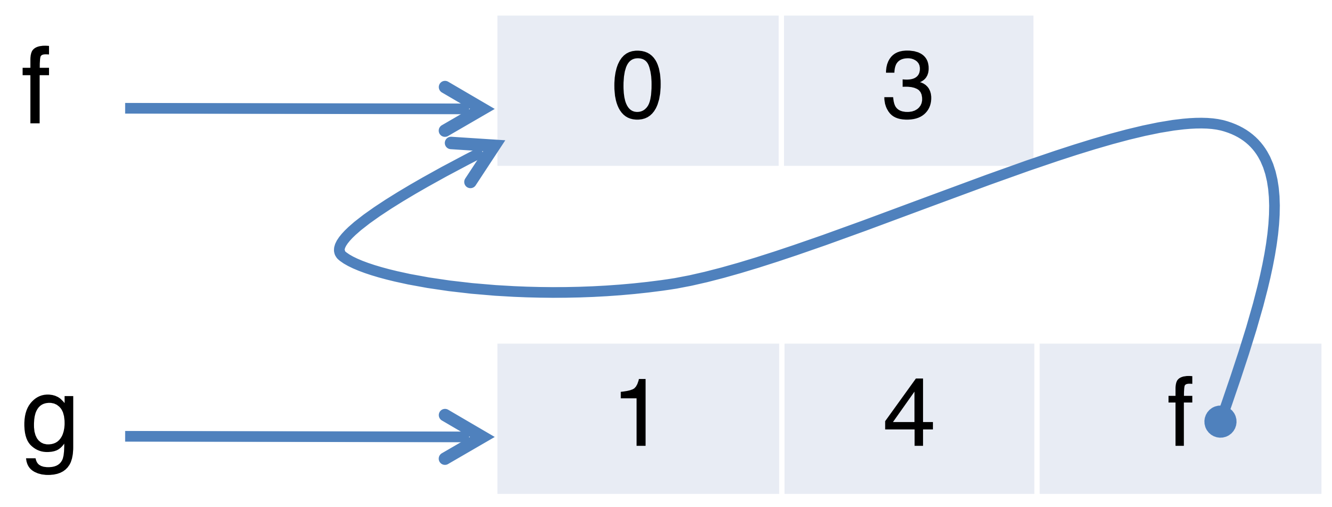
Tagged Datatypes

C-style Enumerations / ML-style datatypes

- In C: `enum Day {sun, mon, tue, wed, thu, fri, sat} today;`
- In OCaml: `type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat`
- Associate an integer *tag* with each case: sun = 0, mon = 1, …
  - C lets programmers choose the tags
- OCaml datatypes can also carry data: `type foo = Bar of int | Baz of int * foo`
- Representation: a foo value is a pointer to a pair: (tag, data)
- Example: tag(Bar) = 0, tag(Baz) = 1

⟦let f = Bar(3)⟧ =

⟦let g = Baz(4, f)⟧ =



Switch Compilation

- Consider the C statement:

```
switch (e) {
    case sun: s1; break;
    case mon: s2; break;
    …
    case sat: s3; break;
}
```

- How to compile this?
  - What happens if some of the break statements are omitted? (Control falls through to the next branch.)

Cascading ifs and Jumps

⟦switch(e) {case tag1: s1; case tag2 s2; …}⟧ =

- Each $tag1…$tagN is just a constant int tag value.
- Note: ⟦break;⟧ (within the switch branches) is:

  br %merge

```
        %tag = ⟦e⟧;
        br label %l1

l1: %cmp1 = icmp eq %tag, $tag1
        br %cmp1 label %b1, label %l2
b1: ⟦s1⟧
        br label %l2

l2: %cmp2 = icmp eq %tag, $tag2
        br %cmp2 label %b2, label %l3
b2: ⟦s2⟧
        br label %l3
…
lN: %cmpN = icmp eq %tag, $tagN
        br %cmpN label %bN, label %merge
bN: ⟦sN⟧
        br label %merge

merge:
```

Alternatives for Switch Compilation

- Nested if-then-else works OK in practice if # of branches is small
  - (e.g. < 16 or so).

- For more branches, use better data structures to organise the jumps:
  - Create a table of pairs (v1, branch_label) and loop through
  - Or, do binary search rather than linear search
  - Or, use a hash table rather than binary search

- One common case: the tags are dense in some range [min…max]
  - Let N = max – min
  - Create a branch table Branches[N] where Branches[i] = branch_label for tag i.
  - Compute tag = $[\![e]\!]$ and then do an *indirect jump*: J Branches[tag]
- Common to use heuristics to combine these techniques.

ML-style Pattern Matching

- ML-style match statements are like C's switch statements except:
  - Patterns can bind variables
  - Patterns can nest

```
match e with
| Bar(z) -> e1
| Baz(y, Bar(w)) -> e2
| _ -> e3
```

```
match e with
| Bar(z) -> e1
| Baz(y, tmp) ->
    (match tmp with
         | Bar(w) -> e2
         | Baz(_, _) -> e3)
```

- Compilation strategy:
  - "Flatten" nested patterns into matches against one constructor at a time.
  - Compile the match against the tags of the datatype as for C-style switches.
  - Code for each branch additionally must copy data from ⟦e⟧ to the variables bound in the patterns.

- There are many opportunities for optimisations, many papers about "pattern-match compilation"
  - Many of these transformations can be done at the AST level

Datatypes in LLVM IR

Structured Data in LLVM

- LLVM's IR is uses types to describe the structure of data.

```
t ::=
    void
    i1 | i8 | i64                N-bit integers
    [<#elts> x t]                arrays
    fty                          function types
    {t1, t2, … , tn}             structures
    t*                           pointers
    %Tident                      named (identified) type

fty ::=              Function Types
    t (t1, .., tn)               return, argument types
```

- <#elts> is an integer constant >= 0
- Structure types can be named at the top level:

```
%T1 = type {t1, t2, … , tn}
```

- Such structure types can be recursive

Example LL Types

- A static array of `4230` integers: `[ 4230 x i64 ]`

- A two-dimensional array of integers: `[ 3 x [ 4 x i64 ] ]`

- Structure for representing dynamically-allocated arrays with their length:

  ```
  { i64 , [0 x i64] }
  ```

  - There is no array-bounds check; the static type information is only used for calculating pointer offsets.

- C-style linked lists (declared at the top level):

  ```
  %Node = type { i64, %Node*}
  ```

- Structs from the C program shown earlier:

  ```
  %Rect = { %Point, %Point, %Point, %Point }
  %Point = { i64, i64 }
  ```

getelementptr

- LLVM provides the `getelementptr` instruction to compute pointer values
  - Given a pointer and a "path" through the structured data pointed to by that pointer, `getelementptr` computes an address
  - This is the abstract analog of the X86 LEA (load effective address). It **does not** access memory.
  - It is a "type indexed" operation, since the size computations depend on the type

```
insn ::= …
      |  getelementptr t* %val, t1 idx1, t2 idx2 ,…
```

- Example: access the x component of the first point of a rectangle:

```
%tmp1 = getelementptr %Rect* %square, i32 0, i32 0
%tmp2 = getelementptr %Point* %tmp1, i32 0, i32 0
```

- The first is **i32 0** a "step through" the pointer to, e.g., %square, with offset 0.

See "Why is the extra 0 index required?": https://llvm.org/docs/GetElementPtr.html#why-is-the-extra-0-index-required

GEP Example*

```
struct RT {
    int A;
    int B[10][20];
    int C;
}
struct ST {
    struct RT X;
    int Y;
    struct RT Z;
}
int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

1. `%s` is a pointer to an (array of) `%ST` structs, suppose the pointer value is ADDR
2. Compute the index of the 1st element by adding `size_ty(%ST)`.
3. Compute the index of the `Z` field by adding `size_ty(%RT) + size_ty(i32)` to skip past `X` and `Y`.
4. Compute the index of the `B` field by adding `size_ty(i32)` to skip past `A`.
5. Index into the 2d array.

```
%RT = type { i32, [10 x [20 x i32]], i32 }
%ST = type { %RT, i32, %RT }
define i32* @foo(%ST* %s) {
entry:
    %arrayidx = getelementptr %ST* %s, i32 1, i32 2, i32 1, i32 5, i32 13
    ret i32* %arrayidx
}
```

Final answer: ADDR + `size_ty(%ST) + size_ty(%RT) + size_ty(i32)`
`+ size_ty(i32) + 5*20*size_ty(i32) + 13*size_ty(i32)`

*adapted from the LLVM documentation: see http://llvm.org/docs/LangRef.html#getelementptr-instruction

getelementptr

- GEP *never* dereferences the address it's calculating:
  - GEP only produces pointers by doing arithmetic
  - It doesn't actually traverse the links of a data structure

- To index into a deeply nested structure, one has to "follow the pointer" by loading from the computed pointer

Compiling Data Structures via LLVM

1. Translate high level language types into an LLVM representation type.
   - For some languages (e.g. C) this process is straightforward
     - The translation simply uses platform-specific alignment and padding
   - For other languages, (e.g. OO languages) there might be a fairly complex elaboration.
     - e.g. for OCaml, arrays types might be translated to pointers to length-indexed structs.
       `⟦int array⟧ = { i32, [0 x i32]}*`

2. Translate accesses of the data into getelementptr operations:
   - e.g. for OCaml array size access:
     ⟦length a⟧ =
     `%1 = getelementptr {i32, [0 x i32]}* %a, i32 0, i32 0`

Type Casting

- What if the LLVM IR's type system isn't expressive enough?
  - e.g. if the source language has subtyping, perhaps due to inheritance
  - e.g. if the source language has polymorphic/generic types
- LLVM IR provides a `bitcast` instruction
  - This is a form of (potentially) unsafe cast. Misuse can cause serious bugs (segmentation faults, or silent memory corruption)

```
%rect2 = type { i64, i64 }              ; two-field record
%rect3 = type { i64, i64, i64 }         ; three-field record

define @foo() {
  %1 = alloca %rect3         ; allocate a three-field record
  %2 = bitcast %rect3* %1 to %rect2*      ; safe cast
  %3 = getelementptr %rect2* %2, i32 0, i32 1   ; allowed
  …
}
```

LLVMlite Specification

https://ilyasergey.net/YSC4230/hw03-llvmlite-spec.html

LLVMlite features

- A C-like "weak type system" to statically rule out some malformed programs.
- A variety of different kinds of integer values, pointers, function pointers, and structured data including strings, arrays, and structs.
- Top-level mutually-recursive function definitions and function calls as primitives.
- An infinite number of "locals" (also known as "pseudo-registers", "SSA variables", or "temporaries") to hold intermediate results of computations.
- An abstract memory model that doesn't constrain the layout of data in memory.
- Dynamically allocated memory associated with a function invocation (in C, the stack).
- Static and dynamically (heap) allocated structured data.
- A control-flow graph representation of function bodies.

Syntax

Example

```
define i64 @fac(i64 %n) {                 ; (1)
  %1 = icmp sle i64 %n, 0                 ; (2)
  br i1 %1, label %ret, label %rec        ; (3)
ret:                                      ; (4)
  ret i64 1
rec:                                      ; (5)
  %2 = sub i64 %n, 1                      ; (6)
  %3 = call i64 @fac(i64 %2)              ; (7)
  %4 = mul i64 %n, %3
  ret i64 %4                              ; (8)
}

define i64 @main() {                      ; (9)
  %1 = call i64 @fac(i64 6)
  ret i64 %1
}
```

function definition, argument prefixed with %
signed comparison, result assigned to %1
"terminator", marks the end of the block
label, indicates the beginning of the new block
return the result (1)
another block
subtract 1 from %n, name result %2
call function @fac, assign the result for %3

return result

call @fac with the argument 6

Good place for a break

LLVMlite types

Concrete Syntax	Kind	Description
`void`	void	Indicates the instruction does not return a usable value.
`i1, i64`	simple	1-bit (boolean) and 64-bit integer values.
`T*`	simple	Pointer that can be dereferenced if its target is compatible with T
`i8*`	simple	Pointer to the first character in a null-terminated array of bytes. Note: `i8*` is a valid type, but just `i8` is not. LLVMlite programs do not operate over byte-sized integer values.
`F*`	simple	Function pointer
`S(S1, ..., SN)`	function	A function from S1, ..., SN to S
`void(S1, ..., SN)`	function	A function from S1, ..., SN to void
`{ T1, ..., TN }`	aggregate	Tuple of values of types T1, ..., TN
`[ N x T ]`	aggregate	Exactly N values of type T
`%NAME`	*	Abbreviation defined by a top-level named type definition

- Simple types appear on stack and as arguments to functions
- Aggregate types that may only appear in global and heap-allocated data
- One can define abbreviations for types:
  %IDENT = type T

Global Definitions

```
@IDENT = global T G
```

```
@foo = global i64 42
@bar = global i64* @foo
@baz = global i64** @bar
```

Concrete Syntax	Type	Description
`null`	`T*`	The null pointer constant.
`[0-9]+`	`i64`	64-bit integer literal.
`@IDENT`	`T*`	Global identifier. The type is always a pointer of the type associated with the global definition.
`c"[A-z]*\00"`	`[ N x i8 ]`	String literal. The size of the array N should be the length of the string in bytes, including the null terminator `\00`.
`[ T G1, ..., T GN ]`	`[ N x T ]`	Array literal.
`{ T1 G1, ..., TN GN }`	`{T1,...,TN}`	Struct literal.
`bitcast (T1* G1 to T2*)`	`T2*`	Bitcast.

Operands of functions

Concrete Syntax	Type	Description
`null`	`T*`	The null pointer constant
`[0-9]+`	`i64`	64-bit integer literal
`@IDENT`	`T*`	Global identifier. The type can always be determined from the global definitions and is always a pointer
`%IDENT`	`S`	Local identifier: can only name values of simple type. The type determined by an local definition of `%IDENT` in scope

Types of instructions

Concrete Syntax	Operand → Result Types
`%L = BOP i64 OP1, OP2`	`i64 x i64 → i64`
`%L = alloca S`	`– → S*`
`%L = load S* OP`	`S* → S`
`store S OP1, S* OP2`	`S x S* → void`
`%L = icmp CND S OP1, OP2`	`S x S → i1`
`%L = call S1 OP1(S2 OP2, ..., SN OPN)`	`S1(S2, ..., SN)* x S2 x ... x SN → S1`
`call void OP1(S2 OP2, ... ,SN OPN)`	`void(S2, ..., SN)* x S2 x ... x SN → void`
`%L = getelementptr T1* OP1, i32 OP2, ..., i32 OPN`	`T1* x i64 x ... x i64 -> ` **`GEPTY`**`(T1, OP1, ..., OPN)*`
`%L = bitcast T1* OP to T2*`	`T1* → T2*`

- Let's discuss the meaning of these types
- The **getelementptr** instruction has some additional well-formedness requirements (see the specification)

GEP Type

```
GEPTY : T -> operand list -> T
GEPTY T operand::path' = GEPTY' T path'

GEPTY' : T -> operand list -> T
GEPTY' T                             [] = T
GEPTY' { T1, ..., TN } (Const m)::path' = GEPTY' Tm path' when m <= N
GEPTY' [ _ x T ]         operand::path' = GEPTY'  T path'
```

- GEPTY is a partial function.
- When GEPTY is not defined, the corresponding instruction is malformed.
- This happens when, for example:
  - The list of index operands provided is empty
  - An operand used to index a struct is not a constant
  - The type is not an aggregate and the list of indices is not empty

Notes on GEP

- Real LLVM requires that constants appearing in getelementptr be declared with type i32:

```
%struct = type { i64, [5 x i64], i64}

@gbl = global %struct {i64 1,
   [5 x i64] [i64 2, i64 3, i64 4, i64 5, i64 6], i64 7}

define void @foo() {
  %1 = getelementptr %struct* @gbl, i32 0, i32 0
  …
}
```

- LLVMlite ignores the i32 annotation and treats these as i64 values
  - we keep the i32 annotation in the syntax to retain compatibility with the clang compiler
  - we assume the arguments of getelementptr always fall in the range [0, Int32.max_int].

Blocks, CFGs, and Function Definitions

- A block is just a sequence of instructions followed by a terminator

Concrete Syntax	Operand → Result Types
`ret void`	`– → –`
`ret S OP`	`S → –`
`br label %LAB`	`– → –`
`br i1 OP, label %LAB1, label %LAB2`	`i1 → –`

- The body of a function is represented by a control flow graph (CFG).
- A CFG consists of a distinguished entry block and a sequence blocks of prefixed with a label.
- The full syntax of a function definition:

```
define [S|void] @IDENT(S1 OP, ... , SN OP) { BLOCK (LAB: BLOCK)...}
```

Semantics

LLVMlite Semantics

- Like for X86lite, we define the semantics of LLVMlite by describing the execution of an *abstract machine*.
- LLVMlite machine explicitly differentiates between stack, heap, code, and global memory (X86 was treating all of those uniformly).
- A definitional (reference) interpreter for LLVMlite is provided in HW3: check **llinterp.ml**
- If you have a question about a detail of the semantics, you can simply run a program through the interpreter!

Memory Model

- The memory state of the LLVMlite machine is represented by a mapping between **block identifiers** and memory values.
  We will refer to a top-level memory value that is not a subtree of another as a **memory block**.

```
{ bid0 -> node     bid1 -> node     bid2 -> node }
         /  |  \               |                |
        L   L   L             node              L
                             /    \
                            L      node
                                  / | | \
                                  L L L* L
```

- Even simple values, such as a single global i64 will be represented using a node with one leaf.
- To identify the leaf marked * we provide the indices 0, 1, 2 along with the identifier **bid1**.
- This means that we're selecting the 2nd child of the 1st child of the 0th child of the root node.

Memory Model

```
{ bid0 -> node     bid1 -> node     bid2 -> node }
         /  |  \            |                |
        L   L   L          node              L
                          /    \
                         L      node
                               / | | \
                               L L L* L
```

- The simple values include:
  - 1-bit (boolean) and 64-bit 2's complement signed integers
  - Pointers to a subtree of a particular memory block containing a block identifier and path
  - A special **undef** value that represents an unusable value

```
{ bid0 -> node        bid1 ->  node       bid2 -> node }
           |                    \                  |
          node                   node             undef
        /  |  \                /  |  \
   "foo" "bar" "baz"   (Ptr Null)  4  (Ptr bid0, 0, 1)
```

Interpreter

- **interp_call** takes
  - the global identifier of a function in an LLVMlite program,
  - a list of (simple) values to serve as arguments, and
  - an initial memory state; and
  - returns the memory state after the function call has completed and the return value.

- **interp_cfg** does most of the work. It takes
  - a control-flow graph,
  - an initial locals map, and
  - a memory state; and
  - evaluates the cfg, returning the new memory state and the return value of the function body.

Some Instructions

(see implementation)

`%L = alloca S`	Allocate a slot in the current stack frame and return a pointer to it. This involves adding a subtree of undef to the root node of the memory block representing the frame at the next available index.
`%L = load S* OP`	OP must be a pointer or **undef**. Find the value referenced by the pointer in the current memory state. Update locals(`%L`) with the result. If OP is not a valid pointer, either because it evaluates to **undef**, no memory value is associated with its block identifier or its path does not identify a valid subtree, then the operation raises an error and the machine crashes. If the pointer is valid, but the value in memory is not a simple value of type S, the operation raises an error and the machine crashes.
`store S OP1, S* OP2`	Update the memory state by setting the target of OP2 to the value of OP1. If OP2 is not a valid pointer, or if the target of OP2 is not a simple value in memory of type S, the operation raises an error and the machine crashes.

Some Instructions (c'd)

(see implementation)

`%L = call S1 OP1(S2 OP2, ... ,SN OPN)`	Evaluate all of the operands and use them to recursively invoke the interpreter through **interp_call** with the current memory state. If OP1 does not evaluate to a function pointer that identifies a function with return type `S1` and argument types `S2, ... , SN,` then the operation raises an error and the machine crashes. Update the local (`%L`) to the result of **interp_call** and continue with the return memory state.
`call void OP1(S2 OP2, ... ,SN OPN)`	The same as a non-void call, but no locals are updated with the returned value.

Some Instructions (c'd)

(see implementation)

`%L = getelementptr T1* OP1, i64 OP2, ... , i64 OPN`	Create a new pointer by *adding* the first index operand OP2 to the last index of the pointer value of OP1 and then *concatenating* the remaining indices onto the path. If the target of the resulting pointer is not a valid memory value *compatible* with the type %L, then update locals(`%L`) with the **undef** value. Otherwise, update locals(`%L`) with the new pointer. See the following section for a more detailed explanation.

GEP Indexing

```
%t1 = type { A, B, C }
%t2 = type [ 2 x %t1 ]

@pn1 = global %t2 [ {a0, b0, c0}, {a1, b1, c1} ]

; Memory:
; { ... bid0 -> root ... }
;                 |
;                 n1
;               /    \
;             n2      n3
;           / | \    / | \
;          a0 b0 c0 a1 b1 c1

...
%pn2 = getelementptr %t2* pn1, i32 0, i32 0    ; %t1* -> n2
%pb1 = getelementptr %t1* pn2, i32 1, i32 1    ; B* -> b1
```

- Start with the pointer **pn1 = (bid0, 0)** pointing to **n1**.
- The first GEP instruction above will compute the pointer **(bid0, 0, 0)**, by first adding 0 to the last index of **pn1** and then concatenating the rest of the indices to the end of the path.
- The next GEP instruction will compute the pointer **(bid0, 0, 1, 1)**, which points to **b1**.
- Why?
- Indexing into a sibling (rather than a child) of a node using GEP with a non-zero first index is only legal if sibling nodes are allocated as *part of an array*.
- In our example, n1 was allocated as [2 x %t1], so this is the case.

- Check out **effective_tag** in the interpreter code.
- Some examples in llprograms: gep3.ll, gep5.ll, gep6.ll

Compiling LLVMlite to X86

Compiling LLVMlite Types to X86

- ⟦`i1`⟧, ⟦`i64`⟧, ⟦`t*`⟧ = quad word (8 bytes, 8-byte aligned)
- raw `i8` values are not allowed (they must be manipulated via `i8*`)
- array and struct types are laid out *sequentially* in memory
- getelementptr computations must be relative to the LLVMlite size definitions
  - i.e. ⟦i1⟧ = quad (quite wasteful!)

Compiling LLVM locals

- How do we manage storage for each %uid defined by an LLVM instruction?

- Option 1:
  - Map each %uid to a x86 register
  - Efficient!
  - Difficult to do effectively: many %uid values, only 16 registers
  - We will see how to do this later in the semester

- Option 2:
  - Map each %uid to a stack-allocated space
  - Less efficient!
  - Simple to implement

- For HW3 we will follow Option 2

Other LLVMlite Features

- Globals
  - must use %rip relative addressing
- Calls
  - Follow x64 AMD ABI calling conventions
  - Should interoperate with C programs
- getelementptr
  - trickiest part

Tour of HW3

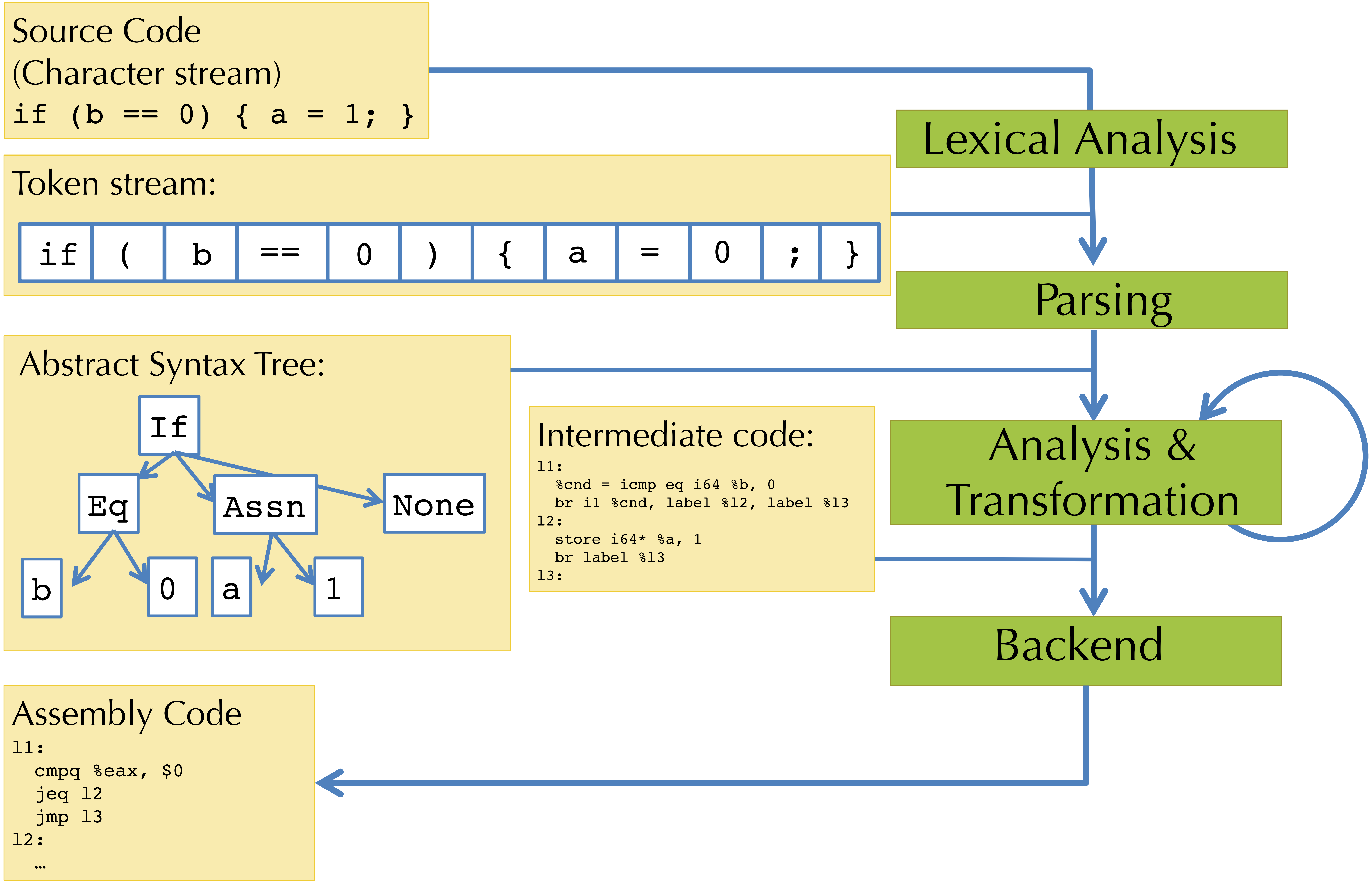
- See HW3 description and `README.md`
- Main definitions: `ll.ml`
- Compiler in the pipeline: driver.ml and process_ll_file.
- Using `main.native`
- Compiling with `clang`

A break?

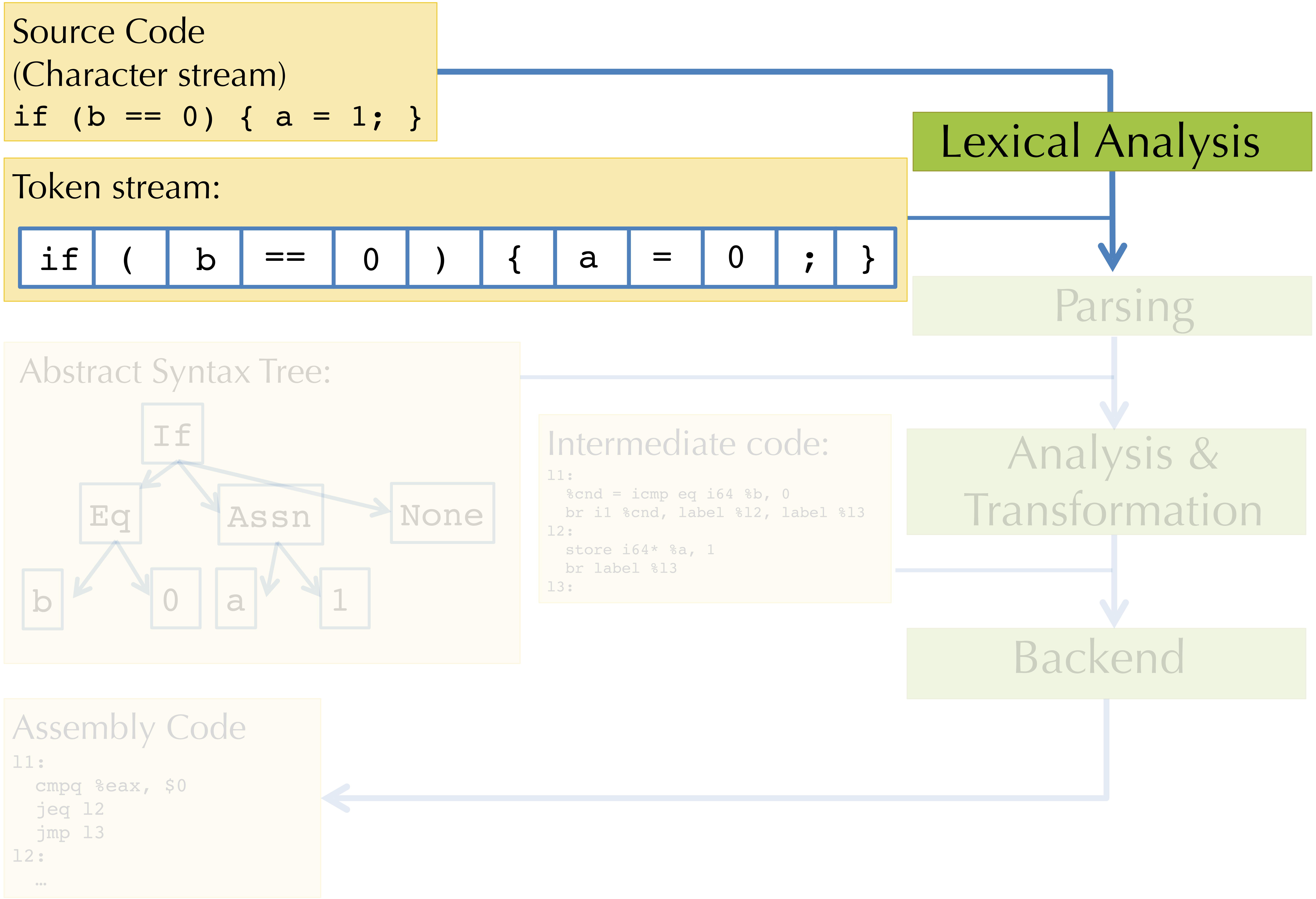
Lexing

Lexical analysis, tokens, regular expressions, automata

Compilation in a Nutshell



Today: Lexing



First Step: Lexical Analysis

- Change the *character stream* “`if (b == 0) a = 0;`” into *tokens*:

if	(	b	==	0	)	{	a	=	0	;	}

```
IF; LPAREN; Ident(“b”); EQEQ; Int(0); RPAREN; LBRACE; Ident(“a”);
EQ; Int(0); SEMI; RBRACE
```

- Token: data type that represents indivisible “chunks” of text:
  - Identifiers: `a   y11   elsex   _100`
  - Keywords: `if   else   while`
  - Integers: `2   200   -500   5L`
  - Floating point: `2.0   .02   1e5`
  - Symbols: `+   *   `` ` ``   {   }   (   )   ++   <<   >>   >>>`
  - Strings: `“x”   “He said, \”Are you?\””`
  - Comments: `(* YSC4230: Project 1 … *)   /* foo */`
- Often delimited by *whitespace* (‘ ‘, \t, etc.)
  - In some languages (e.g. Python or Haskell) whitespace is significant

Demo: Handlex

How hard can it be?

See handlex.ml

https://github.com/ysc4230/week-05-lexing

Lexing By Hand

- How hard can it be?
  - Tedious and painful!
- Problems:
  - Precisely define tokens
  - Matching tokens simultaneously
  - Reading too much input (need look ahead)
  - Error handling
  - Hard to compose/interleave tokeniser code
  - Hard to maintain

A Principled Solution to Lexing

Regular Expressions

- Regular expressions precisely describe sets of strings.

- A regular expression R has one of the following forms:
  - ε — Epsilon stands for the empty string
  - `‘a’` — An ordinary character stands for itself
  - $R_1 \mid R_2$ — Alternatives, stands for choice of R_1 or R_2
  - R_1R_2 — Concatenation, stands for R_1 followed by R_2
  - `R*` — Kleene star, stands for *zero or more* repetitions of `R`

- *Useful extensions:*
  - `“foo”` — Strings, equivalent to `'f''o''o'`
  - `R+` — One or more repetitions of `R`, equivalent to `RR*`
  - `R?` — Zero or one occurrences of `R`, equivalent to $(\varepsilon|R)$
  - `['a'-'z']` — One of `a` or `b` or `c` or … `z`, equivalent to `(a|b|…|z)`
  - `[^'0'-'9']` — Any character except `0` through `9`
  - `R as x` — Name the string matched by `R` as `x`

Example Regular Expressions

- Recognise the keyword "if": `"if"`
- Recognise a digit: `['0'-'9']`
- Recognise an integer literal: `'-'?['0'-'9']+`
- Recognise an identifier:
  `(['a'-'z']|['A'-'Z'])(['0'-'9']|'_'|['a'-'z']|['A'-'Z'])*`

- In practice, it's useful to be able to *name* regular expressions:

```
let lowercase = ['a'-'z']
let uppercase = ['A'-'Z']
let character = uppercase | lowercase
```

How to Match?

- Consider the input string: `ifx = 0`
  - Could lex as:

`if`	`x`	`=`	`0`

or as:

`ifx`	`=`	`0`

- Regular expressions alone are *ambiguous*, need a rule to choose between the options above
- Most languages choose "longest match"
  - So the 2nd option above will be picked
  - Note that only the first option is "correct" for parsing purposes

- Conflicts: arise due to two tokens whose regular expressions have a shared prefix
  - Ties broken by giving some matches **higher priority**
  - Example: keywords have priority over identifiers
  - Usually specified by order the rules appear in the lex input file

Lexer Generators

- Reads a list of regular expressions: `R1,…,Rn` , one per token.
- Each token has an attached “action” `Ai`
  (just a piece of code to run when the regular expression is matched)

```
rule token = parse
| '-'?digit+                        { Int (Int32.of_string (lexeme lexbuf)) }
| '+'                               { PLUS }
| 'if'                              { IF }
| character (digit|character|'_')*  { Ident (lexeme lexbuf) }
| whitespace+                       { token lexbuf }
```

token regular expressions

actions

- Generates scanning code that:

1. Decides whether the input is of the form `(R1|…|Rn)*`
2. Whenever the scanner matches a (longest) token, it runs the associated action

Demo: Ocamllex

lexlex.mll

Next week

- Basic automata theory for lexing
- Syntactic analysis (parsing)
- Building program ASTs from text