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

Example - cont

- **Problem:** shift or reduce?
- You can shift-shift-reduce-reduce or reduce-shift-shift-reduce
- Shift first - right associative
- Reduce first- left associative

Reduce - Reduce Conflicts

- **Problem:** can't decide between two different rules to reduce by
- **Symptom:** RHS of one production suffix of another
- Requires examining grammar and rewriting it
- Harder to solve than shift-reduce errors

Example

- S ::= A | aB   A ::= abc   B ::= bc

● abc	shift
a ● bc	shift
ab ● c	shift
abc ●	

- Problem: reduce by B ::= bc then by S ::= aB, or by A::= abc then S::A?

Disambiguating a Grammar

- Given ambiguous grammar G, with start symbol S, find a grammar G' with same start symbol, such that

  language of G = language of G'
- Not always possible
- No algorithm in general

Disambiguating a Grammar

- Idea: Each non-terminal represents all strings having some property
- Identify these properties (often in terms of things that can't happen)
- Use these properties to inductively guarantee every string in language has a unique parse

Steps to Grammar Disambiguation

- Identify the rules and a smallest use that display ambiguity
- Decide which parse to keep; why should others be thrown out?
- What syntactic restrictions on subexpressions are needed to throw out the bad (while keeping the good)?
- Add a new non-terminal and rules to describe this set of restricted subexpressions (called stratifying, or refactoring)
- **Characterize each non-terminal by a language invariant**
- Replace old rules to use new non-terminals
- Rinse and repeat

Predence in Grammar

- Higher precedence translates to longer derivation chain
- Example:

<exp> ::= 0 | 1 | <exp> + <exp>
     | <exp> * <exp>

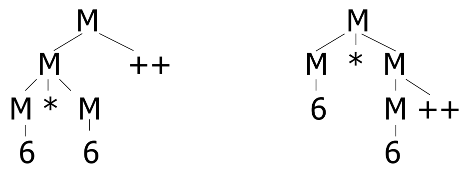
- Becomes

  <exp> ::= <mult_exp>
     | <exp> + <mult_exp>
  <mult_exp> ::= <id> | <mult_exp> * <id>
  <id> ::= 0 | 1

- <mult_exp> = maybe mult, not plus

More Disambiguating Grammars

- M ::= M * M | (M) | M ++ | 6
- Ambiguous because of associativity of *
- Because of conflict between * and ++:
- 6 * 6 ++   6 * 6 ++



M ::= M * M | (M) | M ++ | 6

- How to disambiguate?
- Choose associativity for *
- Choose precedence between * and ++
- Four possibilities
- Three - four different approaches
- Some easier than others
- Will do --- all?

M ::= M * M | (M) | M ++ | 6

- Think about 6 * 6 ++ * 6 * 6 ++

M ::= M * M | (M) | M ++ | 6

- Think about 6 * 6 ++ * 6 * 6 ++
- Let's start with observations
- If * binds less tightly than ++, then no * can be the immediate subtree to a ++.
  - We would need a language for things that don't parse as *
- If * binds more tightly than ++, then …
- The right subtree to * can't be a ++
- But the left can!
  - Need different languages of the left and right

M ::= M * M | (M) | M ++ | 6

- Think about 6 * 6 ++ * 6 * 6 ++
- ++ higher prec than *
  - P == maybe ++, not *
  - A == not *, not ++
- A ::= (M) | 6
- P ::= A | P ++
- M ::= M * P | P   * assoc left OR
- M ::= P * M | P   * assoc right

M ::= M * M | (M) | M ++ | 6

- * higher prec than ++, * assoc left
  - 6 * 6 ++ * 6 ++ * 6
- M :: = M++ | S
- S == maybe *, not ++
- M++ == is ++, not *
- A ::= (M) | 6
- S ::= S * A | M++ * A | A

M ::= M * M | (M) | M ++ | 6

- * higher prec than ++, * assoc left
  - 6 * 6 ++ * 6 ++ * 6
- M :: = M++ | S
- S == maybe *, not ++
- M++ == is ++, not *
- A ::= (M) | 6
- S ::= S * A | M++ * A | A
- S ::= M * A | A

M ::= M * M | (M) | M ++ | 6

- * higher prec than ++, * assoc left
  - 6 * 6 ++ * 6 ++ * 6
- M :: = M++ | M * A | A
- S == maybe *, not ++
- M++ == is ++, not *
- A ::= (M) | 6
- S ::= S * A | M++ * A | A
- S ::= M * A | A

M ::= M * M | (M) | M ++ | 6

- * higher prec than ++, * assoc left
  - 6 * 6 ++ * 6 ++ * 6
- M :: = M++ | M * A | A
- A ::= (M) | 6

- M++ == must be ++
- M * A == must be *
- A == not ++ or *

M ::= M * M | (M) | M ++ | 6

- * higher prec than ++, * assoc right
  - 6 * 6 ++ * 6 ++ * 6
- M :: = M++ | S
- S == maybe *, not ++
- S ::= A | A * S ......
- But ... 6 * 6 ++ * 6, how does that parse?
- ((6 * 6)++) * 6 so .... S ::= M ++ * S as well
- S ::= A | A * S | M++ S
- A | M++ == possibly ++, not *

M ::= M * M | (M) | M ++ | 6

- * higher prec than ++, * assoc right
  - 6 * 6 ++ * 6 ++ * 6
- M :: = M++

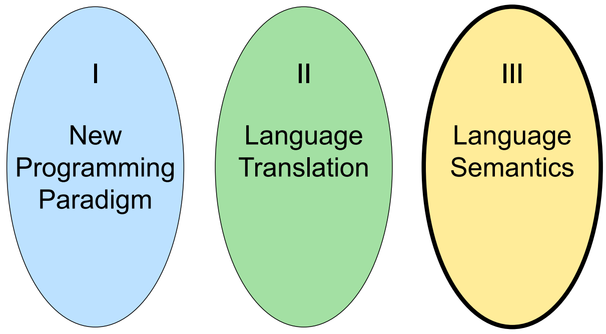
  | S
- S ::= A

  | A * S

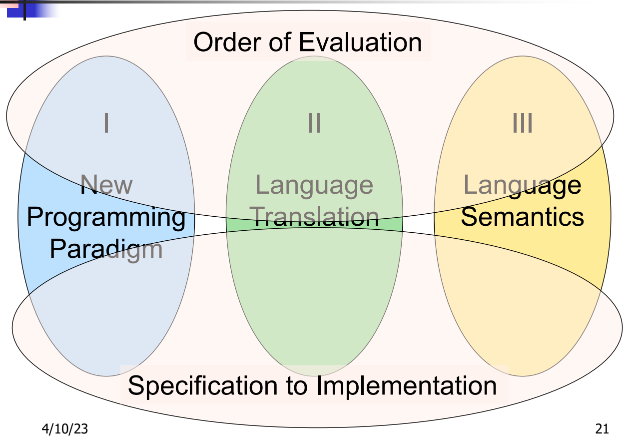
  | M++ * S
- Notice the doubling of rules for *

Programming Languages & Compilers

Three Main Topics of the Course

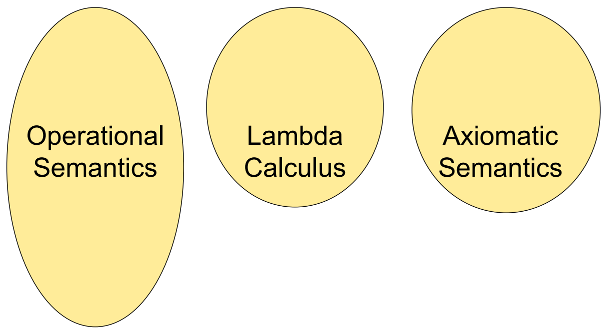


Programming Languages & Compilers

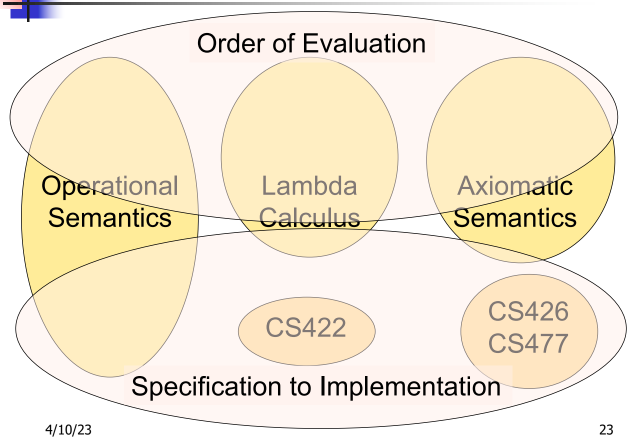


Programming Languages & Compilers

III : Language Semantics



Programming Languages & Compilers



Semantics

- Expresses the meaning of syntax
- Static semantics
  - Meaning based only on the form of the expression without executing it
  - Usually restricted to type checking / type inference

Dynamic semantics

- Method of describing meaning of executing a program
- Several different types:
  - Operational Semantics
  - Axiomatic Semantics
  - Denotational Semantics

Dynamic Semantics

- Different languages better suited to different types of semantics
- Different types of semantics serve different purposes

Operational Semantics

- Start with a simple notion of machine
- Describe how to execute (implement) programs of language on virtual machine, by describing how to execute each program statement (ie, following the *structure* of the program)
- Meaning of program is how its execution changes the state of the machine
- Useful as basis for implementations

Axiomatic Semantics

- Also called Floyd-Hoare Logic
- Based on formal logic (first order predicate calculus)
- Axiomatic Semantics is a logical system built from *axioms* and *inference rules*
- Mainly suited to simple imperative programming languages

Axiomatic Semantics

- Used to formally prove a property (*post-condition*) of the *state* (the values of the program variables) after the execution of program, assuming another property (*pre-condition*) of the state before execution
- Written :

  {Precondition} Program {Postcondition}
- Source of idea of *loop invariant*

Denotational Semantics

- Construct a function $\mathcal{M}$ assigning a mathematical meaning to each program construct
- Lambda calculus often used as the range of the meaning function
- Meaning function is compositional: meaning of construct built from meaning of parts
- Useful for proving properties of programs

1450 minutes