Compiler Design

Lecture-13

Predictive Parsing Algorithms

Topics Covered

- Predictive Parser
- Left Recursive Grammars
- Constructing the Parsing Table
- LR Parsing
- SLR Parsing

Predictive Parser: Generalities

- In many cases, by carefully writing a grammar—eliminating left recursion from it and left factoring the resulting grammar—we can obtain a grammar that can be parsed by a recursive-descent parser that needs no backtracking.
- Such parsers are called *predictive parsers.*

Left Recursive Grammars I

- A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \rightarrow A\alpha$, for some string α
- Top-down parsers can loop forever when facing a left-recursive rules. Therefore, such rules need to be eliminated.
- A left-recursive rule such as $A \rightarrow A\ \alpha \mid \beta$ can be eliminated by replacing it by:
  - $A \rightarrow \beta\ R$ where R is a new non-terminal
  - $R \rightarrow \alpha\ R \mid \epsilon$ and ϵ is the empty string
- The new grammar is right-recursive

Left-Recursive Grammars II

- The general procedure for removing direct left recursion—recursion that occurs in one rule—is the following:
  - Group the A-rules as

    $A \rightarrow A\alpha 1 \mid \ldots \mid A\alpha m \mid \beta 1 \mid \beta 2 \mid \ldots \mid \beta n$

    where none of the β's begins with A
  - Replace the original A-rules with
    - $A \rightarrow \beta 1 A' \mid \beta 2\, A' \mid \ldots \mid \beta n\, A'$
    - $A' \rightarrow \alpha 1\, A' \mid \alpha 2\, A' \mid \ldots \mid \alpha m\, A'$
- This procedure will not eliminate indirect left recursion of the kind:
  - $A \rightarrow BaA$
  - $B \rightarrow Ab$ [Another procedure exists that is not given here]
- Direct or Indirect Left-Recursion is problematic for all top-down parsers. However, it is not a problem for bottom-up parsing algorithms

Left-Recursive Grammars III

- Here is an example of a (directly) left-recursive grammar:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

- This grammar can be re-written as the following non left-recursive grammar:

$$E \rightarrow T\ E' \qquad E' \rightarrow + TE' \mid \epsilon$$
$$T \rightarrow F\ T' \qquad T' \rightarrow * F\ T' \mid \epsilon$$
$$F \rightarrow (E) \mid id$$

Left-Factoring a Grammar I

- Left Recursion is not the only trait that disallows top-down parsing.
- Another is whether the parser can always choose the correct Right Hand Side on the basis of the next token of input, using only the first token generated by the leftmost nonterminal in the current derivation.
- To ensure that this is possible, we need to left-factor the non left-recursive grammar generated in the previous step.

Left-Factoring a Grammar II

- Here is the procedure used to left-factor a grammar:
  - For each non-terminal A, find the longest prefix α common to two or more of its alternatives.
  - Replace all the A productions:

    $A \rightarrow \alpha\beta1 \mid \alpha\beta2 \ldots \mid \alpha\beta n \mid \gamma$

  (where γ represents all alternatives that do not begin with α)
  - By:

    $A \rightarrow \alpha\ A' \mid \gamma$

    $A' \rightarrow \beta1 \mid \beta2 \mid \ldots \mid \beta n$

Left-Factoring a Grammar III

- Here is an example of a common grammar that needs left factoring:

$$S \rightarrow iEtS \mid iEtSeS \mid a$$
$$E \rightarrow b$$

(i stands for “if”; t stands for “then”; and e stands for “else”)

- Left factored, this grammar becomes:

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \epsilon$$
$$E \rightarrow b$$

Predictive Parser: Details

- The key problem during predictive parsing is that of determining the production to be applied for a non-terminal.
- This is done by using a parsing table.
- A parsing table is a two-dimensional array M[A,a] where A is a non-terminal, and a is a terminal or the symbol $, menaing “end of input string”.
- The other inputs of a predictive parser are:
  - The input buffer, which contains the string to be parsed followed by $.
  - The stack which contains a sequence of grammar symbols with, initially, $S (end of input string and start symbol) in it.

Predictive Parser: Informal Procedure

- The predictive parser considers X, the symbol on top of the stack, and a, the current input symbol. It uses, M, the parsing table.
  - If X=a=$ ➔ halt and return success
  - If X=a≠$ ➔ pop X off the stack and advance input pointer to the next symbol
  - If X is a non-terminal ➔ Check M[X,a]
    - If the entry is a production rule, then replace X on the stack by the Right Hand Side of the production
    - If the entry is blank, then halt and return failure

Predictive Parser: An Example

	id	+	*	(	)	$
E	E→ TE’			E→ TE’		
E’		E’ →+ TE’			E’ →ϵ	E’ →ϵ
T	T→ FT’			T→ FT’		
T’		T’ →ϵ	T’ →* FT’		T’ →ϵ	T’ →ϵ
F	F→ id			F→ (E)		

Parsing Table

Stack	Input	Output
$E	id+id*id$	
$E’T	id+id*id$	E → TE’
$E’T’F	id+id*id$	T → FT’
$E’T’id	id+id*id$	F → id
$E’T’	+id*id$	
$E’	+id*id$	T’ → ϵ
$E’T+	+id*id$	E’ → +TE’
$E’T	id*id$	
$E’T’F	id*id$	T → FT’
$E’T’id	id*id$	F → id
$E’T’	*id$	
$E’T’F*	*id$	T’ → *FT’
$E’T’F	id$	
$E’T’id	id$	F → id
$E’T’	$	
$E’	$	T’ → ϵ
$	$	E’ → ϵ

Algorithm Trace →

Constructing the Parsing Table I: First and Follow

- First(α) is the set of terminals that begin the strings derived from α. Follow(A) is the set of terminals a that can appear to the right of A. First and Follow are used in the construction of the parsing table.
- Computing First:
  - If X is a terminal, then First(X) is {X}
  - If X → ϵ is a production, then add ϵ to First(X)
  - If X is a non-terminal and X → Y1 Y2 … Yk is a production, then place a in First(X) if for some i, a is in First(Yi) and ϵ is in all of First(Y1)…First(Yi-1)

Constructing the Parsing Table II: First and Follow

- Computing Follow:
  - Place $ in Follow(S), where S is the start symbol and $ is the input right endmarker.
  - If there is a production A → αBβ, then everything in First(β) except for ϵ is placed in Follow(B).
  - If there is a production A → αB, or a production A → αBβ where First(β) contains ϵ, then everything in Follow(A) is in Follow(B)

Example: E → TE' E' → +TE' | ϵ
T → FT' T' → *FT' | ϵ
F → (E) | id

First(E) = First(T) = First(F) = {(, id} First(E') = {+, ϵ}
First(T') = {*, ϵ}

Follow(E) = Follow(E') = {),$}
Follow(F)={+,*,),$}
Follow(T) = Follow(T') = {+,),$}

Constructing the Parsing Table III

- Algorithm for constructing a predictive parsing table:
  1. For each production A → α of the grammar, do steps 2 and 3
  2. For each terminal a in First(α), add A → α to M[A, a]
  3. If ϵ is in First(α), add A → α to M[A, b] for each terminal b in Follow(A). If ϵ is in First(α), add A → α to M[A,b] for each terminal b in Follow(A). If ϵ is in First(α) and $ is in Follow(A), add A → α to M[A, $].
  4. Make each undefined entry of M be an error.

LL(1) Grammars

- A grammar whose parsing table has no multiply-defined entries is said to be LL(1)
- No ambiguous or left-recursive grammar can be LL(1).
- A grammar G is LL(1) iff whenever A → α | β are two distinct productions of G, then the following conditions hold:
  - For no terminal a do both α and β derive strings beginning with a
  - At most one of α and β can derive the empty string
  - If β can (directly or indirectly) derive ϵ, then α does not derive any string beginning with a terminal in Follow(A).

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 power. Our

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]})}

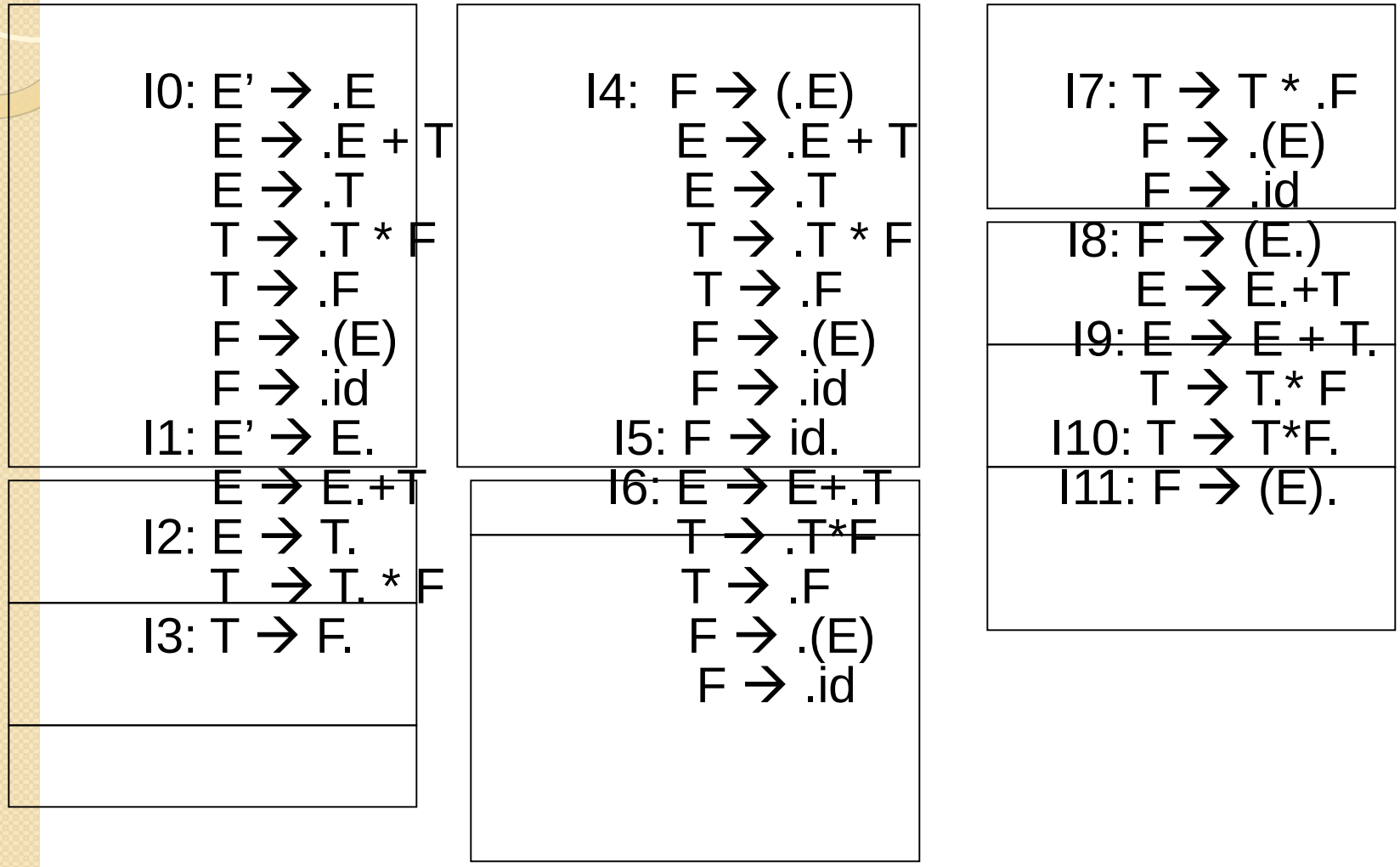
Repeat

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



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