# *Parsing V*
# *Introduction to LR(1) Parsers*

## LR(1) Parsers

- LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 word) 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

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

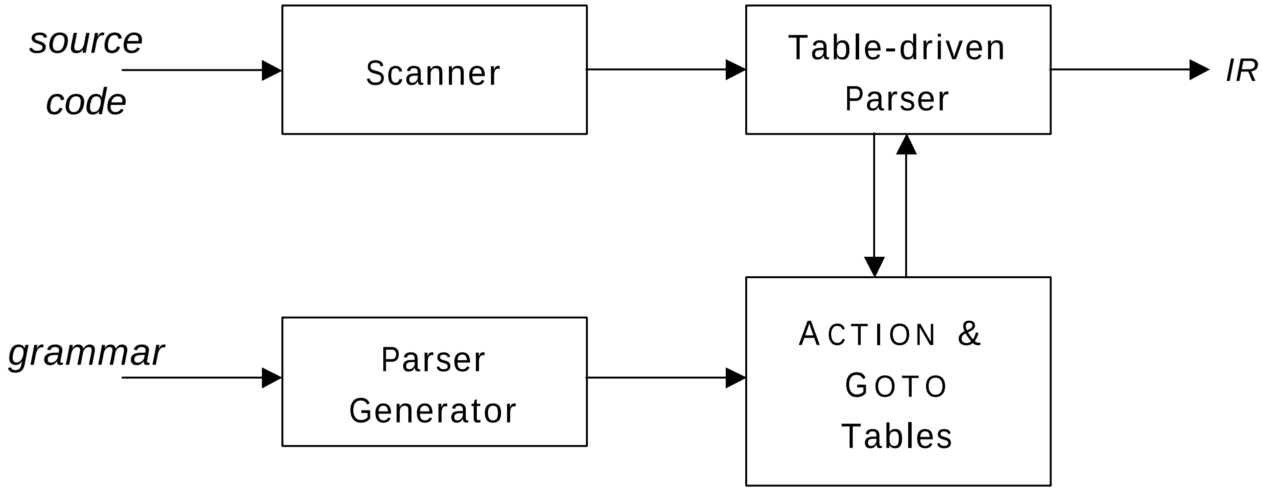
We can

1. *isolate the handle of each right-sentential form* $\gamma_i$, and
2. *determine the production by which to reduce,*

by scanning $\gamma_i$ from left-to-right, going at most 1 symbol beyond the right end of the handle of $\gamma_i$

# *LR(1) Parsers*

A table-driven LR(1) parser looks like




Tables *can* be built by hand (*homework # 2*)

It is a perfect task to automate

## *LR(1) Parsers* *(the skeleton parser)*

```
push INVALID
push s_0
token ← next_token()
repeat forever
  s ← top of stack
  if ACTION[s,token] = "reduce N→β"
    then
      pop 2*|β| symbols
      s ← top of stack
      push N
      push GOTO[s,N]
  else if ACTION[s,token] = "shift s_i"
    then
      push token ; push s_i
      token ← next_token()
  else if ACTION[s,token] = "accept"
       and token = EOF
     then break;
  else report a syntax error
report success
```

*The skeleton parser*

- uses ACTION & GOTO
- does |*words*| shifts
- does |derivation| reductions
- does 1 accept
- detects errors by failure of 3 other cases

## *LR(1) Parsers (parse tables)*

To make a parser for *L(G)*, need a set of tables

The grammar

| | | | |
|---|---|---|---|
| 1 | *Goal* | $\rightarrow$ | SheepNoise |
| 2 | *SheepNoise* | $\rightarrow$ | SheepNoise <u>baa</u> |
| 3 | | \| | <u>baa</u> |

The tables

| ACTION | | |
|---|---|---|
| State | EOF | <u>baa</u> |
| 0 | — | shift 2 |
| 1 | accept | shift 3 |
| 2 | reduce 3 | reduce 3 |
| 3 | reduce 2 | reduce 2 |

| GOTO | |
|---|---|
| State | *SheepNoise* |
| 0 | 1 |
| 1 | 0 |
| 2 | 0 |
| 3 | 0 |

## *Example Parses*

The string “baa”

| Input | Stack | Action |
|---|---|---|
| baa EOF | $\$\ s_0$ | shift 2 |
| EOF | $\$\ s_0$ baa $s_2$ | reduce 3 |
| EOF | $\$\ s_0\ SN\ s_1$ | accept |

The string “baa”

| Input | Stack | Action |
|---|---|---|
| baa baa EOF | $\$\ s_0$ | shift 2 |
| baa EOF | $\$\ s_0$ baa $s_2$ | reduce 3 |
| baa EOF | $\$\ s_0\ SN\ s_1$ | shift 3 |
| EOF | $\$\ s_0\ SN\ s_1$ baa $s_3$ | reduce 2 |
| EOF | $\$\ s_0\ SN\ s_1$ | accept |

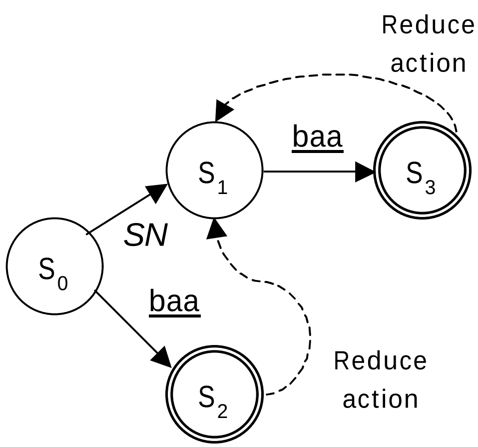
We cannot have a syntax error with *SN*, because it only has 1 terminal symbol!

“baa woof” is a lexical problem, not a syntax error!

## *LR(1) Parsers*

How does this LR(1) stuff work?

- Unambiguous grammar ⇒ unique rightmost derivation
- Keep upper fringe on a stack
  - > All active handles include 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
- To match subterms, recurse & leave DFA's state on stack
- Final state in DFA ⇒ a *reduce* action
  - > New state is GOTO[*lhs*,state at TOS]
  - > For *SN*, this takes the DFA to $S_1$




*Control DFA for SN*

# *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)

The Big Picture

- Model the state of the parser
- Use two functions *goto( s,N )* 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*

An LR($k$) item is a pair [$A$,$B$], where

> $A$ is a production $\alpha \rightarrow \beta\gamma\delta$ with a • at some position in the *rhs*
>
> $B$ is a lookahead string of length $k$ (words or EOF)

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

[$\alpha \rightarrow \bullet\beta\gamma\delta,\underline{a}$] means that the input seen so far is consistent with the use of $\alpha \rightarrow \beta\gamma\delta$ immediately after the symbol on top of the stack

[$\alpha \rightarrow \beta\gamma\bullet\delta,\underline{a}$] means that the input sees so far is consistent with the use of $\alpha \rightarrow \beta\gamma\delta$ at this point in the parse, ***and*** that the parser has already recognized $\beta\gamma$.

[$\alpha \rightarrow \beta\gamma\delta\bullet,\underline{a}$] means that the parser has seen $\beta\gamma\delta$, ***and*** that a lookahead symbol of $\underline{a}$ is consistent with reducing to $\alpha$.

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

## *LR(1) Items*

The production $\alpha \rightarrow \bullet\beta\gamma\delta$, with lookahead $\underline{a}$, generates 4 items

$$[\alpha \rightarrow \bullet\beta\gamma\delta,\underline{a}],\ [\alpha \rightarrow \beta\bullet\gamma\delta,\underline{a}],\ [\alpha \rightarrow \beta\gamma\bullet\delta,\underline{a}],\ \&\ [\alpha \rightarrow \beta\gamma\delta\bullet,\underline{a}]$$

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 correct reduction *(if a choice occurs)*
- Lookaheads are bookkeeping, unless item has • at right end
  - Has no direct use in $[\alpha \rightarrow \beta\gamma\bullet\delta,\underline{a}]$
  - In $[\alpha \rightarrow \beta\gamma\delta\bullet,\underline{a}]$, a lookahead of $\underline{a}$ implies a reduction by $\alpha \rightarrow \beta\gamma\delta$
  - For $\{\ [\alpha \rightarrow \gamma\bullet,\underline{a}],[\beta \rightarrow \gamma\bullet\delta,\underline{b}]\ \}$, $\underline{a} \Rightarrow$ ***reduce*** to $\alpha$; FIRST($\delta$) $\Rightarrow$ ***shift***

$\Rightarrow$ Limited right context is enough to pick the actions

# *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, $i_0$

   → [$S' \rightarrow \bullet S$,EOF], along with any equivalent items

   → Derive equivalent items as *closure(* $i_0$ *)*

  b Repeatedly compute, for each $i_k$, and each $\alpha$, *goto(*$i_k$, $\alpha$*)*

   → 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*

## *Back to Finding Handles*

Revisiting an issue from last class

Parser in a state where the stack (the fringe) was

*Expr* - *Term*

With lookahead of *

How did it choose to expand *Term* rather than reduce to *Expr?*

- *Lookahead* symbol is the key
- With lookahead of + or -, parser should reduce to *Expr*
- With lookahead of * or /, parser should shift
- Parser uses lookahead to decide
- All this context from the grammar is encoded in the handle recognizing mechanism

Remember this slide from last lecture?

## 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 |

shift here

reduce here

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

## *Next Class*

- Algorithms for FIRST, *goto*, & *closure*
- Work an example — a simplified expression grammar

To prepare

- Look at the book or web pages
- Work through the *SheepNoise* example

## *Computing* F*IRST* *Sets*

Define FIRST as

- If $\alpha \Rightarrow^* \underline{a}\beta$, $\underline{a} \in T$, $\beta \in (T \cup NT)^*$, then $\underline{a} \in$ FIRST($\alpha$)
- If $\alpha \Rightarrow^* \varepsilon$, then $\varepsilon \in$ FIRST($\alpha$)

To compute FIRST

- Use a fixed-point method
- FIRST($\alpha$) $\in 2^{(T \cup \varepsilon)}$
- Loop is monotonic

$\Rightarrow$ Algorithm halts

For *SheepNoise:*

FIRST(*Goal*) = { <u>baa</u> }

FIRST(*SN*) = { <u>baa</u> }

FIRST(<u>*baa*</u>) = { <u>baa</u> }

***For each*** $\alpha \in$ ***T***
    ***FIRST(a) ¬ a***
***For each*** $\alpha \in$ ***NT,***
    ***FIRST(a) ¬ Ø***

***While (FIRST sets are still changing)***
  ***for each p*** $\in$ ***P, of the form a ® b,***
    ***if b is e then***
      ***FIRST(a) ¬ FIRST(a)*** $\cup$ ***{ e }***
    ***else if b is*** $b_1 b_2 \ldots b_k$ ***then***
      ***FIRST(a) ¬ FIRST(a)*** $\cup$ ***FIRST***$(b_1)$
      ***i ¬ 1***
      ***while(e*** $\in$ ***FIRST***$(b_i)$ ***& i k - 1)***
        ***FIRST(a) ¬ FIRST(a)*** $\cup$ ***FIRST***$(b_{i+1})$
        ***i ¬ i + 1***