Compiler Design

Bottom-up Parser (I)

Hwansoo Han

Bottom-up Parsing *(definitions)*

- **Bottom-up parsing and reverse rightmost derivation**
  - A derivation consists of a series of rewrite steps
  - A bottom-up parser builds a derivation by working from the input sentence back toward the start symbol *S*

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \textit{sentence}$$

$$\xleftarrow{\qquad\qquad\qquad\qquad\qquad} \text{bottom-up}$$

- **In terms of the parse tree, this is working from leaves to root**
  - Nodes with no parent in a partial tree form its *upper fringe*
  - Since each replacement of β with *A* shrinks the upper fringe, we call it a *reduction*.

Finding Reductions (handles)

- **Parser must find a substring β of the tree's frontier**
  - Matches some production $A \rightarrow \beta$ that occurs as one step in the rightmost derivation
  - Informally, we call this substring β a *handle*
- **Formally,**
  - A *handle* of a right-sentential form γ is a pair $<A\rightarrow\beta, \boldsymbol{k}>$
    - $A\rightarrow\beta \in P$
    - $\boldsymbol{k}$ is the position in γ of β's rightmost symbol.
  - If $<A\rightarrow\beta, \boldsymbol{k}>$ is a handle, then replace β at $\boldsymbol{k}$ with A

- **Handle pruning**
  - The process of discovering a handle & reducing it to the appropriate left-hand side (non-terminal) is called *handle pruning*
  - Because γ is a right-sentential form, the substring to the right of a handle contains only terminal symbols

Bottom-Up Parser Example

The expression grammar

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	<u>number</u>
9		|	<u>id</u>
10		|	<u>(</u> *Expr* <u>)</u>

Handles for rightmost derivation of <u>x</u> <u>-</u> <u>2</u> <u>*</u> <u>y</u>

Prod'n.	*Sentential Form*	*Handle*
—	*Goal*	—
1	*Expr*	1,1
3	*Expr* - *Term*	3,3
5	*Expr* - *Term* * *Factor*	5,5
9	*Expr* - *Term* * <id,<u>y</u>>	9,5
7	*Expr* - *Factor* * <id,<u>y</u>>	7,3
8	*Expr* - <num,<u>2</u>> * <id,<u>y</u>>	8,3
4	*Term* - <num,<u>2</u>> * <id,<u>y</u>>	4,1
7	*Factor* - <num,<u>2</u>> * <id,<u>y</u>>	7,1
9	<id,<u>x</u>> - <num,<u>2</u>> * <id,<u>y</u>>	9,1

Reverse rightmost derivation (RRD)

<u>Handles</u> are specified in blue

One of Bottom-up Parsers

❖ *Shift-reduce parser*

```
push INVALID
token ← next_token( )
repeat until (top of stack = Goal and token = EOF)
    if the top of the stack can reduce using a handle <A→β.k> then
        // reduce β to A
            pop |β| (=k) symbols off the stack
            push A onto the stack
    else if (token != EOF)  then
        // shift
            push token
            token ← next_token( )
    else // need to shift, but out of input
            report an error
```

How do errors show up?

- **failure to find a handle**
- **hitting EOF & needing to shift (final else clause)**

Either generates an error

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id		

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id		

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id		

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$ *Expr* – *Term*	* id		

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$ *Expr* – *Term*	* id	*none*	shift
$ *Expr* – *Term* *	id	*none*	shift
$ *Expr* – *Term* * id			

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x – 2 * y

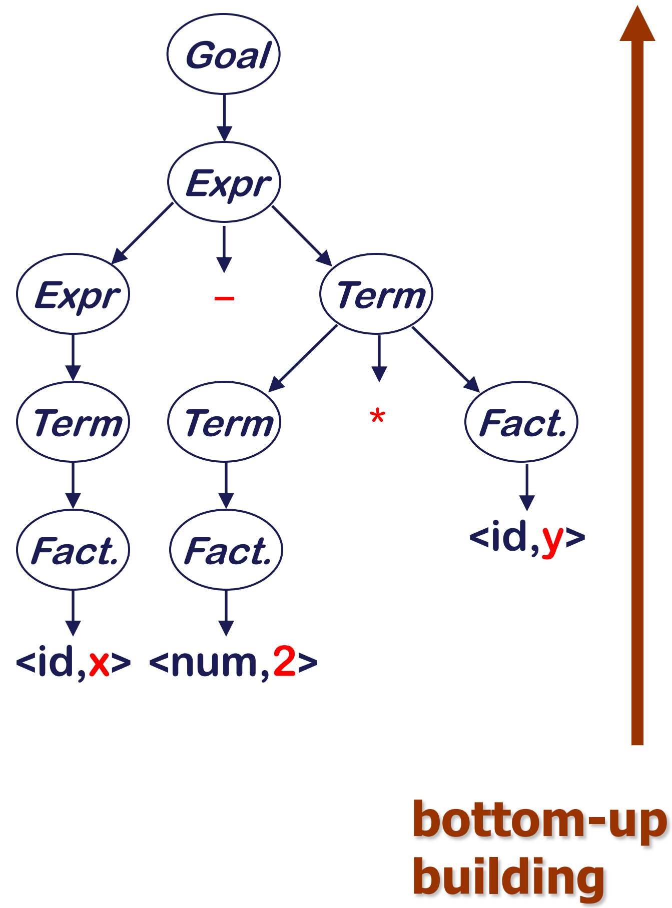
Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$ *Expr* – *Term*	* id	*none*	shift
$ *Expr* – *Term* *	id	*none*	shift
$ *Expr* – *Term* * id		9,5	red. 9
$ *Expr* – *Term* * *Factor*		5,5	red. 5
$ *Expr* – *Term*		3,3	red. 3
$ *Expr*		1,1	red. 1
$ *Goal*		*none*	accept

5 shifts +
9 reduces +
1 accept

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle within stack & reduce

Example

Stack	Input	Action
$	id – num * id	shift
$ id	– num * id	red. 9
$ *Factor*	– num * id	red. 7
$ *Term*	– num * id	red. 4
$ *Expr*	– num * id	shift
$ *Expr* –	num * id	shift
$ *Expr* – num	* id	red. 8
$ *Expr* – *Factor*	* id	red. 7
$ *Expr* – *Term*	* id	shift
$ *Expr* – *Term* *	id	shift
$ *Expr* – *Term* * id		red. 9
$ *Expr* – *Term* * *Factor*		red. 5
$ *Expr* – *Term*		red. 3
$ *Expr*		red. 1
$ *Goal*		accept



Shift-reduce Parsing

- **Shift reduce parsers are easily built and easily understood**
- **A shift-reduce parser has just four actions**
  - *Shift* — next word is shifted onto the stack
  - *Reduce* — right end of handle is at top of stack
    - Locate left end of handle within the stack
    - Pop handle off stack & push appropriate *lhs*
  - *Accept* — stop parsing & report success
  - *Error* — call an error reporting/recovery routine

Handle finding is key
• handle is on stack
• finite set of handles
⇒ use a DFA !

- **Critical Question: How can we know when we have found a handle without generating lots of different derivations?**
  - Answer: we use look ahead in the grammar along with tables produced as the result of analyzing the grammar.
  - *LR(1)* parsers build a DFA that runs over the stack & finds them

Another Bottom-Up Parser

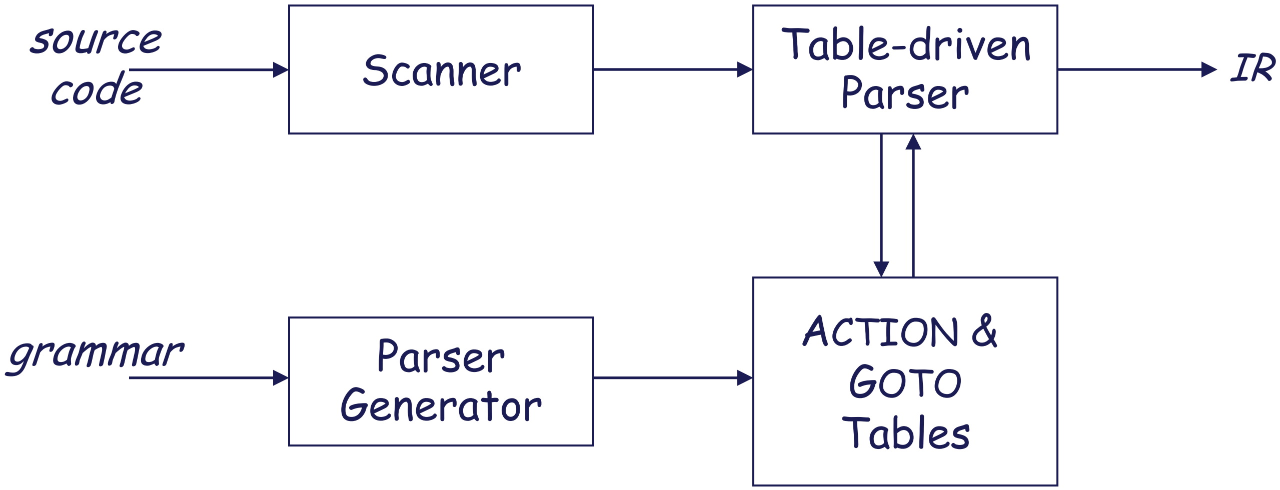
- **LR(1) Parsers**
  - LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition
  - LR(1) parsers recognize languages that have an LR(1) grammar

- **Informal definition:**
  - A grammar is LR(1) if, given a rightmost derivation

    $S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \textit{sentence}$
  - We can
    1. *isolate the handle of each right-sentential form* γ_i, and
    2. *determine the production with which to reduce, by scanning* γ_i *from left-to-right, going at most 1 symbol beyond the right end of the handle of* γ_i

LR(1) Parsers

- **A table-driven LR(1) parser looks like**



- Tables *can* be built by hand
- However, this is a perfect task to automate

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] == "shift s_next" ) then {
            stack.push(token); stack.push(s_next);
            token ← scanner.next_token();
      }
      else 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] == "accept"
                              & token == EOF ) then {
            not_found = false;
      }
      else report a syntax error and recover;
}
report success;
```

The skeleton parser

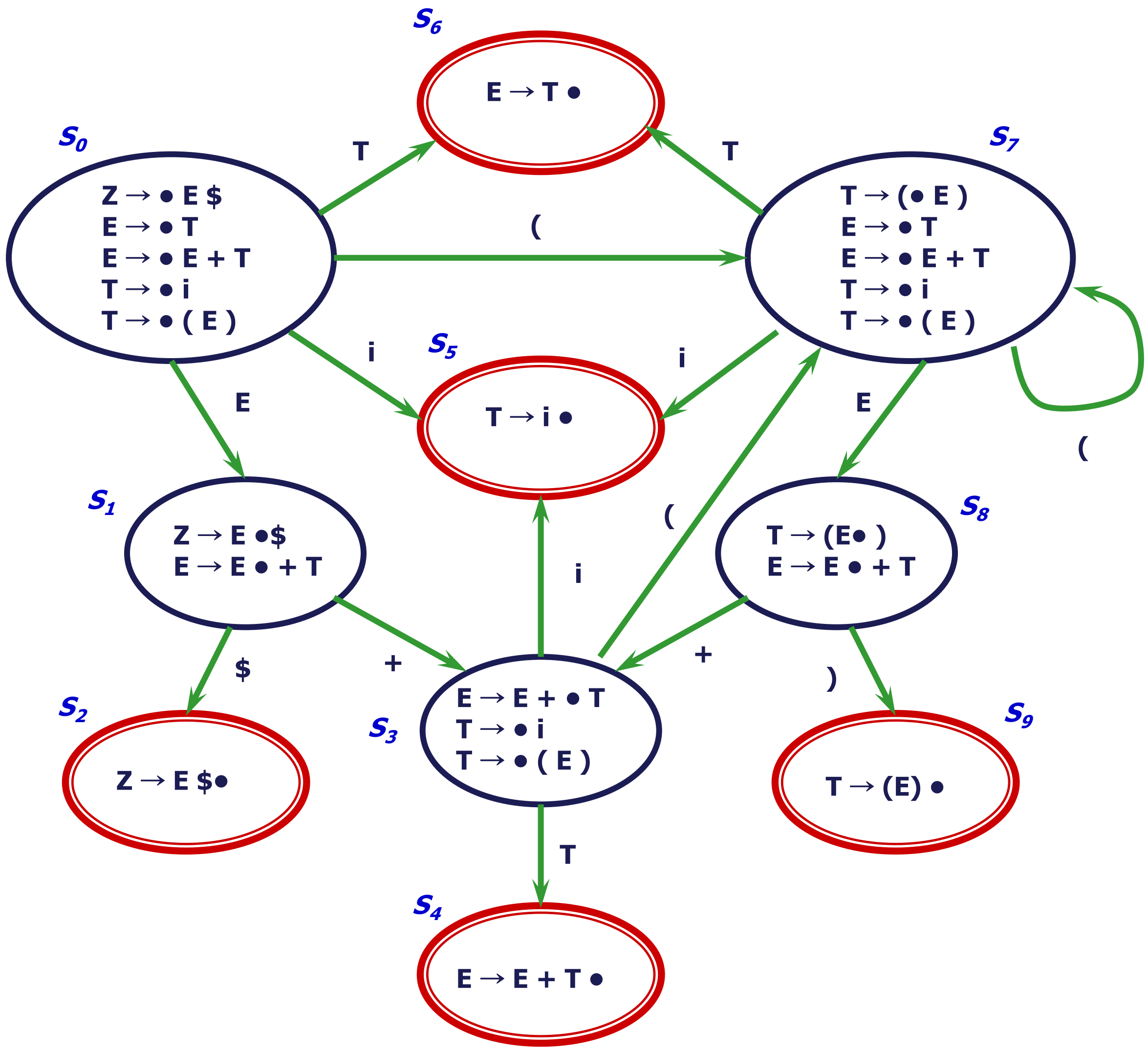
- push tokens & NTs along with DFA states
- uses ACTION & GOTO tables (DFA)
- does |*words*| shifts
- does |derivation| reductions
- does 1 accept
- detects errors by failure of 3 other cases

LR(1) Parsers

- **How does this LR(1) stuff work?**
  - Unambiguous grammar $\Rightarrow$ unique rightmost derivation
  - Keep upper fringe on a stack
    - All active handles include top of stack (TOS)
    - Shift inputs until TOS is right end of a handle
  - Language of handles is regular (finite)
    - Build a handle-recognizing DFA
    - ACTION & GOTO tables encode the DFA
- **The Big Picture**
  - Model the state of the parser
  - Use two functions *goto(s, X)* and *closure(s)*
    - *goto()* is analogous to *move()* in *subset construction* (NFA$\rightarrow$DFA)
    - *closure()* adds information to form a state
  - Build up the states and transition functions of the DFA
  - Use this information to fill in the ACTION and GOTO tables

LR(0) example

Z → E
E → E + T | T
T → i | (E)



Summary

- **Bottom-up parser**
  - Reverse rightmost derivation
  - Handle pruning, reduction
- **Shift-reduce parser**
  - Reduce if found a handle in stack
  - Otherwise, shift a token (push on to stack)
- **LR(1) parser**
  - Discover handles from DFA
  - ACTION, GOTO tables from DFA