Lecture 9

CIS 341: COMPILERS
Announcements

• HW3: LLVM lite
  – Available on the course web pages.
  – Due: Monday, Feb. 26th at 11:59:59pm
  – Only one group member needs to submit
  – Three submissions per group

• TODAY @ 4:30 – The Programmer
  – 20 minute documentary about the women behind ENIAC
  – Wu & Chen Auditorium
  – Free Food!

START EARLY!!
DATATYPES IN THE LLVM IR
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];
}

%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 datastructure

- To index into a deeply nested structure, need to “follow the pointer” by loading from the computed pointer
  - See list.ll from HW3
Compiling Datastructures 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.

\[
\text{[int array]} = \{ \text{i32, [0 x i32]}\}*
\]

2. Translate accesses of the data into getelementptr operations:
   - e.g. for Ocaml array size access:
     \[
     \text{[length a]} = \%
     \text{1} = \text{getelementptr} \{\text{i32, [0xi32]}\}* \%a, \text{i32 0, i32 0}
     \]
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)

```assembly
%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
    ...
}
```
Lexical analysis, tokens, regular expressions, automata
Compilation in a Nutshell

Source Code
(Character stream)
if (b == 0) { a = 1; }

Token stream:
if ( b == 0 ) { a = 0 ; }

Abstract Syntax Tree:
If
  Eq
    b
  Assn
    0
  Assn
    a
  None
  1

Intermediate code:
11:  %cnd = icmp eq i64 %b, 0
    br i1 %cnd, label %l2, label %l3
12:  store i64* %a, 1
    br label %l3
13:

Assembly Code
11:
  cmpq %eax, $0
  jeq 12
  jmp 13
12:
  ...
Today: Lexing

Source Code (Character stream)
```c
if (b == 0) { a = 1; }
```

Token stream:
```c
if ( b == 0 ) { a = 0 ; }
```

Abstract Syntax Tree:
```
If
  Eq
    b
  Assn
    0
    a
  None
    1
```

Intermediate code:
```assembly
11:
    %cnd = icmp eq i64 %b, 0
    br i1 %cnd, label %l2, label %l3
12:
    store i64* %a, 1
    br label %l3
13:
```

Assembly Code
```assembly
11:
    cmpq %eax, $0
    jeq 12
    jmp 13
12:
    ...
```
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: (* CIS341: 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?
  – Tedium 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
  – R₁ | R₂ Alternatives, stands for choice of R₁ or R₂
  – R₁R₂ Concatenation, stands for R₁ followed by R₂
  – 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:

```javascript
let lowercase = ['a'-'z']
let uppercase = ['A'-'Z']
let character = uppercase | lowercase
```
How to Match?

• Consider the input string: \texttt{ifx = 0}
  – Could lex as: \texttt{if x = 0} or as: \texttt{ifx = 0}

• Regular expressions alone are ambiguous, need a rule for choosing between the options above

• Most languages choose “longest match”
  – So the 2\textsuperscript{nd} 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):

\[
\text{rule token = parse}
\begin{align*}
&| '-'? \text{digit+} \quad \{ \text{Int (Int32.of_string (lexeme lexbuf))} \} \\
&| '+' \quad \{ \text{PLUS} \} \\
&| 'if' \quad \{ \text{IF} \} \\
&| \text{character (digit|character|'}_\text{'}\text{')}* \quad \{ \text{Ident (lexeme lexbuf)} \} \\
&| \text{whitespace+} \quad \{ \text{token lexbuf} \}
\end{align*}
\]

- 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
DEMO: OCAMLLLEX
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:

<table>
<thead>
<tr>
<th></th>
<th>&quot;&quot;</th>
<th>Non-&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>ERROR</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>ERROR</td>
<td>ERROR</td>
</tr>
</tbody>
</table>

- 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'

  [Diagram: a transition labeled 'a']

  ε

  [Diagram: an empty transition]

  $R_1R_2$

  [Diagram: a transition labeled with $R_1$, a transition labeled with $R_2$]

  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 $\varepsilon$ (which does not consume input)
- *Nondeterministic*: two arrows leaving the same state may have the same label
Converting regular expressions to NFAs is easy.
Assume each NFA has one start state, unique accept state
• Sums and Kleene star are easy with NFAs
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 its 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…
DEMO: OCAMLLLEX

lexlex.mll, olex.mll, piglatin.mll