450 COMPILERS

COMPUTER SCIENCE
News & Info

- Who’s Hiring May 2016
  - [https://news.ycombinator.com/item?id=11611867](https://news.ycombinator.com/item?id=11611867)

- SoCal Code Camp | San Diego, CA 6/25-6/26
Administrivia

- Lab 06
  - Due Thursday
Review: Shift Reduce Parsing

Bottom-up parsing uses two actions:

**Shift**

\[ ABC|xyz \Rightarrow ABCx|yz \]

**Reduce**

\[ Cbxy|ijk \Rightarrow CbA|ijk \]
Recall: The Stack

• Left string can be implemented by a stack
  - Top of the stack is the |

• Shift pushes a terminal on the stack

• Reduce
  - pops 0 or more symbols off of the stack
    • production rhs
  - pushes a non-terminal on the stack
    • production lhs
Key Issue

• How do we decide when to shift or reduce?

• Example grammar:
  \[ E \rightarrow T + E \mid T \]
  \[ T \rightarrow \text{int} \ast T \mid \text{int} \mid (E) \]

• Consider step \text{int} \mid \ast \text{int} + \text{int}
  - We could reduce by \[ T \rightarrow \text{int} \] giving \[ T \mid \ast \text{int} + \text{int} \]
  - A fatal mistake!
    • No way to reduce to the start symbol \[ E \]
Handles

• Intuition: Want to reduce only if the result can still be reduced to the start symbol

• Assume a rightmost derivation

\[ S \rightarrow^{*} \alpha X \omega \rightarrow \alpha \beta \omega \]

• Then \( X \rightarrow \beta \) in the position after \( \alpha \) is a handle of \( \alpha \beta \omega \)
Handles continued

- Handles formalize the intuition
  - A handle is a string that can be reduced and also allows further reductions back to the start symbol (using a particular production at a specific spot)

- We only want to reduce at handles

- Note: We have said what a handle is, not how to find handles
Important Fact #2

In shift-reduce parsing, handles appear only at the top of the stack, never inside.
Why?

• Informal induction on # of reduce moves:
  • True initially, stack is empty
  • Immediately after reducing a handle
    - right-most non-terminal on top of the stack
    - next handle must be to right of right-most non-terminal, because this is a right-most derivation
    - Sequence of shift moves reaches next handle
Summary of Handles

- In shift-reduce parsing, handles always appear at the top of the stack.

- Handles are never to the left of the rightmost non-terminal.
  - Therefore, shift-reduce moves are sufficient; the need never move left.

- Bottom-up parsing algorithms are based on recognizing handles.
Recognizing Handles

- There are no known efficient algorithms to recognize handles
- Solution: use heuristics to guess which stacks are handles
- On some CFGs, the heuristics always guess correctly
  - For the heuristics we use here, these are the SLR grammars
  - Other heuristics work for other grammars
Grammars

- All CFGs
- Unambiguous CFGs
- SLR CFGs

will generate conflicts
Viable Prefixes

- It is not obvious how to detect handles

- At each step the parser sees only the stack, not the entire input; start with that . . .

\[ \alpha \text{ is a viable prefix if there is an } \omega \text{ such that } \alpha | \omega \text{ is a state of a shift-reduce parser} \]
Huh?

- What does this mean? A few things:

- A viable prefix does not extend past the right end of the handle
- It’s a viable prefix because it is a prefix of the handle
- As long as a parser has viable prefixes on the stack no parsing error has been detected
Important Fact #3 about bottom-up parsing:

For any grammar, the set of viable prefixes is a regular language.
Items

- An item is a production with a “.” somewhere on the rhs

- The items for $T \rightarrow (E)$ are
  
  $T \rightarrow .(E)$
  $T \rightarrow (.E)$
  $T \rightarrow (E.)$
  $T \rightarrow (E).$
Items continued

- The only item for $X \rightarrow \epsilon$ is $X \rightarrow \epsilon$.

- Items are often called “LR(0) items”
Intuition

• The problem in recognizing viable prefixes is that the stack has only bits and pieces of the rhs of productions
  - If it had a complete rhs, we could reduce

• These bits and pieces are always prefixes of rhs of productions
Example

Consider the input (int)

- Then (E|) is a state of a shift-reduce parse

- (E is a prefix of the rhs of T → (E)
  • Will be reduced after the next shift

- Item T → (E.) says that so far we have seen (E of this production and hope to see )
Generalization

- The stack may have many prefixes of rhs's
  \[ \text{Prefix}_1 \text{ Prefix}_2 \ldots \text{Prefix}_{n-1} \text{Prefix}_n \]

- Let \( \text{Prefix}_i \) be a prefix of rhs of \( X_i \rightarrow \alpha_i \)
  - \( \text{Prefix}_i \) will eventually reduce to \( X_i \)
  - The missing part of \( \alpha_{i-1} \) starts with \( X_i \)
  - i.e. there is a \( X_{i-1} \rightarrow \text{Prefix}_{i-1} X_i \beta \) for some \( \beta \)

- Recursively, \( \text{Prefix}_{k+1} \ldots \text{Prefix}_n \) eventually reduces to the missing part of \( \alpha_k \)
Example

Consider the string $(\text{int} \times \text{int})$: 
$(\text{int} \times |\text{int})$ is a state of a shift-reduce parse

“(” is a prefix of the rhs of $T \rightarrow (E)$
“ε” is a prefix of the rhs of $E \rightarrow T$
“int *” is a prefix of the rhs of $T \rightarrow \text{int} \times T$
The “stack of items”

\[ T \rightarrow (E) \]
\[ E \rightarrow .T \]
\[ T \rightarrow \text{int} \ast .T \]

Says

We’ve seen “(” of \[ T \rightarrow (E) \]
We’ve seen \( \varepsilon \) of \[ E \rightarrow T \]
We’ve seen \text{int} \ast \) of \[ T \rightarrow \text{int} \ast T \]
Recognizing Viable Prefixes

Idea: To recognize viable prefixes, we must

- Recognize a sequence of partial rhs’s of productions, where

- Each sequence can eventually reduce to part of the missing suffix of its predecessor
An NFA Recognizing Viable Prefixes

1. Add a dummy production $S' \rightarrow S$ to $G$

2. The NFA states are the items of $G$
   - Including the extra production

3. For item $E \rightarrow \alpha.X\beta$ add transition
   
   $E \rightarrow \alpha.X\beta \xrightarrow{X} E \rightarrow \alpha.X.\beta$

4. For item $E \rightarrow \alpha.X\beta$ and production $X \rightarrow \gamma$ add
   
   $E \rightarrow \alpha.X\beta \xrightarrow{\epsilon} X \rightarrow .\gamma$
5. Every state is an accepting state

6. Start state is $S' \rightarrow .S$
The states of the DFA are
“canonical collections of items”
or
“canonical collections of LR(0) items”

The Dragon book gives another way of constructing LR(0) items
Valid Items

Item $X \rightarrow \beta.\gamma$ is valid for a viable prefix $\alpha\beta$ if:

$$S' \rightarrow^* \alpha X \omega \rightarrow \alpha \beta \gamma \omega$$

by a right-most derivation.

After parsing $\alpha \beta$, the valid items are the possible tops of the stack of items.
Items Valid for a Prefix

An item \( I \) is valid for a viable prefix \( \alpha \) if the DFA recognizing viable prefixes terminates on input \( \alpha \) in a state \( s \) containing \( I \).

The items in \( s \) describe what the top of the item stack might be after reading input \( \alpha \).
Valid Items Example

- An item is often valid for many prefixes

- Example: The item \( T \rightarrow (.E) \) is valid for prefixes

\[
( \\
( \\
( \\
( \\
\ldots
\]
LR(0) Parsing

• Idea: Assume
  - stack contains $\alpha$
  - next input is $\dagger$
  - DFA on input $\alpha$ terminates in state $s$
• Reduce by $X \rightarrow \beta$ if
  - $s$ contains item $X \rightarrow \beta$.
• Shift if
  - $s$ contains item $X \rightarrow \beta.\dagger\omega$
  - equivalent to saying $s$ has a transition labeled $\dagger$
LR(0) Conflicts

- LR(0) has a reduce/reduce conflict if:
  - Any state has two reduce items:
    - $X \rightarrow \beta$ and $Y \rightarrow \omega$.

- LR(0) has a shift/reduce conflict if:
  - Any state has a reduce item and a shift item:
    - $X \rightarrow \beta$ and $Y \rightarrow \omega \cdot \delta$
- LR = “Left-to-right scan”
- SLR = “Simple LR”

- SLR improves on LR(0) shift/reduce heuristics
  - Fewer states have conflicts
SLR Parsing

- Idea: Assume
  - stack contains $\alpha$
  - next input is $t$
  - DFA on input $\alpha$ terminates in state $s$

- Reduce by $X \rightarrow \beta$ if
  - $s$ contains item $X \rightarrow \beta$
  - $t \in \text{Follow}(X)$

- Shift if
  - $s$ contains item $X \rightarrow \beta \cdot t \omega$
SLR Parsing continued

- If there are conflicts under these rules, the grammar is not SLR

- The rules amount to a heuristic for detecting handles
  - The SLR grammars are those where the heuristics detect exactly the handles
Precedence Declarations Digression

- Lots of grammars aren’t SLR
  - including all ambiguous grammars

- We can parse more grammars by using precedence declarations
  - Instructions for resolving conflicts
Precedence Declarations continued

- Consider our favorite ambiguous grammar:
  \[ E \rightarrow E + E \mid E \ast E \mid (E) \mid \text{int} \]
- The DFA for this grammar contains a state with the following items:
  \[ E \rightarrow E \ast E \quad E \rightarrow E + E \]
  - shift/reduce conflict!
- Declaring 
  
  "* has higher precedence than +"
  
  resolves this conflict in favor of reducing
Precedence Declarations continued

- The term “precedence declaration” is misleading
- These declarations do not define precedence; they define conflict resolutions
  - Not quite the same thing!
Naïve SLR Parsing Algorithm

1. Let $M$ be DFA for viable prefixes of $G$
2. Let $|x_1...x_n|$ be initial configuration
3. Repeat until configuration is $S|$
   • Let $\alpha|\omega$ be current configuration
   • Run $M$ on current stack $\alpha$
   • If $M$ rejects $\alpha$, report parsing error
     • Stack $\alpha$ is not a viable prefix
   • If $M$ accepts $\alpha$ with items $I$, let $a$ be next input
     • Shift if $X \rightarrow \beta. \alpha \gamma \in I$
     • Reduce if $X \rightarrow \beta. \in I$ and $a \in \text{Follow}(X)$
     • Report parsing error if neither applies
- If there is a conflict in the last step, grammar is not SLR(k)

- k is the amount of lookahead
  - In practice k = 1