Lecture 9

COMPILER DESIGN
(Simplified) Compiler Structure

Source Code
(character stream)
if (b == 0) a = 0;

Lexical Analysis
Token Stream
Parsing
Abstract Syntax Tree
Intermediate Code Generation
Intermediate Code
Code Generation

Assembly Code
CMP ECX, 0
SETBZ EAX

Front End
(machine independent)

Middle End
(compiler dependent)

Back End
(machine dependent)
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 = 1 ; }

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 %12, label %13
12:
    store i64* %a, 1
    br label %13
13:

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

Lexical Analysis
Parsing
Analysis & Transformation
Backend
Today: Lexing

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

Token stream:
if ( b == 0 ) { a = 1 ; }

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

Intermediate code:
  l1:  %cnd = icmp eq i64 %b, 0  
       br i1 %cnd, label %l2, label %l3
  l2:  store i64* %a, 1  
       br label %l3
  l3:

Assembly Code
  l1:  cmpq %eax, $0
       jeq l2
       jmp l3
  l2:  ...

Lexical Analysis
Parsing
Analysis & Transformation
Backend
First Step: Lexical Analysis

• Change the character stream “if (b == 0) a = 1;” into tokens
  
  if ( b == 0 ) { a = 1 ; }

  IF; LPAREN; Ident(“b”); EQEQ; Int(0); RPAREN; LBRACE;
  Ident(“a”); EQ; Int(1); 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: (* Compiler Design: 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, 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
Tricky/Fun Examples

• (Early) FORTRAN
  – Whitespace insignificant, so \texttt{VAR1} same as \texttt{VA\_R1}
  – Then, how about
    \begin{verbatim}
    DO 5  I = 1,25  (looping)
    DO 5  I = 1.25  (assignment)
    \end{verbatim}

• C++
  – Template syntax: Foo<Bar>
  – Stream syntax: \texttt{cin \textgreater\textgreater x}
  – But, how about nested templates
    \texttt{Foo<Bar<Bazz>>>}

PRINCIPLED SOLUTION TO LEXING
Regular Expressions

- Regular expressions precisely describe sets of strings
- A regular expression $R$ has one of the following forms
  - $\varepsilon$  
    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 ($\varepsilon \mid R$)
  - [‘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 is useful to be able to name regular expressions

  ```
  let lowercase = [ 'a'–'z' ]
  let uppercase = [ 'A'–'Z' ]
  let character = uppercase | lowercase
  ```
Chomsky Hierarchy

- Regular
- Context-Free
- Context-Sensitive
- Recursively Enumerable
• Consider the input string:  \texttt{if x = 0}
  – Could lex as \texttt{if x = 0} or as \texttt{if x = 0}  

• Thus, regular expressions alone can be ambiguous

• We 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
How to Match?

• Conflicts: tokens whose regular expressions have a shared prefix
  
  Keyword
    "if"
  
  Identifier
    ([ 'a'−'z' ] | [ 'A'−'Z' ]) ([ '0'−'9' ] | '_' | [ 'a'−'z' ] | [ 'A'−'Z' ]) *

• How to resolve?
  – 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 Generator

- 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)

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

- Generates scanning code that
  1. Decides whether the input is of the form $(R_1 | \ldots | R_n)^*$
  2. After matching a (longest) token, runs the associated action
lexlex.mll

**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
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

\[ \text{Non-"} \]

\[ \text{Non-"} \]

\[ \text{Non-"} \]
• Can we build a finite automaton for every regular expression?
  – Yes!

• Strategy: consider every possible regular expression (by induction on the structure of the regular expressions)
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 stars are easy with NFAs

### RE to NFA (cont’d)

\[ R_1 \mid R_2 \]

\[ R^* \]
DFA vs. NFA

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

- **NFA**
  - Automaton potentially has a choice at every step
  - Accepts an input 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