Programming Language & Compiler

Bottom-up Parser (II)

Hwansoo Han

LR(k) Items

- **A state of parser == a set of LR(k) items**
- **An LR(*k*) item is a pair [*P*, δ], where**
  - *P* is a production $A \rightarrow \beta$ with a • at some position in the *rhs*
  - δ is a lookahead string of length $\leq k$ (words/tokens or EOF)

LR(k) Items

LR(1) items

- The • in an item indicates the position of the top of the stack
- $[A\to\bullet\beta\gamma,\underline{a}]$ means that the input seen so far is consistent with the use of $A\to\beta\gamma$ immediately after the symbol on top of the stack. *(possibility)*
- $[A\to\beta\bullet\gamma,\underline{a}]$ means that the input seen so far is consistent with the use of $A\to\beta\gamma$ at this point, ***and*** that the parser has already recognized β. *(partially complete)*
- $[A\to\beta\gamma\bullet,\underline{a}]$ means that the parser has seen $\beta\gamma$, ***and*** that a lookahead symbol of $\underline{a}$ is consistent with reducing to A. *(complete)*

Computing goto()

- ***goto(s,x)* computes the state that the parser would reach if it recognized an *x* while in state *s***
  - *goto(* { $[A \rightarrow \beta \bullet X\delta, \underline{a}]$ }, *X)* produces $[A \rightarrow \beta X \bullet \delta, \underline{a}]$ *(easy part)*
  - Should also includes *closure(* $[A \rightarrow \beta X \bullet \delta, \underline{a}]$ *)* *(fill out the state)*

- **The algorithm**

goto(s, X)
    moved $\leftarrow \emptyset$
    for each item $[A \rightarrow \beta \cdot X\delta, \underline{a}] \in s$
        moved $\leftarrow$ *moved* $\cup$ $[A \rightarrow \beta X \cdot \delta, \underline{a}]$
    return closure(moved)

- Not a fixed-point method!
- Straightforward computation
- Uses *closure* ( )

goto() moves forward

Computing closure()

- ***Closure(s)* adds all the items implied by items already in *s***
  - Any item $[A \rightarrow \beta \bullet B\delta, \underline{a}]$ implies $[B \rightarrow \bullet\tau, x]$ for each production with B on the *lhs*, and each $x \in$ FIRST($\delta\underline{a}$)
  - Since $\beta B\delta$ is valid, any way to derive $\beta B\delta$ is valid, too
- **The algorithm**

closure(s)
while (s is still changing)
for each item $[A \rightarrow \beta \cdot C\delta, \underline{a}] \in s$
for each production $C \rightarrow \tau \in P$
for each $\underline{b} \in$ FIRST($\delta\underline{a}$) // δ might be ε
$s \leftarrow s \cup [C \rightarrow \cdot\tau, \underline{b}]$

- Classic fixed-point method
- Halts because $s \subset$ ITEMS
- Worklist version is faster

Closure "fills out" a state

LR(1) Table Construction

- **High-level overview** ***(Algorithm)***

1 **Build the *canonical collection* of sets of LR(1) Items, *I***

   a Begin in an appropriate state, cc_0

   - [$S' \rightarrow \cdot S$,EOF], along with any equivalent items
   - Derive equivalent items as *closure*(cc_0)

   b Repeatedly compute, for each cc_k, and each X, $goto(cc_k, X)$

   - If the set is not already in the collection, add it
   - Record all the transitions created by *goto( )*

   This eventually reaches a fixed point

2 **Fill in the table from the collection of sets of LR(1) items**

The canonical collection completely encodes the transition diagram for the handle-finding *DFA*

Canonical Collection

- **Building CC : all possible states**
  - Start from cc_0 = *closure*([$S' \rightarrow S$,EOF])
  - Repeatedly construct new states, until all are found

- **The algorithm**

```
cc0 ← closure( [S'→S,EOF] )
CC  ← { cc0 }
k  ← 1
while ( CC is still changing )
  for each ccj ∈ CC and for each x ∈ ( T ∪ NT )
    cck ← goto(ccj,x)
    record ccj → cck on x
    if  cck ∉ CC then
      CC ← CC ∪ cck     // new state in DFA
      k ← k + 1
```

- Fixed-point computation
- Loop adds to CC
- $CC \subseteq 2^{ITEMS}$, so CC is finite

Worklist version is faster

Example (grammar & sets)

- **Simplified, <u>right</u> recursive expression grammar**

Goal → *Expr*
Expr → *Term* - *Expr*
Expr → *Term*
Term → *Factor* * *Term*
Term → *Factor*
Factor → <u>*ident*</u>

Symbol	FIRST
Goal	{ <u>ident</u> }
Expr	{ <u>ident</u> }
Term	{ <u>ident</u> }
Factor	{ <u>ident</u> }
-	{ - }
*	{ * }
<u>ident</u>	{ <u>ident</u> }

Example (building the collection)

- **Initialization Step**

***cc_0* ← *closure(* {** [*Goal* → •*Expr* , EOF] } *)*

{ [*Goal* → • *Expr* , EOF], [*Expr* → • *Term* – *Expr* , EOF],
[*Expr* → • *Term* , EOF], [*Term* → • *Factor* * Term , EOF],
[*Term* → • *Factor* * *Term* , –], [*Term* → • *Factor* , EOF],
[*Term* → • *Factor* , –], [*Factor* → • <u>ident</u> , EOF],
[*Factor* → • <u>ident</u> , –], [*Factor* → • <u>ident</u> , *] }

Add **cc_0** ***to a set of states, CC* ← {*cc_0* }**

Example (building the collection)

Iteration 1

$cc_1 \leftarrow goto(cc_0, Expr)$

$cc_2 \leftarrow goto(cc_0, Term)$

$cc_3 \leftarrow goto(cc_0, Factor)$

$cc_4 \leftarrow goto(cc_0, \underline{\text{ident}})$

Iteration 2

$cc_5 \leftarrow goto(cc_2, -)$

$cc_6 \leftarrow goto(cc_3, *)$

Iteration 3

$cc_7 \leftarrow goto(cc_5, Expr)$, # *Term, Factor,* <u>ident</u> $\Rightarrow$ *existing states*

$cc_8 \leftarrow goto(cc_6, Term)$ # *Factor,* <u>ident</u> $\Rightarrow$ *existing states*

Example (Summary)

cc_0 : { [*Goal* → • *Expr* , $\underline{\text{EOF}}$], [*Expr* → • *Term* - *Expr* , $\underline{\text{EOF}}$], [*Expr* → • *Term* , $\underline{\text{EOF}}$],
[*Term* → • *Factor* * *Term* , $\underline{\text{EOF}}$], [*Term* → • *Factor* * *Term* , -],
[*Term* → • *Factor* , $\underline{\text{EOF}}$], [*Term* → • *Factor* , -],
[*Factor* → • $\underline{\text{ident}}$, $\underline{\text{EOF}}$], [*Factor* → • $\underline{\text{ident}}$, -], [*Factor*→ • $\underline{\text{ident}}$, *] }

cc_1 : { [*Goal* → *Expr* •, $\underline{\text{EOF}}$] }

cc_2 : { [*Expr* → *Term* • - *Expr* , $\underline{\text{EOF}}$], [*Expr* → *Term* •, $\underline{\text{EOF}}$] }

cc_3 : { [*Term* → *Factor* • * *Term* , $\underline{\text{EOF}}$],[*Term* → *Factor* • * *Term* , -],
[*Term* → *Factor* •, $\underline{\text{EOF}}$], [*Term* → *Factor* •, -] }

cc_4 : { [*Factor* → $\underline{\text{ident}}$ •, $\underline{\text{EOF}}$],[*Factor* → $\underline{\text{ident}}$ •, -], [*Factor* → $\underline{\text{ident}}$ •, *] }

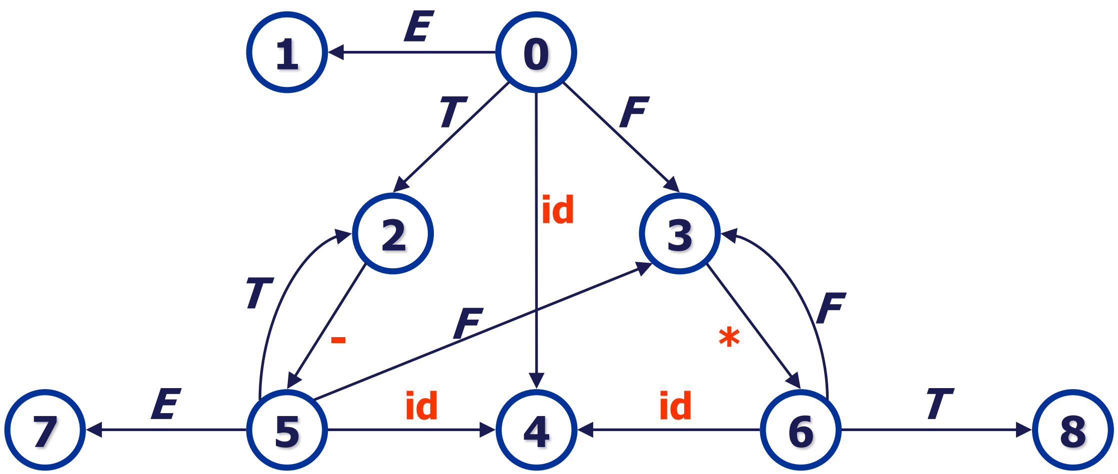
cc_5 : { [*Expr* → *Term* - • *Expr* , $\underline{\text{EOF}}$],
[*Expr* → • *Term* - *Expr* , $\underline{\text{EOF}}$], [*Expr* → • *Term* , $\underline{\text{EOF}}$],
[*Term* → • *Factor* * *Term* , $\underline{\text{EOF}}$], [*Term* → • *Factor* * *Term* , -],
[*Term* → • *Factor* , $\underline{\text{EOF}}$], [*Term* → • *Factor* , -],
[*Factor* → • $\underline{\text{ident}}$, $\underline{\text{EOF}}$], [*Factor* → • $\underline{\text{ident}}$, -], [*Factor* → • $\underline{\text{ident}}$, *] }

Example (Summary)

cc_6 : { [*Term* → *Factor* * • *Term* , EOF], [*Term* → *Factor* * • *Term* , -],
[*Term* → • *Factor* * *Term* , EOF], [*Term* → • *Factor* * *Term* , -],
[*Term* → • *Factor* , EOF], [*Term* → • *Factor* , -],
[*Factor* → • ident , EOF], [*Factor* → • ident , -], [*Factor* → • ident , *] }

cc_7 : { [*Expr* → *Term* - *Expr* •, EOF] }

cc_8 : { [*Term* → *Factor* * *Term* •, EOF], [*Term* → *Factor* * *Term* •, -] }



Example (Summary)

- **The *goto()* Relationship *(from the construction)***

State	Expr	Term	Factor	-	*	Ident
0	1	2	3			4
1						
2				5		
3					6	
4						
5	7	2	3			4
6		8	3			4
7						
8						

Filling in the ACTION and GOTO Tables

- **The algorithm**

```
for each set cc_x ∈ CC
   for each item i ∈ cc_x
      if  i is [A→β •aγ,b] and goto(cc_x,a) = cc_k , a ∈ T
          then ACTION[x,a] ← "shift k"
      else if  i is [S'→S •,EOF]
          then ACTION[x ,EOF] ← "accept"
      else if  i is [A→β •,a]
          then ACTION[x,a] ← "reduce A→β"
   for each nt ∈ NT
      if  goto(cc_x ,nt) = cc_k
          then GOTO[x,nt] ← k
```

***x* is the current state number**

***k* is the next state number**

- **Ignores many items where the • precedes non-terminal**
  - *closure( )* instantiates items where • precedes FIRST(X)
    $[A \rightarrow \beta \cdot X\gamma, \underline{a}\,]$ forces to have $[X \rightarrow \cdot \underline{b}\delta, \underline{c}\,]$,
    where $\underline{b} \in$ FIRST(X), $\underline{c} \in$ FIRST($\gamma\underline{a}$), $X \Rightarrow^* \underline{b}\delta$

Example (Filling in the tables)

- **The algorithm produces the following table**

	ACTION				GOTO		
	Ident	-	*	EOF	Expr	Term	Factor
0	s 4				1	2	3
1				acc			
2		s 5		r 3			
3		r 5	s 6	r 5			
4		r 6	r 6	r 6			
5	s 4				7	2	3
6	s 4					8	3
7				r 2			
8		r 4		r 4			

Plugs into the skeleton LR(1) parser

What can go wrong?

- **What if set *s* contains [*A*→β•<u>a</u>γ,<u>b</u>] and [*B*→β•,<u>a</u>] ?**
  - First item generates "shift", second generates "reduce"
  - Both define ACTION[s,<u>a</u>] — cannot do both actions
  - This is a fundamental ambiguity, called a *shift/reduce error*
  - Modify the grammar to eliminate it *(if-then-else)*
  - Shifting will often resolve it correctly
- **What if set *s* contains [*A*→γ•, <u>a</u>] and [*B*→γ•, <u>a</u>] ?**
  - Each generates "reduce", but with a different production
  - Both define ACTION[s,<u>a</u>] — cannot do both reductions
  - This fundamental ambiguity is called a *reduce/reduce error*
  - Modify the grammar to eliminate it *(PL/I's overloading of (...))*

- ***In either case, the grammar is not LR(1)***

Shrinking the Tables

- **Combine terminals - <u>number</u> & <u>identifier</u>, <u>+</u> & <u>-</u>, <u>*</u> & <u>/</u>**
  - Directly removes a column, may remove a row
  - For expression grammar, 198 (vs. 384) table entries
- **Combine rows or columns**
  - Implement identical rows once & remap states
  - Requires extra indirection on each lookup of ACTION & GOTO
  - Use separate mapping for ACTION & for GOTO
- **Use another construction algorithm**
  - Both LALR and SLR produce smaller tables with LR(0) items
  - Implementations are readily available

SLR vs. LR(1) vs. LALR

- **SLR parsing**
  - States are constructed from LR(0) items
  - State transitions based on symbols(X) right after •

    $A \rightarrow \beta \cdot X\gamma$
- **LR(1) parsing**
  - States are constructed from LR(1) items
  - State transitions based on symbols(X) right after •

    $A \rightarrow \beta \cdot X\gamma$, s
- **LALR parsing**
  - States are constructed with LR(0) items and refine states if different actions are needed depending on look-ahead symbol

    Or
  - states are constructed with LR(1) items and merge states if cores are the same and the same action is needed for the merged look-ahead symbols

LR(k) vs. LL(k)

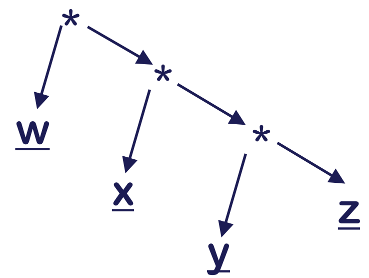
- **Finding Reductions**
  - LR(*k*) $\Rightarrow$ Each reduction in the parse is detectable with
    1. the completed left context,
    2. the reducible phrase, itself, and
    3. the *k* terminal symbols to its right
  - LL(*k*) $\Rightarrow$ Parser must select the reduction based on
    1. The completed left context
    2. The next *k* terminals
- **Thus, LR(*k*) examines more context**

"... in practice, programming languages do not actually seem to fall in the gap between LL(1) languages and deterministic languages"

[*J.J. Horning, "LR Grammars and Analysers", in Compiler Construction, An Advanced Course, Springer-Verlag, 1976*]

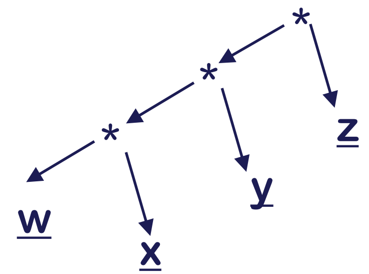
Left Recursion vs. Right Recursion

- **Right recursion**
  - Required for termination in top-down parsers
  - Uses (on average) more stack space
  - Produces right-associative operators



w * (x * (y * z))

- **Left recursion**
  - Works fine in bottom-up parsers
  - Limits required stack space
  - Produces left-associative operators



((w * x) * y) * z

- **Rule of thumb**
  - Left recursion for bottom-up parsers
  - Right recursion for top-down parsers

Hierarchy of Context-Free Languages