Midterm Review and Catch Up: Front End

Wei Le

2015.10
Topics

- Overview
- Lexical analysis
- Syntax analysis
- Semantic analysis
Overview

- What is a compiler? (tasks, difference between compilers and interpreters)
- Compiler history (IBM 704, 1954 FORTRAN project)
- Front end (machine independent, is it valid code), back end (code generation, language independent)
- Phase (conceptual stages) and pass (program representation a compiler goes through)
The Structure of A Compiler

```
1  position ...
2    initial ...
3     rate   ...

SYMBOL TABLE
```

```
position = initial + rate * 60

Lexical Analyzer

(id, 1) (=) (id, 2) (+) (id, 3) (*) (60)

Syntax Analyzer

(id, 1) = (id, 2) + (id, 3)

(id, 2) + (id, 3) * inttofloat 60

Semantic Analyzer

Intermediate Code Generator

t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

Code Optimizer

t1 = id3 * 60.0
id1 = id2 + t1

Code Generator

LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```
Lexical Analysis

- What is a token/lexeme?
- Specifications of tokens: regular expressions and operations on regular expression
- Recognizers of tokens: finite automata
  - Regular language to NFA, NFA to DFA (algorithms)
- Conflicts (lookahead and rules): < or <=, /*/*/  
  - nested comments, stack based lexer (Advanced reading: Lexical Analysis with ANTLR)
Token and Lexeme

- **Token**: a syntactic category
- **Lexeme**: instance of the token

<table>
<thead>
<tr>
<th>Token</th>
<th>Sample lexemes</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyword</td>
<td>if, else, for, while,...</td>
</tr>
<tr>
<td>whitespace</td>
<td>‘ ‘, ‘\t’, ‘\n’, ...</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt;, &gt;, =, =!, =!, ...</td>
</tr>
<tr>
<td>identifier</td>
<td>total, score, name, ...</td>
</tr>
<tr>
<td>number</td>
<td>1, 3.14159, 0, ...</td>
</tr>
<tr>
<td>literal</td>
<td>“Super nice cool compiler”, “ComS”, ...</td>
</tr>
</tbody>
</table>
Operations on Languages

- Single character
  - \( c = \{"c"\} \)

- Epsilon
  - \( \varepsilon = \{"\"\} \)

- Union
  - \( A + B = \{s | s \in A \text{ or } s \in B\} \)

- Concatenation
  - \( AB = \{ab | a \in A \text{ and } b \in B\} \)

- Iteration
  - \( A^* = \bigcup_{i \geq 0} A^i \) where \( A^i = A \ldots (i \text{ times}) \ldots A \)
Example

- $L = \{A, B, ..., Z, a, b, ..., z\}$, $D = \{0, 1, ..., 9\}$
- $L + D$
  - set of letters and digits, each of which strings is either one letter or one digit
  - $\{A\}, \{g\}, \{1\}, ...$
- $LD$
  - set of strings of length two, each consisting of one letter followed by one digit
  - $\{c4\}, \{j8\}, \{y6\}, ...$
- $L^4$
  - set of all 4-letter strings
  - $\{1234\}, \{7416\}, \{2592\}, ...$
Lexical Analysis

Regular expression and DFA/NFA

Diagram of a finite automaton with transitions labeled by symbols a, b, c, d, e, f, and g.
Lexical Analysis

Step 0

Step 1
Lexical Analysis

Step 2

Step 3

Step 4
Syntax Analysis - Basic Concepts

- Specification of language syntax: CFG
- Recognize the language (whether a string belongs to a language): parsing
- Parse tree
- Derivation (a sequence of production): leftmost derivation and rightmost derivation (are you deriving it yourself or algorithm derives it?)
Syntax Analysis - Basic Concepts

- Top-down parsing, bottom-up parsing
- Grammar ambiguity, left factored grammar, left recursion
- Recursive Descendant, LL(1), LR(1), LALR(1) parsing, grammar and language – intuitive understandings
  - general rules
    - LL(1): when you expand a non-terminal, look ahead of one token, do you have only one choice? Ambiguity: one parse tree?
    - LR(1): can handle left recursion
    - Left recursion: not LL(1), also cannot be parsed by recursive descendant parser
  - formal proof based on the definition – constructing parsing tables
  - the grammar is some known language, we then can explain the grammar using the language properties
Syntax Analysis - Algorithms

- Know what is the algorithm
- Simulate the algorithm on input
- Recursive descendant parsing algorithm
- Table driven parsing algorithm
- Constructing LL(1) parsing table: compute FIRST, FOLLOW set
- Constructing LR(1) parsing table: LR(1) items, LALR(1) items, DFA
Table Driven Predictive Parser
WR1-Q6

- LR(1) item
- LALR(1) item: [, a/b]
- LR(1), LALR(1), LL(1) languages: LALR(1) ⊂ LR(1), LL(1) ⊂ LR(1)
- DFA (final state – reduce state)
- Constructing LR Parsing Table
Closure Operation

- The operation of extending the context with items is called the **closure operation**

\[
\text{Closure(Items)} = \\
\quad \text{repeat} \\
\quad \quad \text{for each } [X \rightarrow \alpha \bullet Y \beta, a] \text{ in Items} \\
\quad \quad \quad \text{for each production } Y \rightarrow \gamma \\
\quad \quad \quad \quad \text{for each } b \in \text{First}(\beta a) \\
\quad \quad \quad \quad \quad \text{add } [Y \rightarrow \bullet \gamma, b] \text{ to Items} \\
\quad \text{until } \text{Items is unchanged}
\]
Closure Operation: An Example

- Consider a context with the item
  \[ E \rightarrow E + ( \cdot E ), + \]
- We expect next a string derived from \( E \) +
- There are two productions for \( E \)
  \[ E \rightarrow \text{int} \quad \text{and} \quad E \rightarrow E + ( E ) \]
- We describe this by extending the context with two more items:
  \[ E \rightarrow \cdot \text{int}, ) \]
  \[ E \rightarrow \cdot E + ( E ), ) \]
The table for a fragment of our DFA

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g6</td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r_{E \rightarrow \text{int}}</td>
<td>r_{E \rightarrow \text{int}}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s8</td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r_{E \rightarrow E+(E)}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

E → int
E → int on ), +
E → E + (E)
on $, +

Parsing Table
Computing First Sets

Definition \[ \text{First}(X) = \{ b \mid X \Rightarrow^* b\alpha \} \cup \{ \varepsilon \mid X \Rightarrow^* \varepsilon \} \]

1. \[ \text{First}(b) = \{ b \} \]

2. For all productions \( X \rightarrow A_1 \ldots A_n \)
   - Add \( \text{First}(A_1) - \{ \varepsilon \} \) to \( \text{First}(X) \). Stop if \( \varepsilon \not\in \text{First}(A_1) \)
   - Add \( \text{First}(A_2) - \{ \varepsilon \} \) to \( \text{First}(X) \). Stop if \( \varepsilon \not\in \text{First}(A_2) \)
   - \( \ldots \)
   - Add \( \text{First}(A_n) - \{ \varepsilon \} \) to \( \text{First}(X) \). Stop if \( \varepsilon \not\in \text{First}(A_n) \)
   - Add \( \varepsilon \) to \( \text{First}(X) \)
   (ignore \( A_i \) if it is \( X \))
Computing Follow Sets

Definition \( \text{Follow}(X) = \{ b | S \rightarrow^* \beta X b \in \omega \} \)

1. Compute the First sets for all non-terminals first
2. Add \$$ to \text{Follow}(S)$$ (if \( S \) is the start non-terminal)

3. For all productions \( Y \rightarrow \ldots X A_1 \ldots A_n \)
   - Add \( \text{First}(A_1) - \{ \varepsilon \} \) to \( \text{Follow}(X) \). Stop if \( \varepsilon \notin \text{First}(A_1) \)
   - Add \( \text{First}(A_2) - \{ \varepsilon \} \) to \( \text{Follow}(X) \). Stop if \( \varepsilon \notin \text{First}(A_2) \)
   - ...
   - Add \( \text{First}(A_n) - \{ \varepsilon \} \) to \( \text{Follow}(X) \). Stop if \( \varepsilon \notin \text{First}(A_n) \)
   - Add \( \text{Follow}(Y) \) to \( \text{Follow}(X) \)
WA1-Q3

- Recursive descendant parsers
- Simulate recursive descendant parsing algorithm
- Left recursion
- Parser capabilities
leftmost derivation, rightmost derivation (people parsing)
parse tree
grammar ambiguity
left factor, left recursion
First, Follow
LL(1) parsing table
Left-Factoring Example

• Recall the grammar
  \[ E \rightarrow T + E | T \]
  \[ T \rightarrow \text{int} | \text{int} \ast T | (E) \]

• Factor out common prefixes of productions
  \[ E \rightarrow T \times \]
  \[ X \rightarrow + E | \varepsilon \]
  \[ T \rightarrow (E) | \text{int} \ Y \]
  \[ Y \rightarrow * T | \varepsilon \]
Elimination of Left Recursion

- Consider the left-recursive grammar
  \[ S \rightarrow S \alpha \mid \beta \]

- \( S \) generates all strings starting with a \( \beta \) and followed by a number of \( \alpha \)

- Can rewrite using right-recursion
  \[ S \rightarrow \beta \ S' \]
  \[ S' \rightarrow \alpha \ S' \mid \varepsilon \]
Elimination of Left Recursion

- In general
  \[ S \rightarrow S \alpha_1 | \ldots | S \alpha_n | \beta_1 | \ldots | \beta_m \]
- All strings derived from \( S \) start with one of \( \beta_1, \ldots, \beta_m \) and continue with several instances of \( \alpha_1, \ldots, \alpha_n \)
- Rewrite as
  \[ S \rightarrow \beta_1 S' | \ldots | \beta_m S' \]
  \[ S' \rightarrow \alpha_1 S' | \ldots | \alpha_n S' | \varepsilon \]
First Sets Example

• Recall the grammar
  \( E \rightarrow T \ X \)
  \( T \rightarrow ( \ E \ ) \mid \text{int} \ Y \)
  \( X \rightarrow + \ E \mid \varepsilon \)
  \( Y \rightarrow * \ T \mid \varepsilon \)

• First sets
  \( \text{First}(\ ( \ )\ ) = \{\ ( \ )\} \)
  \( \text{First}(\ ( \ )\ ) = \{\ ( \ )\} \)
  \( \text{First}(\ \text{int}\ ) = \{\ \text{int}\} \)
  \( \text{First}(\ + \ ) = \{\ + \} \)
  \( \text{First}(\ * \ ) = \{\ * \} \)
  \( \text{First}(\ T \ ) = \{\text{int}, ( \ )\} \)
  \( \text{First}(\ E \ ) = \{\text{int}, ( \ )\} \)
  \( \text{First}(\ X \ ) = \{+, \varepsilon\} \)
  \( \text{First}(\ Y \ ) = \{\ast, \varepsilon\} \)
Follow Sets Example

- Recall the grammar
  \[
  E \to T \, X \\
  T \to ( \, E \, ) \mid \text{int} \, Y \\
  X \to + \, E \mid \epsilon \\
  Y \to * \, T \mid \epsilon
  \]

- Follow sets
  \[
  \text{Follow( + )} = \{ \text{int}, ( ) \} \\
  \text{Follow( * )} = \{ \text{int}, ( ) \} \\
  \text{Follow( )} = \{ \text{int}, ( ) \} \\
  \text{Follow( E )} = \{ ( ) \}, \$ \} \\
  \text{Follow( X )} = \{ \$, ( ) \} \\
  \text{Follow( T )} = \{ +, ( ) \}, \$ \} \\
  \text{Follow( )} = \{ +, ( ) \}, \$ \} \\
  \text{Follow( int) } = \{ *, +, ( ) \}, \$ \}
  \]
Constructing a Parsing Table

- Construct a parsing table $T$ for CFG $G$

- For each production $A \rightarrow \alpha$ in $G$ do:
  - For each terminal $b \in \text{First}(\alpha)$ do
    - $T[A, b] = \alpha$
  - If $\alpha \rightarrow^* \epsilon$, for each $b \in \text{Follow}(A)$ do
    - $T[A, b] = \alpha$
LL(1) Parsing Table: an Example

- Left-factored grammar
  \[ E \rightarrow TX \]
  \[ T \rightarrow (E) | \text{int } Y \]
  \[ X \rightarrow +E | \varepsilon \]
  \[ Y \rightarrow *T | \varepsilon \]

- The LL(1) parsing table ($\$ is a special end marker)$:

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th></th>
<th></th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td>(E)</td>
<td></td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>E</td>
<td>TX</td>
<td></td>
<td>TX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>+E</td>
<td></td>
<td>ε</td>
<td>ε</td>
<td>ε</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>*T</td>
<td>ε</td>
<td>ε</td>
<td>ε</td>
<td>ε</td>
<td></td>
</tr>
</tbody>
</table>
WA1-Q2

- left factor
- left recursion
- top down parsing (LL(k))
Top Down Parsing

- top-down parsing expands a parse tree from the start symbol to the leaves
- terminals are seen in order of appearance in the token stream
- the parse tree is constructed:
  - from the top
  - from left to right
- Basic issue: when expanding a nonterminal, which right hand side should be selected?
- Solution I: exhaustively try them all (recursive descendant)
- Solution II: look at input tokens to select (LL(k))
Grammar and language types
WA1-Q5

- Bottom up parsing algorithm
Semantic Analysis - Basic Concepts

- Scope: static and dynamic scope
- Scoping rules
- Symbol table
- Type (static type, dynamic type), type system (specifies which operations are valid for which types), type rules (analysis of the type rules), type inference (what is the type for each expression), type checking (is inferred type consistent with static declared types? – static and dynamic type checking), subtype, type environment
- Soundness of the type system
- Tradeoffs of type system design: safety and expressiveness
- strongly typed language and weakly typed language
- Advanced: SELF_TYPE (type variable for returning subtypes, not a dynamic type – where we can and cannot use it?)
Semantic Analysis - Algorithms

- Symbol table implementation
- Type checking
Understand type environment and type rules
Type Rules and Type environment

Type rules have general format:

\[
\vdash O, M, C \vdash e : T
\]

- \(O\) environment for object (types of free variables)
- \(M\) environment for methods
- \(C\) containing class
- The dots above the horizontal bar stand for other statements about the types of sub-expressions of \(e\). These other statements are hypotheses of the rule; if the hypotheses are satisfied, then the statement below the bar is true.
Subtyping: inheritance tree
Static type, dynamic type, type rule analysis
Let With Initialization

Now consider \texttt{let} with initialization:

\[
\begin{align*}
O & \vdash e_0 : T_0 \\
O[T_0/x] & \vdash e_1 : T_1
\end{align*}
\]

\[O \vdash \texttt{let} \ x : T_0 \gets e_0 \ \text{in} \ e_1 : T_1\]  \hspace{1cm} \text{[Let-Init]}

This rule is weak. Why?
Examples of Wrong Typing Rules

- Consider the following Cool class definitions

```java
Class A { a() : int { 0 }; }
Class B inherits A { b() : int { 1 }; }
```

- An instance of B has methods “a” and “b”
- An instance of A has method “a”
  - A type error occurs if we try to invoke method “b” on an instance of A
Examples of Wrong Typing Rules

• Now consider a hypothetical let rule:

\[
\frac{O \vdash e_0 : T \quad T_0 \leq T \quad O[T_0/x] \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1}
\]

• How is it different from the correct rule?

• The following bad program is well typed

  let x : B ← new A in x.b()

• Why is this program bad?
Examples of Wrong Typing Rules

• Now consider a hypothetical let rule:

\[
\frac{O \vdash e_0 : T \quad T \leq T_0 \quad O[T/x] \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1}
\]

• How is it different from the correct rule?

• The following good program is not well typed

\[
\text{let } x : A \leftarrow \text{new } B \text{ in } \{ \ldots x \leftarrow \text{new } A; x.a(); \}
\]

• Why is this program not well typed?
Examples of Wrong Typing Rules

How it is different from the previous Let rule?

\[ O \vdash e_0 : T \quad T \leq T_0 \quad O \vdash e_1 : T_1 \]

\[ O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1 \]
Examples of Wrong Typing Rules

• The following good program does not typecheck
  \[
  \text{let } x : \text{Int} \leftarrow 0 \text{ in } x + 1
  \]

• Why?
Soundness of the type rules and type systems
Type checking algorithm
Let Example

```
AST
Type env.
Types

O ⊨ let x : T₀ in : int

O[T₀/x] ⊨ let y : T₁ in : int

(O[T₀/x])[T₁/y] ⊨ E(x, y) : int

(O[T₀/x])[T₁/y] ⊨ x : T₀

(O[T₀/x])[T₂/x] ⊨ F(x, y) : int

O[T₀/x] ⊨ let x : T₂ in : int
```

+ : int
Scope and symbol table implementation
Self-Type

- Static type checking but with "dynamic type flavor"
- Understand the flexibility and complexity tradeoffs for type system design
Self_Type: a Motivating Example

class Count {
    i : int ← 0;
    inc () : Count {
        i ← i + 1;
        self;
    }
};

- **Class Count** incorporates a counter
- The **inc** method works for any subclass
- But there is disaster lurking in the type system
Self-Type: a Motivating Example

- Consider a subclass **Stock of Count**

```java
class Stock inherits Count {
    name : String; -- name of item
}
```

- And the following use of **Stock**:

```java
class Main {
    Stock a ← (new Stock).inc ();    Type checking error !
    ... a.name ...
}
```
Self_Type: a Motivating Example

- (new Stock).inc() has dynamic type Stock
- So it is legitimate to write
  Stock a ← (new Stock).inc()
- But this is not well-typed
  (new Stock).inc() has static type Count
- The type checker “loses” type information
- This makes inheriting inc useless
  - So, we must redefine inc for each of the subclasses, with a specialized return type
Self_Type: a Motivating Example

- **SELF_TYPE** allows the return type of `inc` to change when `inc` is inherited
- Modify the declaration of `inc` to read
  
  ```
  inc() : SELF_TYPE 
  ```
- The type checker can now prove:

  ```
  O, M ⊢ (new Count).inc() : Count
  O, M ⊢ (new Stock).inc() : Stock
  ```

- The program from before is now well typed