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
LR Parsing Tables

- Build a pair of tables, Action and Goto, from the grammar
  - This is the hardest part, we omit here
  - Rows labeled by states
  - For Action, columns labeled by terminals and “end-of-tokens” marker
    - (more generally strings of terminals of fixed length)
  - For Goto, columns labeled by non-terminals
Action and Goto Tables

- Given a state and the next input, Action table says either
  - **shift** and go to state \( n \), or
  - **reduce** by production \( k \) (explained in a bit)
  - **accept** or **error**

- Given a state and a non-terminal, Goto table says
  - go to state \( m \)
LR(i) Parsing Algorithm

- Based on push-down automata
- Uses states and transitions (as recorded in Action and Goto tables)
- Uses a stack containing states, terminals and non-terminals
LR(i) Parsing Algorithm

0. Insure token stream ends in special "end-of-tokens" symbol
1. Start in state 1 with an empty stack
2. Push \texttt{state}(1) onto stack
3. Look at next $i$ tokens from token stream ($toks$) (don’t remove yet)
4. If top symbol on stack is \texttt{state}($n$), look up action in Action table at ($n$, $toks$)
LR(i) Parsing Algorithm

5. If action = \textbf{shift} $m$,
   
   a) Remove the top token from token stream and push it onto the stack
   
   b) Push $\textbf{state}(m)$ onto stack
   
   c) Go to step 3
LR(i) Parsing Algorithm

6. If action = \textbf{reduce} \( k \) where production \( k \) is \( E ::= u \)
   a) Remove \( 2 \times \text{length}(u) \) symbols from stack (\( u \) and all the interleaved states)
   b) If new top symbol on stack is \( \text{state}(m) \), look up new state \( p \) in Goto(\( m,E \))
   c) Push \( E \) onto the stack, then push \( \text{state}(p) \) onto the stack
   d) Go to step 3
LR(i) Parsing Algorithm

7. If action = **accept**
   - Stop parsing, return success

8. If action = **error**, 
   - Stop parsing, return failure
Adding Synthesized Attributes

- Add to each `reduce` a rule for calculating the new synthesized attribute from the component attributes.
- Add to each non-terminal pushed onto the stack, the attribute calculated for it.
- When performing a `reduce`,
  - gather the recorded attributes from each non-terminal popped from stack.
  - Compute new attribute for non-terminal pushed onto stack.
Shift-Reduce Conflicts

- **Problem**: can’t decide whether the action for a state and input character should be **shift** or **reduce**
- Caused by ambiguity in grammar
- Usually caused by lack of associativity or precedence information in grammar
Example: $\langle \text{Sum}\rangle = 0 \mid 1 \mid (\langle \text{Sum}\rangle) \\
\mid \langle \text{Sum}\rangle + \langle \text{Sum}\rangle$

- $\circ 0 + 1 + 0$ shift
  -> $0 \circ + 1 + 0$ reduce
- $\langle \text{Sum}\rangle \circ + 1 + 0$ shift
  -> $\langle \text{Sum}\rangle + \circ 1 + 0$ shift
- $\langle \text{Sum}\rangle + 1 \circ + 0$ reduce
  -> $\langle \text{Sum}\rangle + \langle \text{Sum}\rangle \circ + 0$
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
- Again caused by ambiguity in grammar
- **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
- $a \bullet bc$
- shift
- $ab \bullet c$
- shift
- $abc \bullet$

Problem: reduce by $B ::= bc$ then by $S ::= aB$, or by $A ::= abc$ then $S ::= A$?
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 (*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:
  
  \[
  \{\text{Precondition}\} \, \text{Program} \, \{\text{Postcondition}\}
  \]

- Source of idea of *loop invariant*
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.
Natural Semantics

- Aka Structural Operational Semantics, aka “Big Step Semantics”
- Provide value for a program by rules and derivations, similar to type derivations
- Rule conclusions look like
  
  \[(C, m) \Downarrow m'\]
  
  or
  
  \[(E, m) \Downarrow v\]
Simple Imperative Programming Language

- $I \in \text{Identifiers}$
- $N \in \text{Numerals}$
- $B ::= \text{true} \mid \text{false} \mid B \& B \mid B \text{ or } B \mid \text{not } B \mid E < E \mid E = E$
- $E ::= N \mid I \mid E + E \mid E * E \mid E - E \mid - E$
- $C ::= \text{skip} \mid C;C \mid I ::= E \mid \text{if } B \text{ then } C \text{ else } C \text{ fi} \mid \text{while } B \text{ do } C \text{ od}$
Natural Semantics of Atomic Expressions

- Identifiers: \((I,m) \downarrow m(I)\)
- Numerals are values: \((N,m) \downarrow N\)
- Booleans: \((\text{true},m) \downarrow \text{true}\)
  \((\text{false},m) \downarrow \text{false}\)
Booleans:

\[
\begin{align*}
(B, m) &\downarrow \text{false} & (B, m) &\downarrow \text{true} & (B', m) &\downarrow b \\
(B & \land B', m) &\downarrow \text{false} & (B & \land B', m) &\downarrow b \\
(B, m) &\downarrow \text{true} & (B, m) &\downarrow \text{false} & (B', m) &\downarrow b \\
(B & \lor B', m) &\downarrow \text{true} & (B & \lor B', m) &\downarrow b \\
(B, m) &\downarrow \text{true} & (B, m) &\downarrow \text{false} & (B', m) &\downarrow b \\
(\neg B, m) &\downarrow \text{false} & (\neg B, m) &\downarrow \text{true}
\end{align*}
\]
Relations

\[(E, m) \Downarrow U \quad (E', m) \Downarrow V \quad U \sim V = b \]
\[(E \sim E', m) \Downarrow b \]

- By \( U \sim V = b \), we mean does (the meaning of) the relation \( \sim \) hold on the meaning of \( U \) and \( V \).
- May be specified by a mathematical expression/equation or rules matching \( U \) and \( V \).
Arithmetic Expressions

\[(E, m) \Downarrow U \quad (E', m) \Downarrow V \quad U \text{ op } V = N\]

\[(E \text{ op } E', m) \Downarrow N\]

where \(N\) is the specified value for \(U \text{ op } V\)
Commands

Skip: \[(\text{skip, } m) \downarrow m\]

Assignment: \[\frac{(E,m) \downarrow V}{(I:=E,m) \downarrow m[I <-- V]}\]

Sequencing: \[\frac{(C,m) \downarrow m'}{(C';m, m') \downarrow m''} \quad \frac{(C;C', m) \downarrow m''}{(C;C', m) \downarrow m''} \]
If Then Else Command

\[
(B,m) \Downarrow \text{true} \quad (C,m) \Downarrow m' \\
\text{(if } B \text{ then } C \text{ else } C' \text{ fi, } m) \Downarrow m'
\]

\[
(B,m) \Downarrow \text{false} \quad (C',m) \Downarrow m' \\
\text{(if } B \text{ then } C \text{ else } C' \text{ fi, } m) \Downarrow m'
\]
While Command

\[
(B, m) \Downarrow \text{false} \quad (\text{while } B \text{ do } C \text{ od}, m) \Downarrow m
\]

\[
(B, m) \Downarrow \text{true} \quad (C, m) \Downarrow m' \quad (\text{while } B \text{ do } C \text{ od}, m') \Downarrow m''
\]

\[
(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m'''
\]
Example: If Then Else Rule

(if x > 5 then y := 2 + 3 else y := 3 + 4 fi, 
{x -> 7}) ↓ ?
Example: If Then Else Rule

\[(x > 5, \{x -> 7\}) \downarrow ?\]

\[(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,}
\{x -> 7\}) \downarrow ?\]
Example: Arith Relation

\[ x > 7 \Rightarrow 5 \Downarrow \]
\[ (x, \{x \rightarrow 7\}) \Downarrow \]
\[ (x > 5, \{x \rightarrow 7\}) \Downarrow \]
\[ (\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \rightarrow 7\}) \Downarrow \]
Example: Identifier(s)

\[
\begin{align*}
7 &> 5 = \text{true} \\
(x, \{x -> 7\}) &\Downarrow 7 \quad (5, \{x -> 7\}) \Downarrow 5 \\
(x > 5, \{x -> 7\}) &\Downarrow ? \\
\hline
\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \\
\{x -> 7\}) &\Downarrow ?
\end{align*}
\]
Example: Arith Relation

\[
\begin{align*}
7 & > 5 = \text{true} \\
(x, \{x -> 7\}) & \downarrow 7 \quad (5, \{x -> 7\}) \downarrow 5 \\
(x & > 5, \{x -> 7\}) \downarrow \text{true} \\
\text{(if } x & > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \\
\{x -> 7\}) & \downarrow ?
\end{align*}
\]
Example: If Then Else Rule

\[ 7 > 5 = \text{true} \]

\[ (x, \{x \rightarrow 7\}) \downarrow 7 \quad (5, \{x \rightarrow 7\}) \downarrow 5 \]

\[ (x > 5, \{x \rightarrow 7\}) \downarrow \text{true} \]

\[ (\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \]
\[ \{x \rightarrow 7\}) \downarrow ? \]

\[ (y := 2 + 3, \{x \rightarrow 7\}) \downarrow ? \]
Example: Assignment

\[
\begin{align*}
7 > 5 &= \text{true} \\
(x, \{x -> 7\}) \Downarrow 7 &\quad (5, \{x -> 7\}) \Downarrow 5 \\
(x > 5, \{x -> 7\}) \Downarrow \text{true} &\quad (y := 2 + 3, \{x -> 7\}) \Downarrow ? \\
\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} &\quad \{x -> 7\}) \Downarrow ?
\end{align*}
\]
Example: Arith Op

\[
\begin{align*}
? + ? &= ? \\
(2, \{x \rightarrow 7\}) &\downarrow ? \\
7 > 5 &= \text{true} \\
\underbrace{(x, \{x \rightarrow 7\})}_{(x > 5, \{x \rightarrow 7\})} &\downarrow \text{true} \\
(2+3, \{x \rightarrow 7\}) &\downarrow ? \\
(y := 2 + 3, \{x \rightarrow 7\}) &\downarrow ? \\
\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \\
\{x \rightarrow 7\} &\downarrow ?
\end{align*}
\]
Example: Numerals

\[
\begin{align*}
2 + 3 &= 5 \\
(2, \{x \mapsto 7\}) \downarrow 2 & \quad (3, \{x \mapsto 7\}) \downarrow 3 \\
7 > 5 &= \text{true} \\
(\text{x, } \{x \mapsto 7\}) \downarrow 7 & \quad (5, \{x \mapsto 7\}) \downarrow 5 \\
(\text{x > 5, } \{x \mapsto 7\}) \downarrow \text{true} & \quad (\text{y := } 2 + 3, \{x \mapsto 7\}) \downarrow \text{?} \\
\text{if x > 5 then y := 2 + 3 else y := 3 + 4 fi, } \{x \mapsto 7\} & \downarrow \text{?}
\end{align*}
\]
Example: Arith Op

\[
\begin{align*}
2 + 3 &= 5 \\
(2, \{x \rightarrow 7\}) \downarrow 2 &\quad (3, \{x \rightarrow 7\}) \downarrow 3 \\
7 > 5 &= \text{true} \\
(x, \{x \rightarrow 7\}) \downarrow 7 &\quad (5, \{x \rightarrow 7\}) \downarrow 5 \\
(x > 5, \{x \rightarrow 7\}) \downarrow \text{true} &\quad (y := 2 + 3, \{x \rightarrow 7\}) \downarrow ? \\
\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi),} \\
\{x \rightarrow 7\}) \downarrow ?
\end{align*}
\]
Example: Assignment

\[
\begin{align*}
\text{Example: Assignment} \\
2 + 3 &= 5 \\
(2, \{x\rightarrow7\}) \downarrow 2 &\quad (3, \{x\rightarrow7\}) \downarrow 3 \\
7 &> 5 = \text{true} \\
(x, \{x\rightarrow7\}) \downarrow 7 &\quad (5, \{x\rightarrow7\}) \downarrow 5 \\
(x > 5, \{x \rightarrow 7\}) \downarrow \text{true} &\quad (y := 2 + 3, \{x\rightarrow7\}) \downarrow \{x\rightarrow 7, y\rightarrow5\} \\
\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \\
\{x \rightarrow 7\} &\downarrow ?
\end{align*}
\]
Example: If Then Else Rule

\[ 2 + 3 = 5 \]
\[ (2, \{x -> 7\}) \downarrow 2 \quad (3, \{x -> 7\}) \downarrow 3 \]

\[ 7 > 5 = \text{true} \]
\[ (x, \{x -> 7\}) \downarrow 7 \quad (5, \{x -> 7\}) \downarrow 5 \]

\[ (x > 5, \{x -> 7\}) \downarrow \text{true} \]
\[ (\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \]
\[ \{x -> 7\}) \downarrow \{x -> 7, y -> 5\} \]

\[ (y := 2 + 3, \{x -> 7\}) \downarrow 5 \]
\[ (2 + 3, \{x -> 7\}) \downarrow 5 \]
Let in Command

\[
(E, m) \downarrow \nu \ (C, m[I<-\nu]) \downarrow m'
\]

(let \( I = E \) in \( C, m \)) \downarrow m'''

Where \( m''' (y) = m' (y) \) for \( y \neq I \) and \( m''' (I) = m (I) \) if \( m(I) \) is defined, and \( m''' (I) \) is undefined otherwise.
Example

\[(x, \{x -> 5\}) \downarrow 5 \quad (3, \{x -> 5\}) \downarrow 3\]

\[(x+3, \{x -> 5\}) \downarrow 8\]

\[(5, \{x -> 17\}) \downarrow 5 \quad (x := x+3, \{x -> 5\}) \downarrow \{x -> 8\}\]

\[(\text{let } x = 5 \text{ in } (x := x+3), \{x -> 17\}) \downarrow ?\]
Example

\[
(x, \{x \rightarrow 5\}) \downarrow 5 \quad (3, \{x \rightarrow 5\}) \downarrow 3 \\
(x+3, \{x \rightarrow 5\}) \downarrow 8 \\
(5, \{x \rightarrow 17\}) \downarrow 5 \quad (x := x+3, \{x \rightarrow 5\}) \downarrow \{x \rightarrow 8\} \\
(let \ x = 5 \ in \ (x := x+3), \ \{x \rightarrow 17\}) \downarrow \{x \rightarrow 17\}
\]
Comment

- Simple Imperative Programming Language introduces variables *implicitly* through assignment
- The let-in command introduces scoped variables *explicitly*
- Clash of constructs apparent in awkward semantics
A compiler from language L1 to language L2 is a program that takes an L1 program and for each piece of code in L1 generates a piece of code in L2 of same meaning.

An interpreter of L1 in L2 is an L2 program that executes the meaning of a given L1 program.

Compiler would examine the body of a loop once; an interpreter would examine it every time the loop was executed.
An *Interpreter* represents the operational semantics of a language L1 (source language) in the language of implementation L2 (target language).

Built incrementally
- Start with literals
- Variables
- Primitive operations
- Evaluation of expressions
- Evaluation of commands/declarations
Interpreter

- Takes abstract syntax trees as input
  - In simple cases could be just strings
- One procedure for each syntactic category (nonterminal)
  - eg one for expressions, another for commands
- If Natural semantics used, tells how to compute final value from code
- If Transition semantics used, tells how to compute next “state”
  - To get final value, put in a loop
Natural Semantics Example

- compute_exp (Var(v), m) = look_up v m
- compute_exp (Int(n), _) = Num (n)
- ...
- compute_com(IfExp(b,c1,c2),m) =
  if compute_exp (b,m) = Bool(true)
  then compute_com (c1,m)
  else compute_com (c2,m)
Natural Semantics Example

- compute_com(While(b,c), m) =
  if compute_exp (b,m) = Bool(false)
  then m
  else compute_com
      (While(b,c), compute_com(c,m))

- May fail to terminate - exceed stack limits
- Returns no useful information then