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 **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 **state**(*n*), look up action in Action table at (*n*, *toks*)

LR(i) Parsing Algorithm

5. If action = **shift** *m*,
   a) Remove the top token from token stream and push it onto the stack
   b) Push **state**(*m*) onto stack
   c) Go to step 3

LR(i) Parsing Algorithm

6. If action = **reduce** *k* where production *k* is E ::= u
   a) Remove 2 * length(u) symbols from stack (u and all the interleaved states)
   b) If new top symbol on stack is **state**(*m*), look up new state *p* in Goto(*m*,E)
   c) Push E onto the stack, then push **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: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

● 0 + 1 + 0	shift
-> 0 ● + 1 + 0	reduce
-> <Sum> ● + 1 + 0	shift
-> <Sum> + ● 1 + 0	shift
-> <Sum> + 1 ● + 0	reduce
-> <Sum> + <Sum> ● + 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 ● bc   shift
ab ● c   shift
abc ●

- 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 (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

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

- $I \in$ *Identifiers*
- $N \in$ *Numerals*
- *B* ::= true | false | *B* & *B* | *B* or *B* | not *B* | *E* < *E* | *E* = *E*
- *E*::= *N* | *I* | *E* + *E* | *E* * *E* | *E* - *E* | - *E*
- *C*::= skip | *C*;*C* | *I* ::= *E* | if *B* then *C* else *C* fi | while *B* do *C* 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:

$$\frac{(B, m) \Downarrow \text{false}}{(B \,\&\, B', m) \Downarrow \text{false}} \qquad \frac{(B, m) \Downarrow \text{true} \quad (B', m) \Downarrow b}{(B \,\&\, B', m) \Downarrow b}$$

$$\frac{(B, m) \Downarrow \text{true}}{(B \text{ or } B', m) \Downarrow \text{true}} \qquad \frac{(B, m) \Downarrow \text{false} \quad (B', m) \Downarrow b}{(B \text{ or } B', m) \Downarrow b}$$

$$\frac{(B, m) \Downarrow \text{true}}{(\text{not } B, m) \Downarrow \text{false}} \qquad \frac{(B, m) \Downarrow \text{false}}{(\text{not } B, m) \Downarrow \text{true}}$$

Relations

$$\frac{(E, m) \Downarrow U \quad (E', m) \Downarrow V \quad U \sim V = b}{(E \sim E', m) \Downarrow b}$$

- 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

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

where N is the specified value for U op V

Commands

Skip: $(\text{skip}, m) \Downarrow m$

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

Sequencing: $$\frac{(C,m) \Downarrow m' \quad (C',m') \Downarrow m''}{(C;C', m) \Downarrow m''}$$

If Then Else Command

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

$$\frac{(B,m) \Downarrow \text{false} \quad (C',m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$$

While Command

$$\frac{(B,m) \Downarrow \text{false}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m}$$

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

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})⇓?

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7}) ⇓ ?

Example: Arith Relation

? > ? = ?

(x,{x->7})⇓? (5,{x->7})⇓?

(x > 5, {x -> 7})⇓?

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7}) ⇓ ?

Example: Identifier(s)

7 > 5 = true

(x,{x->7})⇓7 (5,{x->7})⇓5

(x > 5, {x -> 7})⇓?

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7}) ⇓ ?

Example: Arith Relation

7 > 5 = true

(x,{x->7})⇓7 (5,{x->7})⇓5

(x > 5, {x -> 7})⇓true

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7}) ⇓ ?

Example: If Then Else Rule

7 > 5 = true

(x,{x->7})⇓7 (5,{x->7})⇓5

(x > 5, {x -> 7})⇓true (y:= 2 + 3, {x-> 7}
⇓ ? .

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7}) ⇓ ?

Example: Assignment

7 > 5 = true

(x,{x->7})⇓7 (5,{x->7})⇓5

(x > 5, {x -> 7})⇓true

(2+3, {x->7})⇓?

(y:= 2 + 3, {x-> 7} ⇓ ?

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi, {x -> 7}) ⇓ ?

Example: Arith Op

? + ? = ?

(2,{x->7})⇓? (3,{x->7}) ⇓?

(2+3, {x->7})⇓?

7 > 5 = true

(x,{x->7})⇓7 (5,{x->7})⇓5

(x > 5, {x -> 7})⇓true

(y:= 2 + 3, {x-> 7} ⇓? .

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi, {x -> 7}) ⇓ ?

Example: Numerals

2 + 3 = 5

(2,{x->7})⇓2 (3,{x->7}) ⇓3

(2+3, {x->7})⇓?

7 > 5 = true

(x,{x->7})⇓7 (5,{x->7})⇓5

(x > 5, {x -> 7})⇓true

(y:= 2 + 3, {x-> 7} ⇓ ?

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi, {x -> 7}) ⇓ ?

Example: Arith Op

2 + 3 = 5

(2,{x->7})⇓2 (3,{x->7}) ⇓3

(2+3, {x->7})⇓5

7 > 5 = true

(x,{x->7})⇓7 (5,{x->7})⇓5

(x > 5, {x -> 7})⇓true

(y:= 2 + 3, {x-> 7} ⇓?

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi, {x -> 7}) ⇓ ?

Example: Assignment

2 + 3 = 5

(2,{x->7})⇓2 (3,{x->7}) ⇓3

(2+3, {x->7})⇓5

7 > 5 = true

(x,{x->7})⇓7 (5,{x->7})⇓5

(x > 5, {x -> 7})⇓true

(y:= 2 + 3, {x-> 7} ⇓ {x->7, y->5}

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi, {x -> 7}) ⇓?

Example: If Then Else Rule

2 + 3 = 5

(2,{x->7})⇓2 (3,{x->7}) ⇓3

(2+3, {x->7})⇓5

7 > 5 = true

(x,{x->7})⇓7 (5,{x->7})⇓5

(x > 5, {x -> 7})⇓true

(y:= 2 + 3, {x-> 7} ⇓ {x->7, y->5}

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi, {x -> 7}) ⇓ {x->7, y->5}

Let in Command

$$\frac{(E,m) \Downarrow v \quad (C, m[I<-v]) \Downarrow m'}{(\text{let } I = E \text{ in } C,\ m) \Downarrow m''}$$

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

$$\frac{(5,\{x\text{->}17\}) \Downarrow 5 \qquad \dfrac{\dfrac{(x,\{x\text{->}5\}) \Downarrow 5 \quad (3,\{x\text{->}5\}) \Downarrow 3}{(x+3,\{x\text{->}5\}) \Downarrow 8}}{(x:=x+3,\{x\text{->}5\}) \Downarrow \{x\text{->}8\}}}{(\text{let } x = 5 \text{ in } (x:=x+3),\ \{x \text{ -> } 17\}) \Downarrow ?}$$

Example

$$\frac{(5,\{x\text{->}17\}) \Downarrow 5 \qquad \dfrac{\dfrac{(x,\{x\text{->}5\}) \Downarrow 5 \quad (3,\{x\text{->}5\}) \Downarrow 3}{(x+3,\{x\text{->}5\}) \Downarrow 8}}{(x:=x+3,\{x\text{->}5\}) \Downarrow \{x\text{->}8\}}}{(\text{let } x = 5 \text{ in } (x:=x+3),\ \{x \text{ -> } 17\}) \Downarrow \{x\text{->}17\}}$$

Comment

- Simple Imperative Programming Language introduces variables *implicitly* through assignment
- The let-in command introduces scoped variables *explictly*
- Clash of constructs apparent in awkward semantics

Interpretation Versus Compilation

- 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

Interpreter

- 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