Compiler Design

Lecture-15

Introduction to Bottom-Up Parsing

Topics Covered

Bottom-Up Parsing

Constructing an SLR Parsing Table

Part II
Bottom-Up Parsing

- There are different approaches to bottom-up parsing. One of them is called *Shift-Reduce parsing*, which in turns has a number of different instantiations.

- *Operator-precedence parsing* is one such method as is *LR parsing* which is much more general.

- In this course, we will be focusing on LR parsing. LR Parsing itself takes three forms: *Simple LR-Parsing (SLR)* a simple but limited version of LR-Parsing; *Canonical LR parsing*, the most powerful, but most expensive version; and *LALR* which is intermediate in cost and

LR Parsing: Advantages

- LR Parsers can recognize any language for which a context free grammar can be written.
- LR Parsing is the most general non-backtracking shift-reduce method known, yet it is as efficient as ither shift-reduce approaches
- The class of grammars that can be parsed by an LR parser is a proper superset of that that can be parsed by a predictive parser.
- An LR-parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

LR-Parsing: Drawback/Solution

- The main drawback of LR parsing is that it is too much work to construct an LR parser by hand for a typical programming language grammar.
- Fortunately, specialized tools to construct LR parsers automatically have been designed.
- With such tools, a user can write a context-free grammar and have a parser generator automatically produce a parser for that grammar.
- An example of such a tool is Yacc “Yet Another Compiler-Compiler”

LR Parsing Algorithms: Details I

- An LR parser consists of an input, output, a stack, a driver program and a parsing table that has two parts: action and goto.
- The driver program is the same for all LR Parsers. Only the parsing table changes from one parser to the other.
- The program uses the stack to store a string of the form $s_0X_1s_1X_2\ldots X_ms_m$, where s_m is the top of the stack. The S_k's are state symbols while the X_i's are grammar symbols. Together state and grammar symbols determine a shift-reduce parsing decision.

LR Parsing Algorithms: Details II

- The parsing table consists of two parts: a parsing *action* function and a *goto* function.
- The LR parsing program determines sm, the state on top of the stack and a_i, the current input. It then consults action[s_m, a_i] which can take one of four values:
    - Shift
    - Reduce
    - Accept
    - Error

LR Parsing Algorithms: Details III

- If action[s_m, a_i] = Shift s, where s is a state, then the parser pushes a_i and s on the stack.
- If action[s_m, a_i] = Reduce A → β, then a_i and s_m are replaced by A, and, if s was the state appearing below a_i in the stack, then goto[s, A] is consulted and the state it stores is pushed onto the stack.
- If action[s_m, a_i] = Accept, parsing is completed
- If action[s_m, a_i] = Error, then the parser discovered an error.

LR Parsing Example: The Grammar

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

LR-Parser Example: The Parsing Table

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				Acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		R1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

LR-Parser Example: Parsing Trace

Stack	Input	Action
(1) 0	id * id + id $	Shift
(2) 0 id 5	* id + id $	Reduce by F → id
(3) 0 F 3	* id + id $	Reduce by T → F
(4) 0 T 2	* id + id $	Shift
(5) 0 T 2 * 7	id + id $	Shift
(6) 0 T 2 * 7 id 5	+ id $	Reduce by F → id
(7) 0 T 2 * 7 F 10	+ id $	Reduce by T → T * F
(8) 0 T 2	+ id $	Reduce by E →T
(9) 0 E 1	+ id $	Shift
(10) 0 E 1 + 6	id $	Shift
(11) 0 E 1 + 6 id 5	$	Reduce by F → id
(12) 0 E 1 + 6 F 3	$	Reduce by T → F
(13) 0 E 1 + 6 T 9	$	E → E + T
(14) 0 E 1	$	Accept

SLR Parsing

- ***Definition:*** An LR(0) item of a grammar G is a production of G with a dot at some position of the right side.
- ***Example:*** A → XYZ yields the four following items:
  - A → .XYZ
  - A → X.YZ
  - A → XY.Z
  - A → XYZ.
- The production A → ϵ generates only one item, A → .
- Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process.

SLR Parsing

- To create an SLR Parsing table, we define three new elements:
  - An augmented grammar for G, the initial grammar. If S is the start symbol of G, we add the production S' → .S . The purpose of this new starting production is to indicate to the parser when it should stop parsing and accept the input.
  - The closure operation
  - The goto function

SLR Parsing: The Closure Operation

- If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:
  1. Initially, every item in I is added to closure(I)
  2. If A → α . B β is in closure(I) and B → γ is a production, then add the item B → . γ to I, if it is not already there. We apply this rule until no more new items can be added to closure(I).

SLR Parsing: The Closure Operation – Example

Original grammar

- E → E + T
- E → T
- T → T * F
- T → F
- F → (E)
- F → id

Augmented grammar

0. E’ → E
1. E → E + T
2. E → T
3. E → T * F
4. T → F
5. F → (E)
6. F → id

Let I = {[E’ → E]} then

Closure(I)=
{ [E’ → .E], [E → .E + T],
[E → .T], [E → .T*F],
[T → .F], [F → .(E)]
[F → .id] }

SLR Parsing: The Goto Operation

- Goto(I,X), where I is a set of items and X is a grammar symbol, is defined as the closure of the set of all items [A → αX.β] such that [A → α.Xβ] is in I.
- Example: If I is the set of two items {E' → E.], [E → E.+T]}, then goto(I, +) consists of

E → E + .T
T → .T * F
T → .F
F → .(E)
F → .id

SLR Parsing:
Sets-of-Items Construction

Procedure items(G')

C = {Closure({[S' → .S]})}

Repeat

For each set of items I in C and each grammar symbol X such that got(I,X) is not empty and not in C do

add goto(I,X) to C

Until no more sets of items can be added to C

Example: The Canonical LR(0) collection for grammar G

I0: E' → .E
E → .E + T
E → .T
T → .T * F
T → .F
F → .(E)
F → .id

I1: E' → E.
E → E.+T

I2: E → T.
T → T. * F

I3: T → F.

I4: F → (.E)
E → .E + T
E → .T
T → .T * F
T → .F
F → .(E)
F → .id

I5: F → id.

I6: E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id

I7: T → T * .F
F → .(E)
F → .id

I8: F → (E.)
E → E.+T

I9: E → E + T.
T → T.* F

I10: T → T*F.

I11: F → (E).

Constructing an SLR Parsing Table

1. Construct $C=\{I_0, I_1, \ldots I_n\}$ the collection of sets of LR(0) items for G'
2. State i is constructed from I_i. The parsing actions for state i are determined as follows:
   a. If $[A \rightarrow \alpha.a\beta]$ is in I_i and $goto(I_i,a) = I_j$, then set action[i,a] to "shift j". Here, a must be a terminal.
   b. If $[A \rightarrow \alpha.]$ is in I_i, then set action[i, a] to "reduce $A \rightarrow \alpha$" for all a in Follow(A); here A may not be S'.
   c. If $[S' \rightarrow S.]$ is in I_i, then set action[i,$] to "accept"

If any conflicting actions are generated by the above rules, we say that the grammar is not SLR(1). The algorithm then fails to produce a

Constructing an SLR Parsing Table (cont'd)

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If goto(I_i, A) = I_j, then goto[i, A] = j.
4. All entries not defined by rules (2) and (3) are made "error".
5. The initial state of the parser is the one constructed from the set of items containing [S' → S].

See example in class