02/24/2010

Recitation #6

Administrative

- Any questions on Exam?
- Parsing Generator can be downloaded here → great tool to verify FIRST and FOLLOW sets, LL1 parsing tables and conflicts
    - Try grammars from exam #1 and from Dr. Hughes' slides

FIRST / FOLLOW

- Want to show one more, very explicit, example of creating the FIRST and FOLLOW sets, because the mechanics of it are still confusing for many students
- Example from exam #1

$$\begin{aligned} \textit{Stmt} &\rightarrow \text{REPEAT } \textit{Stmt} \text{ UNTIL } \textit{Var} \\ &\mid \text{BASIC} \\ &\mid \varepsilon \\ \textit{Var} &\rightarrow \textit{Qualifier} \text{ ID} \\ \textit{Qualifier} &\rightarrow \text{ ID DOT } \textit{Qualifier} \\ &\mid \varepsilon \end{aligned}$$

- For FIRST set look at all non-terminals ($\textit{Stmt}, \textit{Var}, \textit{Qualifier}$) and **where they appear on the left-hand-side**; then calculate the FIRST set of the statements on the right
    - For non-terminal $\textit{Stmt}$ things are straightforward, because whenever a FIRST starts with a terminal, the result will be the same terminal:

      $\text{FIRST}(\text{REPEAT } \textit{Stmt} \text{ UNTIL } \textit{Var}) = \text{FIRST}(\text{REPEAT}) = \text{REPEAT}$

      $\text{FIRST}(\text{BASIC}) = \text{BASIC}$

      $\text{FIRST}(\varepsilon) = \varepsilon$

      $\Rightarrow \text{FIRST}(\textit{Stmt}) = \{\text{REPEAT, BASIC, } \varepsilon\}$
    - For non-terminal $\textit{Qualifier}$:

      $\text{FIRST}(\text{ID DOT } \textit{Qualifier}) = \text{FIRST}(\text{ID}) = \text{ID}$

      $\text{FIRST}(\varepsilon) = \varepsilon$

      $\Rightarrow \text{FIRST}(\textit{Qualifier}) = \{\text{ID, } \varepsilon\}$
    - Now for $\textit{Var}$ it gets interesting. When calculating the FIRST set of $\textit{Qualifier}$ ID , we need to take all terminals from $\text{FIRST}(\textit{Qualifier})$ excluding ε , but because it contains ε we also need to look at the FIRST set of ID:

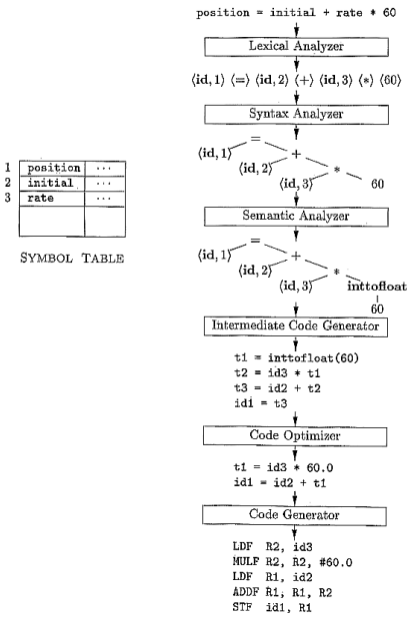
      $\text{FIRST}(\textit{Qualifier} \text{ ID}) = \text{FIRST}(\textit{Qualifier}) \backslash \{\varepsilon\} \cup \text{FIRST}(\text{ID}) = \text{ID}$

      $\Rightarrow \text{FIRST}(\textit{Var}) = \{\text{ID}\}$

- For FOLLOW set look at all non-terminals and **where they appear on the right-hand-side**; then see if the rule in question fits one of these patterns (assume the non-terminal in question is B):
    1. $A \to \alpha B \beta$. Here α could be empty and A could potentially be the same as B. Then everything in $\text{FIRST}(\beta)\backslash\{\varepsilon\}$ is in FOLLOW(B). This makes sense, because any terminal that could follow B would be in the FIRST set of β. If $\text{FIRST}(\beta)$ contains ε, see rule (2).
    2. $A \to \alpha B$ (or a production $A \to \alpha B$ where $\text{FIRST}(\beta)$ contains ε). α could be potentially an empty string. This assumes that $A \neq B$. Then everything in FOLLOW(A) is in FOLLOW(B). Because there is nothing after B, it makes sense that any terminal that could follow A could also follow B.
- Also, the endmarker $\$$ is always placed in the FOLLOW set of the start symbol
- Now, the FOLLOW set for our example is as follows:
    - The only rule where non-terminal $Stmt$ appears on the right side is $Stmt \to \text{REPEAT } Stmt \text{ UNTIL } Var$. This fits rule (1), where $\alpha = \text{REPEAT}$ and $\beta = \text{UNTIL } Var$. Now, $\text{FIRST}(\beta) = \text{FIRST}(\text{UNTIL } Var) = \text{UNTIL}$, so $\Rightarrow \text{FOLLOW}(Stmt) = \{\text{UNTIL}, \$\}$
    - The only rule where non-terminal $Qualifier$ appears on the right side is $Var \to Qualifier \text{ ID}$. Rule(1) applies with, $\alpha = \varepsilon$ and $\beta = \text{ID}$.
      There is another production rule ($Qualifier \to \text{ ID DOT } Qualifier$) that would fit pattern (2) but the non-terminal is the same on left and right, so this rule cannot apply.
      $\Rightarrow \text{FOLLOW}(Qualifier) = \{\text{ID}\}$
    - The only production rule with Var on the right side is $Stmt \to \text{REPEAT } Stmt \text{ UNTIL } Var$. Rule (2) applies with $\alpha = \text{REPEAT } Stmt \text{ UNTIL }$. Hence everything in $\text{FOLLOW}(Stmt)$ is in $\text{FOLLOW}(Var)$.
      $\Rightarrow \text{FOLLOW}(Var) = \{\text{UNTIL}, \$\}$

Review – The Big Picture

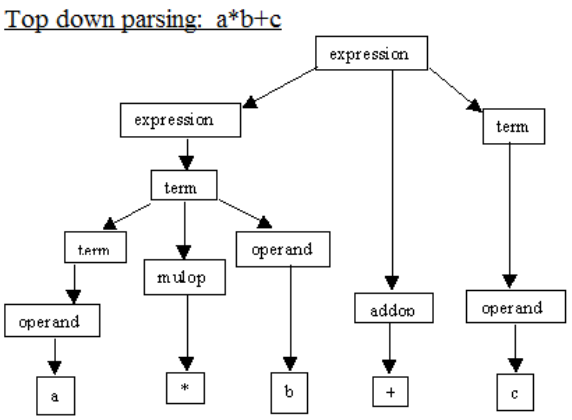
- Why Regular Expressions for Lexical Analyzer, but CFG for Syntax Analysis?
    - Everything that can be described by a regular expression can be described by a grammar → Why not use only CFGs?
    - Lexical rules usually simple, no need for powerful description of grammars
    - Regular expressions easier to understand than grammars
    - More efficient lexical analyzers can be constructed automatically from reg. expressions than from arbitrary grammars
- **Modularity** is important

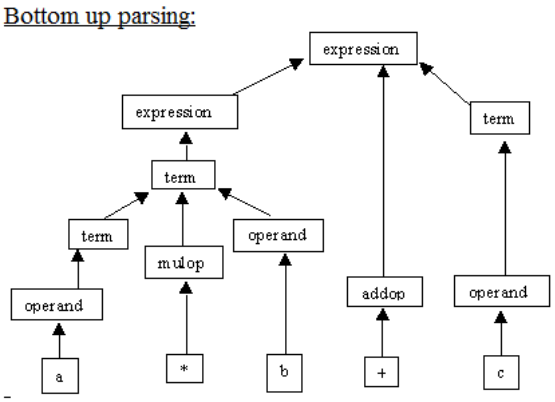


    - It should be possible to simply replace e.g. the lexical analyzer with a new one, without having to throw out the syntactic or semantic analysis
- Still to cover:
    - Semantic Analyis
    - Intermediate Code Generation
    - Code Optimization
    - Code Generation

Recursive Descent Parsing – Top-Down

- We only dealt with **predictive parsing**, which is a special case of recursive descent parsing
- **LL(1) parsing is possible in linear time** $O(N)$, where N is the number of input words. Why?
    - The input only has to be read once
    - At each non-terminal there is a unique entry in the predictive parsing table that decides which rule to apply.
    - Can be implemented by a stack
- Recursive Descent has to possibly deal with backtracking as well, i.e. if we made a wrong decision about a production we might have to scan over the input again
- Set of recursive procedures is used to process the input. **One procedure** is associated with **each non-terminal** in the grammar
- State is maintained implicitly through recursion, not explicitly
- **Runtime possibly exponential**!



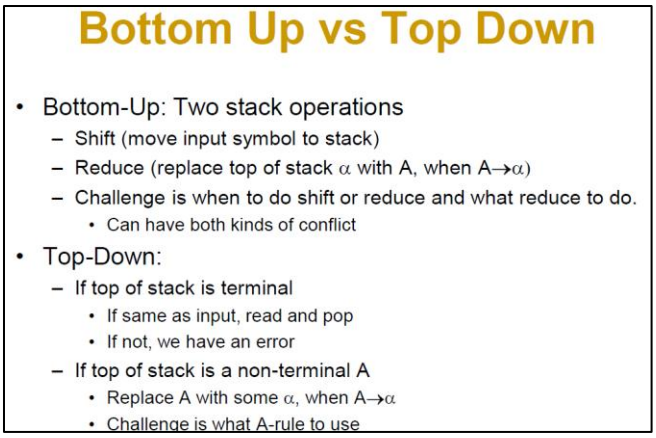


Bottom-Up Parsing

- **Bottom-up Parser**, i.e. we start with the terminals in the input string and subsequently compute recognized parse trees by going from already recognized rhs of productions to the non-terminal on the lefthand side

- In a way it's a flipped parse tree: beginning at the leaves (terminals) and working up towards the root
- Most prevalent type is based on concept of LR(k) parsers; "L" stands for left-to-right scanning, "R" constructs rightmost derivation in reverse, "k" is the number of input symbols of lookahead → usually deal with LR(1)

- **LR parsers** are table-driven (much like LL(1) parsing table)
  - Can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written
  - No backtracking required
  - All LL(1) grammars are proper subset of LR(1) grammars



Cocke, Kasami, Younger Parsing – Bottom-Up

- Parsing in $O(N^3)$ for arbitrary CFG
- Grammar needs to be in Chomsky Normal Form (CNF)
  - All productions rules have the forms
    $A \rightarrow BC$, where A, B, C are non-terminals
    $A \rightarrow a$, where a is a terminal
  - Empty string ε is not part of language
  - **Every context-free grammar** can be converted to CNF
- Uses Dynamic Programming to calculate values in the table → Little review what dynamic programming is
- Show "Exorciser" App here → CYK Parsing nicely illustrated
- In table ignore lower diagonal

- **Algorithm**:

```
CYK ( G, w )
    G = (V, Σ, R, S), Σ ∪ V = {X_1, ..., X_r}, w = w_1 w_2 ... w_n.
begin
  Initialize the 3d array B[1...n, 1...n, 1...r] to FALSE
  for i = 1 to n do
      for (X_j → x) ∈ R do
          if x = w_i then B[i, i, j] ← TRUE.
  for i = 2 to n do      /* Length of span */
      for L = 1 to n − i + 1 do      /* Start of span */
          R = L + i − 1      /* Current span s = w_L w_{L+1} ... w_R */
          for M = L + 1 to R do      /* Partition of span */
              /* x = w_L w_{L+1} ... w_{M−1}, y = w_M w_{M+1} ... w_R, and s = xy */
              for (X_α → X_β X_γ) ∈ R do
                  /* Can we match X_β to x and X_γ to y? */
                  if B[L, M − 1, β] and B[M, R, γ] then
                      B[L, R, α] ← TRUE      /* If so, then can generate s by X_α! */
  for i = 1 to r do
      if B[1, n, i] then return TRUE
  return FALSE
```

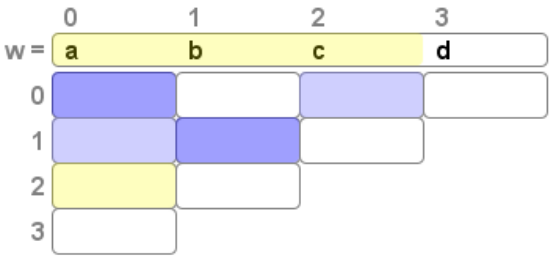
- If input is n characters, (n x n) table is built

	a	b	c	d
1	a	b	c	d
2	ab	bc	cd	
3	abc	bcd		
4	abcd			

- Each cell in the i-th row and the j-th column corresponds to the substring of length i, starting at the j-th character
    - See table on the right for an illustration

- **Intuition**:
    - CYK builds a table containing a cell for each substring
    - The cell for a substring x contains a list of non-terminals that derive x (in one or more steps)
    - First row contains the non-terminals that derive each substring of length 1
    - **For each longer substring x, we have to consider all the ways to divide x into two shorter substrings**, e.g. “abc” can be divided into (1) “a” + “bc” or (2) “ab” + “c”
    - For all possibilities check if there is a grammar rule whose right-hand side contains the concatenation of non-terminals; if yes, add non-terminal that creates rule to cell

- **Illustration**:
    - Basically, for each cell (e.g. yellow cell [2,0]), compare the cell on top in the same column [0,0] with the cell on the first cell in the top-right diagonal [1,1]
    - If there is a production rule mapping the non-terminals in the corresponding cells, add the terminal on the left side of the rule to [2,0]
    - Continue to compare in the column and diagonal, i.e. compare [1,0] and [0,2]
    - Cells that are compared are shaded in the following example:



- **Example**:
  - $S \rightarrow aSb \mid ab$
  - Transformed into CNF $G = (\{S, T, A, B\}, \{a, b\}, S, P)$:

    $S \rightarrow AB \mid AT$

    $T \rightarrow SB$

    $A \rightarrow a$

    $B \rightarrow b$
  - **Input sequence "aabb".** Table is initialized:

	a	a	b	b
1				
2				
3				
4				

  - Look at first row
    - Since we have productions $A \rightarrow a$ and $B \rightarrow b$, we can add these non-terminals

	a	a	b	b
1	A	A	B	B
2				
3				
4				

  - Look at second row
    - Are there any combinations of non-terminals on top and diagonally top right that have existing production rule
    - There is one because of $S \rightarrow AB$

	a	a	b	b
1	A	A	B	B
2		S		
3				
4				

  - Look at third row
    - Are there any combinations of non-terminals
    - There is one because of $T \rightarrow SB$

	a	a	b	b
1	A	A	B	B
2		S		
3		T		
4				

  - Look at fourth row
    - Are there any combinations of non-terminals
    - There is one because of $S \rightarrow AT$

	a	a	b	b
1	A	A	B	B
2		S		
3		T		
4	S			

  - Since the start symbol S is in the lower-left cell, the string was **parsed successfully**

- **Input sequence "aabbb"**

	a	a	b	b	b
1	A	A	B	B	B
2		S			
3		T			
4	S				
5	T				

- Since the start symbol S is NOT in the lower-left cell, the string was **parsed unsuccessfully**

- More complicated example:
  - Parenthesis expressions $G = (\{A, B, C, D, E, F, S\}, \{(,), [,]\}, S, P)$

    $S \rightarrow SS \mid AB \mid AC \mid DE \mid DF$

    $C \rightarrow SB$

    $F \rightarrow SE$

    $A \rightarrow ($

    $B \rightarrow)$

    $D \rightarrow [$

    $E \rightarrow]$

	(	[	]	[	[	]	]	)
1	A	D	E	D	D	E	E	B
2		S			S			
3					F			
4				S				
5				C				
6		S						
7		C						
8	S							

- Input sequence "([ ] [[ ]])"

- From Dr. Hughes' slides
  - $G = (\{S, A, B, C, D, E\}, \{a, b\}, S, P)$

    $S \rightarrow AB \mid BA$

    $A \rightarrow CD \mid a$

    $B \rightarrow CE \mid b$

    $C \rightarrow a \mid b$

    $D \rightarrow AC$

    $E \rightarrow BC$

	a	b	b	a
1	A,C	B,C	B,C	A,C
2	S,D	E	S,E	
3	B	B		
4	S,E			

- Input sequence "abba"

Bison – Bottom-Up Implementation

- Window setup can be downloaded here: http://gnuwin32.sourceforge.net/packages/bison.htm
- **Make sure to install it in a path without any spaces**!
- Parser generator from annotated CFG
- Converts context-free grammar into LALR(1) parser
- Take a look at WHILE language from Dr. Hughes and explain a few things
- The actual language-design process using Bison, from grammar specification to a working compiler or interpreter, has these parts:
    1. Formally specify the grammar in a form recognized by Bison (see Bison Grammar Files). For each grammatical rule in the language, describe the action that is to be taken when an instance of that rule is recognized. The action is described by a sequence of C statements.
    2. Write a lexical analyzer to process input and pass tokens to the parser. The lexical analyzer may be written by hand in C (see The Lexical Analyzer Function yylex). It could also be produced using Lex, but the use of Lex is not discussed in this manual.
    3. Write a controlling function that calls the Bison-produced parser.
    4. Write error-reporting routines.
- To turn this source code as written into a runnable program, you must follow these steps:
    1. Run Bison on the grammar to produce the parser.
    2. Compile the code output by Bison, as well as any other source files.
    3. Link the object files to produce the finished product.