编译原理/Compiler
Parsing

周雅倩
复旦大学计算机科学技术学院
zhouyaqian@fudan.edu.cn
2018/10/11

References

- Partial contents are copied from the slides of Alex Aiken
  - http://www.stanford.edu/class/cs143/
  - https://www.coursera.org/course/compilers
- http://en.wikipedia.org/wiki/Comparison_of_parser_generators

Outline

- Parser overview
- Context-free grammars (CFG)
- LL Parsing (Left to right parse, Leftmost derivation)
- LR Parsing (Left to right parse, Rightmost derivation)

Languages and Automata

- Formal languages are very important in CS
  - Especially in programming languages
- Regular languages
  - The weakest formal languages widely used
  - Many applications
- We will also study context-free languages

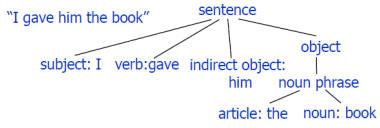
Limitations of Regular Languages

- Intuition: A finite automaton that runs long enough must repeat states
- Finite automaton can't remember # of times it has visited a particular state
- Finite automaton has finite memory
  - Only enough to store in which state it is
  - Cannot count, except up to a finite limit
- E.g., language of balanced parentheses is not regular: $\{ (^i \beta)^i \mid i \geq 0 \}$

语法分析器简介
PARSER OVERVIEW

The Functionality of the Parser

- **Syntax analysis for natural languages:**
  - **recognize whether a sentence is grammatically well-formed & identify the function of each component.**

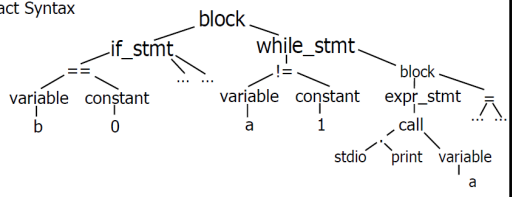


Syntax Analysis Example

Source code (token stream)

```
{
    if (b == (0)) a = b;
    while (a != 1) {
        stdio.print(a);
        a = a - 1;
    }
}
```

Abstract Syntax Tree



Comparison with Lexical Analysis

Phase	Input	Output
Lexer	Sequence of characters	Sequence of tokens
Parser	Sequence of tokens	Parse tree

The Role of the Parser

- **Not all sequences of tokens are programs . . .**
- **. . . Parser must distinguish between valid and invalid sequences of tokens**
- **We need**
  - **A language for describing valid sequences of tokens**
  - **A method for distinguishing valid from invalid sequences of tokens**

上下文无关文法
CONTEXT-FREE GRAMMARS

Context-Free Grammars

- **Programming language constructs have recursive structure**
- **An exp is**
  if (exp) {exp } else {exp } , or
  while (exp) {exp }, or
  ...
- **Context-free grammars are a natural notation for this recursive structure**

CFGs (Cont.)

- A CFG consists of
  - A set of terminals T
  - A set of non-terminals N
  - A start symbol S (a non-terminal)
  - A set of productions r
  - Assuming $X \in N$
  - $X \rightarrow \varepsilon$, or
  - $X \rightarrow Y1\ Y2 \ldots Yn$ where $Yi \subseteq N \cup T$

13

Terminals

- Terminals are called because there are no rules for replacing them
- Once generated, terminals are permanent
- Terminals ought to be tokens of the language

14

Notational Conventions (符号的使用约定)

- In these lecture notes
  - Non-terminals are written upper-case
  - Terminals are written lower-case
  - The start symbol is the left-hand side of the first production
- Productions
  - $A \rightarrow \alpha$, A left side, α right side
  - $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, ... , $A \rightarrow \alpha_k$,
  - $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$,

15

The Language of a CFG

- Read productions as replacement rules:

  $X \rightarrow Y_1 \ldots Y_n$

  means X can be replaced by $Y_1 \ldots Y_n$
- $X \rightarrow \varepsilon$

  means X can be erased (replaced with empty string)

16

Key Ideas

1. Begin with a string consisting of the start symbol "S"
2. Replace any non-terminal X in the string by a right-hand side of some production

   $X \rightarrow Y_1 \ldots Y_n$
3. Repeat (2) until there are no non-terminals in the string

17

The Language of a CFG (Cont.)

More formally, write

$X_1 \ldots X_i \ldots X_n \rightarrow X_1 \ldots X_{i-1}\ Y_1 \ldots Y_m\ X_{i+1} \ldots X_n$

if there is a production

$X_i \rightarrow Y_1 \ldots Y_m$

18

Examples of CFGs

Simple arithmetic expressions:

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

19

The Language of a CFG (Cont.)

Write

$X_1 \ldots X_n \quad * \rightarrow \quad Y_1 \ldots Y_m$

if

$X_1 \ldots X_n \rightarrow \ldots \rightarrow \ldots \rightarrow Y_1 \ldots Y_m$

in 0 or more steps

20

The Language of a CFG

Let *G* be a context-free grammar with start symbol S. Then the language of *G* is:

$\{ a_1 \ldots a_n \mid S * \rightarrow a_1 \ldots a_n$ **and every** a_i **is a terminal** $\}$

21

Examples

- **L(G) is the language of CFG G**

  $\{ (^i)^i \mid i \geq 0 \}$

- **Strings of balanced parentheses**
- **Grammar:**
  - $S \rightarrow (S)$
  - $S \rightarrow \varepsilon$

22

Example

exp → if (exp) {exp } else {exp }
| while (exp) {exp }
| id

23

Example (Cont.)

- **Some elements of the language**
  - **id**
  - **if (id) {id} else {id}**
  - **while(id) {id}**
  - **if(while(id) {id}){id}else{id}**
  - **while(id){while(id){id}}**
  - **…**

24

Arithmetic Example

- Simple arithmetic expressions:

  E→E+E|E*E|(E)|int

- Some elements of the language:

int	int+int
(int)	int*int
(int)*int	int*(int)

25

Notes

The idea of a CFG is a big step. But:

- Membership in a language is "yes" or "no"
  - we also need parse tree of the input
- Must handle errors gracefully
- Need an implementation of CFG's (e.g., bison)

26

More Notes

- Form of the grammar is important
  - Many grammars generate the same language
  - Tools are sensitive to the grammar
  - Note: Tools for regular languages (e.g., flex) are also sensitive to the form of the regular expression, but this is rarely a problem in practice

27

Derivations and Parse Trees

A *derivation*(推导) is a sequence of productions

S → ... → ...

A derivation can be drawn as a tree

- Start symbol is the tree's root
- For a production $X \rightarrow Y_1 \dots Y_n$ add children $Y_1, \dots, Y_n$ to node X

28

Derivation Example

- Grammar

  E→E+E|E*E|(E)|int

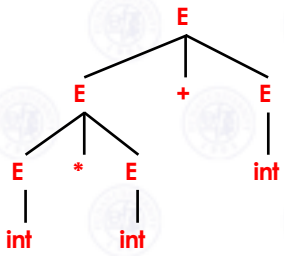
- String

  int*int+int

29

Derivation Example (Cont.)

E
→ E+E
→ E*E+E
→ int*E+E
→ int*int+E
→ int*int+int



30

Notes on Derivations

- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- An in-order traversal of the leaves is the original input
- The parse tree shows the association of operations, the input string does not

31

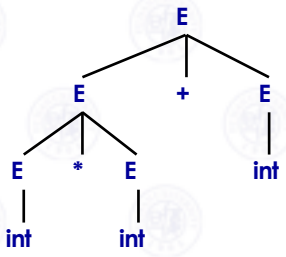
Left-most and Right-most Derivations

- Two choices to be made in each step in a derivation
  - Which nonterminal to replace
  - Which alternative to use for that nonterminal
- The example is a left-most derivation
  - At each step, replace the left-most non-terminal
- There is an equivalent notion of a right-most derivation.
  - At each step, replace the right-most non-terminal

32

Right-most Derivation in Detail

E
→ E+E
→ E+int
→ E*E+int
→ E*int+int
→ int*int+int



33

Derivations and Parse Trees

- Note that right-most and left-most derivations have the same parse tree
- The difference is the order in which branches are added

34

Ambiguity

- Grammar

  E → E + E | E * E | (E) | int

- Strings

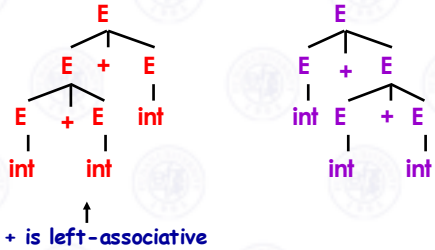
  int + int + int

  int * int + int

35

Ambiguity. Example

This string has two parse trees

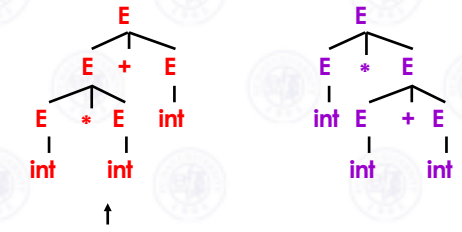


+ is left-associative

36

Ambiguity. Example

This string has two parse trees



* has higher precedence than +

Ambiguous Grammar

- A grammar is *ambiguous* if it has more than one parse tree for some string
- Equivalently, there is more than one right-most or left-most derivation for some string

Ambiguity

- Ambiguity is bad
  - Leaves meaning of some programs ill-defined
- Ambiguity is common in programming languages
  - Arithmetic expressions
  - IF-THEN-ELSE

Dealing with Ambiguity

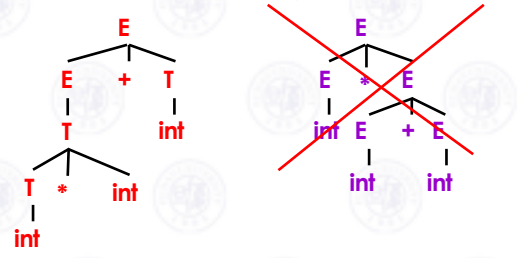
- There are several ways to handle ambiguity
- Most direct method is to rewrite the grammar unambiguously

  E → E + T | T

  T → T * int | int | (E)
- Enforces precedence（优先级） of * over +
- Enforces left-associativity（左结合） of + and *

Ambiguity. Example

The int * int + int has ony one parse tree now



Ambiguity: The Dangling Else

- Consider the grammar

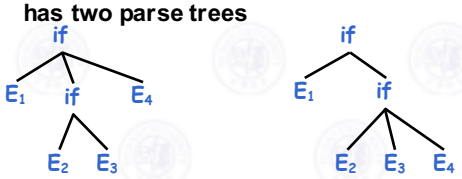
  E → if E then E

  | if E then E else E

  | OTHER
- This grammar is also ambiguous

The Dangling Else: Example

- The expression
  if E_1 then if E_2 then E_3 else E_4
  has two parse trees



- Typically we want the second form

The Dangling Else: A Fix

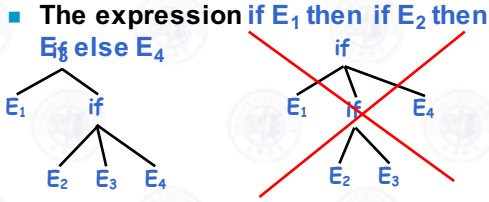
- else matches the closest unmatched then
- We can describe this in the grammar (distinguish between matched and unmatched "then")

```
E → MIF          /* all then are matched */
  | UIF          /* some then are unmatched */

MIF → if E then MIF else MIF
    | OTHER

UIF → if E then E
    | if E then MIF else UIF
```

Describes the same set of strings

The Dangling Else: Example Revisited

- The expression if E_1 then if E_2 then E_3 else E_4



- A valid parse tree (for a UIF)
- Not valid because the then expression is not a MIF

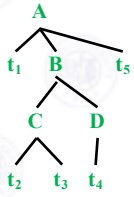
Ambiguity

- No general techniques for handling ambiguity
- Impossible to convert automatically an ambiguous grammar to an unambiguous one
- Used with care, ambiguity can simplify the grammar
  - Sometimes allows more natural definitions
  - We need disambiguation mechanisms

LL(1) PARSING

Intro to Top-Down Parsing

- Terminals are seen in order of appearance in the token stream:
  t_1 t_2 t_3 t_4 t_5
- The parse tree is constructed
  - From the top
  - From left to right



Recursive Descent Parsing

- Consider the grammar
  E → T + E | T
  T → int | int * T | (E)
  Different to previous grammar? Why? Left Recursion
- Token stream is: int * int
- Start with top-level non-terminal E
- Try the rules for E in order

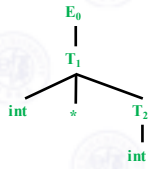
Recursive Descent Parsing. Example (Cont.)

1. Try $E_0 \rightarrow T_1 + E_2$
2. Then try a rule for $T_1 \rightarrow (E_3)$
   But (does not match input token int_5
3. Try $T_1 \rightarrow int$. Token matches.
   But + after T_1 does not match input token *
4. Try $T_1 \rightarrow int * T_2$
   This will match but + after T_1 will be unmatched
5. Have exhausted the choices for T_1
   Backtrack to choice for E_0

Recursive Descent Parsing. Example (Cont.)

6. Try $E_0 \rightarrow T_1$
7. Follow same steps as before for T_1

And succeed with $T_1 \rightarrow int * T_2$ and $T_2 \rightarrow int$
With the following parse tree



Recursive Descent Parsing. Notes.

- Easy to implement by hand
- But does not always work …

Recursive-Descent Parsing

- Parsing: given a string of tokens $t_1\ t_2 \ldots t_n$, find its parse tree
- Recursive-descent parsing: Try all the productions exhaustively
  - At a given moment the fringe of the parse tree is: $t_1\ t_2 \ldots t_k\ A \ldots$
  - Try all the productions for A: if $A \rightarrow BC$ is a production, the new fringe is $t_1\ t_2 \ldots t_k\ B\ C \ldots$
  - Backtrack when the fringe doesn't match the string
  - Stop when there are no more non-terminals

When Recursive Descent Does Not Work

- Consider a production S → S a:
  - In the process of parsing S we try the above rule
  - What goes wrong?
- A left-recursive grammar has a non-terminal S
  $S \rightarrow^+ S\alpha$ for some α
- Recursive descent does not work in such cases
  - It goes into an infinite loop

Elimination of Left Recursion

- Consider the left-recursive grammar
  $S \rightarrow S\,\alpha \mid \beta$
- S generates all strings starting with a β and followed by a number of α
- Can rewrite using right-recursion
  $S \rightarrow \beta\,S'$
  $S' \rightarrow \alpha\,S' \mid \varepsilon$

55

Elimination of Left-Recursion. Example

- Consider the grammar
  $S \rightarrow 1 \mid S\,0$ ($\beta = 1$ and $\alpha = 0$)

  can be rewritten as
  $S \rightarrow 1\,S'$
  $S' \rightarrow 0\,S' \mid \varepsilon$

56

More Elimination of Left-Recursion

- In general
  $S \rightarrow S\,\alpha_1 \mid \ldots \mid S\,\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$
- All strings derived from S start with one of $\beta_1,\ldots,\beta_m$ and continue with several instances of $\alpha_1,\ldots,\alpha_n$
- Rewrite as
  $S \rightarrow \beta_1\,S' \mid \ldots \mid \beta_m\,S'$
  $S' \rightarrow \alpha_1\,S' \mid \ldots \mid \alpha_n\,S' \mid \varepsilon$

57

Summary of Recursive Descent

- Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - … but that can be done automatically
- Unpopular because of backtracking
  - Thought to be too inefficient
- In practice, backtracking is eliminated by restricting the grammar

58

Predictive Parsers

- Like recursive-descent but parser can "predict" which production to use
  - By looking at the next few tokens
  - No backtracking
- Predictive parsers accept LL(k) grammars
  - L means "left-to-right" scan of input
  - L means "leftmost derivation"
  - k means "predict based on k tokens of lookahead"
- In practice, LL(1) is used

59

LL(1) Languages

- In recursive-descent, for each non-terminal and input token there may be a choice of production
- LL(1) means that for each non-terminal and token there is only one production that could lead to success
- Can be specified as a 2D table
  - One dimension for current non-terminal to expand
  - One dimension for next token
  - A table entry contains one production

60

Predictive Parsing and Left Factoring

- Recall the grammar
  E → T + E | T
  T → int | int * T | (E)
- Impossible to predict because
  – For T two productions start with int
  – For E it is not clear how to predict
- A grammar must be left-factored before use for predictive parsing

61

Left-Factoring Example

- Recall the grammar
  E → T + E | T
  T → int | int * T | (E)
- Factor out common prefixes of productions
  E → T X
  X → + E | ε
  T → (E) | int Y
  Y → * T | ε

62

LL(1) Parsing Table Example

- Left-factored grammar
  E → T X X → + E | ε
  T → (E) | int Y Y → * T | ε
- The LL(1) parsing table:

	int	*	+	(	)	$
T	int Y			(E)		
E	T X			T X		
X			+ E		ε	ε
Y		* T	ε		ε	ε

63

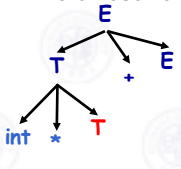
Constructing Parsing Tables

- We want to generate parsing tables from CFG

64

Top-Down Parsing. Review

- Top-down parsing expands a parse tree from the start symbol to the leaves with leftmost non-terminal.



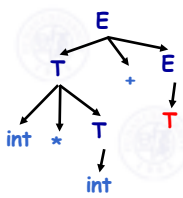
- The leaves at any point form a string βAγ
  - β contains only terminals
  - The input string is βbδ
  - The prefix β matches
  - The next token is b

int * int + int

65

Top-Down Parsing. Review

- Top-down parsing expands a parse tree from the start symbol to the leaves with leftmost non-terminal.



- The leaves at any point form a string βAγ
  - β contains only terminals
  - The input string is βbδ
  - The prefix β matches
  - The next token is b

int * int + int

66

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 $A \rightarrow \alpha$
  - when trying to expand A we use $A \rightarrow \alpha$ if b follows next
- Once we have the table
  - The parsing algorithm is simple and fast
  - No backtracking is necessary

Constructing Predictive Parsing Tables

- Consider the state $S \rightarrow^* \beta A \gamma$
  - With b the next token
  - Trying to match $\beta b \delta$

There are two possibilities:

1. b belongs to an expansion of A
   - Any $A \rightarrow \alpha$ can be used if b can start a string derived from α

     In this case we say that $b \in First(\alpha)$

Or…

Constructing Predictive Parsing Tables (Cont.)

2. b does not belong to an expansion of A
   - The expansion of A is empty and b belongs to an expansion of γ
   - Means that b can appear after A in a derivation of the form $S \rightarrow^* \beta A b \omega$
   - We say that $b \in Follow(A)$ in this case
   - What productions can we use in this case?
     - Any $A \rightarrow \alpha$ can be used if α can expand to ε
     - We say that $\varepsilon \in First(A)$ in this case

Computing First Sets

Definition

$First(X) = \{ b \mid X \rightarrow^* b\alpha \} \cup \{ \varepsilon \mid X \rightarrow^* \varepsilon \}$

1. $First(b) = \{ b \}$
2. For all productions $X \rightarrow A_1 \dots A_n$
   - Add $First(A_1) - \{\varepsilon\}$ to First(X). Stop if $\varepsilon \notin First(A_1)$
   - Add $First(A_2) - \{\varepsilon\}$ to First(X). Stop if $\varepsilon \notin First(A_2)$
   - …
   - Add $First(A_n) - \{\varepsilon\}$ to First(X). Stop if $\varepsilon \notin First(A_n)$
   - Add ε to First(X)

First Sets. Example

- Recall the grammar

  $E \rightarrow T\ X$ $\quad$ $X \rightarrow + E \mid \varepsilon$

  $T \rightarrow (E) \mid int\ Y$ $\quad$ $Y \rightarrow * T \mid \varepsilon$
- First sets

  $First(( ) = \{ (\}$ $\quad$ $First(T) = \{int, (\}$

  $First( )) = \{) \}$ $\quad$ $First(E) = \{int, (\}$

  $First(int) = \{ int \}$ $\quad$ $First(X) = \{+, \varepsilon \}$

  $First(+) = \{ + \}$ $\quad$ $First(Y) = \{*, \varepsilon \}$

  $First(*) = \{ * \}$

Computing Follow Sets

Definition $Follow(X) = \{ b \mid S \rightarrow^* \beta X b \delta \}$

1. Compute the First sets for all non-terminals first
2. Add \$ to Follow(S) (if S is the start non-terminal)
3. For all productions $Y \rightarrow \dots X A_1 \dots A_n$
   - Add $First(A_1) - \{\varepsilon\}$ to Follow(X). Stop if $\varepsilon \notin First(A_1)$
   - Add $First(A_2) - \{\varepsilon\}$ to Follow(X). Stop if $\varepsilon \notin First(A_2)$
   - …
   - Add $First(A_n) - \{\varepsilon\}$ to Follow(X). Stop if $\varepsilon \notin First(A_n)$
   - Add Follow(Y) to Follow(X)

Follow Sets. Example

- Recall the grammar
  - E → T X
  - X → + E | ε
  - T → (E) | int Y
  - Y → * T | ε
- Follow sets
  - Follow(+) = { int, (}
  - Follow(*) = { int, (}
  - Follow(( ) = { int, (}
  - Follow(E) = {), $}
  - Follow(X) = {$,) }
  - Follow(T) = {+,) , $}
  - Follow( )) = {+,) , $}
  - Follow(Y) = {+,) , $}
  - Follow(int) = {*, +,) , $}

73

Constructing LL(1) Parsing Tables

- Construct a parsing table T for CFG G
- For each production A → α in G do:
  - For each terminal b ∈ First(α) do
    - T[A, b] = α
  - If α →* ε, for each b ∈ Follow(A) do
    - T[A, b] = α
  - If α →* ε and $ ∈ Follow(A) do
    - T[A, $] = α

74

Constructing LL(1) Tables. Example

- Recall the grammar
  - E → T X
  - X → + E | ε
  - T → (E) | int Y
  - Y → * T | ε
- Where in the line of Y we put Y → * T ?
  - In the lines of First(*T) = { * }
- Where in the line of Y we put Y → ε ?
  - In the lines of Follow(Y) = { $, +,) }

75

Using Parsing Tables

- Method similar to recursive descent, except
  - For each non-terminal S
  - We look at the next token a
  - And choose the production shown at [S,a]
- We use a stack to keep track of pending non-terminals
- We reject when we encounter an error state
- We accept when we encounter end-of-input

76

LL(1) Parsing Algorithm

initialize stack = <S $> and next (pointer to tokens)

repeat

  case stack of

    <X, rest> : if T[X,*next] = $Y_1 \ldots Y_n$

                then stack = <$Y_1 \ldots Y_n$ rest>;

                else error ();

    <t, rest> : if t == *next ++

                then stack = <rest>;

                else error ();

until stack == < >

77

LL(1) Parsing Example (Table)

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal
Y X $	$	ε
X $	$	ε
$	$	ACCEPT

78

Notes on LL(1) Parsing Tables

- LL(1) languages are those defined by a parsing table for the LL(1) algorithm
- If any entry is multiply defined then G is not LL(1)
  - If G is ambiguous
  - If G is left recursive
  - If G is not left-factored
  - And in other cases as well
- Most programming language grammars are not LL(1)
- There are tools that build LL(1) tables

79

LR PARSING

80

Bottom-up parsing

- Bottom-up parsing is more general than top-down parsing
  - Builds on ideas in top-down parsing
  - Preferred method in practice
- also called as LR Parsing
  - L means that read tokens from left to right
  - R means that it constructs a rightmost derivation
  - LR(0),LR(1),SLR(1),LALR(1)

81

The Idea

- LR parsing *reduces* a string to the start symbol by inverting productions:

  str ← input string of terminals

repeat
Identify β in str such that A → β is a production
(i.e., str = α β γ)
Replace β by A in str (i.e., str becomes α A γ)
until str = S

82

An Introductory Example

- LR parsers don't need left-factored grammars and can also handle left-recursive grammars
- Consider the following grammar:

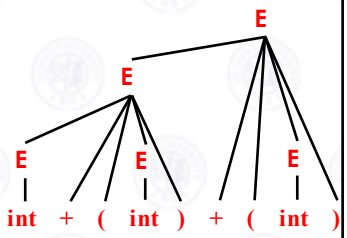
  E → E + (E) | int
  - Why is this not LL(1)?
- Consider the string: int + (int) + (int)

83

A Bottom-up Parse in Detail

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)
E

A rightmost derivation in reverse



84

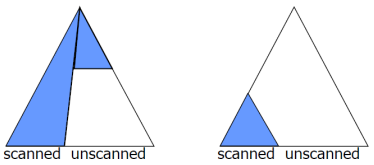
Where Do Reductions Happen

- Some characters:
  - Let αβγ be a step of a bottom-up parse
  - Assume the next reduction is by A→ β
  - Then γ is a string of terminals !
- Why?
  - Because αAγ → αβγ is a step in a right-most derivation

85

Top-down vs. Bottom-up

- Bottom-up: Don't need to figure out as much of the parse tree for a given



Top-down Bottom-up

86

Notation

- Idea: Split string into two substrings
  - Right substring (a string of terminals) is as yet unexamined by parser
  - Left substring has terminals and non-terminals
- The dividing point is marked by a I
  - The I is not part of the string
- Initially, all input is unexamined:

$Ix_1x_2 \ldots x_n$

87

Shift-Reduce Parsing

- Bottom-up parsing uses only three kinds of actions:

Shift
Reduce
Accept

88

Shift

Shift: Move I one place to the right
- Shifts a terminal to the left string

E + (I int) ⇒ E + (int I)

89

Reduce

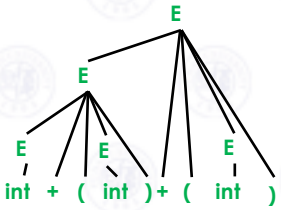
Reduce: Apply an inverse production at the right end of the left string
- If E → E + (E) is a production, then

E + (E + (E) I) ⇒ E +(E I)

90

Shift-Reduce Example

ı int + (int) + (int)$	shift
int ı + (int) + (int)$	red. E → int
E ı + (int) + (int)$	shift 3 times
E + (int ı) + (int)$	red. E → int
E + (E ı) + (int)$	shift
E + (E) ı + (int)$	red.E → E + (E)
E ı + (int)$	shift 3 times
E + (int ı)$	red. E → int
E + (E ı)$	shift
E + (E) ı $	red. E →E + (E)
E ı $	accept

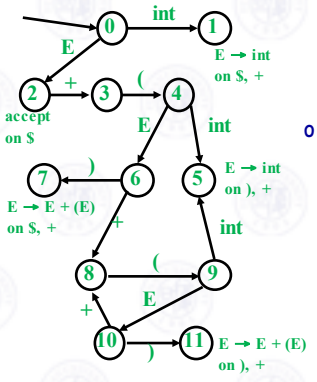


Key Issue: When to Shift or Reduce?

- Decide based on the left string (the stack)
- Idea: use a finite automaton (DFA) to decide when to shift or reduce
  - The DFA input is the stack
  - The language consists of terminals and non-terminals

LR Parsing Engine

- Basic mechanism:
  - Use a set of parser states
  - Use a stack of states
  - Use a parsing table to:
    - Determine what action to apply (shift/reduce)
    - Determine the next state
- The parser actions can be precisely determined from the table



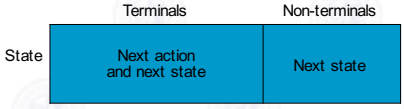
0

ı int + (int) + (int)$	shift
int ı + (int) + (int)$	E → int
E ı + (int) + (int)$	shift(x3)
E + (int ı) + (int)$	E → int
E + (E ı) + (int)$	shift
E + (E) ı + (int)$	E → E+(E)
E ı + (int)$	shift (x3)
E + (int ı)$	E → int
E + (E ı)$	shift
E + (E) ı $	E → E+(E)
E ı $	accept

Representing the DFA

- Parsers represent the DFA as a 2D table
  - Recall table-driven lexical analysis
- Lines correspond to DFA states
- Columns correspond to terminals and non-terminals
- Typically columns are split into:
  - Those for terminals: action table
  - Those for non-terminals: goto table

The LR Parsing Table



Algorithm:

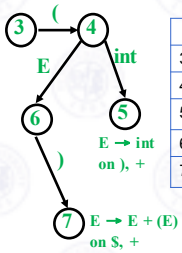
look at entry for current state S and input terminal c

if Table[S,c] = s(S') then shift:
    push(S')

if Table[S,c] = A→α then reduce:
    pop(|α|); S'=top(); push(A); push(Table[S',A])

Representing the DFA. Example

- The table for a fragment of our DFA:



	int	+	(	)	$	E
3			s4			
4	s5					g6
5		r E→ int		r E→ int		
6	s8			s7		
7		r E→ E+(E)			r E→ E+(E)	

The LR Parsing Algorithm

- After a shift or reduce action we return the DFA on the entire stack
  - This is wasteful, since most of the work is repeated
- Remember for each stack element on which state it brings the DFA
- LR parser maintains a stack

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

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

LR Parsing Notes

- Can be used to parse more grammars than LL
- Most programming languages grammars are LR
- Can be described as a simple table
- There are many tools for building the table
- 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 rhs we are looking for
  - What we have seen so far from the rhs
- Each DFA state describes several such contexts
  - E.g., when we are looking for non-terminal E, we might be looking either for an int or a E + (E) rhs

LR(0) Items

- The LR(0) item of a grammar G
  - For example: A->XYZ
    - A → •XYZ
    - A → X • YZ
    - A → XY • Z
    - A → XYZ •
- The parser begins with •S$ (state 0).

The Closure Operation

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

```
Closure(I) =
  repeat
    for each [X → a•Yb] in I
      for each production Y → g
                add [Y → •g] to Items
  until I is unchanged
return I
```

Goto Operation

- Goto(I,X) is defined to be the closure of the set of all items [A→aX•β] such that [A→ a•X β] is in I.

```
I is the set of items
X is a grammar symbol
Goto(I,X)=
set J to the empty set
for any item A→ a•Xβ in I
        add A→ aX•β to J
return Closure(J)
```


The Sets-of-items Construction

```
Initialize T={closure({[S'→ • S$]})};
Initialize E ={};
repeat
  for each state(a set of items) I in T
        let J be Goto(I,X)
          T=T∪{J};
          E=E ∪{I –X→ J};
  until E and I did not change
```

For the symbol $ we do not compute Goto(I,$).

LR Parser

- To be add……

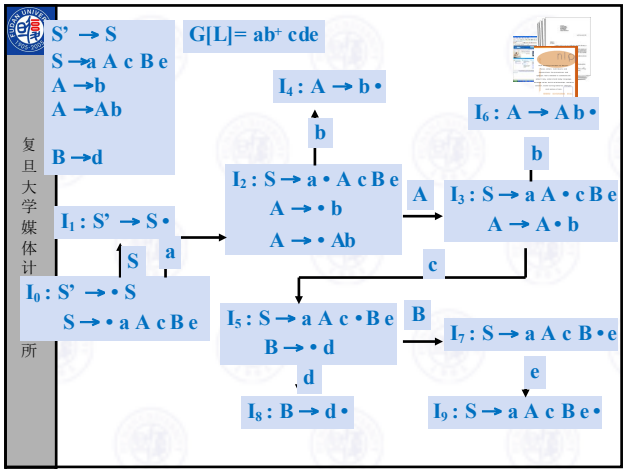
Example(1)

- Grammar G1:

1. S' → S$
2. S → (L)
3. S → x
4. L → S
5. L → L,S

Example(2)

- Grammar G2 :
  - S →a A c B e
  - A →b
  - A →Ab
  - B →d
- abbcde
- abbce



LR(0) Parsing Table

	ACTION						GOTO		
	a	c	e	b	d	#	S	A	B
0	s_2						1		
1						acc			
2				s_4				3	
3		s_5		s_6					
4	r_2	r_2	r_2	r_2	r_2	r_2			
5					s_8				7
6	r_3	r_3	r_3	r_3	r_3	r_3			
7			s_9						
8	r_4	r_4	r_4	r_4	r_4	r_4			
9	r_1	r_1	r_1	r_1	r_1	r_1			

Parsing the String abbcde#

Step	states.	Syms.	The rest of input	action	goto
1	0	#	abbcde$	s2	
2	02	#a	bbcde$	s4	
3	024	#ab	bcde$	r2	3
4	023	#aA	bcde$	s6	
5	0236	#aAb	cde$	r3	3
6	023	#aA	cde$	s5	
7	0235	#aAc	de$	s8	
8	02358	#aAcd	e$	r4	7
9	02357	#aAcB	e$	s9	
10	023579	#aAcBe	$	r1	1
11	01	#S	$	acc	

Parsing the String abbce#

Step	states.	Syms.	The rest of input	action goto
1	0	#	abbce#	s2
2	02	#a	bbce#	s4
3	024	#ab	bce#	r2 3
4	023	#aA	bce#	s6
5	0236	#aAb	ce#	r3 3
6	023	#aA	ce#	s5
7	0235	#aAc	e#	error

Example(3)

- Grammar G3:
  1. S → E$
  2. E → T + E
  3. E → T
  4. T → x
- G3 is not LR(0).
  - We can extend LR(0) in a simple way.
  - SLR(1): simple LR
  - Reduce actions are indicated by Follow set.

112

SLR(1)

R={};
for each state *I* in *T*
  for each item A →α• in *I*
    for each token a in Follow(A)
      R=*R*∪{(*I*, a, A →α)}

(*I*, a, A →α) indicates that in state *I*, on lookahead symbol a, the parser will reduce by rule A →α.

113

LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a single reduce action
- With more complex grammar, construction gives states with shift/reduce or reduce/reduce conflicts
- Need to use look-ahead to choose

114

LR(1) Items

- An LR(1) item is a pair:
  $X \rightarrow \alpha \bullet \beta, a$
  - $X \rightarrow \alpha\beta$ is a production
  - a is a terminal (the lookahead terminal)
  - LR(1) means 1 lookahead terminal
- $[X \rightarrow \alpha \bullet \beta, a]$ describes a context of the parser
  - We are trying to find an X followed by an a, and
  - We have α already on top of the stack
  - Thus we need to see next a prefix derived from βa

Convention

- We add to our grammar a fresh new start symbol S and a production $S \rightarrow E\$$
  - Where E is the old start symbol
- For grammar:
  $E \rightarrow int$ and $E \rightarrow E + (E)$
- The initial parsing context contains:
  $S \rightarrow \bullet E\$, ?$
  - Trying to find an S as a string derived from $E\$$
  - The stack is empty

LR(1) Items (Cont.)

- In context containing
  $E \rightarrow E + \bullet (E), +$
  - If (follows then we can perform a shift to context containing
    $E \rightarrow E + (\bullet E), +$
- In context containing
  $E \rightarrow E + (E)\bullet, +$
  - We can perform a reduction with $E \rightarrow E + (E)$
  - But only if a + follows

LR(1) Items (Cont.)

- Consider the item
  $E \rightarrow E + (\bullet E) , +$
- We expect a string derived from $E) +$
- We describe this by extending the context with two more items:
  $E \rightarrow \bullet int,)$
  $E \rightarrow \bullet E + (E) ,)$

The Closure Operation

The operation of extending the context with items

```
Closure(I) =
  repeat
    for each [A → α•Xβ, z] in I
      for each production X → γ
        for each w ∈ First(βz)
          add [X→ •γ, w] to I
  until I is unchanged
return I
```


The Goto Operation

```
Goto(I, X) =
    J={};
    for each item (A → α•Xβ, z) in I
      add (A → αX • β, z) to J;
    return Closure(J);
```

Example(1)

- Grammar:
  E → E + (E)|int
- Construct the start context:
  Closure({S → •E$, ?})

S → •E$, ?
E → • E+(E), $
E → • int, $
E → • E+(E), +
E → • int, +

abbreviate

S → • E$, ?
E → • E+(E), $/+
E → • int, $/+

LR Parsing Tables. Notes

- Parsing tables (i.e. the DFA) can be constructed automatically for a CFG
- But we still need to understand the construction to work with parser generators
  – E.g., they report errors in terms of sets of items
- What kind of errors can we expect?

Shift/Reduce Conflicts

- If a DFA state contains both
  [X → α•aβ, b] and [Y → γ•, a]
- Then on input "a" we could either
  – Shift into state [X → αa•β, b], or
  – Reduce with Y → γ
- This is called a shift-reduce conflict

Shift/Reduce Conflicts

- Typically due to ambiguities in the grammar
- Classic example: the dangling else
  S → if E then S | if E then S else S | OTHER
- Will have DFA state containing
  [S → if E then S •, else]
  [S → if E then S • else S, x]
- If else follows then we can shift or reduce
- Default (bison) is to shift
  – Default behavior is as needed in this case

More Shift/Reduce Conflicts

- Consider the ambiguous grammar
  E → E + E | E * E | int
- We will have the states containing
  [E → E * • E, +] [E → E * E •, +]
  [E → • E + E, +] $\Rightarrow^{E}$ [E → E • + E, +]
  ... ...
- Again we have a shift/reduce on input +
  – We need to reduce (* binds more tightly than +)
  – Recall solution: declare the precedence of * and +

Reduce/Reduce Conflicts

- If a DFA state contains both
  [X → α •, a] and [Y → β •, a]
  – Then on input "a" we don't know which production to reduce
- This is called a reduce/reduce conflict

Reduce/Reduce Conflicts

- Usually due to gross ambiguity in the grammar
- Example: a sequence of identifiers

  S → ε | id | id S
- There are two parse trees for the string id

  S → id

  S → id S → id
- How does this confuse the parser?

127

More on Reduce/Reduce Conflicts

- Consider the states:

  [S'→ •S, $]
  [S'→ •, $]
  [S'→ •id, $]
  [S'→ •id S, $]

  —id→

  [S→id •, $]
  [S→ id • S, $]
  [S→ •, $]
  [S→ •id, $]
  [S→ •id S, $]
- Reduce conflict on input $

  S'→ S→id

  S'→S→id S→id
- Better rewrite the grammar: S → ε|id S

128

LR(1) but not SLR(1)

- Let G have productions

  S → aAb | Ac

  A → a | ε
- V(a) = {

  [S → a.Ab]

  [A → a.]

  [A → .a]

  [A → .]

  }

  FOLLOW(A) = {b,c}

  reduce-reduce conflict

129

LR(1) Parsing Tables are Big

- But many states are similar, e.g.

  1: E → int•, $/+ on $, +

  and

  5: E → int•,)/+ on), +
- Idea: merge the DFA states whose items differ only in the lookahead tokens
  - We say that such states have the same core
- We obtain

  1': E → int•, $/+/) on $, +,)

130

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

  { [X → α•β, b], [Y → γ•δ, d]}

  is

  {X → α•β, Y → γ•δ}

131

LALR States

- Consider for example the LR(1) states

  {[X → α•, a], [Y → β•, c]}

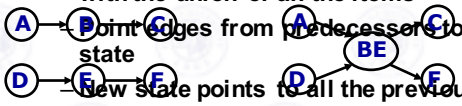
  {[X → α•, b], [Y → β•, d]}
- They have the same core and can be merged
- And the merged state contains:

  {[X → α•, a/b], [Y → β•, c/d]}
- These are called LALR(1) states
  - Stands for LookAhead LR
  - Typically 10 times fewer LALR(1) states than LR(1)

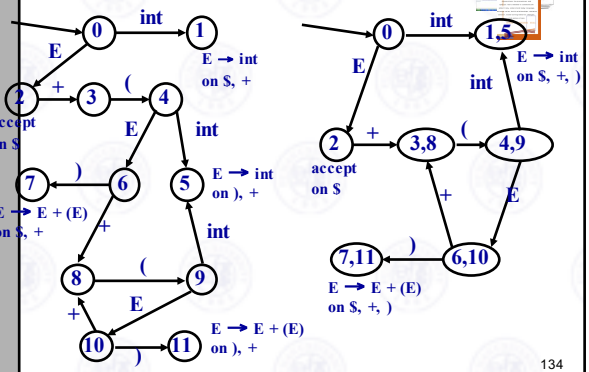
132

A LALR(1) DFA

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



Conversion LR(1) to LALR(1) Example



The LALR Parser Can Have Conflicts

- Consider for example the LR(1) states

  {[X → α•, a], [Y → β•, b]}

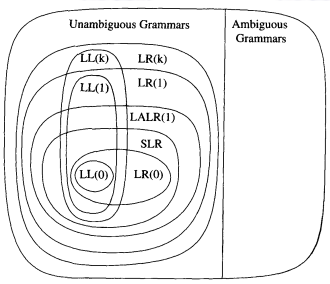
  {[X → α•, b], [Y → β•, a]}
- And the merged LALR(1) state

  {[X → α•, a/b], [Y → β•, a/b]}
- Has a new reduce-reduce conflict
- In practice such cases are rare

LALR vs. LR Parsing

- LALR languages are not natural
  - They are an efficiency hack on LR languages
- Any reasonable programming language has a LALR(1) grammar
- LALR(1) has become a standard for programming languages and for parser generators
- LALR parsers are automatically generated by compiler compilers such as Yacc and GNU Bison.

A Hierarchy of Grammar Classes



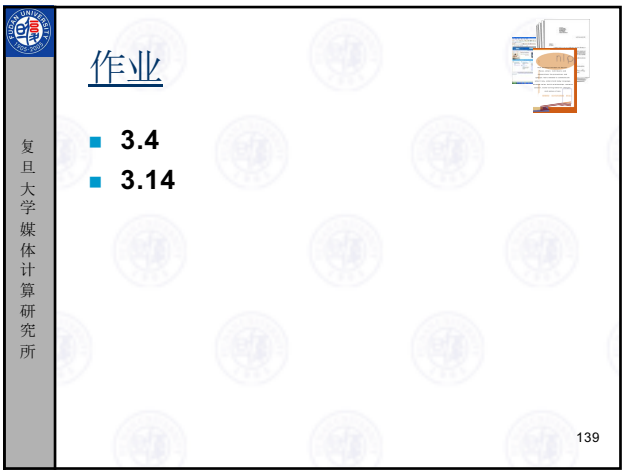
Implementation Issues

LL

- Left derivation
- More intuitive
- Easy for hand-written codes
- LL(*) are also very powerful
- Functional languages
  - Clean, Haskell, Lisp or ML

LR(LALR)

- Right derivation
- Understanding how it works is quite difficult
- Error reporting can be quite hard
- Handle a larger range of languages
- GCC 4.1:
  - replaced by a new, faster hand-written recursive-descent parser
- GLR parser


复旦大学媒体计算研究所
作业
3.4
3.14
139