Lecture 9

CS131: COMPILERS

Announcements

- HW3: LLVM lite
  - Available on Blackboard soon after the class.
  - Due: November 6th at 11:59:59pm

START EARLY!!

- Midterm: November 19th (tentative)
  - In class
  - One-page, letter-sized, double-sided "cheat sheet" of notes permitted
  - See Piazza / Blackboard (soon) for previous exams

Plan for Today

1. Continue tour of datatypes
   – tagged types / matching
2. LLVM IR's types
3. Quick Overview/Recap of HW3
4. (Back to the Compiler Pipeline: Lexing)

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]`	…

- M[i][j] compiles to?

- 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.

`arr`

`Size=7`	`A[0]`	`A[1]`	`A[2]`	`A[3]`	`A[4]`	`A[5]`	`A[6]`

- Other possibilities:
  - Store size and a pointer to array data
  - 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 `%rcx` 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
  - These overheads are particularly bad in an inner loop
- Compiler optimizations can help remove the overhead
  - e.g. In a for loop, if bound on index is known, only do the test once
- Hardware support can improve performance: executing instructions in parallel, branch prediction
  - but speculative execution is behind the Spectre/Meltdown vulnerabilities

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';
```

- 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 ML:

```
type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

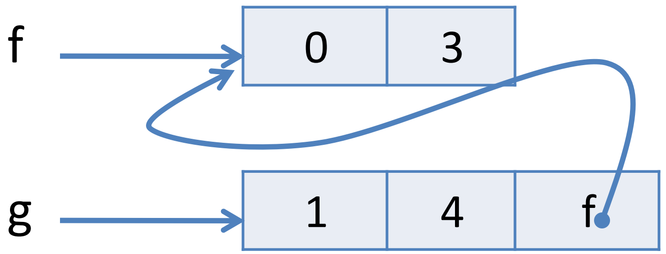
- Associate an integer *tag* with each case: sun = 0, mon = 1, …
  - C lets programmers choose the tags

- ML 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
  rather than
  br %b_(i+1)

```
        %tag = ⟦e⟧;
        br label %l1
l1: %cmp1 = icmp eq %tag, $tag1        ;; compare tags
        br %cmp1 label %b1, label %l2  ;; case 1 or case 2?
b1: ⟦s1⟧
        br label %b2                   ;; fallthru to case 2

l2: %cmp2 = icmp eq %tag, $tag2        ;; compare tags
        br %cmp2 label %b2, label l3   ;; case 2 or case 3?
b2: ⟦s2⟧
        br label %b4 ;;                ;; use %merge if break
…
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 datastructures to organize 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 optimization, many papers about “pattern-match compilation”
  - Many of these transformations can be done at the AST level

DATATYPES IN THE LLVM IR

Structured Data in LLVM

- LLVM's IR 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

- An array of 3410 integers: [3410 x i64]

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

- Structure for representing 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 }

Compiling Datastructures via LLVM

1. Translate high level language types into an LLVM representation type.
   - For some languages (e.g. C) this process is straight forward
     - 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, [0xi32]}* %a, i32 0, i32 0

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 leaq (load effective address). It *does not* access memory.
  - It is a "type directed" operation, since the size computations depend on the type of the pointer

```
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
```

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 documentaion: 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 datastructure
- To index into a deeply nested structure, need to "follow the pointer" by loading from the computed pointer
  - See list.ll from HW3

Bitcast

- 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 notes

- 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

see HW3 and README

ll.ml.

TOUR OF HW 3

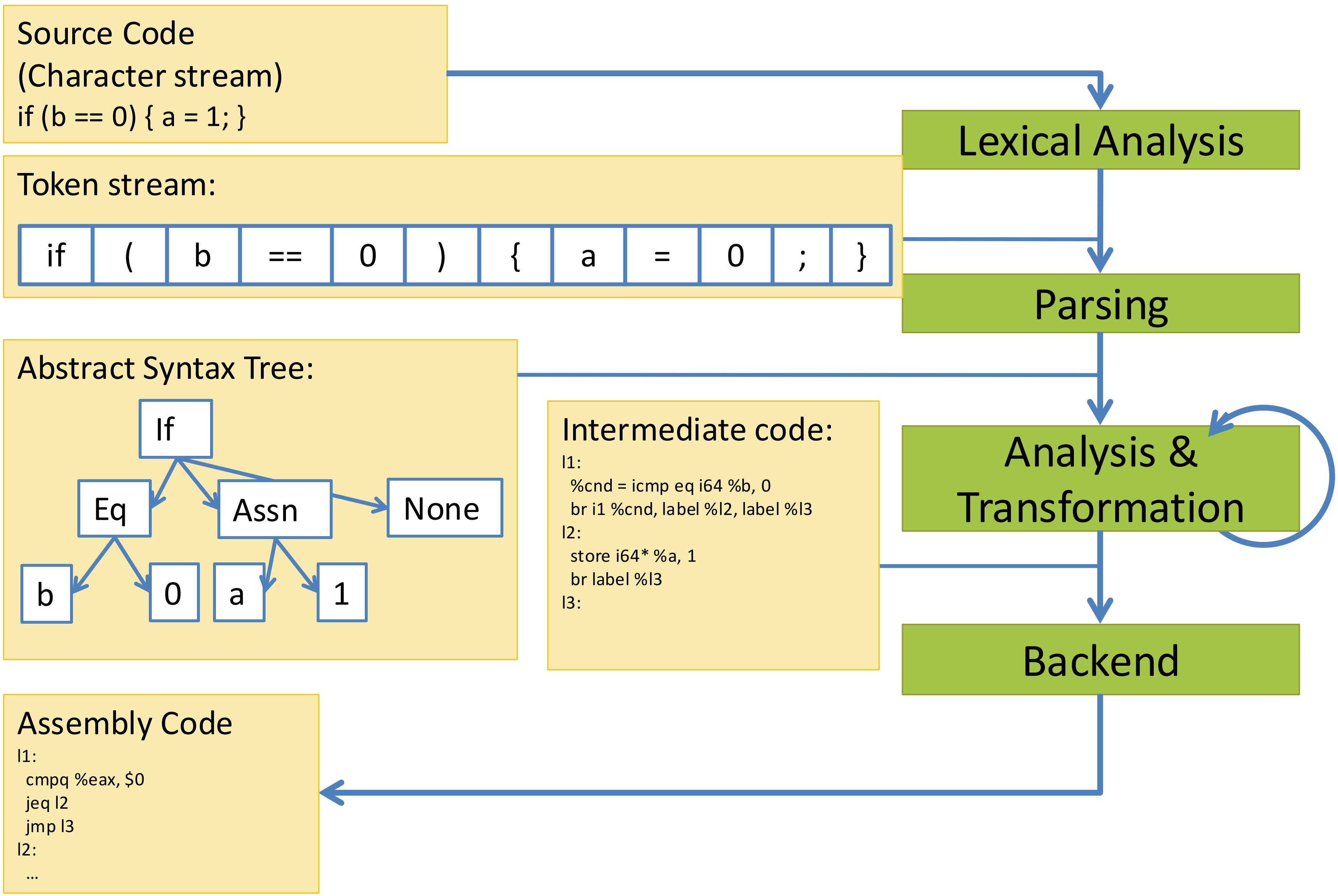
LLVMlite vs "real" LLVM IR

- Full LLVM IR supports a few more types:
  arbitrary bitwidth integers: i3, i17, i128, i12, iX
- Full LLVM IR has has more support for aggregate datatypes
  - alloca can allocate arbitrary types in the stack
  - there are operations for creating/manipulating such values (e.g., extractelement)
- There are a few other instructions:
  - select – choose between values
  - a few other kinds of control flow (for "exceptions" and "switch" statements)
- So-called *intrinsics*
  - special-purpose instructions named llvm.* that are treated by the compiler
    e.g.: llvm.memcpy or llvm.log2

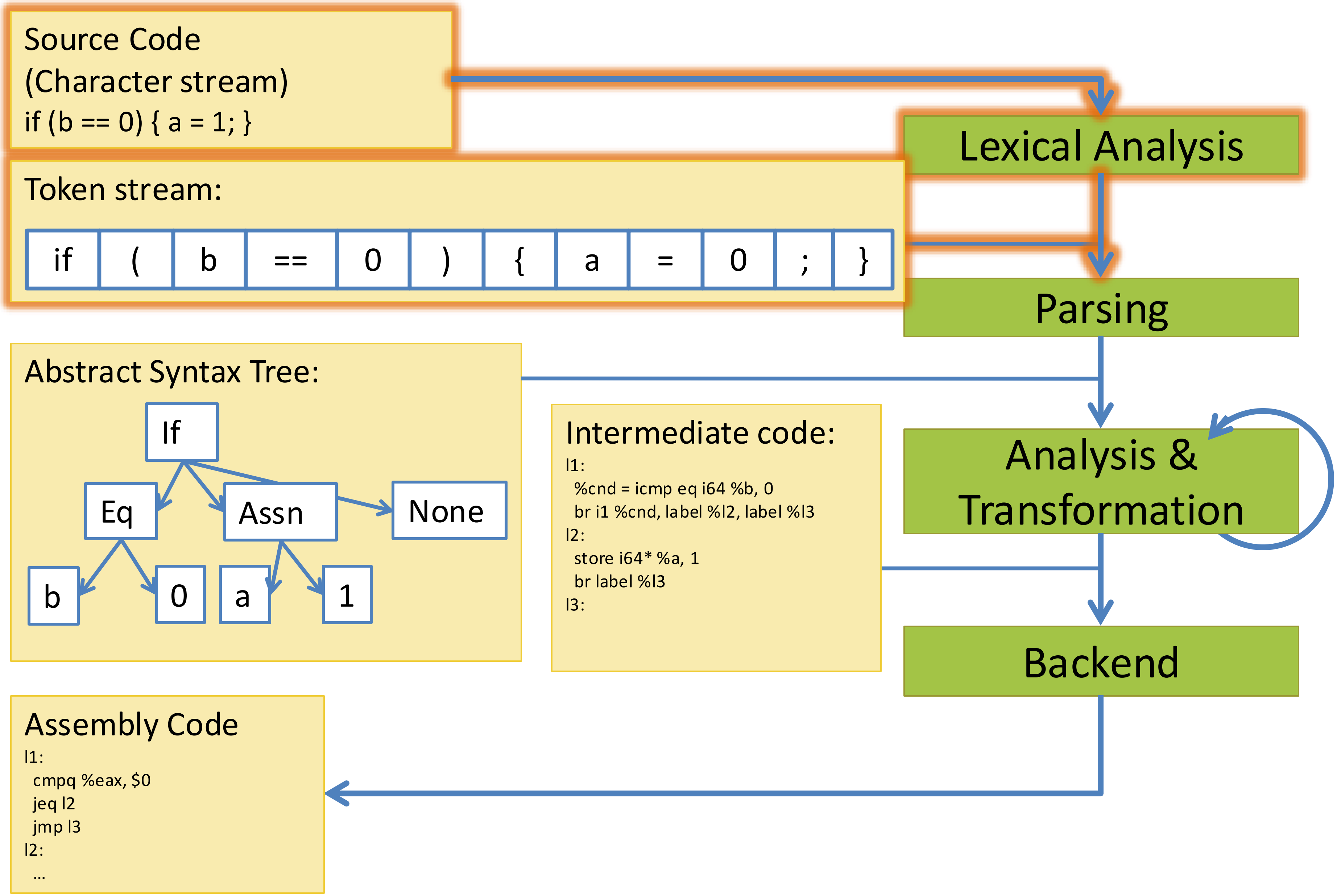
Lexical analysis, tokens, regular expressions, automata

LEXING

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: (* CIS3410: Project 1 … *) /* foo */

- Often delimited by *whitespace* (' ', \t, etc.)
  - In some languages (e.g., Python or Haskell) whitespace is significant

How hard can it be?
handlex0.ml and handlex.ml

DEMO: HANDLEX

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 tokenizer code
  - Hard to maintain

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 | 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 (ε|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

- Recognize the keyword “if”: ”if”
- Recognize a digit: ['0'-'9']
- Recognize an integer literal: '-'?['0'-'9']+
- Recognize 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: they need a rule for choosing 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: $R_1,\ldots,R_n$, one per token.
- Each token has an attached “action” A_i (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 $(R_1|\ldots|R_n)^*$
    2. Whenever the scanner matches a (longest) token, it runs the associated action

lexlex.mll

DEMO: OCAMLLEX

Implementation Strategies

- Most Tools: lex, ocamllex, flex, etc.:
  - Table-based
  - Deterministic Finite Automata (DFA)
  - Goal: Efficient, compact representation, high performance

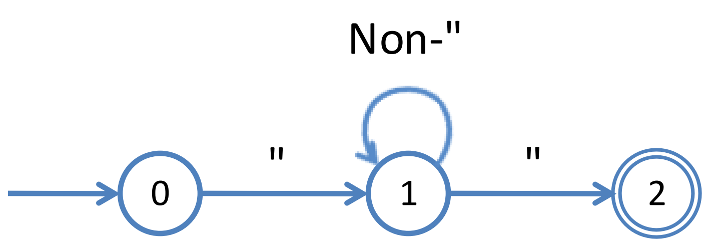
- Other approaches:
  - Brzozowski derivatives
  - Idea: directly manipulate the (abstract syntax of) the regular expression
  - Compute partial "derivatives"
    - Regular expression that is "left-over" after seeing the next character
  - Elegant, purely functional, implementation
  - (very cool!)

Finite Automata

- Consider the regular expression: '"'[^'"']*'"'
- An automaton (DFA) can be represented as:
  - A transition table:

	"	Non-"
0	1	ERROR
1	2	1
2	ERROR	ERROR

  - A graph:

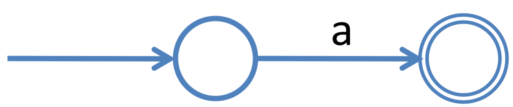


RE to Finite Automaton?

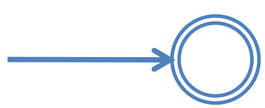
- Can we build a finite automaton for every regular expression?
  - Yes! Recall CIS 262 for the complete theory…

- Strategy: consider every possible regular expression (by induction on the structure of the regular expressions):

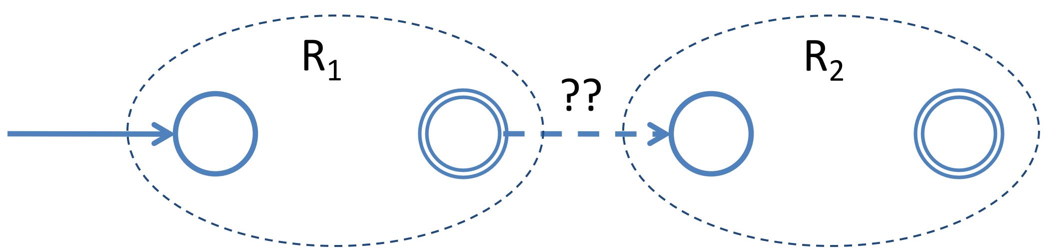
'a'



ε



R_1R_2

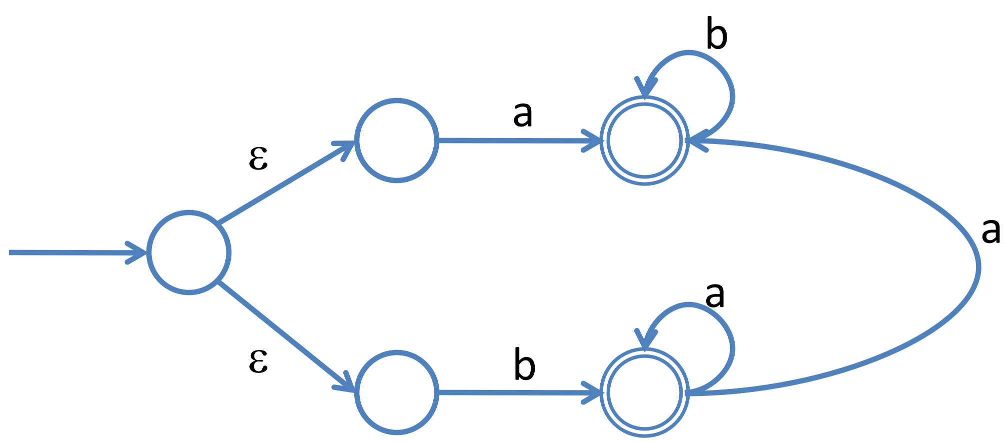


What about?

$R_1 | R_2$

Nondeterministic Finite Automata

- A finite set of states, a start state, and accepting state(s)
- Transition arrows connecting states
  - Labeled by input symbols
  - Or ε (which does not consume input)
- *Nondeterministic*: two arrows leaving the same state may have the same label



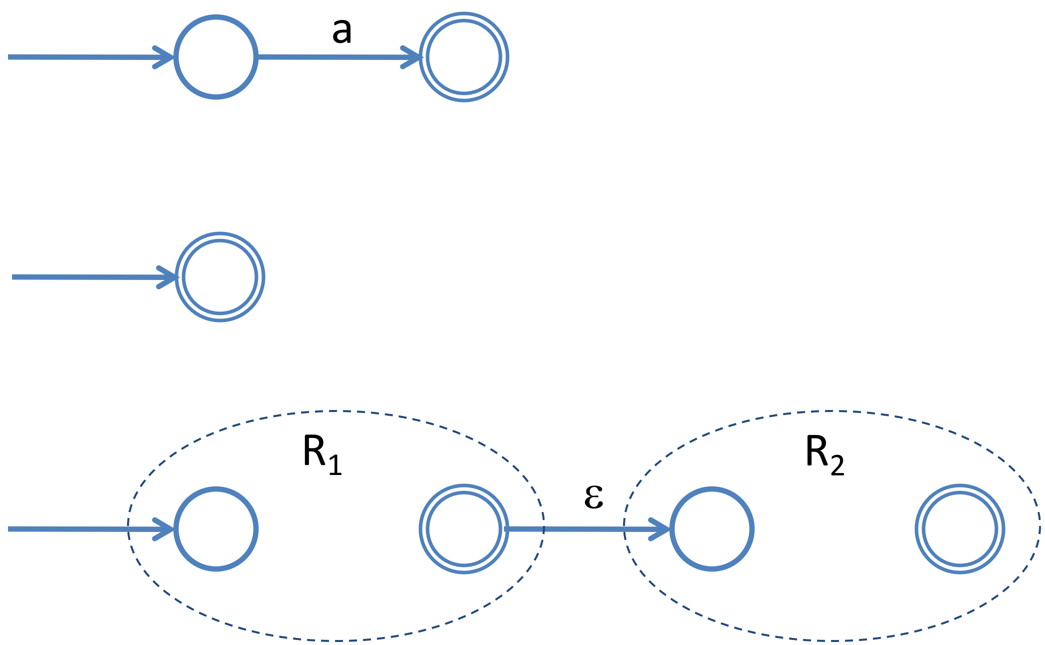
RE to NFA?

- Converting regular expressions to NFAs is easy.
- Assume each NFA has one start state, unique accept state

‘a’

ε

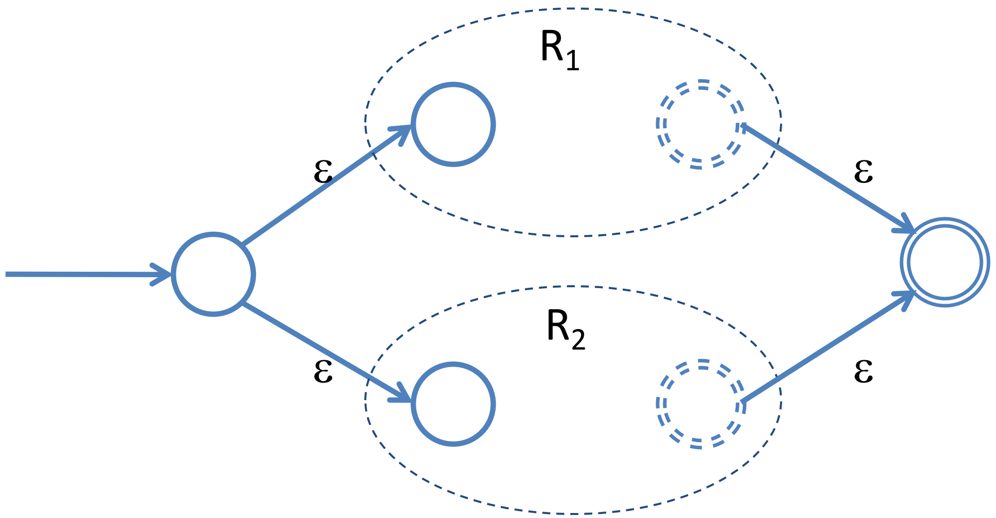
R_1R_2



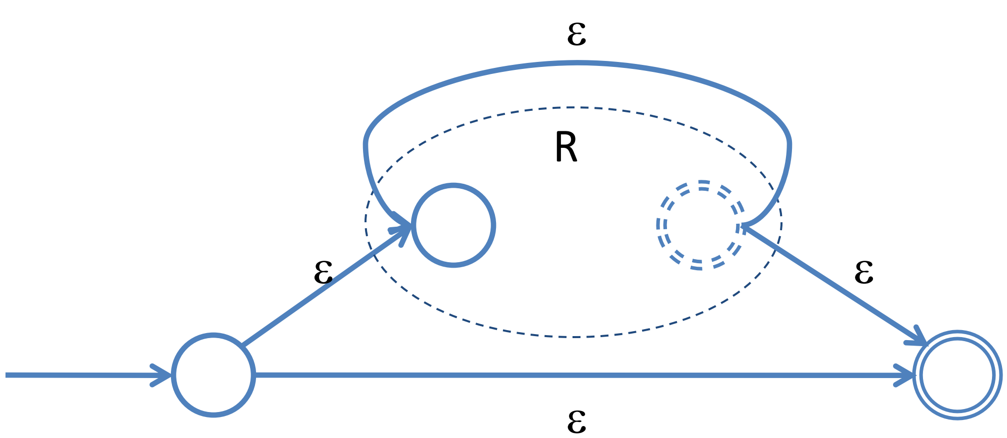
RE to NFA (cont'd)

- Sums and Kleene star are easy with NFAs

$R_1 | R_2$



R^*



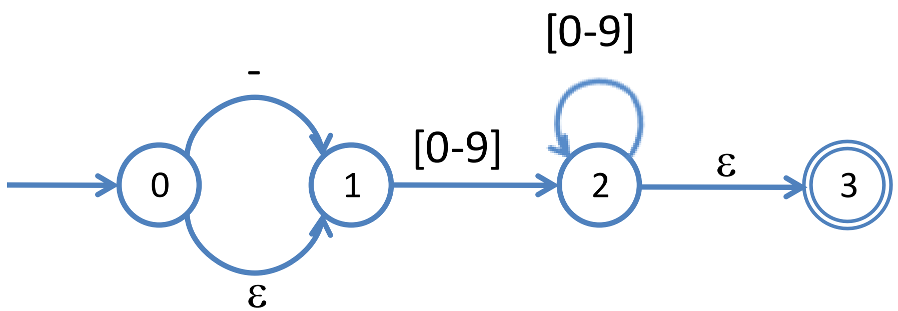
DFA versus NFA

- DFA:
  - Action of the automaton for each input is fully determined
  - Automaton accepts if the input is consumed upon reaching an accepting state
  - Obvious table-based implementation

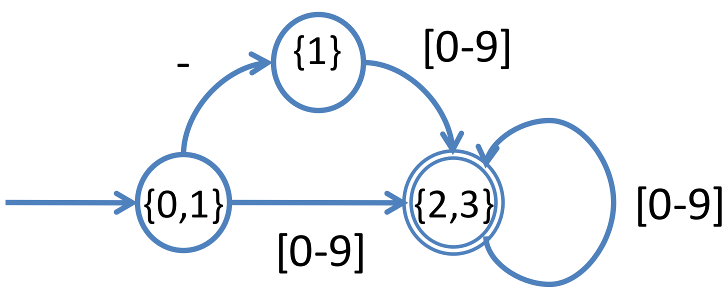
- NFA:
  - Automaton potentially has a choice at every step
  - Automaton accepts an input string if there *exists* a way to reach an accepting state
  - Less obvious how to implement efficiently

NFA to DFA conversion (Intuition)

- Idea: Run all possible executions of the NFA “in parallel”
- Keep track of a set of possible states: “finite fingers”
- Consider: -?[0-9]+
- NFA representation:



- DFA representation:



Summary of Lexer Generator Behavior

- Take each regular expression R_i and it's action A_i
- Compute the NFA formed by $(R_1 \mid R_2 \mid \ldots \mid R_n)$
  - Remember the actions associated with the accepting states of the R_i
- Compute the DFA for this big NFA
  - There may be multiple accept states (why?)
  - A single accept state may correspond to one or more actions (why?)
- Compute the minimal equivalent DFA
  - There is a standard algorithm due to Myhill & Nerode
- Produce the transition table
- Implement longest match:
  - Start from initial state
  - Follow transitions, remember last accept state entered (if any)
  - Accept input until no transition is possible (i.e. next state is "ERROR")
  - Perform the highest-priority action associated with the last accept state; if no accept state there is a lexing error

Lexer Generators in Practice

- Many existing implementations: lex, Flex, Jlex, ocamllex, …
  - For example ocamllex program
    - see lexlex.mll, olex.mll, piglatin.mll on course website
- Error reporting:
  - Associate line number/character position with tokens
  - Use a rule to recognize '\n' and increment the line number
  - The lexer generator itself usually provides character position info.
- Sometimes useful to treat comments specially
  - Nested comments: keep track of nesting depth

- Lexer generators are usually designed to work closely with parser generators…