CMPT 379: Principles of Compiler Design

A Course Overview

Jeffrey Leung
Simon Fraser University

Summer 2019

Contents

1 Introduction

- **Compiler:** Program which translates a source program from user intention in text form into target in machine code
  - **Interpreter:** Program which dynamically executes code
- **Bootstrapping:** Creating a compiler for a language by building a compiler for a simple subset of the language, then using the subset for the rest of the language definition
  - Can be done with a different language
- **Intermediate representation:** Machine code representation of the program which may be modified and optimized before outputting
- Challenges:
  - Differing architectures, memory hierarchies, memory sizes
  - Instruction and algorithm parallelism
  - Branch prediction
- Stages:
  - **Analysis (front-end):** Stage of compiling code which reviews and understands the code in lexical, syntax/parsing, and semantic/type-checking contexts
  - **Synthesis (back-end):** Stage of compiling code which generates and optimizes code

2 Lexical Analysis (Scanning)

- **Lexical analysis (scanning):** Transforming an input program string into tokens
  - Does not validate, simply transforms
  - Challenges:
    - Prove that the implementation captures all tokens specified by the language definition
    - Prove correctness of transformations
- **Token:** Symbol which represents a specific composable section of code
  - E.g. Open bracket, number
  - Can be denoted as T_IDENT, T_LPAREN, etc.
  - **Lexeme/token attribute:** Value of a token
    - Not all tokens have values
    - E.g. For T_INTCONSTANT, a possible value is 1
  - The same character should only represent a single token (e.g. $-$ should be represented by the same token whether it is a unary or binary operator)
- **Loop and switch scanner:** Lexical analyzer which loops over each character sequentially and categorizes it based on context

2.1 Regular Expressions

- **Formal language:** A valid set of strings from a given alphabet of symbols
  - **Symbol:** Single distinct character
    - **Alphabet:** The finite set of symbols
      - Denoted by $\sum$ or $\{\text{symbol}\}$
  - **String:** Sequence of symbols
    - **Empty string:** ε
    - Set of all strings: $\sum *$
- **Formal grammar:** Concise description of a formal language
- **Regular language:** A formal language which can be expressed through specific operations (i.e. any regular expression)
  - For each regular language, there is an equivalent finite-state automaton
  - The set of all regular languages includes:
    - The empty set
    - $\{a\}$ for all a in $\sum \varepsilon$
    - For L_1, L_2, L which are regular languages:
      - Concatenation of $L_1 \cdot L_2$ where $\{xy | x \in L_1 \text{ and } y \in L_2\}$
      - Union of $L_1 \cup L_2$
      - Kleene closure as $L^* = \bigcup_{i=0}^{\infty} L^i$
    - No other regular languages

- **Regular expression:** Concise description of a regular language
  - E.g. The set of all strings over the alphabet $\{a, b\}$ which end in abb is expressed by $(a|b) * abb$
  - Used to define tokens
  - Core operators:
    - **Alternation:** One of several options
      - E.g. $a|b = a \text{ or } b$ is a regular expression where a, b are regular expressions
    - **Concatenation:** Combination of multiple regular expressions
      - E.g. ab, a concatenation of a and b, is a regular expression where a, b are regular expressions
    - **Repetition:** An arbitrary amount of a regular expression
      - E.g. $a*$, any number of a regular expressions, is a regular expression
  - Used in lexical analysis to match the largest valid string
  - Can be expressed as a recursive tree structure or as a finite state automaton

2.2 Finite State Automata

- **Finite State Automaton:** System consisting of a finite set of states and a transition function between states
  - Notations:
    - Alphabet of symbols: Σ
    - Finite set of states: S
    - Start state: Outlined
    - Final/accepting state: Double-outlined
    - Transition function: $\sigma : S \times \Sigma = S$
  - Equivalent to a unique regular language
  - Transition of ϵ refers to no input from the string corresponding to a transition in state
    - Non-deterministic
- **Deterministic Finite Automata (DFA):** Finite State Automata where no state has more than one transition per input, and no ϵ moves exist
  - For any given string, only one path exists from the start state to the final state
  - Runtime complexity: $O(r^s)$
- **Nondeterministic Finite Automata (NFA):** Finite State Automata where at least one state has more than one transition per input, or an ϵ move exists
  - For any given string, multiple paths may exist from the state state to the final state
  - Usually larger than DFA
  - Runtime complexity: $O(2^r)$ where r is the initial cost of creating the automaton
  - **Subset Construction:** Process to convert an NFA into a DFA

* From the set of start states, given any input in the alphabet, output the set of potential resulting states. Repeat until no new sets of states are generated. Use each of the unique potential sets of states as a single DFA state.

* Runtime complexity: Can convert regex of size r to a 2^r state DFA

3 Syntax Analysis (Parsing)

- **Syntax analysis (parsing):** Translation of a set of strings into a valid structure using a grammar
  - **Language:** Description of a valid string of tokens
- Symbols and rules:
  - **Terminal symbol:** Token or base symbol which does not represent any other symbols
  - **Non-terminal symbol:** Set of strings potentially consisting of one or more terminal or non-terminal symbols
  - **Production/Rule/Production rule:** Function on a non-terminal symbol which generates a sequence of terminal and/or non-terminal symbols
    - **Expansion:** Sequence of symbols resulting from the application of a production rule

3.1 Context-Free Grammars

- **Context-Free Grammar (CFG):** Recursive notation for the structure of a valid syntax; subset of regular grammars
  - Generates a *Context-Free Language (CFL)*
  - Defined by a set of non-terminal states N, a start state $S \in N$, a set of terminal states T, and a set of productions to create a set of states
    - Each state is a string consisting of non-terminal and/or terminal symbols
    - If multiple productions exist from the start state, then in order to backtrack from a terminal state to the start state, create a new start state S' which has a single production to the original state state
    - Productions can be implemented using a transition matrix
  - E.g. Given grammar $\left\{(^i)^i | i \geq 0\right\}$ (i.e. The number of opening brackets must match the number of closing brackets) the productions are:
    - $S \to \epsilon$
    - $S \to (S)$
    - $N = \{S\}$
    - $T = \{(,)\}$
  - E.g. Given language $\left\{wcw^R | w \in (a|b)*\right\}$, a CFG which produces this language is $S \to aSa|bSb|c$
  - E.g. Given language $\left\{a^n b^m c^m d^n | n \geq 1, m \geq 1\right\}$, a CFG which produces this language is $S \to aSd|aAd; A \to bAc|bc$
  - **Ambiguous:** CFG which can produce multiple unique parse trees
    - Unacceptable for programming languages
    - Prevention techniques:
      - Rewriting grammar unambiguously
      - Separating non-terminal states by different precedence levels

        - E.g. $E \to E - E; E \to T; T \to T/T$
        - E.g. $A \to A - B; A \to B$

    · **Left/Right associative:** Property of an operator specifying which of multiple operations with the same precedence should be evaluated first

– **Recursive in non-terminal X:** CFG where, in one or more productions, X can derive a sequence of symbols including X

    * **Left recursive in non-terminal X:** CFG where, in one or more productions, X can derive a sequence of symbols which begins with X

    * **Right recursive in non-terminal X:** CFG where, in one or more productions, X can derive a sequence of symbols which ends with X

3.2 Parsing

- **Parse tree:** Tree-shaped sequence of CFG productions used to generate a valid derivation of symbols, with the root being the start symbol and children being productions of the parent (i.e. for a given production $X \to Y_1 \dots Y_n$, the children of X are $Y_1 \dots Y_n$)

  – **Leftmost derivation:** Sequence of CFG productions to create a parse tree by continually processing the leftmost non-terminal state

  – **Rightmost derivation:** Sequence of CFG productions to create a parse tree by continually processing the rightmost non-terminal state

  – E.g. For the production rules in table 1, the parse tree of $id + id * id$ is shown in figure 2

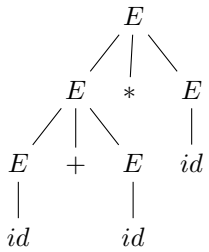
  – E.g. For the production rules in table 3, the parse tree of $id + id * id$ is shown in figure 4

Figure 1: Parse Tree Example 1: Production Rules

$$E \to E + E$$
$$E \to E * E$$
$$E \to id$$

Figure 2: Parse Tree Example 1

String: $id + id * id$



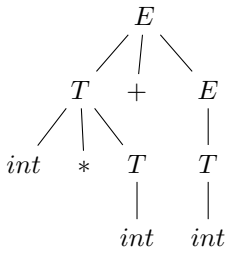
- **Pushdown automaton (PDA):** Finite State Automata system which utilizes a stack to parse a CFL

Figure 3: Parse Tree Example 2: Production Rules

$$E \to T + E$$
$$E \to T$$
$$T \to int$$
$$T \to int * T$$
$$T \to (E)$$

Figure 4: Parse Tree Example 2

String: $int * int + int$



- Notations:
  - Finite set of states: S
  - Start state: Outlined
  - Final/accepting state: Double-outlined
  - Production rules: P
- Uses a stack structure to hold symbols
- For each CFG, there exists an equivalent PDA

▪ **Shift-reduce parsing:** Method of bottom-up parsing which reduces a string to a start symbol, using reversed production rules

- Uses a left substring containing terminal or non-terminal symbols, and a right substring containing as-of-yet unprocessed symbols
- Starts with the entire string in the right substring
- Right substring utilizes a stack with the following options:
  - **Shift:** Retrieving a symbol from the stack (right substring) and placing it on the left substring
  - **Reduce:** Removing one or more symbols from the stack (right substring), processing them with a reversed production rule, and returning the result to the stack
- Example:

Stack|Input	Action
$\|x_1x_2x_3x_4$	Begin
$x_1\|x_2x_3x_4$	Shift x_1
$x_1\|y_1x_4$	Reduce x_2x_3 to y_1
$x_1y_1\|x_4$	Shift y_11
$x_1y_1\|y_2$	Reduce x_4 to y_2
$x_1y_1y_2\|$	Shift y_2

- **Shift-reduce conflict:** Situation in shift-reduce parsing where either a shift or reduce can create valid parses
  - Solution: Precedence/associativity declarations
- **Reduce-reduce conflict:** Situation in shift-reduce parsing where multiple different reduces can create valid parses
  - Solution: Rewrite grammar to no longer be ambiguous
- **Handle:** Sequence of symbol(s) at the top of the stack which is a valid reduction allowing further reductions towards the start symbol
  - **Prune:** Action of reducing a handle and pushing the result onto the stack
  - To find a handle, heuristics are used; no efficient algorithms exist

4 Parsing Algorithms: Bottom-Up

4.1 Introduction

- **Bottom-up parsing:** Reduction of a string of terminal symbols to the start symbol
  - Simulates a reversed rightmost derivation (see *shift-reduce parsing*)
- **LR(k) parsing:** Bottom-up parsing derivation which parses **L**eft-to-right to create a **R**ightmost derivation with k (usually 1) tokens of lookahead
  - Actions:
    - Shift ($f : u \to a$)
    - Reduce ($f :$ lookup production $x \to y_1 \dots y_n$)
    - Accept
    - Error
- **Dotted rule:** Parser state consisting of a single production rule and a dot ($\bullet$) where the parser currently is at
  - Notation: $A \to B \bullet C$
  - All symbols to the left of the dot have been read; all symbols to the right of the dot are predicted
  - **Configuration/itemset/state:** Set of dotted rules which are all possible productions given an in-progress parse
    - E.g. Given a set of production rules and a dotted rule, the resulting configuration set is given in table 5

Figure 5: Configuration Set Example

Production Rules	$T \to F$
	$T \to T * F$
	$F \to id$
	$F \to (T)$
Dotted Rule	$T \to T * \bullet F$
$closure(T \to T * \bullet F)$	$T \to T * \bullet F$
(Configuration set)	$F \to \bullet(T)$
	$F \to \bullet id$

- **Action/Goto (parsing) table:** Set of states and transitions which provides a programmatic method to parse an input
  - Notations:
    - $\text{action}[s, a]; a \in T$ where T is the set of terminal states
    - $\text{goto}[s, X]; X \in N$ where N is the set of non-terminal states
  - Layout: See table 6
  - To create a parsing table, create the layout, then for each itemset/state:
    - If the state transitions to a new state upon symbol X, then:

Figure 6: Layout of an action/goto table

State	**Actions** (Terminal symbols)	$	**Gotos** (Non-terminal symbols)
Number	*Shift STATE or Reduce RULE*		*Change to STATE*

- If X is a terminal symbol, then write Sα (shift to state α) in the action
- If X is a non-terminal symbol, then write α (shift to state α) in the goto

* If the state results in a reduction to non-terminal symbol X, then for each symbol in the $FOLLOW$ set of X, write Rβ (reduce by rule number β) in the action

* If the itemset has an epsilon (ϵ) rule for symbol X (i.e. $X \to \epsilon$), then for each symbol in the $FOLLOW$ set of X, write Rβ (reduce by rule number β) in the action

* If multiple shifts or reduces are possible for a given input from a given state, then use a comma to separate the possibilities

* If the itemset reaches the end of input, then in $, write acc (acceptance) instead of reducing

- **Closure:** Function to create a configuration set given a dotted rule, where the configuration set consists of the given dotted rule, all dotted rules where the symbol X after the bullet is the input of a production rule (i.e. $f : X \to Y$), and all rules which recurse upon Y_1
  - Mathematically: Given dotted rule $T \to X_1 \dots X_i \bullet X_{i+1} \dots X_n$, the configuration set consists of rule T as well as all dotted rules $X_{i+1} \to \bullet Y_1 \dots Y_m$
  - Notation: $closure(A \to B \bullet C) = \{\}$
- Example of a shift-reduce conflict: Rules $F \to id\bullet; F \to id \bullet T$
- Example of a reduce-reduce conflict: Rules $F \to id\bullet; T \to id\bullet$
- **Successor:** Function on a configuration set and a successive symbol which removes non-matching rules in the configuration set and computes a new closure on the remaining dotted rules
  - Notation: $successor(I, X) = \{\}$ where I is a configuration set and X is a successive symbol
  - E.g. Given a set of production rules and a configuration set, an example successor function is given in table 7
  - For an example diagram of a set of production rules and the states generated using the successor function, see figure 8
  - For an example diagram of a set of production rules (including epsilon rules) and the states generated using the successor function, see figure 9
- **Viable prefix:** Set of states representing the stack of a shift-reduce parser where combining it with the remaining symbols is a valid state
  - Notation: γ such that $\gamma|\omega$ is a valid state for some ω
- Process of LR(k) parsing:
  - Requires a set of production rules, an action/goto table, and an input string
  - Begin with state 0 on the stack.
  - From the row number of the current state (given by the top of the stack) and the leftmost symbol in the input, continually take the given action in the action table, given by $\text{action}[]$.

Figure 7: Successor Example

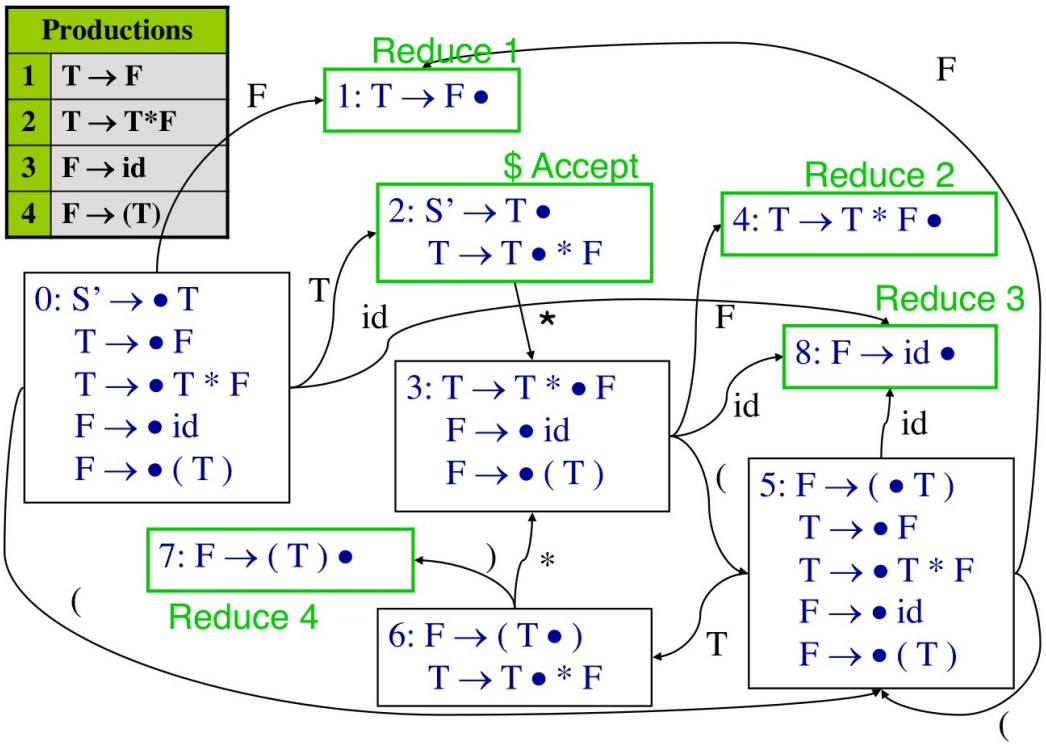
Production Rules	$S' \to T$
	$T \to F$
	$T \to T * F$
	$F \to id$
	$F \to (T)$
Configuration set I	$S' \to \bullet T$
	$T \to \bullet F$
	$T \to \bullet T * F$
	$F \to \bullet id$
	$F \to \bullet (T)$
$successor(I, \text{“(”})$ (Configuration set)	$F \to (\bullet T)$
	$T \to \bullet F$
	$T \to \bullet T * F$
	$F \to \bullet id$
	$F \to \bullet (T)$

    * If the action is $S\alpha$, then push state α onto the stack and consume the leftmost symbol of the input.
    * If the action is Rr, then:
        - Find the production rule numbered r, with $r : X \to Y_1 \dots Y_k$.
        - Pop k states from the stack.
        - Take the state Z at the top of the stack and push state $\text{goto}[Z, X]$ onto the stack.
    * If the action is ACCEPT, then complete the parsing.
    * If no action exists, then return ERROR.
  - Example: Given the production rules in figure 10 and action/goto table in figure 11, the trace of the LR(0) parse is figure 12

4.2 LR(0) Parsing

- **LR(0) parsing:** Form of LR(k) parsing where no tokens of lookahead are used, so a reduction is always executed whenever possible
  - **LR(0) grammar:** Subset of CFGs where an LR(0) construction can be generated without shift-reduce or reduce-reduce conflicts (i.e. the generated pushdown automata is deterministic)
- Grammar validity:
  - A grammar is LR(0) if and only if no shift-reduce or reduce-reduce conflicts exist in the itemsets
  - A grammar is not LR(0) if and only if at least one shift-reduce or reduce-reduce conflict exists in the itemsets

Figure 8: Example Diagram of States



4.3 SLR(1) Parsing

- **LR(1) parsing:** Form of LR(k) parsing where 1 token of lookahead is available for use
- **SLR(1)/SLR/Simple LR parsing:** Subset of LR(0) parsing where each *production rule* includes all possible $FOLLOW$ terminal symbols using 1 character of lookahead
- **SLR(1)/SLR/Simple LR grammar:** Subset of CFGs where an LR(1) construction can be generated without shift-reduce or reduce-reduce conflicts, or conflicts exist but can be resolved due to having a single $FOLLOW$ symbol
  - First/follow:
    - **First:** Set of terminal symbols which are the beginning symbols of all strings which can be derived from a given symbol
      - Notation: $FIRST(S) = \{a, b\}$ where S is a given symbol and a, b are terminal symbols
      - Example: Given a set of production rules, the First sets of the non-terminal symbols are derived in table 13
    - **Follow:** Set of terminal symbols which can follow a given symbol

Figure 9: Example Diagram of States with Epsilon Rules

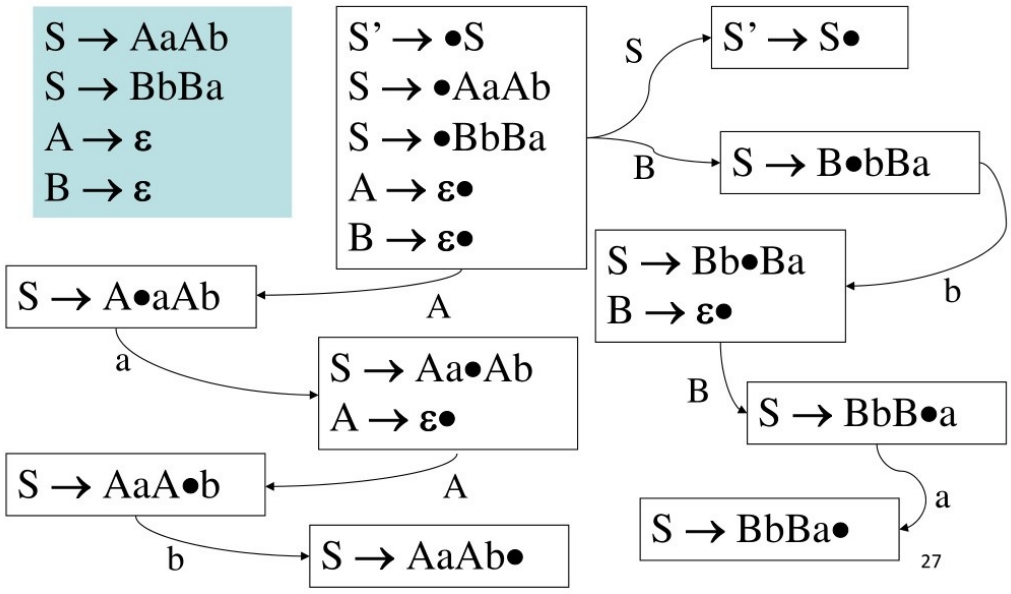


Figure 10: LR(k) Parsing Example: Production Rules

	Production Rules
1	$T \to F$
2	$T \to T * F$
3	$F \to id$
4	$F \to (T)$

- Notation: $FOLLOW(S) = \{a, b\}$ where S is a given symbol and a, b are terminal symbols
- Example: Given a set of production rules, the Follow sets of the non-terminal symbols are derived in table 14

* Example: Given a set of production rules, the First and Follow sets of the non-terminal symbols can be derived as follows:
  - See table 15
  - See table 16

– Process to determine whether a shift-reduce conflict can be eliminated by a 1-token lookahead:
  * Find the $FOLLOW$ set of the reduce rule
  * Find the next accepted symbol of the shift rule

Figure 11: LR(k) Parsing Example: Action/Goto Table

States	Actions					Gotos	
	*	(	)	id	$	T	F
0		S5		S8	2	1	
1	R1	R1	R1	R1	R1		
2	S3				ACC		
3		S5		S8		4	
4	R2	R2	R2	R2	R2		
5		S5		S8		6	1
6	S3		S7				
7	R4	R4	R4	R4	R4		
8	R3	R3	R3	R3	R3		

        * If the symbol is in the $FOLLOW$ set, then the conflict cannot be eliminated

        * Example: Given the set of production rules in table 17, the state $S \to A \bullet xB; B \to A\bullet$ has a shift-reduce conflict which cannot be resolved by 1 token of lookahead, as the parser can still shift or reduce on x because the $FOLLOW$ set of B is $\{x, \$\}$

    – Example: Given the set of production rules and transition diagram in figure 18, the trace of the SLR(1) parse of $id * id$ is table 19

- Grammar validity:

    – A grammar is SLR(1) if and only if, for each shift-reduce or reduce-reduce conflict in the configuration sets, the $FOLLOW$ sets of the conflicting production rules (not dotted rules) do not intersect

    – A grammar is not SLR(1) if and only if there exists at least one shift-reduce or reduce-reduce conflict in the configuration sets where the $FOLLOW$ sets of the conflicting production rules intersect

4.4 Canonical LR(1) Parsing

- **LR(1)/Canonical LR(1) parsing:** Subset of LR(0) parsing where each *dotted rule* includes a 1-character lookahead, rather than each production rule

    – More accurate than SLR(1) parsing

    – Notation: $\text{dotted rule}, \alpha/\beta$ where α, β are the potential $FOLLOW$ characters of the non-terminal symbol if the rule was parsed and reduced

- Grammar validity:

    – A grammar is LR(1) if and only if, for each shift-reduce or reduce-reduce conflict, the $FOLLOW$ sets of the dotted rules do not conflict

    – A grammar is not LR(1) if and only if there exists at least one itemset where conflicting dotted rules have intersecting $FOLLOW$ sets

Figure 12: LR(k) Parsing Example: Trace

Stack	Input	Action
0	(id)*id	Shift 5
0, 5	id)*id	Shift 8
0, 5, 8	)*id	Reduce using production rule 3: $F \to id$
		Pop 1 symbol from the stack (8)
		Push $goto[5, F] = 1$ onto the stack
0, 5, 1	)*id	R1: $T \to F$
		Pop 1
		$goto[5, T] = 6$
0, 5, 6	)*id	S7
0, 5, 6, 7	*id	R4, $F \to (T)$
		Pop 7, 6, 5
		$goto[0, T] = 1$
0, 1	*id	R1: $T \to F$
		Pop 1
		$goto[0, T] = 2$
0, 2	*id	S3
0, 2, 3	id	S8
0, 2, 3, 8	$	R3: $F \to id$
		Pop 8
		$goto]3, F] = 4$
0, 2, 3, 4	$	R2: $T \to T * F$
		Pop 4, 3, 2
		$goto[0, T] = 2$
0, 2	$	ACCEPT

Figure 13: First Example

Production Rules:	$A \to Bc \vert d$
	$B \to e$
$FIRST(A) =$	$\{d, e\}$
$FIRST(B) =$	$\{e\}$

4.5 LALR(1) Parsing

- **LALR(1) parsing:** Subset of LR(0) parsing which combines states when they share the same dotted rules but have different $FOLLOW$ sets
  - More specific than SLR(1) (as resulting $FOLLOW$ sets may not always be the complete $FOLLOW$ set)

Figure 14: Follow Example

Production Rules:	$A \rightarrow Bc$
	$B \rightarrow Bd \vert BC$
	$C \rightarrow e$
$FOLLOW(A) =$	$\{\$\}$
$FOLLOW(B) =$	$\{c, d, e\}$
$FOLLOW(C) =$	$\{c\}$

Figure 15: First/Follow Sets Example 1

Production Rules	$S \rightarrow AB$
	$A \rightarrow c \vert \epsilon$
	$B \rightarrow cbB \vert a$
First sets	$FIRST(A) = \{c, \epsilon\}$
	$FIRST(B) = \{c, a\}$
	$FIRST(S) = FIRST(A) = \{c, \epsilon\}$
Follow sets	$FOLLOW(A) = FIRST(B) = \{c, a\}$
	$FOLLOW(B) = \{\$\}$
	$FOLLOW(S) = \{\$\}$

Figure 16: First/Follow Sets Example 2

Production Rules	$S \rightarrow cAa$
	$A \rightarrow cB \vert B$
	$B \rightarrow bcB \vert \epsilon$
First sets	$FIRST(A) = \{c, b, \epsilon\}$
	$FIRST(B) = \{b, \epsilon\}$
	$FIRST(S) = \{c\}$
Follow sets	$FOLLOW(A) = \{a\}$
	$FOLLOW(B) = FOLLOW(A) = \{a\}$
	$FOLLOW(S) = \{\$\}$

  - Simplified/compact version of canonical LR(1)

- Grammar validity:

  - A grammar is LALR(1) if and only if, when states with matching dotted rules are merged, for each combination of states, for each matching dotted rule in the resulting configuration set, there are no conflicts in the 1-character lookaheads

  - A grammar is not LALR(1) if and only if, when states with matching dotted rules are merged, the $FOLLOW$ sets contain shift-reduce or reduce-reduce conflicts

Figure 17: SLR Shift-Reduce Conflict Example

$$S \to Ax$$
$$S \to B$$
$$B \to A$$
$$A \to yB$$

Figure 18: SLR(1) Parsing Example: Production Rules and Diagram

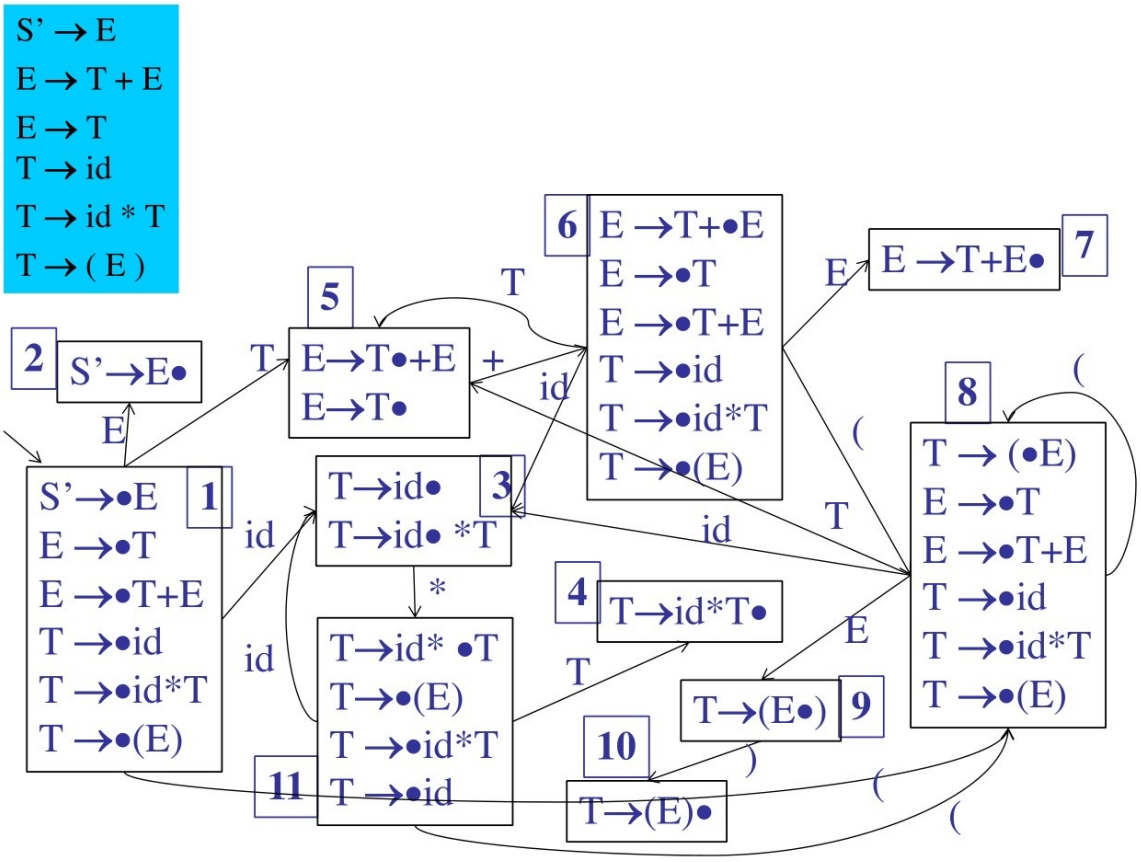


Figure 19: SLR(1) Parsing Example: Trace

Stack and Input	DFA Halt State	Action
$\|id * id\$$	1	Shift
$id\| * id\$$	$3(\notin FOLLOW(T))$	Shift
$id * \|id\$$	11	Shift
$id * id\|\$$	$3(\$ \in FOLLOW(T))$	Reduce $T \to id$
$id * T\|\$$	$4(\$ \in FOLLOW(T))$	Reduce $T \to id * T$
$T\|\$$	$5(\$ \in FOLLOW(T))$	Reduce $E \to T$
$E\|\$$		Accept

5 Parsing Algorithms: Top-Down

5.1 Introduction

- **Top-down/leftmost parsing:** Deterministic creation of a string beginning from a start symbol and continuously expanding the leftmost non-terminal symbol
  - Similarly efficient to bottom-up parsing
  - Constructs a parse tree by expanding symbols from the top to the bottom, left to right
  - As shifts and reduces are not valid actions, neither shift-reduce nor reduce-reduce conflicts are possible

5.2 Recursive Descent Parsing

- **Recursive descent parser:** Top-down parser which applies any valid production rule to the leftmost non-terminal symbol and, when no option are valid, backtracks in a brute-force manner until a valid construction is parsed (i.e. backtracking search)
  - Efficient when using a grammar which is:
    - Unambiguous
    - Left-factored
    - Free of left-recursion
- **Left-recursive grammar:** Grammar which recurses using a symbol on the left of the production rule string
  - E.g. $A \to Ab|c$ is left-recursive
  - Causes top-down parsers to create infinite loops
  - E.g. Given left-recursive grammar $A \to A*B|B; B \to a$, an equivalent non-left-recursive grammar would be $X \to BA; A \to *BA|\epsilon; B \to a$
  - E.g. Given a set of production rules, figure 20 shows the steps to remove left-recursion

Figure 20: Elimination of Left-Recursion: Example

Production Rules	$S \to Aa\|b$
	$A \to Ac\|Sd\|\epsilon$
Remove $A \to S$ recursion	$S \to Aa\|b$
	$A \to bdA'\|A'$
	$A' \to A'c\|A'ad$
Remove A' left-recursion	$S \to Aa\|b$
	$A \to bdA'\|A'$
	$A' \to cA'\|adA'$

- **Left-factoring:** Operation on a grammar which combines all common strings on the leftmost of a production rule string
  - Left-factored grammars create ambiguous rule expansions as a given terminal can expand to multiple potential states

- E.g. Given grammar $A \rightarrow Bc|D; B \rightarrow ef; D \rightarrow e$, left-factoring would create the grammar $A \rightarrow eB; B \rightarrow fc|\epsilon$

5.3 LL(1) Parsing

- **LL(1) parsing:** Top-down parsing algorithm which creates a left-to-right leftmost derivation requiring only 1 symbol of lookahead
  - Requires a grammar which is:
    - Unambiguous
    - Left-factored
    - Free of left-recursion
  - Only one possible production can exist given the next token
- **Predictive parsing table:** Table which correlates the leftmost non-terminal symbol and the next terminal symbol with the only possible production
  - No cell can have more than one string as the expansion to the rule
  - Empty cells mean parsing errors
  - Uses lookahead to resolve conflicts between multiple possible production rule applications for the same symbol
  - Layout: See figure 21

Figure 21: Predictive Parsing Table Layout

		Terminal symbols	$
Non-terminal Symbols	*Symbol*	*Production result*	

- To create a predictive parsing table:
  - For each non-terminal symbol X on the left column:
    - If there is a production rule where X expands to epsilon (ϵ), then:
      - For each terminal symbol which is in the $FOLLOW$ of symbol X, write ϵ
    - If there is a production rule where X expands to a string which is not epsilon (ϵ), then:
      - For each terminal symbol which is in the $FIRST$ of symbol X, write the expansion of the valid production rule
- E.g. In figure 22, $Y \rightarrow \epsilon$ is the valid production rule to be chosen when $+,)$, or $\$$ are the lookahead symbol
- LL(1) parsing trace:
  - Stack is used to keep track of non-terminal symbols
  - Stack and input strings are terminated with the end of input symbol ($)
  - Layout: See figure 23
  - To create an LL(1) parsing trace:
    - For the leftmost input symbol and the leftmost stack symbol:

Figure 22: LL(1) Conflict Resolution with $FOLLOW$

Production Rules	$E \to TX$
	$X \to \epsilon$
	$X \to +E$
	$T \to (E)$
	$T \to idY$
	$Y \to *T$
	$Y \to \epsilon$
$FOLLOW(Y)$	$= FOLLOW(T)$
	$= (FIRST(X) - \{\epsilon\}) + FOLLOW(E)$
	$= \{+,), \$\}$

Figure 23: LL(1) Parsing Trace Example

Stack	**Input**	**Action**
string $	*string $*	*Production rule to expand OR*
		ϵ OR
		Terminal α OR
		acc

- If the leftmost stack symbol is a non-terminal symbol, then use the leftmost input and stack symbols to find the corresponding production rule expansion from the predictive parsing table, and:
  - If the expansion is not an epsilon, then replace the leftmost stack symbol with the expansion
  - If the expansion is an epsilon (ϵ), then remove the leftmost stack symbol
- If the leftmost stack symbol is a terminal symbol and the leftmost stack and input symbols match, then consume both of them with the action *Terminal*
- If the leftmost stack and input symbols are the end of input ($), then accept the parse

- Grammar validity:
  - A grammar is LL(1) if and only if, for each pairing of a non-terminal symbol with a terminal symbol, only one possible production rule can be applied
  - A grammar is not LL(1) if and only if there exists at least one pairing of a non-terminal symbol and a terminal symbol which can expand to multiple production rules (e.g. $A \to bc; A \to bd$)

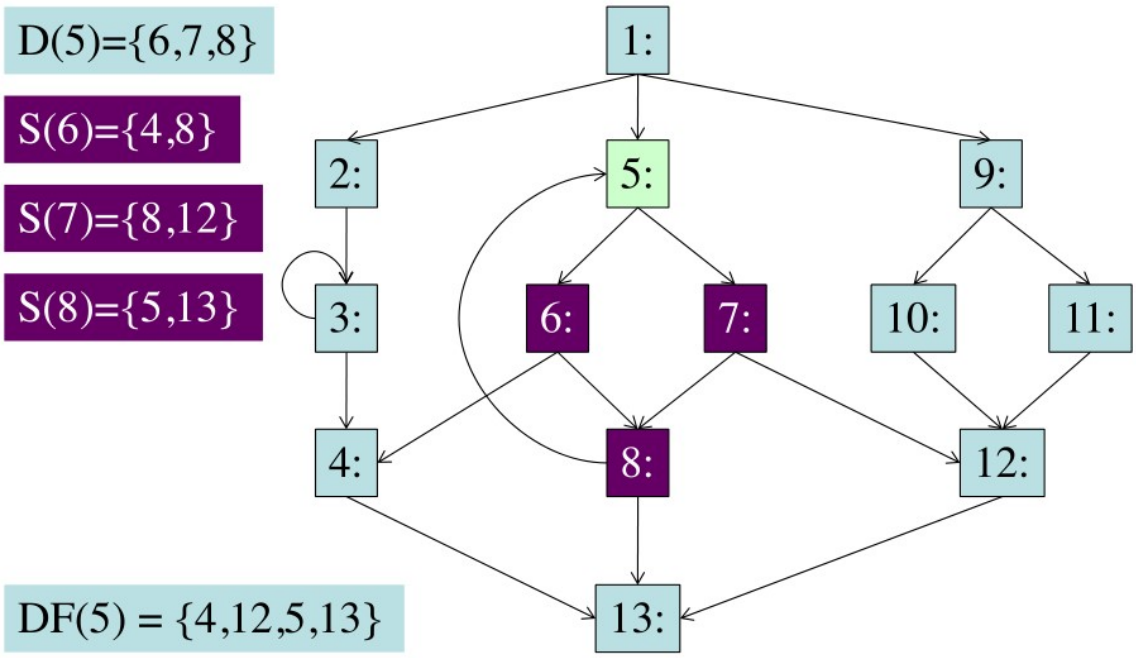
6 Syntax-Directed Transmission

- **Syntax-Directed Transmission:** Compiler implementation where the parser conveys and passes information between syntax trees
  - Generalization of a CFG where each symbol includes an additional attribute
- **Attribute grammar:** Set of syntax-directed rules which construct a meaning from a set of strings
  - Addition to a parse tree
  - **Decorate/annotate:** Description of the effect of an attribute grammar on a parse tree
- **Attribute:** Arbitrary value assigned to each symbol representation
  - **Synthesized attribute:** Parse tree attribute which is determined by the attribute value(s) of the child nodes
  - **Inherited attribute:** Parse tree attribute which is determined by the attribute value(s) of the parent and/or sibling nodes
    - E.g. Parent action { $$.in = 3; }, child action { print($0.in); }
- **S-attributed definition:** Semantic-action grammar with only synthesized attributes
- **L-attributed definition:** Semantic-action grammar where, for each production $A \to B_1 \dots B_{j-1} B_j \dots B_n$, for each $j = 1 \dots n$, each inherited attribute of B_j depends on the inherited attributes of A and the attributes of $B_1 \dots B_{j-1}$
  - Superset of S-attributed definitions

7 Semantic Analysis and Static Single Assignment

- **Semantic analysis:** Validating a program's basic requirements and type compatibilities, processing static semantic checks, and adding run-time semantic checks
  - E.g. Checking that a *main* function exists, that variables are declared, whether operand types are compatible
- **Static Single Assignment (SSA):** Property of an intermediate representation where each variable is only assigned once
  - Used to distinguish and merge values from multiple possible paths of computation to create a single resulting value
- **Dominance:** Characteristic of a node which is a mandatory predecessor of another node (i.e. any path from the root to the lower node must pass through the upper node)
  - Any node dominates itself
  - Definitions of a variable dominate its later usages
  - **Strict dominance:** Characteristic of a node which dominates a different node
- **Dominance Frontier (DF):** Set of nodes which, given a target node, are the convergence of two nodes - one which is dominated by the target node and one which is not (i.e. all nodes such that the target node dominates a predecessor but not the node itself)
  - Notation: $DF(X)$
  - Can include the node itself
  - E.g. See figure 24 which calculates the dominance frontier of node 5
  - Every node in $DF(X)$ requires a ϕ (phi) function for each variable defined in X
- **Dominator tree:** Acyclical hierarchical structure of a set of nodes, where a node dominates all of its children

Figure 24: Dominance Frontier Example



8 Programming Language Structure

- **Procedure:** Series of computational steps to be executed
  - Assumption that execution is sequential (which can be violated by concurrency)
  - Assumption that control is returned to the caller after execution (which can be violated by exceptions)
  - **Activation:** Invocation of a given procedure
    - **Activation tree:** Tree of invoked procedures where each child node was invoked by its parent node
  - **Lifetime (procedure):** Set of all the steps required to execute the associated procedure and its subprocedures
  - Stack is used to keep track of all procedures which are currently active
- **Control Flow Graph (CFG):** Diagram showing logic flow of code, with branching on conditional commands such as if/for/while loops

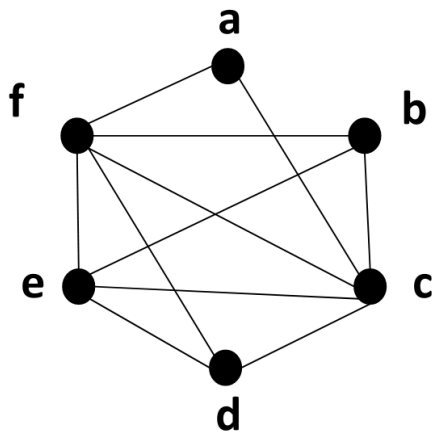
9 Optimization

- **Constant folding:** Compile-time preprocessing of constant values to avoid predictable runtime computation
- **Constant propagation:** If a variable is set to a constant, then replace all future variable uses with the constant (until the variable is reassigned)
- **Copy propagation:** Simplification of multiple computations of the same values
- **Dead code elimination:** Removal of code which will never be executed, used, or have any side effects
- **Algebraic simplification:** Basic arithmetic and algebraic simplifications which are computed during compile-time such as addition by 0 or multiplication by 0 or 1
- **Reduction of strength:** Replacement of exponentiation with multiplication
- Correctness: Optimizations must not change the meaning of a performance

10 Register Allocation

- Intermediate representation uses unlimited temporary storage locations, which must be translated to a limited number of registers
- **Live:** Variable which has been defined and is being used or will be used later
  - Should be loaded into a register for quicker access
- **Liveness analysis:** Detection of when variables are and are not live
  - Variables can share a register if, at all points in the program, no more than one of them are live
  - At the end of each line of code, create a set of all live variables
  - **Live range:** Locations in the program where a variable is live
  - **Live interval:** Start and end line numbers which represent from when a variable is first defined to its last usage
    - Greedy algorithm can be applied to allocate registers based on when the live interval for a variable ends
- **Register Interference Graph (RIG):** Undirected graph where each node represents a unique temporary variable, and an edge (t_1, t_2) exists iff they are both live at any given point in the program
  - For two variables v_1, v_2, if there is no edge connecting them, they can be allocated to the same register
  - E.g. For the RIG in figure 25, a, c cannot be in the same register, but a, d can be in the same register

Figure 25: Example of a Register Interference Graph



- **Spilling:** Operation which allocates a memory location for a variable when there are not enough registers to store all live variables
  - When a variable must be spilled, the live range of the variable is reduced and the RIG is updated
- k-coloring problem: Given a graph, can k colors be applied to the nodes such that each node has only one color, and no connected nodes share the same color?

- Apply the k-coloring problem to find the minimum possible coloring for the graph, where k also equals the minimum number of registers required
- If k is not large enough, then a node is spilled
- *Optimistic coloring heuristic algorithm:*
  * Choose an arbitrary k to attempt
  * Remove a node with less than k neighbors and push it into a stack (repeat until the graph is empty)
  * Pop a node and connect it to its neighbors in the graph, then assign either an existing color or a new color (repeat until the stack is empty)

- **Linear scan register alllocation:** Algorithm which detects when a variable needs to be spilled
  - Efficient
  - Less effective than graph coloring