CSE 131A

Lecture 9

Error Handling

LR Parser Theory and Design

Bottom Up Parser Generation

Scott B. Baden

Department of Computer Science & Engineering

University of California, San Diego

Winter 2007

Today's lecture

- Working with grammar and AST specifications
- Error Handling
- Theory of operation of an LR parser
  - Rightmost derivations and right-sentential forms
  - Handles and handle pruning
  - LR(k) grammars
  - State tables
- Today's reading comes from §4.7 of the text

Roadmap

- Working with the grammar and AST specifications
- Error handling
- Theory of operation of LR parsing

Examples

num.onyx

```
3
```

l2.onyx

```
let $a := 4, $b := 3
return $a+$b
```

fl3.onyx

```
declare function f($a as onyx.types.Integer, $b)
{ let $z := $a + 1
 return $z
};
f(3,4)
```

Num.onyx

num.onyx

```
3
```

AST

```
<onyx.ast.Query>
  <onyx.ast.ExprList>
    <onyx.ast.Constant datatype="onyx.types.Integer">
3
    </onyx.ast.Constant>
  </onyx.ast.ExprList>
</onyx.ast.Query>
```

Parsing a literal

3

Literal ::= tokenInteger | tokenDecimal | tokenBoolean | tokenString
PrimaryExpr ::= Literal | tokenVariable | (ExprList?) | FunctionCall
UnaryExpr ::= UnaryOp? PrimaryExpr
MultiplicativeExpr ::= UnaryExpr (MultiplicativeOp UnaryExpr)*
AdditiveExpr ::= MultiplicativeExpr (AdditiveOp MultiplicativeExpr)*
CompareExpr ::= AdditiveExpr (CompareOp AdditiveExpr)?
AndExpr ::= CompareExpr (and CompareExpr)*
OrExpr ::= AndExpr (or AndExpr)*
ExprSingle ::= OrExpr | FLWRExpr | IfExpr | RangeExpr
ExprList ::= ExprSingle (, ExprSingle)*
QueryBody ::= ExprList?
QueryProlog ::= (FunctionDecl | VariableDecl)*
Query ::= QueryProlog QueryBody

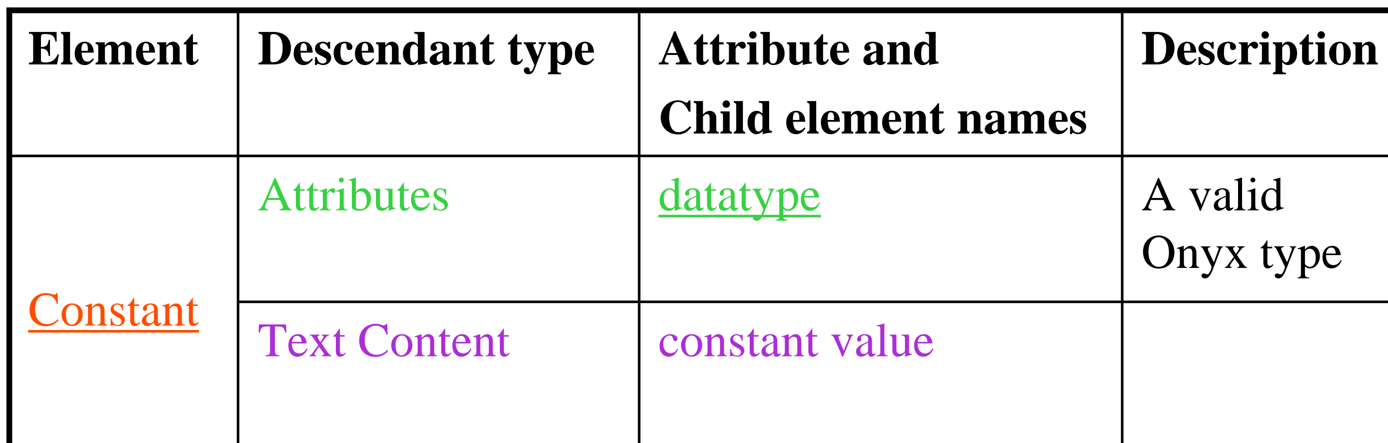
Generating the AST

3

Literal ::= tokenInteger | tokenDecimal | tokenBoolean | tokenString

=> Constant

```
<onyx.ast.Constant datatype="onyx.types.Integer">
  3
</onyx.ast.Constant>
```



Element	Descendant type	Attribute and Child element names	Description
Constant	Attributes	datatype	A valid Onyx type
	Text Content	constant value	

AST for l2.onyx

```
let $a := 4, $b := 3: return $a+$b
```

```
<onyx.ast.LetExpression name="$a">
  <onyx.ast.Constant datatype="onyx.types.Integer">4</onyx.ast.Constant>
  <onyx.ast.LetExpression name="$b">
     <onyx.ast.Constant datatype="onyx.types.Integer">3</onyx.ast.Constant>
     <onyx.ast.Operator name="op:numeric-add">
        <onyx.ast.Variable>$a</onyx.ast.Variable>
        <onyx.ast.Variable>$b</onyx.ast.Variable>
     </onyx.ast.Operator>
  </onyx.ast.LetExpression>
</onyx.ast. LetExpression>
```

Parsing l2.onyx

```
let $a := 4, $b := 3
   return $a+$b
```

```
<onyx.ast.Constantdatatype="onyx.types.Integer”> 4</onyx.ast.Constant>
:=
VAR(A)
LET
```

LetClause ::= let tokenVariable := ExprSingle (, tokenVariable := ExprSingle)*

⇒ **LetExpression**

```
<onyx.ast.LetExpression name="$a">
  <onyx.ast.Constant datatype="onyx.types.Integer">4</onyx.ast.Constant>
   <onyx.ast.LetExpression name="$b">
     <onyx.ast.Constant datatype="onyx.types.Integer">3</onyx.ast.Constant>
     <onyx.ast.Operator name="op:numeric-add"> ….
```

Building the LetExpression node

```
<onyx.ast.LetExpression name="$a">
<onyx.ast.Constant datatype="onyx.types.Integer">4</onyx.ast.Constant>
  <onyx.ast. LetExpression name="$b">
   <onyx.ast.Constant datatype="onyx.types.Integer">3</onyx.ast.Constant>
   …..
  </onyx.ast.LetExpression>
</onyx.ast.LetExpression>
```

Element	Descendant type	Attribute and Child element names	Description
LetAssignment	Attributes	varName	Name of variable
	Elements	*any-expression-element/ any-expression-element*	1st expr generates value bound to name. 2nd expr is the let body

fl3.onyx: function definition

```
declare function f($a as onyx.types.Integer, $b)
{ let $z := $a + 1  return $z }; f(3,4)
```

```
<onyx.ast.FunctionDeclaration datatype="onyx.types.AnyType" funcName="f">
    <onyx.ast.ArgumentDeclaration datatype="onyx.types.Integer" name="$a"/>
    <onyx.ast.ArgumentDeclaration datatype="onyx.types.AnyType"
    name="$b"/>
    <onyx.ast.LetExpression name="$z">
      <onyx.ast.Operator name="op:numeric-add">
        <onyx.ast.Variable>$a</onyx.ast.Variable>
        <onyx.ast.Constant datatype="onyx.types.Integer">1</onyx.ast.Constant>
      </onyx.ast.Operator>
      <onyx.ast.Variable>$z</onyx.ast.Variable>
    </onyx.ast.LetExpression>
  </onyx.ast.FunctionDeclaration>
```

f13.onyx: function call

```
declare function f($a as onyx.types.Integer, $b)
{ let $z := $a + 1  return $z }; f(3,4)
```

```
<onyx.ast.ExprList>
    <onyx.ast.FunctionCall name="f">
      <onyx.ast.Constant datatype="onyx.types.Integer">3</onyx.ast.Constant>
      <onyx.ast.Constant datatype="onyx.types.Integer">4</onyx.ast.Constant>
    </onyx.ast.FunctionCall>
  </onyx.ast.ExprList>
```

Roadmap

- Working with grammar and AST specifications
- Error handling
- Theory of operation of LR parsing

Error Location and Identification

- Error handling is an important feature of a programming tool
- Locating and identifying errors can be done at various stages of translation
- Lexical "spelling" errors can be intercepted by the scanner, e.g.: **if ($x < 0) thev 0 else 1**
- The parser can detect structural errors

  (3 + 4)) + 7
- "Semantic" errors, problems such as an undeclared identifier or a type mismatch may not be expressible using context-free grammar rules

Attributes of a good error handler

- Provide convenient and accurate location of the error

```
    i = 1$5
        ^
Error 24 at (3:t.f) : syntax error
```

- Recover quickly from errors so that parsing may continue, and any additional errors may be handled

```
5: int x = 1$5;          t.c:5: parse error before `$5'
6: int y = 7;
7: f[y);                 t.c:7: parse error before `)‘
```

- Error handling should not slow down the parsing of syntactically correct programs

What makes error handling difficult?

- One syntax error can lead to many others
- A missing semicolon within a declaration causes y to be undeclared

```
7: int x = 1
8: int y = 7;    u.c:8:parse error before `int'
9: f(x);
10: f(y);        u.c:10: `y' undeclared

11: f(x+y);      (Each undeclared identifier is
                  reported only once for a
                  function it appears in.)
```

Handling errors

- What shall we tell the user?
- What action shall we take**?**
- **Panic mode** error recovery consumes tokens until a *synchronizing token* is found (e.g. statement-terminating semicolon) where parsing can begin again. Simple to implement
- **Phrase level** error recovery attempts to repair the input by making small local changes

```
if ($x < 0) thev 0 else 1
if ($x < 0) then 0 else 1
```

Why repair can be useful

- Changing ; to , might avoid many errors

```
call g(i;j)
          ^
Error 3 at (5:t.f) : unbalanced parenthesis
          ^
Error 7 at (5:t.f) : incomplete statement

call g(i,j)
```

Error productions

- The “If we can’t beat ‘em, join ‘em” strategy
- Extend the grammar of the language to include anticipated malformed program constructs
- Then the parser can automatically correct these errors, or report them in an informative way

Roadmap

- Working with grammar and AST specifications
- Error handling
- Theory of operation of bottom up parsing

Inside a bottom up parser

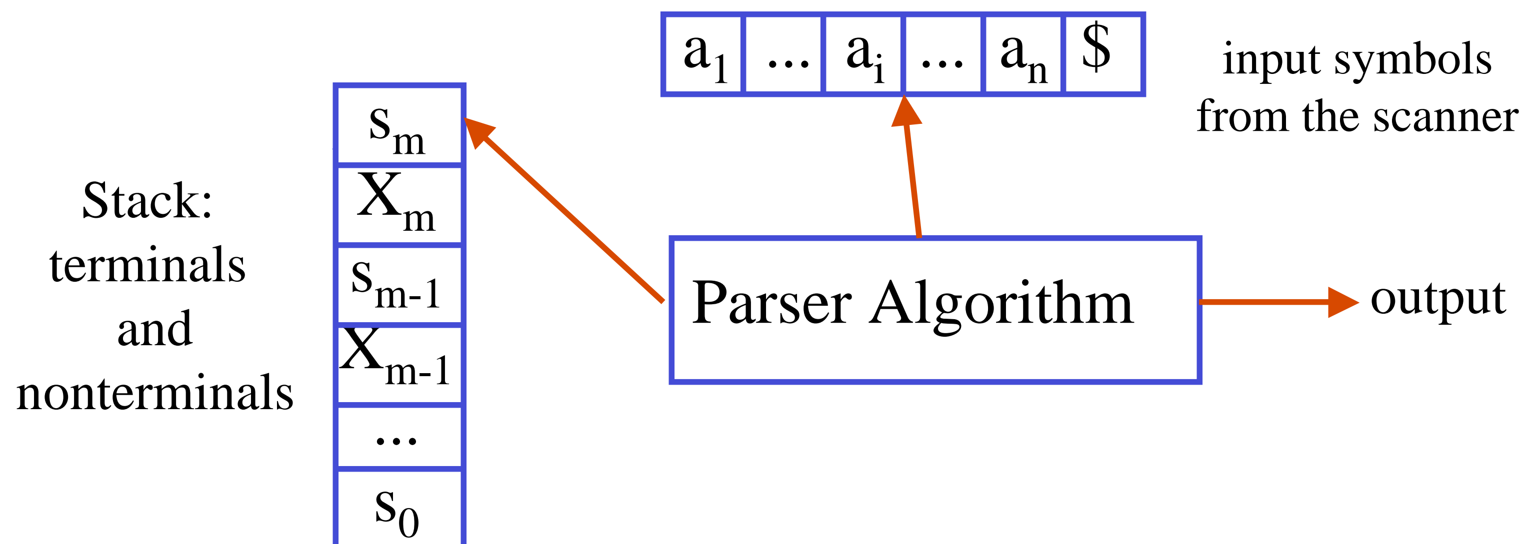
- Toward implementing a shift-reduce parser
- Rightmost derivations and right-sentential forms
- Handles and handle pruning
- LR(k) grammars

Bottom up parsing

- In a bottom up parser, we attempt to reduce a string to the start symbol following a rightmost derivation in reverse
- At each step, the parser locates the unique handle of the sentential form
- When the parser cannot deterministically compute the handle we have a parser conflict

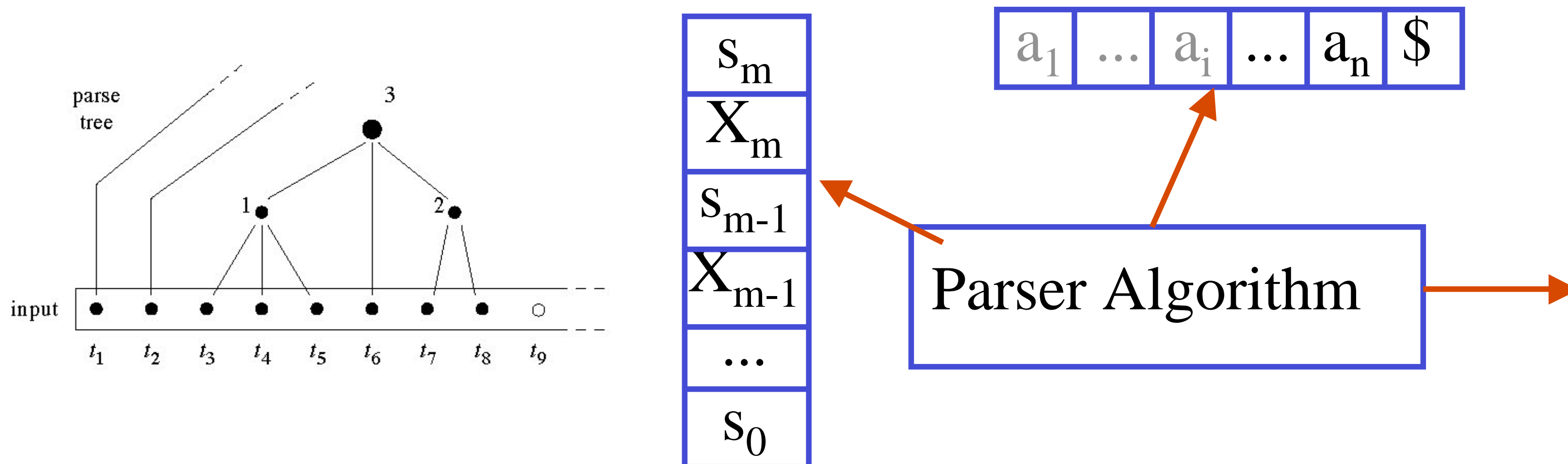
Structure of an LR parser

- The ***stack*** holds grammar symbols for reduction; only symbols at the top of the stack (TOS) are accessed
- The ***input buffer*** holds the string of tokens to be parsed; tokens are read in sequence



Parser actions

- **shift** input tokens onto the stack until a match is found between the symbols on the top of stack (TOS) and the RHS of some production
- **reduce** those symbols to the LHS of the production, replacing with the LHS on TOS
- repeat until the stack contains the start symbol: **accept**
- if there is no matching production: **error**



Theory of operation

- We have not yet revealed how the parser..
  - recognizes handles
  - decides whether to shift or reduce
  - in the case of a reduction, which rule to reduce by

LR(k) Parsers

- We are now considering LR(k) parsers
  - The "L" stands for **L**eft-to-right input scanning; the "R" for constructing a **R**ightmost derivation in reverse
  - ... As opposed to top-down LL(k) parsers that do **L**eft-to-right input scanning but construct a **L**eftmost derivation
  - The "k" represents the number of lookahead input symbols (we assume k=1 if k is omitted)

The attraction of LR Parsers

- LR parsers are time and space efficient
- LR(1) grammars are enough to define typical programming language constructs
- LR(k) parsers are more powerful than LL(k) top-down predictive parsers, for a given value of k
- LR parsers permit error detection
  "as early in the input as possible"
- While LR parser are hard to build by hand, good LR parser generators exist, e.g. CUP

Towards finding the handle

- We know that ..
  - if the grammar is unambiguous, every right-sentential form has exactly one handle
  - a shift-reduce parser can be sure the handle appears at the top of the stack
  - For a LR(1) grammar, our parser can detect the handle with only 1 token of lookahead
- The trick is how to tell that it is there

Viable prefixes

- The notion of a *viable prefix* is helpful
- A *viable prefix* is a prefix of a right sentential form extending now further than the right end of the handle
- In our earlier example grammar, the right sentential form *aAbcde* has the handle *Abc*

$a\underline{b}bcde \rightarrow a\underline{Abc}de \rightarrow aA\underline{d}e \rightarrow \underline{aABe} \rightarrow S$

- The viable prefixes are ε *a aA aAb aAbc*

The significance viable prefixes

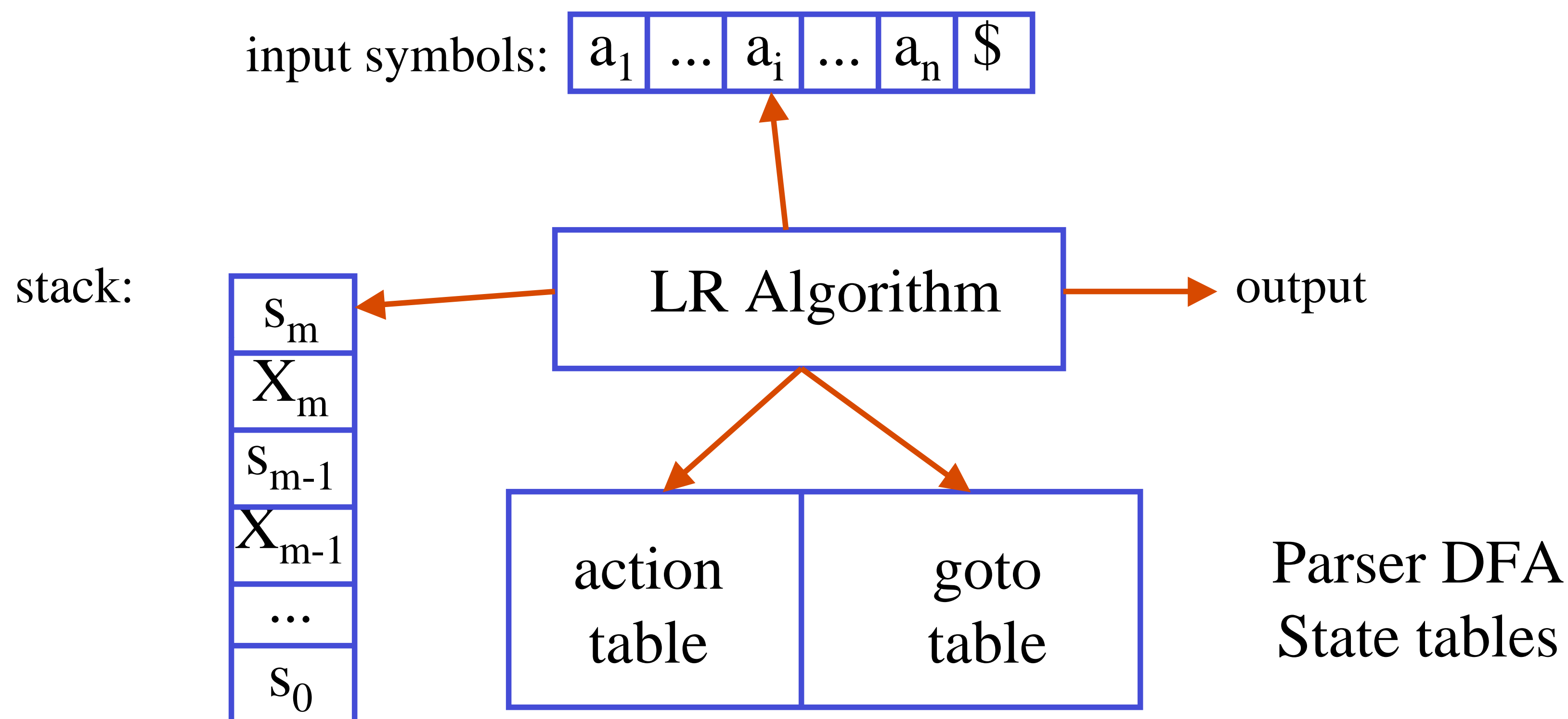
- A *viable prefix* may also be viewed as a sequence of symbols appearing on the stack during a successful parse
- It is always possible to add input symbols to a viable prefix to obtain a right-sentential form
- If a viable prefix is on the stack, the parse can continue
- We need an algorithm that tells us when we have found a viable prefix

Towards finding the handle

- Knuth (1965) proved that the viable prefixes for a context free grammar form a regular language
- We can therefore construct a DFA to recognize the handles on the stack
- Transitions in the DFA depend on the current TOS and the current lookahead symbol(s)
- The problem of LR parser construction is then how to build a state table for this DFA that recognizes viable prefixes

Structure of a LR parser

- All LR parsers have this form;variants differ in the content of the tables: stack, input, parser tables, and a driver



The LR Parser Structures

- The stack stores
  - grammar symbols X_i which are prefixes of the current right sentential form
  - *parser states* s_j
  - we store them in an alternating pattern
    $\$ \; s_0 \; X_1 \; s_1 \; X_2 \; s_2 \; \ldots \; X_m \; s_m$
- The states are states of the DFA that recognizes the handles; s_0 is the DFA start state
- A state summarizes the essential information about the symbols on the stack below it
- It is unnecessary to include grammar symbols on the stack; we show them for clarity only

The parser state tables

- The tables represent the transition function of a DFA that recognizes handles on the stack so they can be reduced at the right time
- They also describe the parser actions
  shift, **reduce**, **accept**, **error**
- There are two parts to the table:
  – ***action:*** maps [state, terminal symbol] → action

  – ***goto:*** maps [state, nonterminal symbol] → state

The action table

- *action*[s_m,$a_{i\ldots}$ a_{i+k-1}] tells the parser what action to take when state s_m is on top of stack and lookahead is the next k symbls $a_{i\ \ldots}\ a_{i+k-1}$
- Possible actions:
  – **shift** the next input symbol onto the stack, together with the new parser state
  – **reduce** a handle on TOS, replacing the handle symbol son the rule's LHS; execute code associated with the rule, if any; use *goto* table to determine the next state, pushed onto the stack
  – **accept** the parse, or report an **error**

The goto table

- When a reduce action happens, symbols at the top of the stack (the handle) are popped
- This exposes the state at the top of the stack
- In place of the removed symbols, we push the LHS of the handle's rules
- *goto*$[s_m , X_i]$ tells the parser what state to push on the stack when a reduction to symbol X_i exposes state s_m on TOS
- This state becomes the current state of the parser

The LR(k) parser algorithm

- Start with stack containing $ and start state s_0
- Loop:
  - Consider the next *k* input tokens as lookahead; and look at the current state at TOS
  - Consult the *action* table entry for this state and lookahead
  - If the action is **error** or **accept**, we're done.
  - If the action is **shift** s_i, push next input token and state s_i on the stack
  - If the action is **reduce** $A \rightarrow \beta$, remove symbols β from TOS, replace with A, and consult *goto* table to determine state to push on stack

Algorithm invariants

- Because the LR algorithm traces a rightmost derivation in reverse it must maintain these invariants
  - The sequence of grammar symbols on the stack must *always* be the prefix of a right-sentential form of the grammar
  - The right end of the handle cannot be below the top of stack
- Equivalently: the parser must maintain a viable prefix on the stack
- We need the concepts of *viable prefix* and *parser configuration*

Configuration

- A ***configuration*** of an LR parser is the ordered pair consisting of the stack and the unconsumed input: $(s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_m\ s_m\ ,\ a_i\ a_{i+1}\ \ldots\ a_n\$)$
- It represents the right sentential form (excluding states)

  $X_1\ \ X_2\ \ldots\ \ X_m\ \ a_i\ \ a_{i+1}\ \ \ldots\ \ a_n$

  where $X_1\ \ X_2\ \ldots\ \ X_m$ is a viable prefix
- The LR algorithm manipulates the configuration by shifting and reducing until it reaches either an accept or error state

Configurations are changed by reduce

- Say that the configuration is (current state $X_m\ s_m$)

  ($\$\ s_0\ X_1\ s_1\ X_2\ s_2 \ldots X_m\ s_m\ ,\ a_i\ a_{i+1} \ldots a_n \$$)

- Thus, current state is s_m, lookahead is $a_i\ a_{i+1}$

- If $action[s_m,a_i]$ table indicates a **shift** of state s :

  ($s_0\ X_1\ s_1\ X_2\ s_2 \ldots X_m\ s_m\ a_i\ s\ ,\ a_{i+1} \ldots a_n \$$)

- We consume the token, and push it onto the stack, together with the new state

Configurations are changed by shift

- Starting with the configuration is (current state $X_m\ s_m$):

  $(\$\ s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_m\ s_m\ ,\ \ a_i\ a_{i+1}\ \ldots\ a_n\ \$)$

- If $action[s_m, a_i]$ results in a **reduce** $A \rightarrow \beta$
- The new configuration is

  $(s_0\ X_1\ s_1\ X_2\ s_2\ \ldots X_{m\text{-}r}\ s_{m\text{-}r}\ A\ s,\ a_i\ a_{i+1}\ \ldots\ a_n\ \$)$

  - The new current state is $s=goto[s_{m\text{-}r}\ , A]$
  - β holds r symbols
  - In a reduction, the input symbol is not consumed
  - The symbols popped off the stack will always match β , the RHS of the reducing production
  - Code associated with the rule is executed at this time (perform a semantic action, e.g. build a node in an AST)

LR(1) state table

- The parser begins in state s_0 with input w
- r x: reduce by rule x
- s x: shift and goto state x
- Parse table for the following grammar

$$
\begin{aligned}
&(1)\ E \rightarrow E + T\\
&(2)\ E \rightarrow T\\
&(3)\ T \rightarrow T * F\\
&(4)\ T \rightarrow F\\
&(5)\ F \rightarrow (\ E\ )\\
&(6)\ F \rightarrow \mathbf{id}
\end{aligned}
$$

s_i	*action*						*goto*		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				**acc**			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

The Moves

Stack	Input	Action
0	**id** * **id** + **id**$	Shift 5
0 **id** 5	* **id** + **id**$	Reduce by F → **id**
0 *F* 3	* **id** + **id**$	Reduce by T → F
0 *T* 2	* **id** + **id**$	Shift 7
0 *T* 2 * 7	**id** + **id**$	Shift 5
0 *T* 2 * 7 **id** 5	+ **id**$	Reduce by F → **id**
0 *T* 2 * 7 *F* 10	+ **id**$	Reduce by T → T * F
0 *T* 2	+ **id**$	Reduce by E → T
0 *E 1*	+ **id**$	Shift 6
0 *E 1* + 6	**id**$	Shift 5
0 *E 1* + 6 **id** 5	$	Reduce by F → **id**
0 *E 1* + 6 *F* 3	$	Reduce by T → F
0 *E 1* + 6 *T* 9	$	Reduce by E → E + T
0 *E 1*	$	**accept**

Tables for recognizing handles

- It is possible to recognize a handle on the stack if you know all the grammar symbols on the stack, plus any lookahead
- Since handles are regular languages, recognition is done by a DFA scanning the stack from bottom to top
- However we don't want to scan the whole stack with a DFA every time we consider an action
- Instead we set up the tables so that the state at the top of the stack is the state the DFA would be in had it scanned the whole stack
- This is not magic: the algorithm knows what symbols are there, since it put them there
- Next time we'll talk about how to build the tables