RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Syntax Analysis
Part 5

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Roadmap for the remainder of the course

- Fourth homework:
  Due Monday, March 28
- Project #2 – Bottom-up parser and compiler
  Will be posted next week, due April 13 (tentative)
- Project #3 – Peephole optimizer for ILOC
  Will be posted April 13, due May 2 (tentative)
- Second midterm on Wednesday, April 6 (60 minutes in class)
- Final exam on May 10 (60 minutes at assigned location)
- At least 3 more homeworks

Bottom-up Parsing (Syntax Analysis)

EAC Chapters 3.4

LR(1) Skeleton Parser

```
stack.push(INVALID); stack.push(s0);
not_found = true;
token = scanner.next_token();
do while (not_found) {
        s = stack.top();
        if ( ACTION[s,token] == "reduce A→β" ) then {
                stack.popnum(2*|β|); // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);
        stack.push(GOTO[s,A]);
        }
        else if ( ACTION[s,token] == "shift si" ) then {
                stack.push(token); stack.push(si);
                token ← scanner.next_token();
        }
        else if ( ACTION[s,token] == "accept"
                                & token == EOF )
                then not_found = false;
        else report a syntax error and recover;
}
report success;
```

The skeleton parser

- uses ACTION & GOTO tables
- does |*words*| shifts
- does |derivation| reductions
- does 1 accept

Building LR(1) Parsers

How do we generate the ACTION and GOTO tables?

- Use the grammar to build a model of the DFA
- Use the model to build ACTION & GOTO tables
- If construction succeeds, the grammar is LR(1)

Terminal or non-terminal

The Big Picture

- Model the state of the parser
- Use two functions *goto(s, X)* and *closure(s)*
  - → *goto()* is analogous to *move()* in the subset construction
  - → *closure()* adds information to round out a state
- Build up the states and transition functions of the DFA
- Use this information to fill in the ACTION and GOTO tables

LR(k) items

The LR(1) table construction algorithm uses LR(1) items to represent valid configurations of an LR(1) parser

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 ≤ k (words or EOF)

The • in an item indicates the position of the top of the stack

LR(1):

[$A \rightarrow \bullet\beta\gamma$,<u>a</u>] means that the input seen so far is consistent with the use of $A \rightarrow \beta\gamma$ immediately after the symbol on top of the stack

[$A \rightarrow \beta\bullet\gamma$,<u>a</u>] means that the input seen so far is consistent with the use of $A \rightarrow \beta\gamma$ at this point in the parse, <u>*and*</u> that the parser has already recognized β.

[$A \rightarrow \beta\gamma\bullet$,<u>a</u>] means that the parser has seen $\beta\gamma$, <u>*and*</u> that a lookahead symbol of <u>a</u> is consistent with reducing to A.

LR(1) Items

The production $A \rightarrow \beta$, where $\beta = B_1 B_1 B_1$ with lookahead $\underline{a}$, can give rise to 4 items

$[A \rightarrow \bullet B_1 B_2 B_3, \underline{a}]$, $[A \rightarrow B_1 \bullet B_2 B_3, \underline{a}]$, $[A \rightarrow B_1 B_2 \bullet B_3, \underline{a}]$, & $[A \rightarrow B_1 B_2 B_3 \bullet, \underline{a}]$

The set of LR(1) items for a grammar is finite

What's the point of all these lookahead symbols?

- Carry them along to choose the correct reduction, *if there is a choice*
- Lookaheads are bookkeeping, unless item has • at the right end
  - → Has no direct use in $[A \rightarrow \beta \bullet \gamma, \underline{a}]$
  - → In $[A \rightarrow \beta \bullet, \underline{a}]$, a lookahead of $\underline{a}$ implies a reduction by $A \rightarrow \beta$
  - → For $\{ [A \rightarrow \beta \bullet, \underline{a}], [B \rightarrow \gamma \bullet c, \underline{b}] \}$, $\underline{a} \Rightarrow$ ***reduce*** to A; $\underline{c} \Rightarrow$ ***shift***

⇒ Limited right context is enough to pick the actions (unique, i.e., deterministic choice)

LR(1) Table Construction

High-level overview

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

  a Begin in an appropriate state, s_0

   - Assume: $S' \rightarrow S$, and S' is unique start symbol that does not occur on any RHS of a production (extended CFG - ECFG)
   - $[S' \rightarrow \cdot S, \underline{EOF}]$, along with any equivalent items
   - Derive equivalent items as *closure*(s_0)

  b Repeatedly compute, for each s_k, and each X, *goto*(s_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

RUTGERS

Computing Closures

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($\delta\underline{a}$)

The algorithm

Closure(s)
  while (s is still changing)
    $\forall$ *items* $[A \rightarrow \beta \cdot B\delta,\underline{a}] \in s$
      $\forall$ *productions* $B \rightarrow \tau \in P$
        $\forall\ \underline{b} \in$ FIRST($\delta\underline{a}$) // δ might be ε
          if $[B \rightarrow \cdot\ \tau,\underline{b}] \notin s$
            then add $[B \rightarrow \cdot\ \tau,\underline{b}]$ *to s*

- Classic fixed-point method
- Halts because $s \subset$ ITEMS

Closure "fills out" a state

Computing Gotos

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)
    new $\leftarrow \emptyset$
    $\forall$ *items* [$A \rightarrow \beta \cdot X \delta$,$\underline{a}$] $\in$ *s*
        new $\leftarrow$ *new* $\cup$ [$A \rightarrow \beta X \cdot \delta$,$\underline{a}$]
    return closure(new)

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

Goto() moves forward

Building the Canonical Collection

Start from s_0 = *closure*([*S'*→*S*, EOF])

Repeatedly construct new states, until all are found

The algorithm

$cc_0 \leftarrow$ *closure*([*S'*→ •*S*, EOF])
CC ← { cc_0 }
while (*new sets are still being added to CC*)
  for each unmarked set $cc_j \in CC$
    mark cc_j *as processed*
    for each x following a • in an item in cc_j
      temp ← *goto*(cc_j, *x*)
      if temp ∉ *CC*
        then CC ← *CC* ∪ { *temp* }
      record transitions from cc_j *to temp on x*

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

Another Example (*grammar & sets*)

Simplified, right recursive expression grammar

1: *Goal* → *Expr*
2: *Expr* → *Term* - *Expr*
3: *Expr* → *Term*
4: *Term* → *Factor* * *Term*
5: *Term* → *Factor*
6: *Factor* → *ident*

Symbol	FIRST
Goal	{ ident }
Expr	{ ident }
Term	{ ident }
Factor	{ ident }
–	{ – }
*	{ * }
ident	{ ident }

Another Example (*grammar & sets*)

1: *Goal* → *Expr*
2: *Expr* → *Term* - *Expr*
3: *Expr* → *Term*
4: *Term* → *Factor* * *Term*
5: *Term* → *Factor*
6: *Factor* → ident

Symbol	FIRST
Goal	{ ident }
Expr	{ ident }
Term	{ ident }
Factor	{ ident }
–	{ – }
*	{ * }
ident	{ ident }

s_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* → • ident , EOF],
[*Factor* → • ident , –], [*Factor* → • ident , *] }

Example (*building the collection*)

Iteration 1

$s_1 \leftarrow goto(s_0, Expr)$

$s_2 \leftarrow goto(s_0, Term)$

$s_3 \leftarrow goto(s_0, Factor)$

$s_4 \leftarrow goto(s_0, \underline{\text{ident}})$

Iteration 2

$s_5 \leftarrow goto(s_2, -)$

$s_6 \leftarrow goto(s_3, *)$

Iteration 3

$s_7 \leftarrow goto(s_5, Expr)$

$s_8 \leftarrow goto(s_6, Term)$

Example (*Summary*)

S_0 : { [*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>, *] }

S_1 : { [*Goal* → *Expr* • , EOF] }

S_2 : { [*Expr* → *Term* • - *Expr* , EOF], [*Expr* → *Term* • , EOF] }

S_3 : { [*Term* → *Factor* • * *Term* , EOF],[*Term* → *Factor* • * *Term* , -],
[*Term* → *Factor* • , EOF], [*Term* → *Factor* • , -] }

S_4 : { [*Factor* → <u>ident</u> • , EOF],[*Factor* → <u>ident</u> • , -], [*Factor* → <u>ident</u> • , *] }

S_5 : { [*Expr* → *Term* - • *Expr* , EOF], [*Expr* → • *Term* - *Expr* , EOF],
[*Expr* → • *Term* , EOF], [*Term* → • *Factor* * *Term* , -],
[*Term* → • *Factor* , -], [*Term* → • *Factor* * *Term* , EOF],
[*Term* → • *Factor* , EOF], [*Factor* → • <u>ident</u> , *],
[*Factor* → • <u>ident</u> , -], [*Factor* → • <u>ident</u> , EOF] }

Example (*Summary*)

S_6 : { [*Term* → *Factor* * • *Term* , EOF], [*Term* → *Factor* * • *Term* , –],
[*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> , *] }

S_7: { [*Expr* → *Term* – *Expr* •, EOF] }

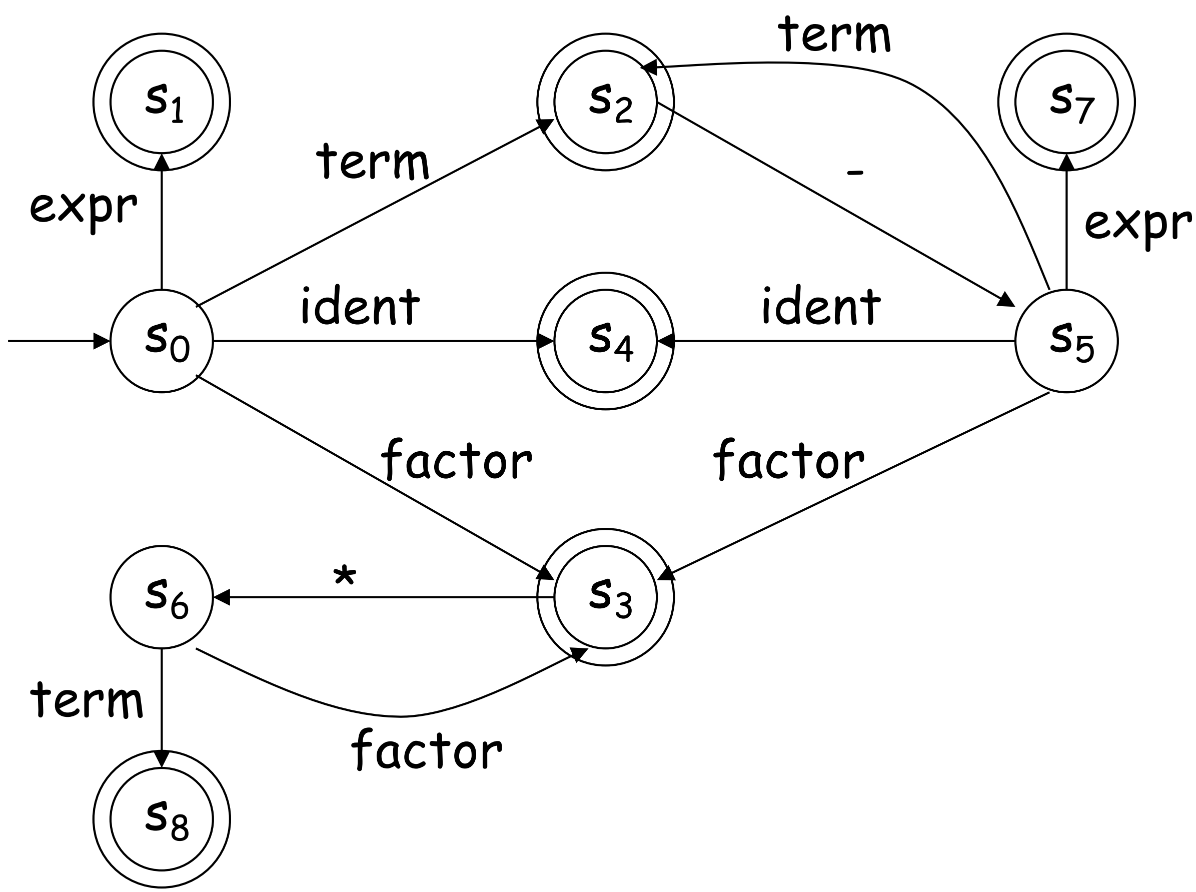
S_8 : { [*Term* → *Factor* * *Term* •, EOF], [*Term* → *Factor* * *Term* •, –] }

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						

Example (DFA)


s1
s2
s7
term
term
-
expr
expr
ident
ident
s0
s4
s5
factor
factor
s6
*
s3
term
factor
s8

Filling in the ACTION and GOTO Tables

x is the number of the state for s_x

The algorithm

$\forall$ *set* $s_x \in S$
  $\forall$ *item* $i \in s_x$
    if i *is* $[A \rightarrow \beta \bullet \underline{a}\delta, \underline{b}]$ *and* $goto(s_x, \underline{a}) = s_k$, $\underline{a} \in T$
      then ACTION$[x, \underline{a}] \leftarrow$ "*shift k*"
    else if i *is* $[S' \rightarrow S \bullet, \text{EOF}]$
      then ACTION$[x, \text{EOF}] \leftarrow$ "*accept*"
    else if i *is* $[A \rightarrow \beta \bullet, \underline{a}]$
      then ACTION$[x, \underline{a}] \leftarrow$ "*reduce* $A \rightarrow \beta$"
  $\forall$ $n \in NT$
    if $goto(s_x, n) = s_k$
      then GOTO$[x, n] \leftarrow k$

Many items generate no table entry

e.g., $[A \rightarrow \beta \cdot B\alpha, \underline{a}]$ does not, but *closure* ensures that all the rhs' for *B* are in s_x

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 $[A \rightarrow \beta \bullet \underline{a}\gamma, \underline{b}]$ and $[B \rightarrow \beta \bullet, \underline{a}]$?

- First item generates "shift", second generates "reduce"
- Both define ACTION[s,$\underline{a}$] — 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

EaC includes a worked example

What is set *s* contains $[A \rightarrow \gamma' \bullet, \underline{a}]$ and $[B \rightarrow \gamma \bullet, \underline{a}]$?

- Each generates "reduce", but with a different production
- Both define ACTION[s,$\underline{a}$] — cannot do both reductions
- This fundamental ambiguity is called a *reduce/reduce error*
- Modify the grammar to eliminate it

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

Computing Closures – LR(0) items

Closure(s) adds all the items implied by items already in *s*

- Any item $[A \to \beta \bullet B\delta]$ implies $[B \to \bullet\tau]$ for each production with *B* on the *lhs*

The algorithm

Closure(s)
  while (s is still changing)
    $\forall$ *items* $[A \to \beta \cdot B\delta] \in s$
      $\forall$ *productions* $B \to \tau \in P$

        if $[B \to \cdot\, \tau] \notin s$
          then add $[B \to \cdot\, \tau]$ *to s*

- Classic fixed-point method
- Halts because $s \subset$ ITEMS

Closure "fills out" a state

Next class

Context-Sensitive Analysis