CS 460 - Compilers

Arian Izadi

Spring 2024

1 Languages

Syntax is the rules for what a syntactically correct program looks like. Semantics is the meaning of a program.

When does it matter the order of evaluation (right to left vs left to right)? When the code has side effects, an example of this is postfix vs prefix increment (a++ vs ++a).

Compilers for a language L, move from front end $\rightarrow$ intermediate representation $\rightarrow$ back end.

- Front end: Lexical Analysis, Syntax Analysis, and Semantic Analysis
- Intermediate: Intermediate Code
- Back end: Optimizer and Code Generation

1.1 Lexical Analysis & Scanning

Lexical analysis, a scanner, is the process of converting a stream of characters into a stream of tokens.

1. Find all terminals in the grammar.
2. Write the Scanner.
   (a) Do we use a DFA, NFA, or PDA?
   (b) Look at token types. All tokens can be expressed by a regular expression.
      i. Symbols: Semicolon, commas, etc.
      ii. Keywords: for, while, etc.
      iii. Variables: x, y, etc.
      iv. Numbers: 1, 3.14, 0x64, etc.

Chomsky Language Hierarchy

- Type 0: Unrestricted (Turing Machines)
- Type 1: Context Sensitive
- Type 2: Context Free (PDA)
- Type 3: Regular Expressions (NFA, DFA)

Both RE and CFG have 1 non-terminal on the left of any combination of terminals and non-terminals on the right.

Example 1:

$S \rightarrow X$ $\quad$ $X \rightarrow aXb|d$ $\quad$ not regular: $a^n db^n$

Example 2:

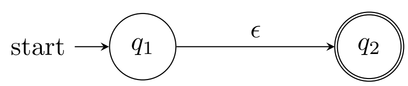
$S \rightarrow X$ $\quad$ $X \rightarrow aX|b$ $\quad$ regular: a^*b

Example 3:

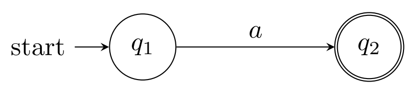
$S \rightarrow X$ $\quad$ $X \rightarrow aY|\epsilon$ $\quad$ $Y \rightarrow bX$ $\quad$ regular: $(ab)^*$

An NFA for recognizing tokens, construct NFA for each construct of RE.

$\mathcal{E}$:



$a\epsilon\Sigma$:



Any RE can be turned into an NFA using these rules. If all the tokens of a language are represented by RE's, $r_1, \ldots, r_n$. Create an NFA for each RE.

2 Parsing

2.1 Types of Parsers

- $LL(k)$ – leftmost derivation
  - (Hard Explain) Always develop the left-most non terminal in a sentential form.
  - Above means: Start at the start symbol and work towards the input.
- $LR(k)$ – rightmost derivation
  - Reverse rightmost derivation.
  - Above means: Start at input and work backwards to the start symbol.

Types of $LL(k)$ parsers:

- $LL(0)$ – No look ahead and no left recursion.
- $LL(1)$ – One look ahead and no left recursion.
- $LL(k)$ – k look ahead and no left recursion.

Example 1: a^*b^* $\quad \delta(a^*b^*) = \epsilon, a, b, aa, ab, bb, \ldots$

$S \to A \qquad A \to aA|B \qquad B \to bB|\epsilon$

Try $w = aabb$:

$S \to_{LM} A \to_{LM} aA \to_{LM} aaA \to_{LM} aaB \to_{LM} aabB \to_{LM} aabbB \to_{LM} aabb$

What if B could generate strings starting with an a?

$S \to A \qquad A \to aA|B \qquad B \to bB|a|\epsilon$

Try $w = aabb$:

It cannot be parsed by an $LL(1)$ parser.

A is not $LL(1)$ when

- $\exists w_1 \in \lambda(\alpha_1)$ or $\exists w_2 \in \lambda(\alpha_2)$
- $w_1 = aw_1$ or $w_2 = aw_2$

Example 2: A set representing tokens that a sentential form can start with: Call it $FIRST(\alpha)$

1. ϵ : FIRST $(\epsilon) = \{\epsilon\}$
2. $a \in \Sigma$: FIRST $(a) = \{a\}$
3. a mixed string: $Y = Y_1Y_2 \ldots Y_n$
   (a) FIRST $(Y_1 \ldots Y_n) \supseteq$ FIRST (Y_1) $\{\epsilon\}$
   (b) if $\epsilon \exists$ FIRST (Y_1):
      i. $Y_1, Y_2, \ldots, Y_n \to \epsilon$
      ii. $Y_1, Y_2, \ldots, Y_n \to Y_2, \ldots, Y_n$

Example 3: $X \to X_1|\ldots|X_m$

FIRST (X) = FIRST $(X_1) \cup$ FIRST $(X_2) \cup \ldots \cup$ FIRST (X_m)

Example 4: General Case

For $A \to \alpha_1|\alpha_2|\ldots|\alpha_n$

if $\exists i, j$ such that $i \neq j$ and FIRST $(\alpha_i) \cap$ FIRST (α_j) $\neq \emptyset$ then A is not $LL(1)$.

MUST BE DONE PAIRWISE, only needs to exist one pair to not be $LL(1)$.

$\forall i, j : i \neq j\hat{}(1 <= i, j <= n) : FIRST(\alpha_i) \cap FIRST(\alpha_j) = \emptyset$ then A is probably (necessary, but not sufficient) $LL(1)$.

Example 5: General Case

For $A \to \alpha_1|\alpha_2|\ldots|\alpha_n$

$\vdots$

$A_m \to \alpha_{m1}|\alpha_{m2}|\ldots|\alpha_{mn_m}$

Example 6:

$A \to B|C$

$\vdots$

$B \to \ldots$

$C \to \ldots$

FIRST (B) = {a, b} FIRST (C) = {c, f}

$w = a\beta$ where $a \in \Sigma$ and $\beta \in \Sigma^*$

* a is the lookahead token.

$A \to_{LM} B \to_{LM} aX$

* aX matches $a\beta$, so we can derive β from X.

2.2 Grammar Transformation

2.2.1 Left vs Right Recursion

$A \to Aa|b$

$\delta(A) = \{b, ba, baa, \ldots\} = ba^*$

FIRST (A) = {b} FIRST (Aa) = {b}

* Both have $b \to$ and is not $LL(1)$.

* Just make it right recursive.

$A \to bA'$

$A' \to aA'|\epsilon$

In General:

$A \to A\alpha_1|A\alpha_2|\ldots|A\alpha_n|\beta_1|\beta_2|\ldots|\beta_m$

$\delta(A) = (B_1|\ldots|B_n)(\alpha_1|\ldots|\alpha_n)$

$A \to \beta_1 A'|\ldots|\beta_m A'$

$A' \to \alpha_1 A'|\ldots|\alpha_n A'|\epsilon$

Example 4.6:

$E \to E + b|b$ similar to $A \to A\alpha|\beta$

$E \to bE'$

$E' \to +bE'|\epsilon$

Example 4.7:

$E \to E + b|E - c|b|C$ similar to $A \to A\alpha_1|A\alpha_2|\beta_1|\beta_2$

$E \to bE'|CE'$

$E' \to +bE'|-cE'|\epsilon$

2.2.2 Common Subexpressions

Example 4.8:

$A \to abC|abD$

* is this LL (1)?

No, because FIRST $(abC) \cap$ FIRST $(abD) \neq \emptyset$.

* if we change it to $A \to abA'$ and $A' \to c|D$, then it is LL (1).

* is this LL (1)?

Only if FIRST $(c) \cap$ FIRST $(D) = \emptyset$.

For the following:

$A \to A_1|\ldots|A_n$

Let α be the longest common prefix of $A_1, \ldots, A_n$ such that $A_i = \alpha A_i'$. Then $A \to \alpha A'$ and $A' \to A_1|\ldots|A_n$.

2.2.3 Grammar Ambiguity

Definition: A grammar is ambiguous if there exists a string w such that there are two or more different parse trees for the same string.

$A \to aA|bA|c$ with $w = ababc$.

$A \to aA \to abA \to abaA \to ababA \to ababc$

Simple Example:

$A \to aA|a|\epsilon$

Example 4.9:

$E \to E + E|E - E|id$ is this ambiguous?

Yes, because $id+id-id$ can be parsed as $(id+id)-id$ or $id + (id - id)$.

PHASE 2 ADVICE: AST is never the answer to the left hand side, use a subclass of it.

What if we want +, -, *, /?

$E \to E + E | E - E | E * E | E/E | id$

Consider $id + id * id$.

* is higher precedence than +, so $id + (id * id)$.

When dealing with precedence, the lower precedence should be higher in the parse tree.

$E \to E + T | E - T | T$

$T \to T * F | T/F | F$

$F \to id | (E)$

* The bracket is the highest precedence.

* This is also bad because it is left recursive.

Eliminate Left Recursion:

$E \to TE'$

$E' \to +TE' | -TE' | \epsilon$

$T \to FT'$

$T' \to *FT' | /FT' | \epsilon$

$F \to id | (E)$

2.3 Recursive Descent Parsing

$A \to B | C$

$B \to b | \epsilon$

$C \to c$

* lookahead = $

$A \to B \to \epsilon$

```
parse_x();
parse_y();
match('a');
parse_z();
```

```
parse_A() {
  if (lookahead == 'b') {
    parse_B();
  } else if (lookahead == 'c') {
    parse_C();
  } else {
    // will not check error vs epsilon
    // this is a problem
    // if none of alpha i are epsilon
  }
}
```

2.4 Rules for Follow Sets

1. if S is the start symbol, then $\$ \in$ FOLLOW (S)
2. if $A \to \alpha B \beta$ then FIRST $(\beta) - \{\epsilon\} \subseteq$ FOLLOW (B)
3. if $A \to \alpha B$ then FOLLOW $(A) \subseteq$ FOLLOW (B)
4. if $A \to \alpha B \beta$ and FIRST (β) contains ϵ then FOLLOW $(A) \subseteq$ FOLLOW (B)