Compiler Design
Syntax Analysis
Bottom-Up Parsing

conf. dr. ing. Ciprian-Bogdan Chirila

chirila@cs.upt.ro

http://www.cs.upt.ro/~chirila

Outline

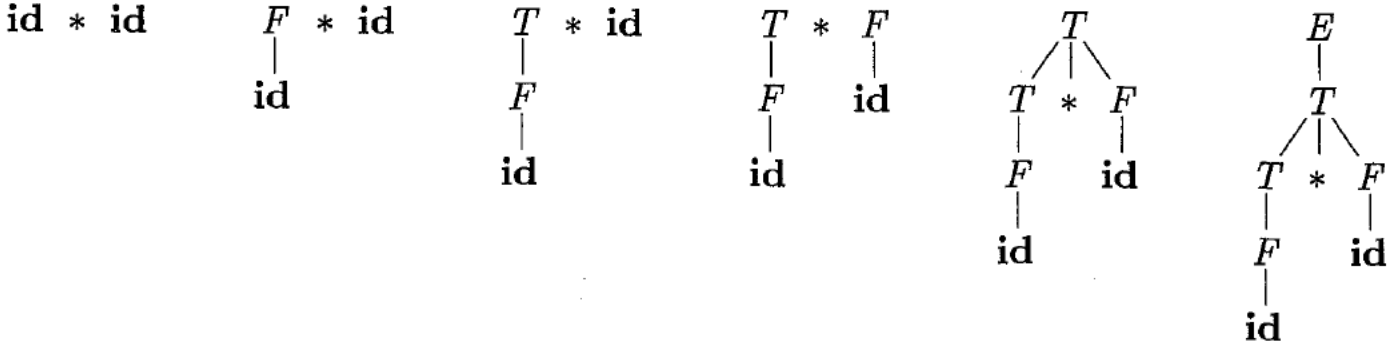
- Reductions
- Handle Pruning
- Shift-Reduce Parsing
- Conflicts During Shift-Reduce Parsing

Introduction

- the construction of a parse tree
  - beginning at the leaves (bottom)
  - working up towards the root (top)
- general style of bottom-up parsing
  - shift-reduce parsing
- large class of grammars for which shift-reduce parsers can be built are LR grammars
- LR parsers
  - difficult to be built by hand
  - generators build efficient LR parsers

Example

- bottom-up parse for id*id



- E->E+T|T
- T->T*F|F
- F->id | (E)

Reductions

- bottom-up parsing = reducing a string w to the starting symbol of the grammar
- reduction step consists in
  - specific substring matching the body of a production is replaced by a non-terminal of that production
- key decisions
  - when to reduce
  - what production to apply

Reductions

- id * id
  - leftmost id is reduced to F using F->id
- F * id
  - F is reduced to T
- T * id
  - T can be reduced to E
  - or
  - id can be reduced to F
- T * F
  - T*F is reduced to T
- T
  - T is reduced to E
- E

- roots of subtrees in the example

Reductions

- the reverse step of derivation
  - a non-terminal is replaced by the body of one of its productions
- bottom-up parsing
  - to construct derivation in reverse
  - using the rightmost derivation
- E=>T=>T*F=>T*id=>F*id=>id*id

Handle Pruning

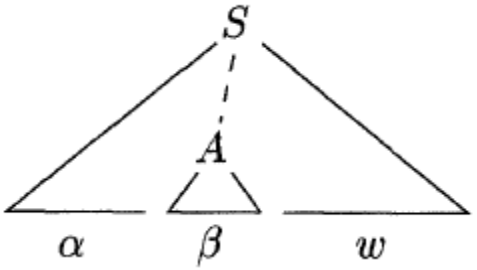
- left to right bottom-up parsing constructs a rightmost derivation in reverse
- handle = substring that matches the body of a production
- handle reduction = a step in the reverse of rightmost derivation

Handles During a Parse id_1*id_2

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$E \to T * F$

- E->T ,T is not a handle in $T*id_2$
- if we replace T by E
  - we get $E*id_2$ which can not be derived from E
- leftmost substring that matches production body need not to be a handle

Handles



- if $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha \beta w$
  - then A->β in the position following α is a handle of αβw
- the handle of right-sentential form γ is a production A->β and a position of γ where β may be found
  - such that replacing β at that position by A produces the previous right sentential form in a rightmost derivation of γ

Handles

- the string w to the right of the handle must contain only terminal symbols
- the body β is the handle
- if the grammar is ambiguous
  - “the handle” becomes “a handle”
- else
  - every right-sentential form has exactly one handle

Handles

- rightmost derivation = handle pruning
- w is the sentence of the grammar
- $w=\gamma_n$ where γ_n is the n-th right-sentential form of some unknown rightmost derivation
- $S=\gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \dots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$
- to rebuild this derivation in reverse order
  - locate handle β_n in γ_n by production of $A_n \rightarrow \beta_n$ to get right-sentential form γ_{n-1}
  - handles must be found with specific methods
  - repeat the process until the start symbol S is found
  - reverse of reductions = rightmost derivation

Shift-Reduce Parsing

- is a form of bottom-up parsing
- the stack holds grammar symbols
- the input buffer holds the rest of the string to be parsed
- the handle appears on the top of the stack
- we mark by $
  - the bottom of the stack
  - the right end of the input
- initially
  - stack input
  - $ w$

Shift-Reduce Parsing

- left-to-right scan of the input string
- shift zero or more input symbols onto the stack
- reduce a string β of grammar symbols on the top of the stack to the appropriate production
- stop when
  - error is detected
  - both
    - the stack contains the start symbol
    - the input is empty

Configurations of a shift-reduce parser on id_1*id_2

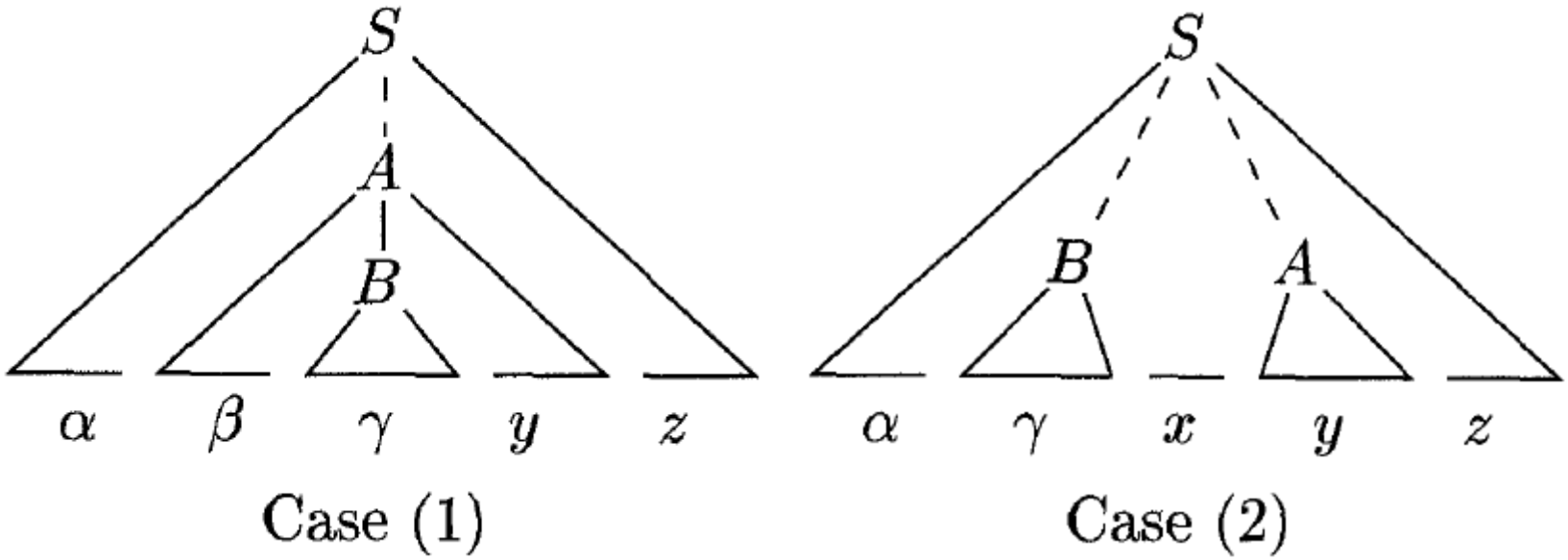
STACK	INPUT	ACTION
\$	$\mathbf{id}_1 * \mathbf{id}_2$ \$	shift
\$ $\mathbf{id}_1$	$* \mathbf{id}_2$ \$	reduce by $F \rightarrow \mathbf{id}$
\$ F	$* \mathbf{id}_2$ \$	reduce by $T \rightarrow F$
\$ T	$* \mathbf{id}_2$ \$	shift
\$ $T *$	$\mathbf{id}_2$ \$	shift
\$ $T * \mathbf{id}_2$	\$	reduce by $F \rightarrow \mathbf{id}$
\$ $T * F$	\$	reduce by $T \rightarrow T * F$
\$ T	\$	reduce by $E \rightarrow T$
\$ E	\$	accept

Possible Actions

- shift
  - the next symbol onto the top of the stack
- reduce
  - the right end of the string when it is on the top of the stack
  - locate the left end of the string
  - decide with what non-terminal to replace the string
- accept
  - announce successful completion of parsing
- error
  - discover a syntax error
  - call an error recovery routine

Two Possible Cases

- (1) $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha Az \underset{rm}{\Rightarrow} \alpha\beta Byz \underset{rm}{\Rightarrow} \alpha\beta\gamma yz$
- (2) $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha BxAz \underset{rm}{\Rightarrow} \alpha Bxyz \underset{rm}{\Rightarrow} \alpha\gamma xyz$



Case 1 in Reverse

STACK	INPUT
$αβγ	yz$
$αβB	yz$
$αβBy	z$
$αA	z$
$αAz	$
$S	$

Case 2 in Reverse

STACK	INPUT
$αγ	xyz$
$αB	xyz$
$αBxy	z$
$αBxA	z$
$αBxAz	$
$S	$

Conclusion

- in both cases
- after making a reduction
- the parser had to shift zero or more symbols to get the next handle on the stack
- the handle will appear always on the top of the stack !!!
- the handle is never found into the stack !!!

Conflicts During Shift-Reduce Parsing

- shift/reduce conflicts
- reduce/reduce conflicts
- not LR(k) grammars
- k number of symbols of lookahead on the input
- grammars used in compiling LR(1)

Example 1

- stmt-> if expr then stmt

| if expr then stmt else stmt

| other

Stack	Input
…if expr then stmt	else…$

- shift/reduce conflict
  - to reduce "if expr then stmt" to stmt
  - shift else, shift another stmt and reduce "if expr then stmt else stmt" to stmt
- to favor shifting

Example 2

1. stmt->id (parameter_list)
2. stmt->expr := expr
3. parameter_list->parameter_list , parameter
4. parameter_list->parameter
5. parameter->id
6. expr->id (expr_list)
7. expr->id
8. expr_list->expr_list , expr
9. expr_list->expr

Example 2

- procedure calls = names and parentheses
- arrays have the same syntax
- statement p(i,j) appears as id(id,id)
- STACK INPUT
- …id(id ,id)…
- to reduce with
  - 5 if p is a procedure
  - 7 if p is an array
- STACK INPUT
- …procid(id ,id)…

Bibliography

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman – Compilers, Principles, Techniques and Tools, Second Edition, 2007