Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Unification Problem

Given a set of pairs of terms ("equations")
{(s₁, t₁), (s₂, t₂), ..., (sₙ, tₙ)}
(the unification problem) does there exist a substitution σ (the unification solution) of terms for variables such that
σ(sᵢ) = σ(tᵢ),
for all i = 1, ..., n?

Uses for Unification

- Type Inference and type checking
- Pattern matching as in OCaml
  - Can use a simplified version of algorithm
- Logic Programming - Prolog
- Simple parsing

Unification Algorithm

- Delete: if s = t (they are the same term) then Unif(S) = Unif(S')
- Decompose: if s = f(q₁, ..., qₘ) and t = f(r₁, ..., rₘ) (same f, same m!), then Unif(S) = Unif({(q₁, r₁), ..., (qₘ, rₘ)} ∪ S')
- Orient: if t = x is a variable, and s is not a variable, Unif(S) = Unif({(x = s)} ∪ S')
- Eliminate: if s = x is a variable, and x does not occur in t (the occurs check), then
  - Let φ = {x → t}
    Unif(S) = Unif(φ(S')) o {x → t}
  - Let ψ = Unif(φ(S'))
    Unif(S) = {x → ψ(t)} o ψ
  - Note: (x → a) o (y → b) = {y → ((x → a)(b))} o {x → a} if y not in a

3/21/24
Tricks for Efficient Unification

- Don’t return substitution, rather do it incrementally
- Make substitution be constant time
  - Requires implementation of terms to use mutable structures (or possibly lazy structures)
  - We won’t discuss these

Example

- \(x, y, z\) variables, \(f, g\) constructors
  - \(S = \{(f(x) = f(g(f(z), y))), (g(y, y) = x)\}\) is nonempty
  - Unify \(\{(f(x) = f(g(f(z), y))), (g(y, y) = x)\}\) = ?

Example

- \(x, y, z\) variables, \(f, g\) constructors
  - Pick a pair: \((g(y, y)) = x)\)
  - Orient: \((x = g(y, y))\)
  - Unify \(\{(f(x) = f(g(f(z), y))), (g(y, y) = x)\}\) = ?
    Unify \(\{(f(x) = f(g(f(z), y))), (x = g(y, y))\}\)
    by Orient

Example

- \(x, y, z\) variables, \(f, g\) constructors
  - Unify \(\{(f(x) = f(g(f(z), y))), (x = g(y, y))\}\) = ?
Example
- \( x, y, z \) variables, \( f, g \) constructors
- \( \{(f(x) = f(g(f(z),y))), (x = g(y,y))\} \) is non-empty
  - Unify \( \{(f(x) = f(g(f(z),y))), (x = g(y,y))\} = ? \)

Example
- \( x, y, z \) variables, \( f, g \) constructors
- Pick a pair: \( (x = g(y,y)) \)
  - Unify \( \{(f(x) = f(g(f(z),y))), (x = g(y,y))\} = ? \)

Example
- \( x, y, z \) variables, \( f, g \) constructors
- Pick a pair: \( (x = g(y,y)) \)
- Eliminate \( x \) with substitution \( \{x \mapsto g(y,y)\} \)
  - Check: \( x \) not in \( g(y,y) \)
  - Unify \( \{(f(x) = f(g(f(z),y))), (x = g(y,y))\} = ? \)

Example
- \( x, y, z \) variables, \( f, g \) constructors
- \( \{(f(g(y,y)) = f(g(f(z),y)))\} \) is non-empty
  - Unify \( \{(f(g(y,y)) = f(g(f(z),y)))\} o \{x \mapsto g(y,y)\} = ? \)
Example

- x, y, z variables, f, g constructors
- Pick a pair: \((f(g(y,y)) = f(g(f(z),y)))\)

Unify \({(f(g(y,y)) = f(g(f(z),y)))}\)

\(\circ \{x \mapsto g(y,y)\} = ?\)

Example

- x, y, z variables, f, g constructors
- Pick a pair: \((f(g(y,y)) = f(g(f(z),y)))\)
- Decompose: \((g(y,y)) = g(f(z),y))\) becomes \({(y = f(z)); (y = y)}\)

Unify \({(g(y,y) = g(f(z),y)})\)

\(\circ \{x \mapsto g(y,y)\} = ?\)

Example

- x, y, z variables, f, g constructors
- Pick a pair: \((g(y,y)) = g(f(z),y))\)

Unify \({(y = f(z)); (y = y)}\) \(\circ \{x \mapsto g(y,y)\} = ?\)
Example

- $x, y, z$ variables, $f, g$ constructors
- \{(y = f(z)); (y = y)\} o \{x \to g(y, y)\} is non-empty
- Unify \{(y = f(z)); (y = y)\} o \{x \to g(y, y)\} = ?

Example

- $x, y, z$ variables, $f, g$ constructors
- Pick a pair: $(y = f(z))$
- Unify \{(y = f(z)); (y = y)\} o \{x \to g(y, y)\} = ?

Example

- $x, y, z$ variables, $f, g$ constructors
- Pick a pair: $(f(z) = f(z))$
- Eliminate $y$ with \{y $\to f(z)$\}
- Unify \{(f(z) = f(z))\} o \{y \to f(z); x \to g(f(z), f(z))\} = ?

Example

- $x, y, z$ variables, $f, g$ constructors
- \{(f(z) = f(z))\} is non-empty
- Unify \{(f(z) = f(z))\} o \{y \to f(z); x \to g(f(z), f(z))\} = ?

Example

- $x, y, z$ variables, $f, g$ constructors
- Pick a pair: $(f(z) = f(z))$
- Unify \{(f(z) = f(z))\} o \{y \to f(z); x \to g(f(z), f(z))\} = ?
Example

- $x, y, z$ variables, $f, g$ constructors
- Pick a pair: $(f(z) = f(z))$
- Delete
- Unify\{(f(z) = f(z))\}
  - $\{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\} = \{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\}$
- Unify $\{\} o \{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\} = \{\} o \{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\}$

Example

- $x, y, z$ variables, $f, g$ constructors
- $\{\}$ is empty
- Unify $\{\} = \text{identity function}$
- Unify $\{\} o \{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\} = \{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\}$

Example of Failure: Decompose

- Unify\{(f(x,g(y)) = f(h(y),x))\}
- Decompose: $(f(x,g(y)) = f(h(y),x))$
  - $\{x = h(y), (g(y) = x)\}$
- Orient: $(g(y) = x)$
  - $\{x = h(y), (x = g(y))\}$
- Eliminate: $(x = h(y))$
- Unify $(h(y) = g(y)) o \{x \rightarrow h(y)\}$
- No rule to apply! Decompose fails!

Example of Failure: Occurs Check

- Unify\{(f(x,g(x)) = f(h(x),x))\}
- Decompose: $(f(x,g(x)) = f(h(x),x))$
  - $\{x = h(x), (g(x) = x)\}$
- Orient: $(g(x) = x)$
  - $\{x = h(x), (x = g(x))\}$
- No rules apply.
### Programming Languages & Compilers

#### Three Main Topics of the Course

I. New Programming Paradigm
II. Language Translation
III. Language Semantics

#### Major Phases of a Compiler

- Source Program
- **Lex**
- Tokens
- **Parse**
- Abstract Syntax
- **Semantic Analysis**
- Symbol Table
- **Translate**
- Intermediate Representation
- **Optimize**
- Optimized IR
- Instruction Selection
- Unoptimized Machine-Specific Assembly Language
- **Optimize**
- Optimized Machine-Specific Assembly Language
- **Emit code**
- Assembly Language
- **Assembler**
- Relocatable Object Code
- **Linker**
- Machine Code

*Modified from "Modern Compiler Implementation in ML", by Andrew Appel*

#### Where We Are Going Next?

- We want to turn strings (code) into computer instructions
- Done in phases
- Turn strings into abstract syntax trees (parse)
- Translate abstract syntax trees into executable instructions (interpret or compile)

#### Meta-discourse

- Language Syntax and Semantics
- Syntax
  - Regular Expressions, DFSAs and NDFSAs
  - Grammars
- Semantics
  - Natural Semantics
  - Transition Semantics

### Language Syntax

- Syntax is the description of which strings of symbols are meaningful expressions in a language
- It takes more than syntax to understand a language; need meaning (semantics) too
- Syntax is the entry point
Syntax of English Language

- **Pattern 1**
  
<table>
<thead>
<tr>
<th>Subject</th>
<th>Verb</th>
</tr>
</thead>
<tbody>
<tr>
<td>David</td>
<td>sings</td>
</tr>
<tr>
<td>The dog</td>
<td>barked</td>
</tr>
<tr>
<td>Susan</td>
<td>yawned</td>
</tr>
</tbody>
</table>

- **Pattern 2**
  
<table>
<thead>
<tr>
<th>Subject</th>
<th>Verb</th>
<th>Direct Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>David</td>
<td>sings</td>
<td>ballads</td>
</tr>
<tr>
<td>The professor</td>
<td>wants</td>
<td>to retire</td>
</tr>
<tr>
<td>The jury</td>
<td>found</td>
<td>the defendant guilty</td>
</tr>
</tbody>
</table>

Elements of Syntax

- **Character set** – previously always ASCII, now often 64 character sets
- **Keywords** – usually reserved
- **Special constants** – cannot be assigned to
- **Identifiers** – can be assigned to
- **Operator symbols**
- **Delimiters** (parenthesis, braces, brackets)
- **Blanks** (aka white space)

Elements of Syntax

- **Expressions**
  
  if ... then begin ... ; ... end else begin ... ; ... end
- **Type expressions**
  
  `typexpr₁ -> typexpr₂`
- **Declarations** (in functional languages)
  
  `let pattern = expr`
- **Statements** (in imperative languages)
  
  `a = b + c`
- **Subprograms**
  
  `let pattern₁ = expr₁ in expr`

Lexing and Parsing

- Converting strings to abstract syntax trees done in two phases
  
  - **Lexing**: Converting string (or streams of characters) into lists (or streams) of tokens (the “words” of the language)
    - Specification Technique: Regular Expressions
  - **Parsing**: Convert a list of tokens into an abstract syntax tree
    - Specification Technique: BNF Grammars

Formal Language Descriptions

- **Regular expressions, regular grammars, finite state automata**
- **Context-free grammars, BNF grammars, syntax diagrams**
- **Whole family more of grammars and automata – covered in automata theory**
Grammars

- Grammars are formal descriptions of which strings over a given character set are in a particular language
- Language designers write grammar
- Language implementers use grammar to know what programs to accept
- Language users use grammar to know how to write legitimate programs

Regular Expressions - Review

- Start with a given character set – a, b, c...
- \( L(\varepsilon) = \{ \varepsilon \} \)
- Each character is a regular expression
  - It represents the set of one string containing just that character
  - \( L(a) = \{a\} \)

Regular Expressions

- If \( x \) and \( y \) are regular expressions, then \( xy \) is a regular expression
  - It represents the set of all strings made from first a string described by \( x \) then a string described by \( y \)
  - If \( L(x) = \{a, ab\} \) and \( L(y) = \{c, d\} \), then \( L(xy) = \{ac, ad, abc, abd\} \)

Example Regular Expressions

- \((0 \lor 1)^*1\)
  - The set of all strings of 0’s and 1’s ending in 1, \{1, 01, 11,…\}

- \(a^*b(a^*)\)
  - The set of all strings of a’s and b’s with exactly one b

- \(((01) \lor (10))^*\)
  - You tell me

- Regular expressions (equivalently, regular grammars) important for lexing, breaking strings into recognized words
Right Regular Grammars

- Subclass of BNF (covered in detail sool)
- Only rules of form
  \[ \text{nonterminal} ::= \text{terminal} \text{nonterminal} \text{ or } \text{nonterminal} ::= \varepsilon \]
- Defines same class of languages as regular expressions
- Important for writing lexers (programs that convert strings of characters into strings of tokens)
- Close connection to nondeterministic finite state automata – nonterminals = states; rule = edge

Example

Right regular grammar:

\[
\text{Balanced} ::= \varepsilon
\]
\[
\text{Balanced} ::= 0\text{OneAndMore}
\]
\[
\text{Balanced} ::= 1\text{ZeroAndMore}
\]
\[
\text{OneAndMore} ::= 1\text{Balanced}
\]
\[
\text{ZeroAndMore} ::= 0\text{Balanced}
\]

Generates even length strings where every initial substring of even length has same number of 0’s as 1’s

Implementing Regular Expressions

- Regular expressions reasonable way to generate strings in language
- Not so good for recognizing when a string is in language
- Problems with Regular Expressions
  - which option to choose,
  - how many repetitions to make
- Answer: finite state automata
- Should have seen in CS374

Example: Lexing

Regular expressions good for describing lexemes (words) in a programming language

- Identifier = (a ∨ b ∨ ... ∨ z ∨ A ∨ B ∨ ... ∨ Z) (a ∨ b ∨ ... ∨ z ∨ A ∨ B ∨ ... ∨ Z ∨ 0 ∨ 1 ∨ ... ∨ 9)*
- Digit = (0 ∨ 1 ∨ ... ∨ 9)
- Number = 0 ∨ (1 ∨ ... ∨ 9)(0 ∨ ... ∨ 9)* ∨ ~ (1 ∨ ... ∨ 9)(0 ∨ ... ∨ 9)*
- Keywords: if = if, while = while,...

Lexing

- Different syntactic categories of “words”: tokens
- Example:
  - Convert sequence of characters into sequence of strings, integers, and floating point numbers.
  - "asd 123 jkl 3.14" will become: [String "asd"; Int 123; String "jkl"; Float 3.14]

Lex, ocamllex

- Could write the reg exp, then translate to DFA by hand
  - A lot of work
- Better: Write program to take reg exp as input and automatically generates automata
- Lex is such a program
- ocamllex version for ocaml
How to do it

To use regular expressions to parse our input we need:
  - Some way to identify the input string — call it a lexing buffer
  - Set of regular expressions,
  - Corresponding set of actions to take when they are matched.

The lexer will take the regular expressions and generate a state machine.
The state machine will take our lexing buffer and apply the transitions...
If we reach an accepting state from which we can go no further, the machine will perform the appropriate action.

Mechanics

Put table of reg exp and corresponding actions (written in ocaml) into a file <filename>.mll
Call ocamlex <filename>.mll
Produces Ocaml code for a lexical analyzer in file <filename>.ml

Sample Input

rule main = parse
[0-'9']+ { print_string "Int\n"}
[0-'9']+'.'[0-'9']+ { print_string "Float\n"}
[0-'9']+'.'[0-'9']+ { print_string "String\n"}
| _ { main lexbuf }
let newlexbuf = (Lexing.from_channel stdin) in
main newlexbuf

General Input

{ header }
let ident = regexp ...  Introduces ident for use in later regular expressions
rule entrypoint [arg1... argn] = parse
  regexp { action }
  | ...
  | regexp { action }
and entrypoint [arg1... argn] = parse ...
and ...
{ trailer }

Ocamlex Input

header and trailer contain arbitrary ocaml code put at top an bottom of <filename>.ml
let ident = regexp ...  Introduces ident for use in later regular expressions
Ocamlllex Input

- `<filename>.ml` contains one lexing function per `entrypoint`
  - Name of function is name given for `entrypoint`
  - Each entry point becomes an Ocaml function that takes \( n+1 \) arguments, the extra implicit last argument being of type `Lexing.lexbuf`
- `arg1... argn` are for use in `action`

Ocamlllex Regular Expression

- Single quoted characters for letters: `'a`
- `_:` (underscore) matches any letter
- `Eof`: special “end_of_file” marker
- Concatenation same as usual
- “string”: concatenation of sequence of characters
- `e1 / e2`: choice - what was `e1 \lor e2`

Ocamlllex Regular Expression

- `[c1 - c2]`: choice of any character between first and second inclusive, as determined by character codes
- `[^c1 - c2]`: choice of any character NOT in set
- `e*`: same as before
- `e+`: same as `e e*`
- `e?`: option - was `e \lor \varepsilon`
- `(e)`: same as `e`

Ocamlllex Manual

- More details can be found at
- Version for ocaml 4.07: https://v2.ocaml.org/releases/4.07/htmlman/lexyacc.html
  (same, except formatting, I think)
Example: test.mll

```ocaml
{ type result = Int of int | Float of float | String of string }
let digit = ['0'-'9']
let digits = digit +
let lower_case = ['a'-'z']
let upper_case = ['A'-'Z']
let letter = upper_case | lower_case
let letters = letter +
```

Example: test.mll

```ocaml
rule main = parse
  (digits)'.'digits as f  { Float (float_of_string f) }
| digits as n              { Int (int_of_string n) }
| letters as s             { String s }
| _ { main lexbuf }
{ let newlexbuf = (Lexing.from_channel stdin) in
  print_newline ();
  main newlexbuf  }
```

Example

```ocaml
# #use "test.ml";;
...
val main : Lexing.lexbuf -> result = <fun>
val __ocaml_lex_main_rec : Lexing.lexbuf -> int -> result = <fun>
hi there 234 5.2
  : result = String "hi"
```

What happened to the rest?!?

Example

```ocaml
# let b = Lexing.from_channel stdin;;
# main b;;
hi 673 there
  - : result = String "hi"
# main b;;
- : result = Int 673
# main b;;
- : result = String "there"
```

Your Turn

- Work on MP8
  - Add a few keywords
  - Implement booleans and unit
  - Implement Ints and Floats
  - Implement identifiers

Problem

- How to get lexer to look at more than the first token at one time?
  - Answer: action has to tell it to -- recursive calls
    - Not what you want to sew this together with ocamlyacc
  - Side Benefit: can add "state" into lexing
  - Note: already used this with the _ case
Example

rule main = parse
(digits) '.' digits as f { Float (float_of_string f) :: main lexbuf}
| digits as n    { Int (int_of_string n) ::
    main lexbuf }
| letters as s   { String s :: main lexbuf}
| eof            { [] }
| _              { main lexbuf }

Example Results

hi there 234 5.2
- : result list = [String "hi", String "there", Int 234; Float 5.2]
#

Used Ctrl-d to send the end-of-file signal

Dealing with comments

First Attempt
let open_comment = "(*)
let close_comment = ")"
rule main = parse
(digits) '.' digits as f { Float (float_of_string f) :: main lexbuf}
| digits as n    { Int (int_of_string n) ::
    main lexbuf }
| letters as s   { String s :: main lexbuf}
| open_comment   { comment  lexbuf}
| eof            { [] }
| _              { main lexbuf }

and comment = parse
    open_comment    { main lexbuf }
| _               { comment lexbuf }

Dealing with nested comments

rule main = parse ...
| open_comment   { comment 1 lexbuf}
| eof            { [] }
| _              { main lexbuf }

and comment depth = parse
    open_comment    { comment (depth+1) lexbuf }
| close_comment  { if depth = 1
    then main lexbuf
    else comment (depth - 1) lexbuf }
| _              { comment depth lexbuf }

Dealing with nested comments

rule main = parse
(digits) '.' digits as f { Float (float_of_string f) ::
    main lexbuf}
| digits as n    { Int (int_of_string n) :: main lexbuf }
| letters as s   { String s :: main lexbuf}
| open_comment   { (comment 1 lexbuf}
| eof            { [] }
| _              { main lexbuf }

and comment depth = parse
    open_comment    { comment (depth+1) lexbuf }
| close_comment  { if depth = 1
    then main lexbuf
    else comment (depth - 1) lexbuf }
| _              { comment depth lexbuf }

Dealing with nested comments

and comment depth = parse

open_comment        { comment (depth+1) lexbuf }
| close_comment       { if depth = 1
then main lexbuf
else comment (depth - 1) lexbuf }
| _                   { comment depth lexbuf }

Types of Formal Language Descriptions

- Regular expressions, regular grammars
- Context-free grammars, BNF grammars, syntax diagrams
- Finite state automata
- Pushdown automata
- Whole family more of grammars and automata – covered in automata theory

Sample Grammar

- Language: Parenthesized sums of 0's and 1's
  - <Sum> ::= 0
  - <Sum ::= 1
  - <Sum ::= <Sum> + <Sum>
  - <Sum ::= (<Sum>)
  - Can be abbreviated as
  - <Sum ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

BNF Grammars

- BNF rules (aka productions) have form
  - \( X ::= y \)
  - where \( X \) is any nonterminal and \( y \) is a string
  - of terminals and nonterminals
- BNF grammar is a set of BNF rules such that
  - every nonterminal appears on the left of
  - some rule

Sample Grammar

- Terminals: 0 1 + ( )
- Nonterminals: <Sum>
- Start symbol = <Sum>
  - <Sum ::= 0
  - <Sum ::= 1
  - <Sum ::= <Sum> + <Sum>
  - <Sum ::= (<Sum>)
  - Can be abbreviated as
  - <Sum ::= 0 | 1
  - | <Sum> + <Sum> | (<Sum>)
BNF Derivations

- Given rules
  
  \[ X ::= yZw \]
  \[ Z ::= \nu \]

  we may replace \( Z \) by \( \nu \) to say
  
  \[ X \Rightarrow yZw \Rightarrow y\nu w \]

- Sequence of such replacements called derivation

- Derivation called right-most if always replace the right-most non-terminal

---

Start with the start symbol:

\[ <\text{Sum}> \Rightarrow \]

---

Pick a non-terminal

\[ <\text{Sum}> \Rightarrow \]

---

Pick a rule and substitute:

\[ <\text{Sum}> ::= <\text{Sum}> + <\text{Sum}> \]

\[ <\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}> \]

---

Pick a non-terminal:

\[ <\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}> \]

---

Pick a rule and substitute:

\[ <\text{Sum}> ::= ( <\text{Sum}> ) \]

\[ <\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}> \]

\[ \Rightarrow ( <\text{Sum}> ) + <\text{Sum}> \]
BNF Derivations

Pick a non-terminal:

<Sum> => <Sum> + <Sum>
=> ( <Sum> ) + <Sum>

BNF Derivations

Pick a non-terminal:

<Sum> => <Sum> + <Sum>
=> ( <Sum> ) + <Sum>
=> ( <Sum> + <Sum> ) + <Sum>

BNF Derivations

Pick a non-terminal:

<Sum> => <Sum> + <Sum>
=> ( <Sum> ) + <Sum>
=> ( <Sum> + 1 ) + <Sum>

BNF Derivations

Pick a non-terminal:

<Sum> => <Sum> + <Sum>
=> ( <Sum> ) + <Sum>
=> ( <Sum> + 1 ) + <Sum>
=> ( <Sum> + 1 ) + 0
BNF Derivations

Pick a non-terminal:

<Sum> => <Sum> + <Sum>
=> ( <Sum> ) + <Sum>
=> ( <Sum> + <Sum> ) + <Sum>
=> ( <Sum> + 1 ) + <Sum>
=> ( <Sum> + 1 ) + 0

BNF Derivations

Pick a rule and substitute

<Sum> ::= 0
<Sum> => <Sum> + <Sum>
=> ( <Sum> ) + <Sum>
=> ( <Sum> + <Sum> ) + <Sum>
=> ( <Sum> + 1 ) + <Sum>
=> ( <Sum> + 1 ) 0
=> ( 0 + 1 ) + 0

BNF Derivations

( 0 + 1 ) + 0 is generated by grammar

<Sum> => <Sum> + <Sum>
=> ( <Sum> ) + <Sum>
=> ( <Sum> + <Sum> ) + <Sum>
=> ( <Sum> + 1 ) + <Sum>
=> ( <Sum> + 1 ) + 0
=> ( 0 + 1 ) + 0

BNF Derivations

Pick a non-terminal:

<Sum> => <Sum> + <Sum>
=> ( <Sum> ) + <Sum>
=> ( <Sum> + <Sum> ) + <Sum>
=> ( <Sum> + 1 ) + <Sum>
=> ( <Sum> + 1 ) + <Sum>