Programming Languages and Compilers: Final

Due on June 8, 2024 at 11:30

Professor CHUNG YUNG

LEE CHIH PIN

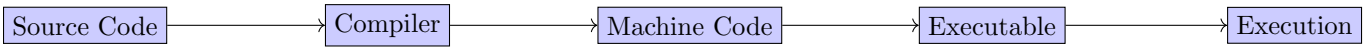
Problem 1

Compilation, interpretation, and hybrid models

- **Compilation** - Use a figure to show the compilation model of language procession, briefly describe each component in the figure
- **Interpretation** - Use a figure to show the interpretation model of language procession, briefly describe each component in the figure
- **Hybrid** - Use a figure to show the hybrid model of language procession, briefly describe each component in the figure

Solution

(A) Compilation Model

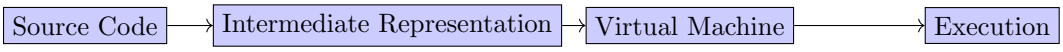


- **Source Code**: The human-readable code written in a high-level programming language.
- **Compiler**: Translates the source code into machine code (binary code).
- **Machine Code**: The low-level code understood by the computer's CPU.
- **Executable**: The machine code in a form that can be directly executed by the computer.
- **Execution**: The process of running the executable on the computer.

(B) Interpretation Model

- **Source Code**: The human-readable code written in a high-level programming language.
- **Interpreter**: Reads and executes the source code directly, line by line.
- **Execution**: The process of running the source code directly by the interpreter.

(C) Hybrid Model



- **Source Code**: The human-readable code written in a high-level programming language.
- **Intermediate Representation (IR)**: A code representation that is between high-level source code and low-level machine code. Often bytecode.
- **Virtual Machine (VM)**: Executes the intermediate representation. Examples include the Java Virtual Machine (JVM) and the .NET Common Language Runtime (CLR).
- **Execution**: The process of running the intermediate representation on the virtual machine.

Problem 2

```
studies(alan, csie35460).
studies(bob, csie35460).
studies(cate, csieb0480).
studies(david, csieb0480).
lectures(edward, csie35460).
lectures(fiona, csieb0480).
teaches(X, Y) :-
lectures(X, C), studies(Y, C).
```

Consider the above program.

- What is the programming language in which the above program is written?
- Briefly describe the functionality of rule teaches.
- What is the return value of teaches(edward, alan)
- What is the return value of teaches(fiona, bob)?

Solution

What is the programming language in which the above program is written?

The program provided is written in **Prolog**

Briefly describe the functionality of rule teaches.

The rule `teaches(X, Y)` states that X teaches Y if X lectures a course C and Y studies the same course C. In other words, for any pair (X, Y), X is considered to be teaching Y if X is a lecturer of a course and Y is a student of that same course.

What is the return value of teaches(edward, alan)

To determine the return value of `teaches(edward, alan)`, we need to check if Edward lectures a course that Alan studies. From the given facts:

- `lectures(edward, csie35460).`
- `studies(alan, csie35460).`

Since Edward lectures `csie35460` and Alan studies `csie35460`, the rule `teaches(edward, alan)` holds true.

Return value: `true`

What is the return value of teaches(fiona, bob)?

To determine the return value of `teaches(fiona, bob)`, we need to check if Fiona lectures a course that Bob studies. From the given facts:

- `lectures(fiona, csieb0480).`
- `studies(bob, csie35460).`

Fiona lectures `csieb0480` and Bob studies `csie35460`, which are different courses. Therefore, the rule `teaches(fiona, bob)` does not hold true.

Return value: `false`

Problem 3

- What is a strongly typed programming language?
- Is each of the following a strongly typed programming language?

Solution

- **Strongly typed programming language** - Java,ML,F#andC#

Is each of the following a strongly typed programming language?

1. **Java**
   - *Yes.* Java is a strongly typed language. It enforces strict type-checking at both compile time and runtime, requiring explicit type conversions and preventing many type-related errors.
2. **C#**
   - *Yes.* C# is a strongly typed language. It has strict type-checking rules and requires explicit conversions between incompatible types, ensuring type safety and reducing runtime errors.
3. **F#**
   - *Yes.* F# is a strongly typed language. It enforces strict type-checking and supports type inference, which helps maintain type safety while reducing the verbosity of type annotations.
4. **ML**
   - *Yes.* ML (Meta Language) is a strongly typed language. It uses a sophisticated type inference system to enforce strict type rules, ensuring that type errors are caught at compile time.

Problem 4

```
var x = 1;
function sub1() {
 document.write("x = " + x + "");
}
function sub2() {
 var x;
 x = 3;
 sub1();
}
x = 2;
sub2();
```

Consider the above JavaScript program.

- Assume the program is interpreted using static-scoping rules. What value of x is displayed in function sub1?
- Under dynamic-scoping rules, what value of x is displayed in function sub1?

Solution

(A) Static Scoping

The scope of a variable is determined by the structure of the program code. the bindings of variables are resolved by looking at the program text and using the environment in which the function was defined.

- `sub1` is defined in the global scope.
- When `sub1` is defined, the global variable `x` is `1`.
- Before `sub2` is called, `x` is updated to `2` globally.
- When `sub2` is called, the local `x` inside `sub2` is `3`, but `sub1` refers to the global `x`, which is now `2`.

Thus, under static scoping, the value of `x` displayed in `sub1` is:

$$\boxed{2}$$

(B) Dynamic Scoping

The scope of a variable is determined by the calling sequence of the functions at runtime. The bindings of variables are resolved by looking at the call stack at runtime and using the environment in which the function was called.

- When `sub2` is called, a new local `x` is declared and set to `3`.
- When `sub1` is called within `sub2`, it looks for `x` in the current call stack. The closest `x` is the local `x` in `sub2`, which is `3`.

Thus, under dynamic scoping, the value of `x` displayed in `sub1` is:

$$\boxed{3}$$

Problem 5

Consider a fixed-decimal literal with no superfluous leading or trailing zeros (called as xnum).

For examples, 0.0, 123.01, and 123005.0 are legal, but 00.0, 001.000, and 002345.1000 are illegal.

- Write the regular expression for xnum
- Draw the transition diagram for xnum.
- Draw the transition table for xnum

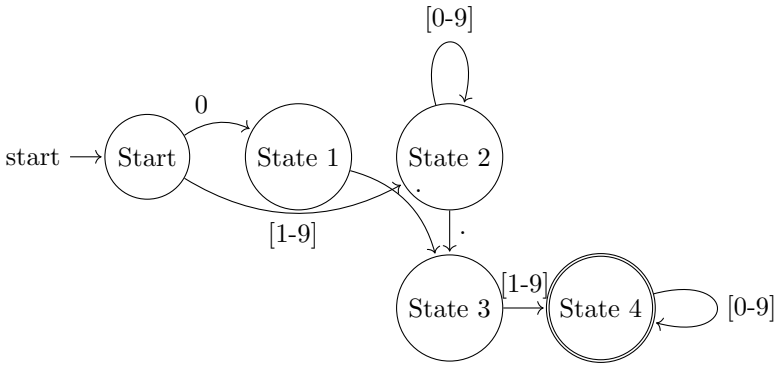
Solution

(A) Regular Expression for xnum

A fixed-decimal literal with no superfluous leading or trailing zeros can be defined with the following regular expression:

```
(0|([1-9]\d*))(\.\d*[1-9])?
```

(B) Transition Diagram for xnum



(C) Transition Table for xnum

Current State	Input	Next State
Start	0	State 1
Start	[1-9]	State 2
State 1	.	State 3
State 2	[0-9]	State 2
State 2	.	State 3
State 3	[1-9]	State 4
State 4	[0-9]	State 4

Problem 6

Grammar:

$$S \to ABc$$
$$A \to dA \mid B$$
$$B \to eB \mid \lambda$$

Compute the First and Follow Sets of the above grammar.

Solution

1. **First(S):**
   - $S \to ABc$
   - To find First(S), we need First(A).
2. **First(A):**
   - $A \to dA$
     - 'd' is terminal, so it goes into First(A).
   - $A \to B$
     - To find First(B), proceed to find First(B).
3. **First(B):**
   - $B \to eB$
     - 'e' is terminal, so it goes into First(B).
   - $B \to \lambda$
     - λ is epsilon (empty string), so λ goes into First(B).
4. **Combining Results:**

$$\text{First}(A) = \{d\} \cup \text{First}(B)$$
$$\text{First}(B) = \{e, \lambda\}$$
$$\text{First}(A) = \{d, e, \lambda\}$$
$$\text{First}(S) = \text{First}(A) = \{d, e, \lambda\}$$

First Sets:

$$\text{First}(S) = \{d, e, \lambda\}$$
$$\text{First}(A) = \{d, e, \lambda\}$$
$$\text{First}(B) = \{e, \lambda\}$$

Steps to Compute Follow Sets:

1. **Follow(S):**
   - S is the start symbol.

- Follow(S) = {\$} (end of input marker)

2. **Follow(A):**
   - From $S \to ABc$
     - Follow(A) $\subseteq$ First(B)
   - From $A \to B$
     - Follow(A) $\subseteq$ Follow(S)
3. **Follow(B):**
   - From $S \to ABc$
     - Follow(B) $\subseteq \{c\}$
   - From $A \to B$
     - Follow(B) $\subseteq$ Follow(A)
4. **Combining Results:**
   - Follow(S) = {\$}
   - B has λ in its First set, so Follow(A) = Follow(S):

$$\text{Follow}(A) = \{\$\}$$

   - Follow(B):

$$\text{Follow}(B) = \{c\} \cup \text{Follow}(A) = \{c, \$\}$$

Follow Sets:

$$\text{Follow}(S) = \{\$\}$$
$$\text{Follow}(A) = \{\$\}$$
$$\text{Follow}(B) = \{c, \$\}$$

Problem 7

Given grammar G:

1. $S \to i\ E\ t\ L\ f$
2. $S \to i\ E\ t\ L\ e\ L\ f$
3. $L \to L\ ;\ S$
4. $L \to S$
5. $E \to v + E$
6. $E \to v$

- In the above grammar G, capital letters are nonterminals and lower-case letters are terminals. Is G an LL(1) grammar? If yes, prove it; otherwise, transform G into LL(1).

Solution

Step 1: Eliminate Left Recursion

The grammar contains left recursion in the productions for L. We can rewrite these productions to eliminate left recursion.
Original productions for L:

$$L \to L\ ;\ S$$
$$L \to S$$

Rewrite to eliminate left recursion:

$$L \to S\ L'$$
$$L' \to ;\ S\ L' \mid \epsilon$$

Step 2: Remove Common Prefixes

Next, we need to handle common prefixes in the grammar. The productions for S have common prefixes.
Original productions for S:

$$S \to i\ E\ t\ L\ f$$
$$S \to i\ E\ t\ L\ e\ L\ f$$

Rewrite to remove common prefixes:

$$S \to i\ E\ t\ L\ S'$$
$$S' \to f \mid e\ L\ f$$

Updated Grammar

After eliminating left recursion and common prefixes, the updated grammar is:

1. $S \to i\ E\ t\ L\ S'$
2. $S' \to f \mid e\ L\ f$
3. $L \to S\ L'$
4. $L' \to ;\ S\ L' \mid \epsilon$
5. $E \to v + E$
6. $E \to v$

Step 3: Verify LL(1) Conditions

We now need to check if the updated grammar is LL(1) by verifying the conditions.
For S:

$$\text{FIRST}(S) = \{i\}$$
$$\text{FOLLOW}(S) = \{\ ,\}, f\}$$

For S':

$$\text{FIRST}(S') = \{f, e\}$$
$$\text{FOLLOW}(S') = \{\}, f\}$$

For L:

$$\text{FIRST}(L) = \{i\}$$
$$\text{FOLLOW}(L) = \{\ ,\}, f, e\}$$

For L':

$$\text{FIRST}(L') = \{\ , \epsilon\}$$
$$\text{FOLLOW}(L') = \{\ ,\}, f\}$$

For E:

$$\text{FIRST}(E) = \{v\}$$
$$\text{FOLLOW}(E) = \{t\}$$

From this, we see that:

$$\text{FIRST}(S) \cap \text{FIRST}(L) = \{i\} \cap \{i\} = \{i\}$$
$$\text{FIRST}(E) \cap \text{FOLLOW}(E) = \{v\} \cap \{t\} = \emptyset$$

Because the grammar has common prefixes and overlapping FIRST sets, it is not an LL(1) grammar in its original form. However, the transformed grammar removes these conflicts.
Thus, the updated grammar is an LL(1) grammar as it satisfies the conditions for LL(1) parsing:

1. No left recursion.
2. No common prefixes.
3. Disjoint FIRST and FOLLOW sets for each nonterminal.

Problem 8

- Compute the Predict Set of each rule in the above grammar.
- Give the LL(1) parse table of the above grammar.

Solution

Part (A): Compute the Predict Set of each rule

Step 1: Compute First sets

$$\begin{aligned}
\text{First}(E) = \text{First}(T) = \text{First}(F) &= \{i, (\} \\
\text{First}(X) &= \{+, \epsilon\} \\
\text{First}(T) = \text{First}(F) &= \{i, (\} \\
\text{First}(Y) &= \{*, \epsilon\} \\
\text{First}(F) &= \{i, (\}
\end{aligned}$$

Step 2: Compute Follow sets

$$\begin{aligned}
\text{Follow}(E) &= \{\$,)\} \\
\text{Follow}(X) &= \{\$,)\} \\
\text{Follow}(T) &= \text{First}(X) \cup \text{Follow}(E) = \{+, \$,)\} \\
\text{Follow}(Y) = \text{Follow}(T) &= \{+, \$,)\} \\
\text{Follow}(F) = \text{Follow}(T) &= \{+, *, \$,)\}
\end{aligned}$$

Step 3: Compute Predict sets for each production

$$\begin{aligned}
\text{Predict}(E \to TX\$) &= \text{First}(T) = \{i, (\} \\
\text{Predict}(X \to +E) &= \text{First}(+) = \{+\} \\
\text{Predict}(X \to \epsilon) &= \text{Follow}(X) = \{\$,)\} \\
\text{Predict}(T \to FY) &= \text{First}(F) = \{i, (\} \\
\text{Predict}(Y \to *T) &= \text{First}(*) = \{*\} \\
\text{Predict}(Y \to \epsilon) &= \text{Follow}(Y) = \{+, \$,)\} \\
\text{Predict}(F \to i) &= \{i\} \\
\text{Predict}(F \to (E)) &= \{(\}
\end{aligned}$$

Problem 8 continued on next page...

Part (B): LL(1) Parse Table

Nonterminal	(	)	**i**	+	*	**$**
E	$E \to TX\$$		$E \to TX\$$			
X		$X \to \epsilon$		$X \to +E$		$X \to \epsilon$
T	$T \to FY$		$T \to FY$			
Y		$Y \to \epsilon$			$Y \to *T$	$Y \to \epsilon$
F	$F \to (E)$		$F \to i$			

Problem 9

Computation and Parse Table

1. $S \to aAc$
2. $\quad | \; aBd$
3. $A \to b$
4. $B \to b$

- Prove or disprove the above grammar is LR(0).

Solution

Grammar

1. $S \to aAc$
2. $S \to aBd$
3. $A \to b$
4. $B \to b$

Augmented Grammar

1. $S' \to S$
2. $S \to aAc$
3. $S \to aBd$
4. $A \to b$
5. $B \to b$

Canonical Collection of LR(0) Items

I0:

$[S' \to \cdot S]$
$[S \to \cdot aAc]$
$[S \to \cdot aBd]$

Goto(I0, a):

$$[S \to a \cdot Ac]$$
$$[S \to a \cdot Bd]$$
$$[A \to \cdot b]$$
$$[B \to \cdot b]$$

I1:

$$[S \to a \cdot Ac]$$
$$[S \to a \cdot Bd]$$
$$[A \to \cdot b]$$
$$[B \to \cdot b]$$

Goto(I1, b):

$$[A \to b\cdot]$$
$$[B \to b\cdot]$$

I2:

$$[A \to b\cdot]$$
$$[B \to b\cdot]$$

Goto(I1, A):

$$[S \to aA \cdot c]$$

I3:

$$[S \to aA \cdot c]$$

Goto(I1, B):

$$[S \to aB \cdot d]$$

I4:

$$[S \to aB \cdot d]$$

Goto(I3, c):

$$[S \to aAc\cdot]$$

I5:

$$[S \to aAc\cdot]$$

Goto(I4, d):

$$[S \to aBd\cdot]$$

I6:

$$[S \to aBd\cdot]$$

LR(0) Parse Table

State	a	b	c	d	A	B
I0	s1					
I1		s2			3	4
I2		r3, r4				
I3			s5			
I4				s6		
I5						
I6						

Conflict Analysis

In state I2, there is a reduce/reduce conflict:

- Reduce $A \to b$ (r3)
- Reduce $B \to b$ (r4)

Conclusion

The given grammar is not LR(0) because there is a reduce/reduce conflict in state I2.

Problem 10

- Give a grammar G that is LR(0) but not LL(1).
- Prove that G is LR(0)
- Prove that G is not LL(1).

Solution

Consider the following grammar G:

1. $S \to Aa$
2. $S \to bAc$
3. $A \to d$

This grammar is LR(0) but not LL(1).

(B) Prove that G is LR(0).

To prove that G is LR(0), we need to construct the LR(0) items and the canonical collection of LR(0) sets of items.

Step 1: LR(0) Items

1. $S \to \cdot Aa$
2. $S \to \cdot bAc$
3. $A \to \cdot d$
4. $S \to A \cdot a$
5. $S \to bA \cdot c$
6. $A \to d\cdot$
7. $S \to Aa\cdot$
8. $S \to bAc\cdot$

Step 2: Canonical Collection of LR(0) Sets of Items

$$\begin{aligned} I_0 &= \text{Closure}(\{S \to \cdot Aa, S \to \cdot bAc\}) \\ &= \{S \to \cdot Aa, S \to \cdot bAc, A \to \cdot d\} \end{aligned}$$

Move on terminal a:

$$\text{Goto}(I_0, a) = \{S \to A \cdot a, S \to bA \cdot c\}$$

Move on terminal b:

$$\text{Goto}(I_0, b) = \{S \to b \cdot Ac, A \to \cdot d\}$$

Move on terminal A:

$$\text{Goto}(I_0, A) = \{S \to A \cdot a, S \to bA \cdot c\}$$

Move on terminal d:

$$\text{Goto}(I_0, d) = \{A \to d\cdot\}$$

The construction of the full table confirms that there are no conflicts, thus proving that G is LR(0).

(C) Prove that G is not LL(1).

To prove that G is not LL(1), we need to show that the grammar does not meet the conditions required for LL(1) grammars, which involve having a unique production rule for each combination of non-terminal and lookahead terminal.

First sets

$$\text{First}(S) = \{b, d\}$$
$$\text{First}(A) = \{d\}$$

Follow sets

$$\text{Follow}(S) = \{\$\}$$
$$\text{Follow}(A) = \{a, c\}$$

Predict sets

$$\text{Predict}(S \to Aa) = \text{First}(Aa) = \{d\}$$
$$\text{Predict}(S \to bAc) = \text{First}(bAc) = \{b\}$$
$$\text{Predict}(A \to d) = \text{First}(d) = \{d\}$$

In the predict sets, we see that:

- $S \to Aa$ has predict set $\{d\}$
- $A \to d$ has predict set $\{d\}$

The predict sets overlap for non-terminal A, which indicates that the parser will have difficulty determining which production to use when encountering the lookahead 'd'. This demonstrates that the grammar is not LL(1), as LL(1) grammars require non-overlapping predict sets for each non-terminal and lookahead combination.

Thus, grammar G is LR(0) but not LL(1).