Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC

https://courses.engr.illinois.edu/cs421/sp2023

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

Programming Languages & Compilers

Three Main Topics of the Course

I
New Programming Paradigm

II
Language Translation

III
Language Semantics

Programming Languages & Compilers

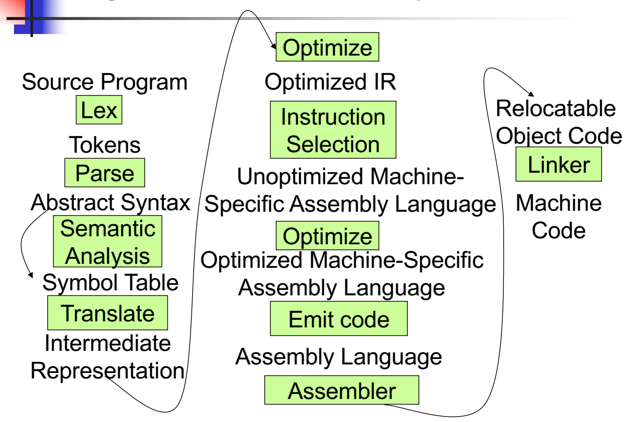
II : Language Translation

Type Systems

Lexing and Parsing

Interpretation

Major Phases of a Compiler



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

Subject	Verb
David	*sings*
The dog	*barked*
Susan	*yawned*

- Pattern 2

Subject	Verb	Direct Object
David	*sings*	*ballads*
The professor	*wants*	*to retire*
The jury	*found*	*the defendant guilty*

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
  - $typexpr_1$ -> $typexpr_2$
- Declarations (in functional languages)
  - let *pattern* = *expr*
- Statements (in imperative languages)
  - a = b + c
- Subprograms
  - let $pattern_1$ = $expr_1$ in *expr*

Elements of Syntax

- Modules
- Interfaces
- Classes (for object-oriented languages)

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**…
- $L(\varepsilon)$ = {"" }
- Each character is a regular expression
  - It represents the set of one string containing just that character
  - $L(\mathbf{a})$ = {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}

Regular Expressions

- If **x** and **y** are regular expressions, then **x**∨**y** is a regular expression
  - It represents the set of strings described by either **x** or **y**

If $L(x)$={a,ab} and $L(y)$={c,d}
then $L(x \vee y)$={a,ab,c,d}

Regular Expressions

- If **x** is a regular expression, then so is (**x**)
  - It represents the same thing as **x**
- If **x** is a regular expression, then so is **x***
  - It represents strings made from concatenating zero or more strings from **x**

If $L(x)$ = {a,ab} then $L(x^*)$ ={"",a,ab,aa,aab,abab,…}

- ε
  - It represents {""}, set containing the empty string
- Φ
  - It represents { }, the empty set

Example Regular Expressions

- **(0∨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) ∨(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
  <nonterminal>::=<terminal><nonterminal> or
  <nonterminal>::=<terminal> or
  <nonterminal>::=ε
- 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:
  <Balanced> ::= ε
  <Balanced> ::= 0<OneAndMore>
  <Balanced> ::= 1<ZeroAndMore>
  <OneAndMore> ::= 1<Balanced>
  <ZeroAndMore> ::= 0<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,…