DAC718 - Compiler Design

Error Recovery and Bottom-Up Parsing

Jonas Lundberg

Jonas.Lundberg@msi.vxu.se

http://w3.msi.vxu.se/users/jonasl/dac718

11 februari 2007

Agenda

- **Previous Lecture:**
  - LL-parsing – a table-driven top-down parsing technique
  - Table M(non-terminal, lookahead) $\mapsto$ production
  - Can be constructed systematically using *help* functions *Nullable*, *FIRST*, *FOLLOWS*
- **This Lecture:**
  - Error handling: detect, report, and recover
  - LR-parsing: A bottom-up parsing technique
  - Formal Languages - The Big Picture

Error Handling

Handling = Detect + Report + Recover

- **Detecting Errors**
  We don't detect the actual errors,
  we detect that an error has occured.

  $a{*}b{+}c{*}d{+}e)$ could be $\begin{cases} (a*b+c*d+e) & \text{or,} \\ a*b+c*(d+e) & \text{or,} \\ a*b+c*d+e & \text{or, ...} \end{cases}$

- **Report the error location and type**
  For example; `Syntax Error: Missing semicolon at line 235`
- **Recover from the error and analyze the rest of the program**
  Try to recover such that the resulting effect of earlier errors are minimized.

Compile-Time Errors

1. Lexical Errors
   - Illegal characters not included in the alphabet (e.g. π or $\$$)
   - Error in lexeme (e.g. a digit must follow the floating point)
   - End quotation mark missing in strings
2. Syntactical Errors
   - Incorrect sequence of tokens
   - Misspelled keywords (e.g. `TEHN`)
3. Semantical Errors
   - Incorrect composition of types (e.g. Boolean + String)
   - Multiple declaration of a variable
   - Variable use before assignment

Run-Time Errors

1. Semantical Errors
   - Division by zero
   - Array index out of range
   - Resulting number to big in an assignment
   - Illegal type casts
2. Logical Errors
   - Never ending loops
   - Error in algorithm

Run-time errors will not be treated in this course

Lexical Errors

Lexical errors are in most cases easy to handle

1. Detection: Termination (no transition possible) in a non-final state.
2. Report the location (line and column number) and the unidentified character sequence.
3. Abort the analysis of the current token.
4. Go to the initial state of the finite automata
5. Start the analysis of the next token

We don't have to handle lexical errors in the practical assignment. JavaCC takes care of that.

```
Exception in thread "main" TokenMgrError:
   Lexical error at line 3, column 8.
   Encountered: "&"
```

Syntactical Errors

Reasons for detection

- The parser can't make a prediction,
  (can't find suitable production for the given lookahead)
- Token sequence not in agreement with chosen prediction/production

We don't have to **detect** syntactical errors in the practical assignment. JavaCC takes care of that and gives reasonable error messages.

What to do when error detected?

- Report and stop?
- Report, recover and continue?
- Report, fix it if possible, continue?

The second choice is the prefered solution. We usually don't want the compiler to make any changes in the program. Not even to fix bugs.

Problems with Syntactical Error Recovery

- We can't restart from some *initial* state when an error has occured.
- Example, restart from

```
Program  --> ClassDeclaration+
```

  doesn't work for a missing semi-colon.
- In general, given a problematic token t, we don't now if
  - t is an erroneus token
  - t is an extra token
  - we are missing some tokens
- Simple "skip the erroneus token and continue" often makes the parser *unsynchronized* with the remaining token sequence
  $\Rightarrow$ everything that follows results in errors.
- Hence, the major problem is to synchronize the parser with the remaining token sequence.

Error Recovery Methods

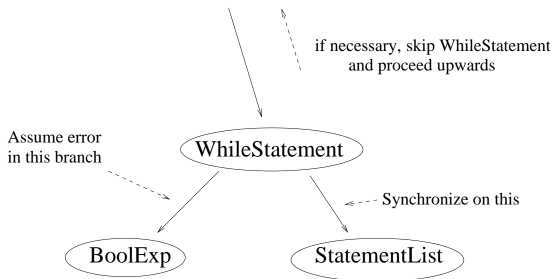
- **Panic Mode:** Discard tokens until we recognize some specific tokens (for example tokens that starts a new statement, or a new method, or a new variable declaration . . .) that we know how to handle.
- **Correcting Methods:** Guesses what it should look like, insert it temporarily, and continoue.
  Used to handle frequently occuring errors (missing semicolon) and often together with panic mode.
- **Error Grammars:** Switch to **error grammar** especially designed to handle errors. This is an advanced solution
  $\Rightarrow$ difficult to develop and implement.

Panic Mode - Basic idea

- Skip tokens until we find one that we now how to handle
  $\Rightarrow$ *Synchronization*.
- Assume we are parsing $X \in N$ and finds an erroneus token t.
  What is a good point to restart the parsing?
- The $FIRST(X)$ and $FOLLOW(X)$ sets can be used to help us.
- **Remember**
  - Recovery is a *service* provides by the parser,
    it is not essential that we recover.
  - We try to recover from most errors. Not all errors.
    $\Rightarrow$ Error recovery often stops after (say) 5 errors.

Panic Mode (Example)

- Approach: Consider current production as lost, synchronize on the following one.



- The goal is often to find next statement (or next method declaration, or next class declaration) $\Rightarrow$ we recover upwards the syntax tree.

A LL(1) Parse Table for Arithmetic Expressions

	id	$+$	$*$	$($	$)$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$	
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$	

Parse Tables

- Given a non-terminal A and lookahead t, $M[A, t]$ returns the appropriate production to use.
- Rule: *reduce* iff TOP element equals LA, otherwise *shift*.
- shift $\Rightarrow$ replace top element with $M[TOP, LA]$ right-hand side
- reduce $\Rightarrow$ pop element (a terminal) and set lookahead to next input.

Algorithm for table driven LL-parsing

$stack.push(StartSymbol)$
$LA = input.nextToken()$
repeat
  $X = stack.top()$
  if $X \in T$ or $X = EOF$ **then**
    if $X = LA$ **then**
      $stack.pop()$
      $LA = input.nextToken()$
    else
      error(stack,LA,input) (Token not in agreement with prediction)
    end if
  else
    if $M[X, t] = X \rightarrow Y_1 \ldots Y_n$ **then**
      $stack.pop()$
      push $Y_n \ldots Y_1$ onto *stack*, with Y_1 on top
      add $X \rightarrow Y_1 \ldots Y_n$ to parse tree
    else
      error(stack,LA,input) (Can't make a prediction, empty slot in $M[X, t]$)
    end if
  end if
until $LA = EOF$

The FIRST and FOLLOW Approach

- **FIRST(X):** is the set of terminals that can **begin** strings derived from X.
- **FOLLOW(X):** is the set of terminals that can immediately follow X.

Basic idea: Assume error when parsing $X \in N$. Repeat $LA = nextToken()$ until

- $LA \in FIRST(X) \Rightarrow$ We can restart parsing X
- $LA \in \text{FOLLOW}(X) \Rightarrow$ We can start parsing the next nonterminal

Procedure error(stack,LA,input)

$X = stack.top()$
repeat
  $LA = input.nextToken()$
  if $LA \in \text{FIRST}(X)$ **then**
    return
  else if $LA \in \text{FOLLOW}(X)$ **then**
    $stack.pop()$
    return
  end if
until $LA = EOF$
We couldn't recover - stop parsing!

Parse Table for Arithmetic Expressions

	id	$+$	$*$	$($	$)$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$	
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$	

Simulation: parsing $)idid + id\$$

Input	LA	Stack	Error recovery
$idid + id\$$	$)$	E	error $\Rightarrow LA = id \Rightarrow LA \in FIRST(E) \Rightarrow return$
$id + id\$$	id	E	
...	...	...	continue until ...
$id + id\$$	id	$idT'E'$	
$+ id\$$	id	$T'E'$	error $\Rightarrow LA = + \Rightarrow LA \in FIRST(T') \Rightarrow return$
$id\$$	$+$	$T'E'$	no more troubles ...

Bottom-up Parsing: Intro

Consider the grammar

$$(1) \quad S \rightarrow aABe \qquad (2) \quad A \rightarrow b \qquad (3) \quad A \rightarrow Abc \qquad (4) \quad B \rightarrow d$$

- Using a rightmost derivation we can show that *abbcde* is in the language

$$S \stackrel{1}{\Longrightarrow} aABe \stackrel{4}{\Longrightarrow} aAde \stackrel{3}{\Longrightarrow} aAbcde \stackrel{2}{\Longrightarrow} abbcde$$

  This is a top-down approach since we start from the start symbol S (the syntax tree root) and work our way down to the tokens *abbcde* (the leaves of the syntax tree).
- Problem: What production to use.

Bottom-up: Intro (II)

$$(1) \quad S \rightarrow aABe \qquad (2) \quad A \rightarrow b \qquad (3) \quad A \rightarrow Abc \qquad (4) \quad B \rightarrow d$$

- Bottom-up approaches starts with the leaves and uses the grammar productions to reduce the input to the start symbol S.

$$abbcde \xrightarrow{2} aAbcde \xrightarrow{3} aAde \xrightarrow{4} aABe \xrightarrow{1} S$$

- That is, bottom-up parsing is a backward rightmost derivation. (More difficult for humans?)
- Problem: What production to use and where to apply it.

Shift-Reduce Parsing – Intro

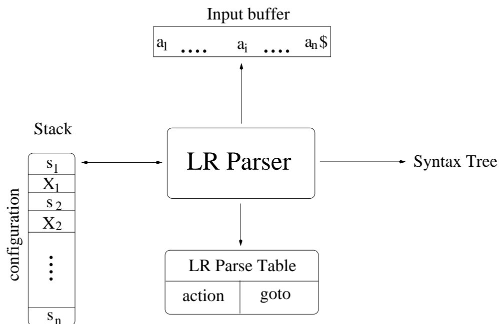
Grammar: $E \rightarrow E + E \mid E * E \mid id$

Stack (top $\rightarrow$)	Input	Action
	$id + id * id$	shift
id	$+id * id$	reduce by $E \rightarrow id$
E	$+id * id$	shift
$E+$	$id * id$	shift
$E + id$	$*id$	reduce by $E \rightarrow id$
$E + E$	$*id$	shift
$E + E*$	id	shift
$E + E * id$		reduce by $E \rightarrow id$
$E + E * E$		reduce by $E \rightarrow E * E$
$E + E$		reduce by $E \rightarrow E + E$
E		*accept*

Note

- The shift with $E + E$ on the stack is not obvious from the given grammar. (We look into the stack when doing the reductions $E \rightarrow E + E \mid E * E$)
- Hence, we need rules to resolve shift/reduce conflicts and ways to use the stack content information.

LR Parsing



- The stack keeps the current **configuration** consisting of symbols and **states**.
- The parse table contains two tables. One called *actions*$[s, a]$ saying what action (shift or reduce) to take given state s and lookahead a. One called *goto*$[s, X]$ saying what state to goto given symbol X and state s.

Configurations

- A configuration is a sequence of the form

$$s_0 X_1 s_1 X_2 s_2 \ldots X_n s_n$$

  where s_n is on top. Each X_i is a grammar symbol (terminal or non-terminal) and each s_i is a state.
- Each state summarizes the information contained in the stack below it.
  $\Rightarrow$ We have a state transition diagram (finita automaton) working in the background.
- The combination of the top state s_n and the current lookahead (k tokens in a LR(k) parser) are used to index the parse table.

Example: Expression Grammar

(1)	$E \rightarrow E + T$				(4)	$T \rightarrow F$	
(2)	$E \rightarrow T$				(5)	$F \rightarrow (E)$	
(3)	$T \rightarrow T * F$				(6)	$F \rightarrow id$	

Notice, non-ambiguous and left-recursive

Parse Table for Expression Grammar

state	*id*	+	*	(	)	$	*E*	*T*	*F*
0	*s*5			*s*4			*g*1	*g*2	*g*3
1		*s*6				*acc*			
2		*r*2	*s*7		*r*2	*r*2			
3		*r*4	*r*4		*r*4	*r*4			
4	*s*5			*s*4			*g*8	*g*2	*g*3
5		*r*6	*r*6		*r*6	*r*6			
6	*s*5			*s*4				*g*9	*g*3
7	*s*5			*s*4					*g*10
8		*s*6			*s*11				
9		*r*1	*s*7		*r*1	*r*1			
10		*r*3	*r*3		*r*3	*r*3			
11		*r*5	*r*5		*r*5	*r*5			

s*n*	shift	Advance input one token (i.e. push lookahead on stack), push state *n* on stack
r*k*	reduce	Reduce by rule k ($A \rightarrow \beta$), pop $2 * \lvert\beta\rvert$ symbols of the stack. if s' on top, push A and then push state $goto[s', A]$
acc	accept	Stop parsing, report succes

Parsing *id* + *id*$

Stack	Input	Rules
0	$id + id\$$	$s5$
0 id 5	$+id\$$	$r6, g3$
0 F 3	$+id\$$	$r4, g2$
0 T 2	$+id\$$	$r2, g1$
0 E 1	$+id\$$	$s6$
0 E 1 $+$ 6	$id\$$	$s5$
0 E 1 $+$ 6 id 5	$\$$	$r6, g3$
0 E 1 $+$ 6 F 3	$\$$	$r4, g9$
0 E 1 $+$ 6 T 9	$\$$	$r1, g1$
0 E 1	$\$$	acc

Notice:

- *reduce* and *goto* come in pairs.
- On the stack, we always have:

  `state symbol state symbol state symbol ... state`
- The reduction sequence backwards: $r1$, $r4$, $r6$, $r2$, $r4$, $r6$ gives a right-most derivation.

LR Properties

- LR parsers have “seen” the input before it makes a reduction. This information is stored in the top state. This is the reason why LR parsers are more powerful than LL parsers.
- Ambiguities leads to duplicate entries in the parse table. These entries represents shift/reduce and reduce/reduce conflicts.
- The ambiguous grammar $E \rightarrow E + E \mid E * E \mid id$ may find itself in the following position

Stack	Input
$E + E$	$*id$

while parsing the string $id + id * id$. Should it reduce using $E \rightarrow E + E$ or shift?

- Similarly

Stack	Input	
$E + E$	$+id$	reduce or shift?

- The above problem is due to lacking information about operator priority (* has higher than $+$) and associativity (left in both cases) in the grammar.

A Highly Ambiguous Grammar (WA 3)

(1) $E \rightarrow id$ (2) $E \rightarrow E * E$ (3) $E \rightarrow E + E$ (4) $E \rightarrow (E)$

state	*id*	+	*	(	)	$	*E*
1	*s*2			*s*3			*g*6
2		*r*1	*r*1		*r*1	*r*1	
3	*s*2			*s*3			*g*4
4		*s*7	*s*9		*s*5		
5		*r*4	*r*4		*r*4	*r*4	
6		*s*7	*s*9			*acc*	
7	*s*2			*s*3			*g*8
8		*s*7, *r*3	*s*9, *r*3		*r*3	*r*3	
9	*s*2			*s*3			*g*10
10		*s*7, *r*2	*s*9, *r*2		*r*2	*r*2	

Reasons for ambiguity

1. The grammar doesn't show that * has higher priority than +.
2. The grammar doesn't show that both * and + are left associative

LR Advantages

- LR parsers can recognize virtually all *programming constructs* for which non-ambiguous CFG:s can be written.
- LR(1) is sufficient in most cases. Most programming languages can be parsed with LR(1) without any fancy rewritings.
- LALR(1) is a slightly less powerful parser than LR(1) with considerably smaller parse tables. LALR(1) is common in parser generator tools. For example, both Yacc and Bison are LALR(1).
- LR can be implemented just as efficiently as any other parsing method.
- LR Grammars are a proper superset of the LL grammars.

LR Disadvantages

- Deriving the LR parse table is very difficult. However, it can be constructed algorithmicly. (See pages 215-257 in book bu Aho, Sethi, and Ullman.)

Written Assignment 3: Resolving Ambiguity

The LR parse table on slide 24 has duplicate entries. The reason is that the grammar doesn't take operator priority and associativity into account. Your job is to remove one of the entries. **Assignments**

1. Give an example where the shift/reduce conflict $s9/r2$ in state 10 comes into play. What is the reason for this particular conflict?
2. Remove the ambiguity from the parse table by removing one of the rules in each entry containing duplicate rules.
3. Use the corrected table to parse the input string $id * (id + id + id)$.
4. Write an algorithm that describes the parsing process when using an unambiguous parse table. Use the *Algorithm for table driven LL-parsing* (slide 13) as an example of what an algorithm may look like. Remember to clearly define all operations and notations that you use.

Deadline: 2007-03-04

Formal Languages: Revisited

- **Definition:**
  A *formal language* L over Σ, denoted $L(\Sigma)$, is a subset of Σ^*
  $\Rightarrow$ a language $L(\Sigma)$ is a well-defined set of strings formed by the symbols in Σ.
- Languages are often infinite sets.
- Example: $\{ab, aabb, aaabbb, aaaabbbb, \ldots\}$ is a language over $\Sigma = \{a, b\}$.

Common Language Related Problems

- **Definition:** Define a language L over some alphabet Σ.
  (Remember that languages are often infinite sets.)
- **Recognition:** Verify that $s \in L$ for a given string s.
  (Recognition $\longrightarrow$ yes/no)
- **Parsing:** Recognition + building a language structure.
  (e.g. token sequences and syntax trees.)

Regular Languages and Expressions

Regular Expressions: Basic Operators

- $a|b \Rightarrow a$ or b.
- $ab \Rightarrow a$ followed by b.
- $a^* \Rightarrow$ zero or more a.

Regular Languages

- Each alphabet Σ defines a set of regular expressions $RE(\Sigma)$.
- Each $r \in RE(\Sigma)$ defines a regular language $L(r) \subset \Sigma^*$.
- Example, $r = (aa|bb)*$ defines the infinite language
  $$L(r) = \{\varepsilon, aa, bb, aaaa, aabb, bbaa, bbbb, aaaaaa, \ldots\}$$
- Think of a regular expression r as a pattern. Every string that matches the pattern is a part of the regular language $L(r)$

Lexical Analysis (Formally)

- $\Sigma =$ The set of all ASCII characters.
- $\Sigma^* =$ The set of all character sequences.
- $L(r) \subset \Sigma^*$ The set of all lexically correct programs.

Lexical Analysis Problems

- Language definition by regular expressions.
  (Defines a **regular language**.)
- Input string to parse: The program as a sequence of characters.
- Resulting language structure: A sequence of tokens

Context-free Languages

A grammar G with start symbol S defines a set of terminal strings $L(G)$ as:

$$L(G) = \{s \mid S \stackrel{+}{\Rightarrow} s\}$$

(All strings that can be derived from the start symbol.)
We say that $L(G)$ is the **context-free language** generated by G.
Two grammars G_1 and G_2 are equivalent, denoted $G_1 = G_2$, iff $L(G_1) = L(G_2)$.

Syntax Analysis (Formally)

- $\Sigma =$ The set of all tokens.
- $\Sigma^* =$ The set of all token sequences.
- $L(g) \subset \Sigma^*$ The set of all syntactically correct programs.

Syntax Analysis Problems

- Language definition by context-free grammar.
- Input string to parse: The program as a sequence of tokens.
- Resulting language structure: A syntax tree

Context-free vs Regular Languages

- The regular languages are a subset of the context free languages.
- Two well-known languages that can't be described by regular expressions:
  1. The strings with n symbols x followed by n symbols y.

     $$\{x^n y^n \mid n > 1\} \qquad S \rightarrow xSy \mid xy$$

  2. Matching brackets: $B \rightarrow (B) \mid BB \mid \varepsilon$
- We use regular expressions in the lexical analysis because its sufficient and fast.

Limitations of Context Free Languages

- $\{x^n \mid n \text{ is a positive power of } 2\}$
- A variable should be defined before use
- intVariable + stringVariable is not allowed

The Chomsky Hierarchy of Languages

Type 3: Regular Languages

- Grammar: $A \rightarrow a \mid bB$ $\qquad a, b \in T,\ A, B \in N$
- Recognizing Machine: Finite Automata

Type 2: Context-Free Languages

- Grammar: $A \rightarrow \omega$ $\qquad A \in N,\ \omega \in (N \cup T)^*$
- Recognizing Machine: Push-down Automata

Type 1: Context-Sensitive Languages

- Grammar: $\alpha A \beta \rightarrow \alpha \omega \beta$ $\qquad A \in N,\ \alpha, \beta, \omega \in (N \cup T)^*$
- Recognizing Machine: Linear Bounded Automata

Type 0: Natural Languages

- Grammar: $\alpha \rightarrow \omega$ $\qquad \alpha, \omega \in (N \cup T)^*$
- Recognizing Machine: Turing Machine