CSC 425 - Principles of Compiler Design I

Introduction to Bottom-Up Parsing

Outline

- Review *LL* parsing
- Shift-reduce parsing
- The *LR* parsing algorithm
- Constructing *LR* parsing tables

Top-Down Parsing: Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal
- The leaves at any point form a string $\beta A \gamma$
  - β contains only terminals
  - The input string is $\beta b \delta$
  - The prefix β matches (is valid)
  - The next token is b

Predictive Parsing: Review

- A predictive parser is described by a table
  - For each non-terminal A and for each token b we specify a production $A \to \alpha$
  - When trying to expand A we use $A \to \alpha$ if b follows next
- Once we have the table:
  - The parsing algorithm is simple and fast
  - No backtracking is necessary

Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing
  - and just as efficient
  - builds on ideas in top-down parsing
  - preferred method in practice
- Also called *LR* parsing
  - *L* means that tokens are read left-to-right
  - *R* means that it constructs a rightmost derivation

An Introductory Example

- LR parsers do not need left-factored grammars and can also handle left-recursive grammars
- Consider the following grammar:

$$E \rightarrow E + (E) \mid int$$

- This is not $LL(1)$
- Consider the string: $int + (int) + (int)$

The Idea

- *LR* parsing reduces a string to the start symbol by inverting productions
- Given a string of terminals:
  1. Identify β in the string such that $A \to \beta$ is a production
  2. Replace β by A in the string
  3. Repeat steps 1 and 2 until the string is the start symbol (or all possibilities are exhausted)

Bottom-up Parsing Example

- Consider the following grammar:

$$E \to E + (E) \mid int$$

- And input string: $int + (int) + (int)$
- Bottom-up parse:
  1. $int + (int) + (int)$
  2. $E + (int) + (int)$
  3. $E + (E) + (int)$
  4. $E + (int)$
  5. $E + (E)$
  6. E
- A rightmost derivation in reverse

Reductions

- An *LR* parser traces a rightmost derivation in reverse
- This has an interesting consequence
  - Let $\alpha\beta\gamma$ be a step of a bottom-up parse
  - Assume the next reduction is by using $A \rightarrow \beta$
  - The γ is a string of terminals
  - This is because $\alpha A\gamma \rightarrow \alpha\beta\gamma$ is a step in a rightmost derivation

Notation

- Idea: split a string into two substrings
  - the right substring is the partition that has not been examined yet
  - the left substring has terminals and non-terminals
- The dividing point is marked by a $|$
- Initially, all input is unexamined: $|x_1, x_2 \ldots x_n$

Shift-Reduce Parsing

- Bottom-up parsing uses only two kinds of actions: shift and reduce
- Shift: move $|$ one place to the right

$$E + (|int) \rightarrow E + (int|)$$

- Reduce: apply an inverse production at the right end of the left string
  - If $E \rightarrow E + (E)$ is a production, then

$$E + (\underline{E + (E)}|) \rightarrow E + (\underline{E}|)$$

Shift-Reduce Example

- Consider the grammar: $E \to E + (E) \mid int$

String	Action
$\|int + (int) + (int)\$$	shift
$int\| + (int) + (int)\$$	reduce $E \to int$
$E\| + (int) + (int)\$$	shift three times
$E + (int\|) + (int)\$$	reduce $E \to int$
$E + (E\|) + (int)\$$	shift
$E + (E)\| + (int)\$$	reduce $E \to E + (E)$
$E\| + (int)\$$	shift three times
$E + (int\|)\$$	reduce $E \to int$
$E + (E\|)\$$	shift
$E + (E)\|\$$	reduce $E \to E + (E)$
$E\|\$$	accept

The Stack

- The left string can be implemented by a stack
  - The top of the stack is the |
- Shift pushes a terminal on the stack
- Reduce pops zero or more symbols off of the stack (production right hand side) and pushes a non-terminal on the stack (production left hand side).

Question: To Shift or Reduce

- Idea: use a finite automaton (DFA) to decide when to shift or reduce
  - The input is the stack
  - The language consists of terminals and non-terminals
- We run the DFA on the stack and examine the resulting state X and token t after $|$
  - If X has a transition labeled t then shift
  - If X is labeled with "$A \rightarrow \beta$ on t" then reduce

$LR(1)$ DFA Example

- Transitions:
  - $0 \rightarrow 1$ on *int*
  - $0 \rightarrow 2$ on E
  - $2 \rightarrow 3$ on $+$
  - $3 \rightarrow 4$ on (
  - $4 \rightarrow 5$ on *int*
  - $4 \rightarrow 6$ on E
  - $6 \rightarrow 7$ on)
  - $6 \rightarrow 8$ on $+$
  - $8 \rightarrow 9$ on (
  - $9 \rightarrow 5$ on *int*
  - $9 \rightarrow 10$ on E
  - $10 \rightarrow 8$ on $+$
  - $10 \rightarrow 11$ on)
- States with actions:
  - 1: $E \rightarrow \mathit{int}$ on $\$, +$
  - 2: accept on \$
  - 5: $E \rightarrow \mathit{int}$ on $), +$
  - 7: $E \rightarrow E + (E)$ on $\$, +$
  - 11: $E \rightarrow E + (E)$ on $), +$

Representing the DFA

- Parsers represent the DFA as a 2D table similar to table-driven lexical analysis
- Rows correspond to DFA states
- Columns correspond to terminals and non-terminals
- Columns are typically split into:
  - terminals: action table
  - non-terminals: goto table

Representing the DFA Example

	int	$+$	(	)	\$	E
0	s1					g2
1		r($E \rightarrow int$)			r($E \rightarrow int$)	
2		s3				accept
3			s4			
4	s5					g6
5		r($E \rightarrow int$)		r($E \rightarrow int$)		
6	s8		s7			
7		r($E \rightarrow E + (E)$)			r($E \rightarrow E + (E)$)	
8			s9			
9	s5					g10
10		s8		s11		
11		r($E \rightarrow E + (E)$)		r($E \rightarrow E + (E)$)		

The LR Parsing Algorithm

- After a shift or reduce action we rerun the DFA on the entire stack
  - This is wasteful, since most of the work is repeated
- For each stack element remember which state it transitions to in the DFA
- The LR parser maintains a stack

$$\langle sym_1, state_1 \rangle \ldots \langle sym_n, state_n \rangle$$

where $state_k$ is the final state of the DFA on $sym_1 \ldots sym_k$

The *LR* Parsing Algorithm

```
let I = w$ be the initial input
let j = 0
let DFA state 0 be the start state
let stack = <dummy, 0>
repeat
  case action[top_state(stack), I[j]] of
    shift k: push <I[j++], k>
    reduce X -> A:
      pop |A| pairs
      push <X, goto[top_state(stack), X]>
    accept: halt normally
    error: halt and report error
```

LR Parsers

- Can be used to parse more grammars than *LL*
- Most programming languages are *LR*
- *LR* parsers can be described as a simple table
- There are tools for building the table
- Open question: how is the table constructed?

Key Issue: How is the DFA Constructed?

- The stack describes the context of the parse
  - What non-terminal we are looking for
  - What production right hand side we are looking for
  - What we have seen so far from the right hand side
- Each DFA state describes several such contexts
  - Example: when we are looking for non-terminal E, we might be looking either for an *int* of an $E + (E)$ right hand side

$LR(0)$ Items

- An $LR(0)$ item is a production with a "$|$" somewhere on the right hand side
- The items for $T \to (E)$ are:
  - $T \to |(E)$
  - $T \to (|E)$
  - $T \to (E|)$
  - $T \to (E)|$
- The only item for $X \to \epsilon$ is $X \to |$

$LR(0)$ Items: Intuition

- An item $\langle X \to \alpha|\beta\rangle$ says that
  - the parser is looking for an X
  - it has an α on top of stack
  - expects to finr a string derived from β next in the input
- Notes
  - $\langle X \to \alpha|a\beta\rangle$ means that a should follow – then we can shift it and still have a viable prefix
  - $\langle X \to \alpha|\rangle$ means that we could reduce X – but this is not always a good idea

$LR(1)$ Items

- An $LR(1)$ item is a pair:

  $$\langle X \to \alpha|\beta, a\rangle$$

  - $X \to \alpha\beta$ is a production
  - a is a terminal (the lookahead terminal)
  - $LR(1)$ means one lookahead terminal
- $\langle X \to \alpha|\beta, a\rangle$ describes a context of the parser
  - We are trying to find an X followed by an a, and
  - We have (at least) α already on top of the stack
  - Thus, we need to see a prefix derived from βa

Note

- The symbol $|$ was used before to separate the stack from the rest of the input.
  - $\alpha|\gamma$, where α is the stack and γ is the remaining string of terminals
- In items $|$ is used to mark a prefix of a production right hand side:

$$\langle X \rightarrow \alpha|\beta, a\rangle$$

  - Here β might contain terminals as well
- In both cases, the stack is on the left of $|$

Convention

- We add to our grammar a fresh new start symbol S and a production $S \to E$ where E is the old start symbol
- The initial parsing context contains:

$$\langle S \to |E, \$\rangle$$

  - Trying to find an S as a string dervied from $E\$$
  - The stack is empty

$LR(1)$ Items Continued

- In context containing

  $$\langle E \to E + |(E), +\rangle$$

  If "(" follows then we can perform a shift to context containing

  $$\langle E \to E + (|E), +\rangle$$

- In context containing

  $$\langle E \to E + (E)|, +\rangle$$

  We can perform a reduction with $E \to E + (E)$, but only if a "$+$" follows

$LR(1)$ Items Continued

- Consider the item

$$\langle E \to E + (|E), +\rangle$$

- We expect a string derived from $E)+$
- There are two productions for E
  - $E \to int$
  - $E \to E + (E)$
- We describe this by extending the context with two more items:
  - $\langle E \to |int,)\rangle$
  - $\langle E \to |E + (E),)\rangle$

The Closure Operation

- The operation of extending the context with items is called the closure operation

```
Closure(Items) =
  repeat
    for each [X -> alpha | Y beta, a] in Items
      for each production Y -> gamma
        for each b in First(beta a)
          add [Y -> | gamma, b] to Items
  until Items is unchanged
```

Constructing the Parsing DFA (1)

- Construct the start context: $\mathit{Closure}(\{S \to E, \$\})$
  - $\langle S \to |E, \$\rangle$
  - $\langle E \to |E + (E), \$\rangle$
  - $\langle E \to |\mathit{int}, \$\rangle$
  - $\langle E \to |E + (E), +\rangle$
  - $\langle E \to |\mathit{int}, +\rangle$
- We abbreviate as:
  - $\langle S \to |E, \$\rangle$
  - $\langle E \to |E + (E), \$/+\rangle$
  - $\langle E \to |\mathit{int}, \$/+\rangle$

Constructing the Parsing DFA (2)

- A DFA state is a closed set of $LR(1)$ items
- The start state contains $\langle S \to |E, \$\rangle$
- A state that contains $\langle X \to \alpha | b\rangle$ is labelled with "reduce with $X \to \alpha$ on b"

The DFA Transitions

- A state "State" that contains $\langle X \to \alpha | y\beta, b\rangle$ has a transition labeled y to a state that contains the items "Transition(State,y)" where y can be a terminal or non-terminal

```
Transition(State, y) =
  Items = empty set
  for each [X -> alpha | y beta, a] in State
      add [X -> alpha y | beta, b] to Items
  return Closure(Items)
```

LR Parsing Tables: Notes

- Parsing tables (DFA) can be constructed automatically for a CFG
- But, we still need to understand the construction to work with parser generators
- What kinds of errors can we expect?

Shift/Reduce Conflicts

- If a DFA state contains both

  $$\langle X \to \alpha|a\beta, b\rangle$$

  and

  $$\langle Y \to \gamma|, a\rangle$$

- Then on input "a" we could either
  - Shift into state $\langle X \to \alpha a|\beta, b\rangle$
  - Reduce with $Y \to \gamma$
- This is called a shift-reduce conflict

Shift/Reduce Conflicts

- Typically due to ambiguities in the grammar
- Classic example: the dangling else

$$S \to if\ E\ then\ S \mid if\ E\ then\ S\ else\ S \mid OTHER$$

- Will have a DFA state containing
  - $\langle S \to if\ E\ then\ S|, else\rangle$
  - $\langle S \to if\ E\ then\ S|\ else\ S, x\rangle$
- If *else* follows then we can shift or reduce
- The default behavior of tools is to shift

More Shift/Reduce Conflicts

- Consider the ambiguous grammar

$$E \rightarrow E + E \mid E * E \mid int$$

- We will have the states containing
  - $\langle E \rightarrow E * | E, + \rangle \Rightarrow \langle E \rightarrow E * E |, + \rangle$
  - $\langle E \rightarrow | E + E, + \rangle \Rightarrow \langle E \rightarrow E | + E, + \rangle$
  - $\ldots$
- Again we have a shift/reduce on input $+$
  - We need to reduce ($*$ binds tighter than $+$)
  - Recall solution: declare the precedence of $*$ and $+$

More Shift/Reduce Conflicts

- In `yacc` we can declare precedence and associativity

```
%left +
%left *
```

- Precedence of a rule equals that of its last terminal
- Resolve shift/reduce conflict with a shift if:
  - no precedence declared for either rule or terminal
  - input terminal has a higher precedence than the rule
  - the precedences are the same and right associative

Using Precedence to Resolve Shift/Reduce Conflicts

- Back to the example
  - $\langle E \to E * | E, + \rangle \Rightarrow \langle E \to E * E |, + \rangle$
  - $\langle E \to | E + E, + \rangle \Rightarrow \langle E \to E | + E, + \rangle$
  - $\ldots$
- Will choose reduce because precedence of rule $E \to E * E$ is higher than of terminal $+$

Using Precedence to Resolve Shift/Reduce Conflicts

- Another example
  - $\langle E \to E + |E, +\rangle \Rightarrow \langle E \to E + E|, +\rangle$
  - $\langle E \to |E + E, +\rangle \Rightarrow \langle E \to E| + E, +\rangle$
  - $\ldots$
- Now we have a shift/reduce on input $+$: we choose redue because $E \to E + E$ and $+$ have the same precedence and $+$ is left associative

Precedence Declarations Revisited

- The phrase *precedence declaration* is misleading
- These declarations do not define precedence, they define conflict resolutions
- That is, they instruct shift-reduce parsers to resolve conflicts in certain ways – that is not quite the same thing as precedence

Reduce/Reduce Conflicts

- If a DFA state contains both

  $$\langle X \to \alpha|, a\rangle$$

  and

  $$\langle Y \to \beta|, a\rangle$$

  then on "a" we don not know which production to reduce
- This is called a reduce/reduce conflict

Reduce/Reduce Conflicts

- Usually due to gross ambiguity in the grammar
- Example:

$$S \to \epsilon \mid id \mid id\ S$$

- There are two parse trees for the string *id*
- This grammar is better if we rewrite it as

$$S \to \epsilon \mid id\ S$$

Using Parser Generators

- A parser generator automatically contructs the parsing DFA given a context free grammar
  - Use precedence declarations and default conventions to resolve conflicts
  - The parser algorithm is the same for all grammars
- But, most parser generators do not construct the DFA as described before because the $LR(1)$ parsing DFA has thousands of states for even simple languages

$LR(1)$ Parsing Tables are Big

- But, many states are similar:

$$\langle E \rightarrow int|, \$/+\rangle \text{ and } \langle E \rightarrow int|,)/+\rangle$$

- Idea: merge the DFA states whose items differ only in the lookahead tokens
- We say that that such states have the same core
- In this example, we obtain

$$\langle E \rightarrow int|, \$/+/)\rangle$$

The Core of a Set of LR Items

- Definition: The core of a set of LR items is the set of first components without the lookahead terminals
- Example: the core of

  $$\{\langle X \to \alpha|\beta, b\rangle, \langle Y \to \gamma|\delta, d\rangle\}$$

  is

  $$\{X \to \alpha|\beta, Y \to \gamma|\delta\}$$

LALR States

- Consider for example the $LR(1)$ states
- Example: the core of

$$\{\langle X \to \alpha|, a\rangle, \langle Y \to \beta|, c\rangle\}$$
$$\{\langle X \to \alpha|, b\rangle, \langle Y \to \beta|, d\rangle\}$$

- They have the same core and can be merged
- The merged state contains:

$$\{\langle X \to \alpha|, a/b\rangle, \langle Y \to \beta|, c/d\rangle\}$$

- These are called $LALR(1)$ states
  - Stands for LookAhead LR
  - Typically 10 times fewer $LALR(1)$ states than $LR(1)$

A $LALR(1)$ DFA

- Repeat until all states have a distinct core
  - Choose two distinct states with the same core
  - Merge the states by creating a new one with the union of all the items
  - Point edges from the predecessors to the new state
  - New state points to all previous states

The *LALR* Parser Can Have Conflicts

- Consider for example the $LR(1)$ states

$$\{\langle X \rightarrow \alpha|, a\rangle, \langle Y \rightarrow \beta|, b\rangle\}$$
$$\{\langle X \rightarrow \alpha|, b\rangle, \langle Y \rightarrow \beta|, a\rangle\}$$

- And the merged $LALR(1)$ state

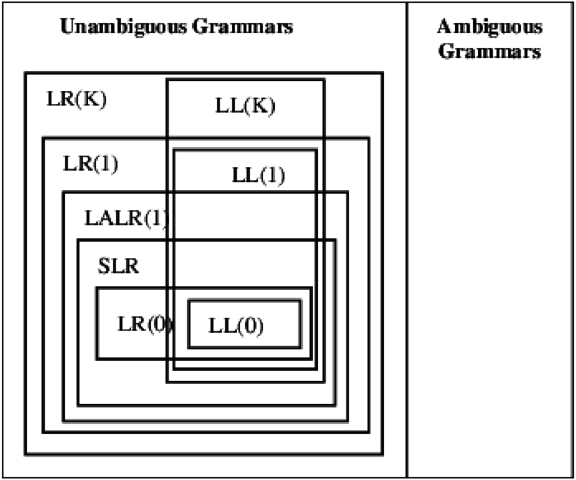
$$\{\langle X \rightarrow \alpha|, a/b\rangle, \langle Y \rightarrow \beta|, a/b\rangle\}$$

- Has a new reduce/reduce conflict
- In practice such cases are rare

LALR versus *LR* Parsing

- *LALR* languages are not natural; they are an efficiency hack on *LR* languages
- Most reasonable programming languages has an *LALR*(1) grammar
- *LALR*(1) parsing has become a standard for programming languages and for parser generators.

A Hierarchy of Grammar Classes



Semantic Actions in LR Parsing

- We can now illustrate how semantic actions are implemented for LR parsing
- Keep attributes on the stack:
  - On shifting a, push the attribute for a on the stack
  - On reduce $X \to \alpha$
    1. pop attributes for α
    2. compute attribute for X
    3. push it on the stack

Performing Semantic Actions: Example

- Recall the example

$$
\begin{aligned}
E &\rightarrow T + E_1 && \{E.val = T.val + E_1.val\} \\
&\mid T && \{E.val = T.val\} \\
T &\rightarrow int * T_1 && \{T.val = int.val + T_1.val\} \\
&\mid int && \{T.val = int.val\}
\end{aligned}
$$

- Consider parsing the string: $4 * 9 + 6$

Performing Semantic Actions: Example

- Recall the example

$$
\begin{aligned}
E &\rightarrow T + E_1 && \{E.val = T.val + E_1.val\} \\
&\mid T && \{E.val = T.val\} \\
T &\rightarrow int * T_1 && \{T.val = int.val + T_1.val\} \\
&\mid int && \{T.val = int.val\}
\end{aligned}
$$

- Consider parsing the string: $4 * 9 + 6$

Performing Semantic Actions: Example

String	Action
$\|int * int + int$	shift
$int(4)\| * int + int\$$	shift
$int(4) * \|int + int\$$	shift
$int(4) * int(9)\| + int\$$	reduce $T \rightarrow int$
$int(4) * T(9)\| + int\$$	reduce $T \rightarrow int * T$
$T(36)\| + int\$$	shift
$T(36) + \|int\$$	shift
$T(36) + int(6)\|\$$	reduce $T \rightarrow int$
$T(36) + T(6)\|\$$	reduce $E \rightarrow T$
$T(36) + E(6)\|\$$	reduce $E \rightarrow T + E$
$E(42)\|\$$	accept

Notes on Parsing

- Parsing
  - A solid foundation: context-free grammars
  - A simple parser: $LL(1)$
  - A more powerful parser: $LR(1)$
  - An efficiency hack: $LALR(1)$
  - $LALR(1)$ parser generators