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

- \( S ::= A \mid aB \quad A ::= abc \quad B ::= bc \)

- \( \bullet abc \quad shift \)
- \( a \bullet bc \quad shift \)
- \( ab \bullet c \quad shift \)
- \( abc \bullet \)

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
  \[ \text{language of } G = \text{language of } G' \]
- 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:
  \[<\text{exp}> ::= 0 \mid 1 \mid <\text{exp}> + <\text{exp}> \]
  \[\quad \mid <\text{exp}> * <\text{exp}>\]
- Becomes
  \[<\text{exp}> ::= <\text{mult}\_exp> \]
  \[\quad \mid <\text{exp}> + <\text{mult}\_exp>\]
  \[<\text{mult}\_exp> ::= <\text{id}> \mid <\text{mult}\_exp> * <\text{id}>\]
  \[<\text{id}> ::= 0 \mid 1\]
- \[<\text{mult}\_exp> = \text{maybe mult, not plus}\]
More Disambiguating Grammars

- $M ::= M \ast M \mid (M) \mid M++ \mid 6$
- Ambiguous because of associativity of $\ast$
- Because of conflict between $\ast$ and $++$:

\[
\begin{align*}
6 \ast 6 & \quad ++ & \quad 6 \ast 6 & \quad ++ \\
& \quad M & \quad M & \quad M
\end{align*}
\]

4/10/23
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?
$M ::= M \times M \mid (M) \mid M++ \mid 6$

- Think about $6 \times 6 ++ \times 6 \times 6 ++$
Think about $6 \times 6 \, \text{++} \times 6 \times 6 \, \text{++}$

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
\[ M ::= M \ast M \mid (M) \mid M++ \mid 6 \]

- * higher prec than ++, * assoc left
  - \[ 6 \ast 6 \ast 6 \ast 6 \ast * 6 \]

- \[ M ::= M++ \mid S \]

- \[ S == \] maybe *, not ++

- \[ M++ == \] is ++, not *

- \[ A ::= (M) \mid 6 \]

- \[ S ::= S \ast A \mid M++ \ast A \mid A \]
\[ M ::= M \ast M \mid (M) \mid M ++ \mid 6 \]

- * higher prec than ++, * assoc left
  - \[ 6 \ast 6 ++ \ast 6 ++ \ast 6 \]
- \[ M ::= M++ \mid S \]
- \[ S == \text{maybe } *, \text{not } ++ \]
- \[ M++ == \text{is } ++, \text{not } * \]
- \[ A ::= (M) \mid 6 \]
- \[ S ::= S \ast A \mid M++ \ast A \mid A \]
- \[ S ::= M \ast A \mid A \]
\[ M ::= M \star M \mid (M) \mid M \+\+ \mid 6 \]

- * higher prec than ++, * assoc left
- \[ 6 \star 6 \+\+ \star 6 \+\+ \star 6 \]

- \[ M ::= M \+\+ \mid M \star A \mid A \]
- \[ S == maybe \star, not ++ \]
- \[ M \+\+ == is ++, not * \]
- \[ A ::= (M) \mid 6 \]
- \[ S ::= S \star A \mid M \+\+ \star A \mid A \]
- \[ S ::= M \star A \mid A \]
\[ M ::= M \times M \mid (M) \mid M ++ \mid 6 \]

- \( \times \) higher prec than ++, \( \times \) assoc left
- \( 6 \times 6 ++ \times 6 ++ \times 6 \)

\[ M ::= = M++ \mid M \times A \mid A \]
\[ A ::= = (M) \mid 6 \]

- \( M++ \) == must be ++
- \( M \times A \) == must be *
- \( A \) == not ++ or *
\[ M ::= M \ast M \mid ( M ) \mid M++ \mid 6 \]

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

4/10/23
Three Main Topics of the Course

I
New Programming Paradigm

II
Language Translation

III
Language Semantics
Programming Languages & Compilers

Order of Evaluation

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

Specification to Implementation
III : Language Semantics

- Operational Semantics
- Lambda Calculus
- Axiomatic Semantics
Programming Languages & Compilers

Order of Evaluation

Operational Semantics
Lambda Calculus
Axiomatic Semantics

Specification to Implementation

CS422
CS426
CS477
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 (i.e., 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 (\textit{post-condition}) of the \textit{state} (the values of the program variables) after the execution of program, assuming another property (\textit{pre-condition}) of the state before execution.

- Written:
  \{\textit{Precondition}\} Program \{\textit{Postcondition}\}

- Source of idea of \textit{loop invariant}
Denotational Semantics

- Construct a function $M$ 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