Rn's programming language course midterm note

Rn's programming language course midterm note

i wrote this note for my midterm exam. core keywords and concepts are included in this note.

Table of contents

[1주] Introduction (1)

Why Should We lean PL?

목적에 맞는 언어를 선택하고, 그 언어를 잘 사용하기 위해.

Language Evaluation Criteria

Characteristic	Readability	Writability	Reliability
Simplicity	O	O	O
Orthogonality	O	O	O
Data Types	O	O	O
Syntax Design	O	O	O
Support for Abstraction	X	O	O
Expressivity	X	O	O
Type Checking	X	X	O
Exception Handling	X	X	O
Restricted Aliasing	X	X	O

Readability (가독성)

- 현재 프로그래밍 언어에서 중요한 기준이다. (하드웨어가 좋아져서)
- 가독성이 좋으면 유지보수하는데 도움이 된다.

Simplicity (단순성)

```
count = count + 1
count += 1
count++
++count
```

위의 코드들은 같은 의미를 가진다. 하지만 단순한 코드는 아니다.

> 같은 일을 하는 여러 문법이 존재 -> 혼란을 줄 수 있음

연산자 오버로딩을 하면 단순함이 떨어짐. 같은 연산자라도 다른 의미를 가질 수 있음.

Orthogonality (직교성)

직교성은 쉽게 설명하면 반대 기능도 된다. 혹은 같은 명령으로 만들 수 있는 모든 조합의 명령을 모두 수행 가능함을 나타낸다.

다음 예제를 참고하면 좋다.

IBM 컴퓨터의 경우 더하기를 위해 다음 두 가지 명령을 제공한다.

```
A   Reg1, Memory Cell
AR  Reg1, Reg2
```

메모리 + 메모리 혹은 메모리 + 레지스터 가 불가능하다.

반면 VAX 컴퓨터는 다음과 같이 더하기를 수행한다.

```
ADDL    operand1, operand2
```

메모리 + 메모리, 메모리 + 레지스터, 레지스터 + 레지스터 모두 가능하다.

따라서 VAX 컴퓨터가 IBM 컴퓨터보다 직교성이 뛰어나다고 할 수 있다.

C언어는 직교성이 낮은 언어다. int + int 같은건 되지만 struct + int 같은건 안 되니까.

직교성이 떨어지면 예외가 많아진다.

직교성이 너무 높아져도 안 좋다. (primitive type을 너무 많이 제공해야 함)

Data Types (데이터 타입)

```
timeOut = 1

timeOut = true
```

위 코드는 둘 다 가능하지만, T/F를 나타낼거면 boolean type을 채용해서 아래 문법으로 쓰는게 가독성에 더 좋다.

Syntax Design (문법 디자인)

문법이 복잡하면 가독성이 떨어진다.

블록을 {}로 쓰는거보다 begin end로 쓰는게 가독성이 더 좋다. 하지만 {}로 쓰는게 더 심플하다.

begin, end같은 예약어가 많아지면 변수로 쓸 수 있는 양이 줄어든다.

Form and meaning (형태와 의미)

C언어에서 static은 함수에 쓸 때, 전역 변수에 쓸 때, 지역 변수에 쓸 때 서로 다른 의미를 갖는다. (사실은 아니긴 함)

그래서 이런 경우 가독성을 떨어뜨린다.

Writability (쓰기 쉬움)

- 쓰기 쉬운 언어는 프로그램을 빨리 작성할 수 있다.
- 쓰기 쉬운 언어는 프로그램을 빨리 배울 수 있다.
- 가독성과 연관이 많다. (가독성이 높으면 쓰기도 쉬움)

Simpolicity and orthogonality

프로그래머는 항상 유저 친화적인 코드를 작성하지 않는다. (심플한 코드 말고 빠른 코드를 작성하기 위함)

심플하지 않은 코드를 쓰면 실수하기 쉬움

직교성이 너무 높으면 쓰기 어려워진다. (직교성이 높다 -> 지원하는 primitive type이 많다 -> 배울게 많다 -> 쓰기 어렵다)

Support for Abstraction (추상화 지원)

추상화를 지원하면 쓰기 쉬워진다. (데이터 관점에서 디자인 가능)

Expressivity (표현력)

표현력이 높으면 쓰기 쉬워진다. (표현력이 높으면 코드를 짧게 쓸 수 있다)

```
count++
```

위와 같은 코드를 보면 짧게 우리가 원하는 일을 할 수 있다. -> 쓰기 쉽다.

Reliability (신뢰성)

- 신뢰성이 높으면 프로그래머가 실수할 확률을 낮춰준다.
- 실수하더라도 실수한 부분을 더 정확하게 알려줄 수 있다.

Type Checking (타입 체킹)

컴파일러가 잘못된 타입 연산을 체크해주면 실행 전에 문제를 알 수 있어서 신뢰성이 높아진다.

런타임 체킹 (실행 중에 체킹)은 신뢰성을 떨어뜨린다. 컴파일 타임에 체킹하는게 더 좋다.

Exception Handling (예외 처리)

런타임에 에러를 핸들링 하는 방법이다.

프로그램은 유저 입력 등에 따라 의도치 않은 에러가 발생할 수 있다. 이런 에러를 잘 핸들링하면 죽을 프로그램도 살릴 수 있기에 신뢰성이 높아진다.

Restricted Aliasing (제한된 별칭)

별칭이란 같은 메모리를 가리키는 포인터 두 개 같은 것들을 말한다.

별칭이 많으면, 하나를 바꿨을 때 나머지도 다 바뀌기 때문에 의도하지 않았다면 프로그램이 제대로 동작하지 않을 수 있다.

하지만, 별칭이 없으면 절대 해결할 수 없는 문제가 있다. 그래서 필요하긴 하다. (포인터 없으면 동적할당해서 자료구조 못 만듬)

Readability and Writability

가독성이 높으면 신뢰성이 높아진다. (가독성이 높으면 실수를 줄일 수 있음)

쓰기 쉬운 언어는 신뢰성이 높아진다. (쓰기 쉬운 언어는 가독성이 높다)

[1주] Introduction (2)

Language Evaluation Criteria

Cost

1. 언어를 배우는 비용 (Related to Simplicty and Orthogonality)
2. 생산성 (Related to Writability)
3. 컴파일 시간
4. 실행 시간
5. 언어를 지원하는 시스템 비용 (유닉스같은거 쓰면 유닉스 설치비가 많이 나옴)
6. 신뢰성이 낮은 언어는 비용이 높아진다. (실수를 많이 하면 고치는데 시간이 많이 걸림)
7. 유지보수 비용 (코드를 계속 고쳐야 되는데 유지보수가 어려우면 시간이 많이 걸림)

컴파일 시간과 실행 시간은 trade off 관계이다.

Portablility (이식성)

플랫폼에 상관없이 동작하는 언어가 이식성이 높다.

이식성이 낮으면 OS별로 다른 코드를 작성해야 한다. -> 비용이 높아진다.

Generality (일반성)

일반성이 높으면 다양한 분야에 사용할 수 있다.

C같은건 쓸 데가 많이 없는데, JS배우면 프론트 백 어플리케이션 이런거 다 할 수 있음.

Well-Definedness (잘 정의되어 있음)

언어가 공식 도큐멘트같은게 잘 돼있으면 배우기 쉽다.

Criteria

- Readability
- Writability
- Reliability
- Cost

위와 같은 정보(Criteria)들은 언어를 디자인할 때 중요한 기준이 된다.

언어마다 어떤 것을 중점으로 둘 지 다르다.

보통 언어마다 겹치는 내용들이 많다.

Other factors for PL design

Computer Architecture

컴퓨터 아키텍쳐는 언어 디자인에 많은 영향을 준다.

ex) 폰 노이만 아키텍쳐

- 데이터와 프로그램 코드 모두 같은 메모리에 저장된다.
- 데이터와 프로그램 코드는 CPU로 전송된다. (vice versa)
- Fetch-Execute Cycle

Imperative Language (명령형 언어)는 폰 노이만 아키텍쳐를 기반으로 디자인되었다.

- 변수를 메모리 셀에 저장함
- 폰 노이만 아키텍쳐에는 Functional Language보다 Imperative Language가 더 잘 맞는다. (라고 믿는 사람들이 있음)

Programming Design Methodologies

기존에는 그냥 한 함수에 다 때려박았다. 코드의 flow를 바꾸기 위해서는 goto를 써야했고, 다른 기능은 없었다.

그리고 1 computer 1 task 였다.

즉, 그냥 계산기에 가까웠던 것.

근데 1960년부터 structured/procedural-programming이 (구조적/절차지향 프로그래밍) 나오기 시작했다.

함수라는 개념이 나오면서 재사용성을 높이고 가독성을 높였다.

1970년에는 procedure-oriented programming이 data-oriented programming (데이터지향(?) 프로그래밍)으로 변화하기 시작했다.

추상화 데이터 타입을 사용했다.

1980년도에는 object-oriented programming이 나왔다.

- Encapsulation: 데이터 은닉
- Inheritance: 상속
- Dynamic Binding: flexible한 상속 가능

현재는 다양한 언어가 object-oriented programming을 지원한다.

Language Categories

언어는 크게 네 가지 카테고리로 나뉜다.

- Imperative
- Functional
- Logic
- Object-Oriented

Imperative Language (명령형 언어)

알고리즘은 명령어들의 연속이다. 모든 코드를 디테일하게 작성하고, 실행 순서를 정의한다.

폰 노이만 아키텍쳐를 따른다. (변수를 메모리에 저장 (vice versa))

Functional Language (함수형 언어)

모든 문제를 pure functions의 체이닝으로 해결한다.

수학적인 함수를 따르며, 가독성이 높다. (유지보수가 쉬움)

Assignment가 없다. (변수도 없다.)

Logic Language (논리형 언어)

규칙은 특별한 순서로 정해지지 않으며, 구현 할 때 어떤 규칙을 적용할지 선택한다.

Prolog가 대표적인 예시다.

Object-Oriented Language (객체지향 언어)

객체지향 프로그래밍은 데이터와 함수를 하나로 묶어서 사용한다.

객체지향 프로그래밍은 데이터를 캡슐화하고, 추상 클래스와 상속을 통해 코드 재사용성을 높인다.

다음 네 가지는 추가적으로 존재하지만, 위 네 가지가 가장 중요하다.

- Visual Language
- Scripting Language
- Markup Language
- Special-Purpose Language

Visual Language (시각 언어)

.NET같은 언어를 말하며, Imperative Language의 sub category이다.

Scripting Language (스크립트 언어)

Python, Ruby, Perl 같은 언어를 말한다.

언어 설계보다는 구현이 중요(?) 모르겠다.

Markup Language (마크업 언어)

HTML, XML 같은 언어를 말한다. (문서를 구조화하는 언어)

Special-Purpose Language (특수 목적 언어)

이건 몰라도 될듯. 그냥 특수한 목적성을 가진 문제를 해결하기 위해 존재하는 언어들이다.

Language Design Trade-Offs

언어를 디자인할 때, 다음과 같은 trade-off가 존재한다.

- Reliability가 올라가면 execution efficiency가 떨어진다. (vice versa)
- APL은 표현력이 높지만 가독성이 떨어진다.
- C언어에서 포인터를 사용하면 Writeability가 높아지지만, Reliability가 떨어진다.

Implementation Methods

컴퓨터는 메모리와 프로세서(CPU)로 이루어져 있다.

프로세서(CPU)는 low-level 명령어 집합으로 다룰 수 있다. 하드웨어를 다룰 수 있는 API를 유저에게 제공한다. (system call 등)

프로그래밍 언어는 OS에서 제공하는 API를 사용해서 하드웨어를 다룬다.

Compilation

Source Code -> Lexical Analyzer -> Syntax Analyzer -> Intermediate Code Generator and Sementic Analyzer -> Code Optimizer -> Code Generator -> Machine Code (Result)

컴파일을 위 과정을 거쳐 실행된다.

Lexical Analyzer (어휘 분석기)

프로그램 코드를 Lexical Units로 나눈다.

Syntax Analyzer (구문 분석기)

Parse Trees를 만들어 문법을 검사한다.

Intermediate Code Generator (중간 코드 생성기)

C -> Machine Code가 아닌 C -> LLVM -> Machine Code처럼 변경된다.

이때 LLVM과 같은 중간 코드로 변형한다. 이는 LLVM이 다양한 플랫폼에서 지원하면 C언어를 LLVM으로 바꾸는 것으로 다양한 플랫폼에서 실행할 수 있다.

Semantic Analyzer (의미 분석기)

의미 분석을 해서 잘못된 코드를 찾는다. (타입 체킹 등)

Code Optimizer (코드 최적화기)

코드를 최적화한다. (최적화는 다양한 방법이 있다.)

코드 사이즈를 줄이거나 더 빠르게 동작하도록 바꾼다.

Code Generator (코드 생성기)

최적화된 중간 언어에서 Machine Code로 변역된 결과물을 만든다.

Symbol Table (심볼 테이블)

컴파일에 필요한 심볼들을 저장한다.

머신 코드 링킹 과정에서 사용된다. (변수가 저장된 주소 등을 기록한다) Lexical and Syntax Analyzer에서 작성된다.

Linker (링커)

여러 파일을 하나로 합친다.

Pure Interpretation

Source Code -> Interpreter (Result)

컴파일 없이 바로 실행한다.

구현하기 쉽고, 디버깅하기도 쉽다. 하지만 느리다. Symbol Table에 더 많은 메모리가 필요하다.

Hybrid Implementation Systems

컴파일러와 인터프리터를 혼합한 방식이다.

JVM같은게 이런 방식으로 작동한다. Java Code -> (Compile) -> Byte Code -> JVM (Interpret)

Pure Interpretation보다 빠르다. (컴파일된 코드를 실행하기 때문)

JVM만 있으면 어디서든 실행할 수 있다. (이식성이 높다)

Preprocessors

컴파일 전에 코드를 변경하는 프로그램이다. (C언어의 #include 같은 것)

[2주] Introduction (1)

Describing Syntax and Semantics

Syntax (문법)

Syntax는 언어의 문법을 나타낸다.

Semantics (의미)

Semantics는 문법에 따라 어떤 일이 일어나는지를 나타낸다.

Syntax와 Semantics는 서로 연관이 있다.

Syntax를 설명하는 것이 Semantics를 설명하는 것보다 쉽다.

언어를 배울 때 Syntax를 배우는 것과 Semantics를 배우는 것 두 개로 나눌 수 있다.

프로그램은 주어진 String(Source Code)을 Lexemes와 Tokens로 결정해야 한다.

- Lexemes: Source Code의 한 단어
- Tokens: Lexemes의 의미

Ex)

Source Code: `index = 2 * count + 17;`

Lexemes	Tokens
index	identifier
=	equal sign
2	integer literal
*	mult op
count	identifier
+	plus op

Lexemes	Tokens
17	integer literal
;	semicolon

BNF (Backus-Naur Form) Notation - Context-Free Grammars

Context Free Grammar를 나타내기 위해 BNF가 사용된다.

Extended BNF는 BNF를 확장해 더 많은 문법을 나타낼 수 있다.

Attribute Grammars (속성 문법)

속성 문법은 문법에 의해 생성된 구조에 속성을 추가한다.

프로그래밍 언어의 Static Semantics를 나타낼 때 사용된다.

Description Method for Semantics (의미 설명 방법)

구문을 설명하기 위해 다음 세 가지 방법이 존재한다.

- Operational Semantics
- Axiomatic Semantics
- Denotational Semantics

언어는 다음 두 가지 형태로 정의될 수 있다.

- Recognition: 사용자의 인풋 (Source Code)을 보고 어떤 언어인지 판단하는 것 (올바른 문법대로 작성되었는지 확인) - LR Parser
- Generation: 언어의 문법을 보고 Source Code를 생성하는 것 (생성이 된다면 올바른 문법) - LL Parser

두 방법은 매우 비슷하다.

Grammar (문법)

문법은 언어 생성 매커니즘이다.

Regular Grammar, Context-Free Grammar, Context-Sensitive Grammar, Unrestricted Grammar가 있다.

Context-Free Grammar (CFG)

CFG는 양 옆 구문에 따라 의미가 변하지 않는 문법을 말한다. (문맥 자유 문법)

CFG는 BNF로 나타낼 수 있다.

대개 프로그래밍 언어는 CFG이다.

BNF

다음과 같은 구조를 가지고 있다.

LHS -> RHS

`<assign> -> <var> = <expression>`

LHS에서 RHS로 Derivation 하면서 문법을 나타낸다.

<>로 감싼 것은 non-terminal symbol이다. 그 외는 terminal symbol이다.

- terminal symbol: 실제 코드에 나타나는 것 (더이상 변화하지 않음)
- non-terminal symbol: non-terminal symbol + terminal symbol 변환될 수 있는 것 (모든 Derivation이 끝나면 남아있으면 안 된다.)

Parse Tree (파스 트리)

Derivation을 나타내는 트리이다.

Terminal Symbol을 Derivation한 결과를 자식으로 연결하면 Parse Tree를 구할 수 있다.

Ambiguity (모호성)

하나의 문장이 여러 가지 Parse Tree를 가질 때 발생한다.

모호성이 있다면, 어떻게 분석하냐에 따라 매번 다른 결과가 나올 수 있기에 없애야 한다.

Operator Precedence (연산자 우선순위)

(Left | Right) Most Derivation 등을 사용하면 Ambiguity를 없앨 수 있다.

하지만, 연산자 우선순위에 문제가 생길 수 있다.

이는 문법이 잘못된 경우로, 문법을 수정해야 한다.

Associativity (연산자 결합성)

Left Most Derivation 등을 사용하면 Ambiguity를 없앨 수 있다. 하지만

a ** b ** c 같은 문법이 주어진다면 (a의 (b의 c승) 승)이 되어야 하지만, Left Most Derivation을 할 경우 ((a의 b승)의 c승)이 된다.

이런 경우 Right Most Derivation을 사용해야 한다.

`<if_stmt> -> if <logic_expr> then <stmt>[else <stmt>]` 같은 Production Rule이 있다고 가정해 보겠습니다.

이때 `<if_stmt> => if <logic_expr> then if <logic_expr> then <stmt> else <stmt>` 같은 문법이 있다면 Ambiguity가 발생한다.

else가 어떤 if에 속해야 하는지 모호하기 때문이다.

따라서 이런 문제를 해결하기 위해 문법을 수정해야 한다.

Extended BNF (EBNF)

EBNF는 BNF를 확장한 것이다.

1. brackets[]: 있어도 되고 없어도 되는 부분 (0번 혹은 1번 등장)
2. braces{}: 0번 이상 반복
3. braces{}+: 1번 이상 반복
4. multiple choices: ('*' | '+' | '-' | '/') 같이 네 개 중 하나 선택해서 적용

[2주] Introduction (2)

Static Semantics

Attribute Grammar (속성 문법)

- 속성 문법은 BNF에 의해 정의하기 어려운 언어이다.
- CFG의 확장 버전이다.
- Static Semantics Rule이 올바른지 확인하고 설명한다.

Static Semantic (정적 의미)

- Type Compatibility (타입 호환성): BNF에 의해 정의하려면, 문법은 확장되어야 한다.
- 모든 변수는 참조되기 전에 선언되어야 한다. 그렇지 않으면 BNF로 표현이 불가능하다.
- 컴파일 타임에 모든 정적 의미를 분석할 수 있다.

Attribute Grammar - Context-Free Grammar

-
  - Attributes: 문법 Symbol과 연관 있음
-
  - Attribute Computation Functions: Attribute 값이 어떻게 계산되는지와 연관 있음
-
  - Predicate Functions: Boolean Expression (Attribute set과 Literal Attribute Value에 대해 계산함 - 올바른지 체크) 쉽게 설명하면 속성이 올바른지 확인함

Attribute Grammar - Synthesized and Inherited Attributes

- Synthesized: 하위 노드에서 상위 노드로 전달되는 속성 (하위 속성을 통해 상위 속성이 결정됨)
- Inherited: 상위 노드에서 하위 노드로 전달되는 속성 (상위 속성과 주변 속성을 통해 하위 속성이 결정됨)

Intrinsic Attributes

- Intrinsic Attributes = Terminal Node의 Synthesized Attributes (터미널 노드의 고유 속성)
- Parse Tree의 밖에서 정의됨 (Symbol Table에서 참조 - 변수 이름 <-> 타입 등)
- Semantic Function을 통해 Non Terminal Node의 Attributes를 계산할 수 있음

Ex)

Syntax Rule: `<proc_def> -> procedure <proc_name>[1] <proc_body> end <proc_name>[2]`

위처럼 Syntax Rule이 주어졌다고 가정해보자.

이때 `<proc_name>` 이 Derivation 돼서 결국 나온 터미널 노드에서 Intrinsic Attributes를 계산할 수 있다.
이후 Semanctic Function을 통해 `<proc_name>`의 Attributes를 계산할 수 있다.

이후 Predicate (`<proc_name>[1].string == <proc_name>[2].string`) Function을 통해 올바른 Semantic인지 아닌지 확인할 수 있다.

Decorating Parse Trees (Parse Tree에 속성을 부여)

Parse Tree에 속성을 부여하는 방법은 다음과 같다.

1. Intrinsic Attributes를 계산한다.
2. Semantic Function을 통해 모든 노드의 속성을 계산한다.
3. Predicate Function을 통해 올바른 Semantic인지 확인한다.

Dynamic Semantics

Dynamic Semantics는 Expressions, Statements, Program Units의 의미를 나타낸다.

위 세 가지는 실제 실행 결과라고 생각하면 된다.

Operational Semantics

Statement or Program의 의미를 설명한다.

- Machine에서 동작하면 어떤 효과를 남기는지 정의하는 것으로 설명이 가능하다.
- 실제 실행 파일을 컴퓨터에서 실행하는 것이라고 생각하면 된다.

몇 가지 문제가 있다.

- 실제 반영되는 결과는 너무 작은 변화이고 다양성이 너무 많다.
- 실제 컴퓨터에서 Storage를 변화시키는 행동은 매우 복잡하다.

따라서 실제로 어떤 동작을 가지는지를 완벽하게는 분석할 수 없다.

> High Level에서는 결국 연산이 가지는 의미가 무엇인지 분석한다.

> Low Level에서는 각 명령이 어떤 상태에 머무는지 검사한다.

결과론적으로 어떤 의미를 갖는지만 분석하는데 쓰일 수 있다.

C언어의 For문을 분석한다고 가정하면, 결국 이 프로그램이 어떤 실행 흐름을 가지는지 분석하는 것이다.

Operational Semantics는 작은 스케일을 가진 프로그램에 대해 심플하고 직관적이다.

실제 구현에 도움이 된다.

하지만 프로그램 스케일이 커지면 분석하기 어렵다. 결과를 예측할 때 수학적으로 분석하기가 까다로워진다.

Denotational Semantics

재귀함수 이론을 기초로 엄격하게 정의된다.

다음 두 가지를 포함하고 있다.

- Mathematical Object
- Mapping Function (instance of lanugage -> mathematical object)
- Domain: Syntatic domain (문법적 도메인) Collection of Parameter's Values - 쉽게 설명하면 Denotational Semantics를 분석하기 위해 필요한 파라미터 집합이다.
- Range: Semantic domain (의미적 도메인) Collection of Mapped Mathematical Objects - 쉽게 설명하면 Denotational Semantics를 계산한 결과물 집합이다.

Ex)

Syntax Rule: `<bin_num> -> [<bin_num>]('0' | '1')`

Semantic Function:

- `M_bin('0') = 0`
- `M_bin('1') = 1`
- `M_bin(<bin_num>'0') = 2 * M_bin(<bin_num>)`
- `M_bin(<bin_num>'1') = 2 * M_bin(<bin_num>) + 1`

0, 1, 10001, 1010111010 같은 값들을 Accept한다.

- Syntatic domain: `['0', '1']` - Denotational Semantics를 분석하기 위해 필요한 파라미터는 01이다.
- Semantic domain: `{0, 1, 2, 3, ... 9, 10, 11, ... Infinity}` - 결과로 나오는 값은 음이 아닌 정수다.

State는 다음과 같이 정의된다.

`s = {<i_1, v_1>, <i_2, v_2>, ... <i_n, v_n>}`

- s: Stage
- i: Name of Variable
- v: Value of Variable
- VARMAP: Mapping Function from Name to Value (Ex, VARMAP(i_3, s) = v_3)

[3주] Semantics_Lexical

Dynamic Semantics

Axiomatic Semantics

가장 추상적인 접근 방식이다.

프로그램을 검증하고 의미를 분석한다.

프로그램의 정당성을 증명한다.

Variables와 Constants의 관계를 베이스로 분석한다.

Assertions

프로그램이 올바르게 동작하는 것을 검증한다. 쉽게 설명하면, 나오면 안 되는 상태가 나오는지 분석한다.

- Pre-Condition: 명령어 실행 전에 참이어야 하는 조건
- Post-Condition: 명령어 실행 후에 참이어야 하는 조건

Ex)

`sum = 2 * x + 1 {sum > 1}` 은 Post-Condition이다.

이 Post-Condition(sum > 1)을 만족하는 Pre-Condition은 `{x > 10}`, `{x > 50}` 등이 있다.

Weakest Pre-Condition (가장 빡빡한 조건)은 `{x > 0}`이다.

Inference Ruls (추론 규칙)

다른 Assertsion을 기저로 새로운 Assertion을 추론한다.

보통 마지막 명령어의 Post-Condition을 가지고 Weakest Pre-Condition을 구하고 전 명령어의 Post-Condition을 구한다. 이를 반복하며 모든 Condition을 구한다.

- Antecedent: 규칙의 조상 (초기 조건)
- Consequent: 규칙의 결과 (결과 조건) - 추론한 결과로 나온 Condition

`Axiom`이란 당연히 참이 되어야 하는 조건을 말한다. -> Inference Rule Without an Antecedent 즉 추론을 하지 않아도 자명한 사실을 뜻한다.

Axiomatic Semantics: `{P}S{Q}`

- P: Pre-Condition
- S: Statement
- Q: Post-Condition

Ex)

```
x = 2 * y - 3 {x > 25}
-> 2 * y - 3 > 25
-> 2 * y > 28
-> y > 14
```

즉 `y > 14`는 Weakest Pre-Condition이다.

```
x = x + y - 3 {x > 10}
-> x + y - 3 > 10
-> y > 13 - x
```

- `Axiom`: Assignment Statements
- `Inference Rule`: Statement Sequences, Selection Statements, Logical Pre-Test Loop Statements 등이 있다.

쉽게 설명하면, Axiom은 하나의 Post-Condition에서 Pre-Condition을 구하는 것이고, 여러 Statement를 가지고 Condition을 구하는 것이 Inference Rule이다.

다음과 같이 추론을 할 수 있다.

`{P}S{Q}`와 `P' => P`, `Q => Q'` 이 주어지면 `{P'}S{Q'}`는 참임을 증명할 수 있다.

`P => Q`, `P' => P => Q => Q'` 이 주어지면 `P' => Q'`는 참임을 증명할 수 있다.

Ex)

`{x > 5} x = x - 3 {x > 0}`
-> `x = x - 3 {x > 0}` -> `{x > 3} != {x > 5}`

`{x > 3} x = x - 3 {x > 0}`, 와 `{x > 5} => {x > 3}`, `{x > 0} => {x > 0}`이 주어지면

- P': `x > 5`
- P: `x > 3`
- Q: `x > 0`
- Q': `x > 0`

따라서 `{P'}S{Q'}` = `{x > 5} x = x - 3 {x > 0}`이 참임을 증명할 수 있다.

`{P1}S1{P2}`, `{P2}S2{P3}`가 주어지면 `{P1}S1; S2{P3}`이 참임을 증명할 수 있다.

General form: `if B then S1 else S2`

If문은 다음과 같은 Inference Rule을 가진다.

`{B and P} S1 {Q}`, `{(not B) and P} S2 {Q}`가 주어지면 `{P} if B then S1 else S2 {Q}`이 참임을 증명할 수 있다.

Logical Pre-Test Loop Statements

루프는 보통 Mathematical Induction(수학적 귀납법) 으로 정의해서 사용합니다. 따라서 P와 Q가 모두 만족하는 조건을 찾을 수 있습니다. (for의 조건식 같은걸 생각하면 끝났을 때 상태까지 포함해서 조건을 세우면 시작부터 끝까지 만족하는 조건을 찾을 수 있습니다.)

따라서 다음과 같은 Inference Rule을 가집니다.

`{I and B} S {I}`가 주어지면 `{I} while B do S {I and (not B)}`이 참임을 증명할 수 있습니다.

물론 이를 만족하는 I를 찾을 수 없을 수도 있습니다.

하지만, 이를 만족하는 I를 찾을 수 있다면, 루프가 끝나는 것을 보장할 수 있습니다.

I를 찾는 것 외에도 다음 네 가지를 만족해야 합니다.

1. `P => I` : 이 조건은 P = I로 놓으면 P => I를 만족합니다.
2. `{I and B} S {I}` : 위에서 명시한대로 for의 조건식에 끝나는 조건까지 포함해서 계산하면 I를 구할 수 있습니다. B는 주어진 거라 따로 구할 필요 X
3. `(I and (not B)) => Q` : for의 조건식에 끝나는 조건까지 포함해서 계산했으므로, I and (not B)는 당연히 Q를 만족합니다.
4. `the loop terminates` : 만약 위 세 조건을 만족하긴 하지만, 루프가 끝나지 않는 조건일 수 있습니다. 그래서 문제가 점점 작아지는 방향으로 루프하는지 확인해야 합니다.

Ex)

```
while y <> x do y = y + 1 {y = x}
```

위 조건에서는 y가 커지는 방향이므로 I = `y <= x`로 생각할 수 있습니다.

이렇게 되면 위 네 조건을 만족하는 I를 찾을 수 있습니다.

따라서 위 반복문은 결국 언젠가는 Terminate되는 것을 보장할 수 있습니다.

Lexical and Syntax Analysis

Semantic Analyzer와 Intermediate Code Generator에 영향을 준다.

컴파일 할 때 중요하다.

Syntax Analyzer

- CFG or BNF를 베이스로 사용한다.
- 간단하고 투명하다.
- 따로 분리되어 있으므로 (모듈화 되어있음) 유지보수 하기 용이하다.

Lexical Analysis

- Lexical Analysis (어휘 분석)은 따로 분리되어 있다.
- 변수 이름과 상수 등을 관리한다.
- 복잡하지 않다.
- 컴파일 타임의 대부분을 차지한다.
- 플랫폼에 의존성을 가진다.
- 자동으로 만들어주는 툴을 쓴다.
- Transition Diagram을 설계하고 구현한다. (토큰 패턴을 설명한다.)
- NFA같은걸 만들어서 어떤 식으로 Lexical Analysis를 할지 설계한다.

프로그램은 주어진 String(Source Code)을 Lexemes와 Tokens로 결정해야 한다.

- Lexemes: Source Code의 한 단어
- Tokens: Lexemes의 의미

Ex)

Source Code: `index = 2 * count + 17;`

Lexemes	Tokens
index	identifier
=	equal sign
2	integer literal
*	mult op
count	identifier
+	plus op
17	integer literal
;	semicolon

[3주] Semantics_Lexical (2)

Lexical and Syntax Analysis

Parsing

Syntax Analysis는 Parsing이라고도 불린다. Parsing에는 두 가지 방법이 있다.

- Top-Down Parsing (하향식 파싱 - From Root - LL Parser - Recognition)
- Bottom-Up Parsing (상향식 파싱 - From Leaves - LR Parse - Generation)

Parse Tree와 Derivation을 사용한다.

결국, Parsing이 하는 일은, 유저가 입력한 Source Code가 문법적으로 올바른지 확인하는 것이다.

Top-Down Parser (LL Parser)

LL Parser는 Left-to-Right, Leftmost Derivation을 사용한다.

Production Rule에서 가장 왼쪽에 나타나는 Non-Terminal Symbol을 먼저 Derivation한다.

이 과정을 반복해서 유저의 입력을 만들 수 있다면, 문법적으로 올바른 것을 확인할 수 있다.

Bottom-Up Parser (LR Parser)

LR Parser는 Left-to-Right, Reverse Order of Rightmost Derivation을 사용한다.

유저의 입력을 보고 RHS를 LHS로 바꾸는 과정을 반복하며 결국 Production Rule의 Start Symbol로 도달하면 문법적으로 올바른 것이다.

파싱은 보통 오래 걸리므로 조금 빠른 방법이 필요하다.

Recursive Descent Parsing

Top-Down Parsing의 한 종류로, Recursive Descent Parsing이 있다.

EBNF가 적합하다.

다음과 같이 사용할 수 있다.

Production Rule: `<expr> -> <term> {(+ | -) <term>}`

```
void expr() {
    term();
    while (nextToken == PLUS_OP || nextToken == MINUS_OP) {
        lex(); // lex mean get next token(lexeme)
        term();
    }
}
```

term 함수도 비슷한 형태로 구현할 수 있다.

하지만 이 방법에는 문제가 있다.

다음과 같은 코드가 있다고 가정하면, 함수는 무한 루프에 빠질 것이다.

Production Rule: `<bin_num> -> [<bin_num>]('0' | '1')`

```
void bin_num() {
    if (nextToken == '0' || nextToken == '1') bin_num();
    if (nextToken == '0' || nextToken == '1') {
        lex();
    }
}
```

따라서 다음과 같이 right recursion으로 바꿔서 문제를 해결할 수 있다.

Production Rule: `<bin_num> -> ('0' | '1')[<bin_num>]`

```
void bin_num() {
    if (nextToken == '0' || nextToken == '1') {
        lex();
    }
    if (nextToken == '0' || nextToken == '1') bin_num();
}
```

Non-Terminal을 Derivation할 수 있는 Rule이 여러 개라면, 어떤 Rule을 사용할지 결정해야 한다. 혹은 모두 탐색해야 한다.

이는 성능에 직접적인 영향을 주고 매우 느려지는 부분이다. 따라서, 다음 규칙을 적용하면 파싱 시 성능을 향상시킬 수 있다.

만약 Production Rule이 다음과 같다고 가정해보자.

Production Rule:

- `A -> aB | bAb | Bb`
- `B -> cB | d`

이렇게 되면 A를 Derivation할 때 세 가지 중 어떤 걸 선택하냐에 따라 첫 번째로 나오는 Terminal Symbol이 다르다.

FIRST(A) = `{ {a}, {b}, {c, d} }`

따라서 다음 나오는 Terminal Symbol이 무엇인지에 따라 어떤 Rule을 사용할지 결정할 수 있다.

하지만 이 방법은 모든 경우 되는 건 아니다. 다음 예시를 살펴보자.

Production Rule:

- `A -> aB | BAb`
- `B -> aB | b`

FIRST(A) = `{ {a}, {a, b} }`

만약 다음 Terminal Symbol이 b라면, 두 번째 Rule을 사용해야 한다. 하지만, a라면 두 Rule중 어떤 것을 사용해야 할 지 확정 지을 수 없다. 따라서 이 경우는 모든 방법을 다 탐색해야 한다.

이 문제를 해결하기 위해선, 공통 부분을 묶어주는 것으로 해결이 가능하다.

다음 예제를 보자

Production Rule: `<variable> -> identifier | identifier [ <expr> ]`

다음 Terminal Symbol이 identifier이라고 할때 어떤 Rule을 사용해야 할지 확정할 수 없다.

하지만 다음과 같이 변경한다면 확정할 수 있다.

Production Rule:

- `<variable> -> identifier <variable_tail>`
- `<variable_tail> -> [ <expr> ] | ε`

이렇게 하면 다음 Terminal Symbol이 무엇이든 확정할 수 있다.

[4주] Syntax_Names_bindings_Scopes

Lexical and Syntax Analysis

Bottom-Up Parsing

Bottom-Up Parsing은 LR Parser로 불린다. LR Parser는 가장 많이 사용되는 Parser 중 하나이다.

Left Recursion 문제가 없다.

RHS를 LHS로 바꾸는 과정을 반복하며 문법적으로 올바른지 확인한다.

그 과정에서 Handle이라는 걸 정의한다

Handle: focues symbol during parsing (reduction)

즉, 현재 없앨 RHS를 Handle이라고 한다.

만약 `S =>* aAw => abw`라면 Handle은 `b`이다.

RHS인 `b`를 LHS인 `A`로 바꾸는 과정을 반복하며 문법적으로 올바른지 확인한다.

그 과정에서 정의하는 phrase가 있다.

- Phrase: 현재 파스트리에서 존재하는 모든 Terminal Symbol
- Simple Phrase: 현재 상태에서 한 번 Derivation해서 나올 수 있는 Phrase

Shift-Reduce Algorithm

LR Parsing을 위한 알고리즘 중 하나이다.

스택을 사용하며, 다음 두 가지 액션을 취한다.

- Shift: 다음 Terminal Symbol을 스택에 넣는다.
- Reduce: Handle을 찾아서 RHS를 LHS로 바꾼다.

이때 shift 과정에서 현재 상태도 같이 스택에 넣는다.

Names, Bindings and Scopes

Variables

Variable은 다음과 같은 네 가지 요소를 가진다.

- Name: 변수의 이름
- Address: 변수의 주소
- Type: 변수의 타입
- Value: 변수의 값

Names

변수의 이름은 문자로 이루어진 구분 가능한 문자열이다.

기본 속성이며, 함수나 파라미터 등에 적용된다.

설계 시 다음 두 가지를 고려해야 한다.

- Case Sensitivity: 대소문자를 구분할 것인지
- Reserved(Special) Words: 예약어를 변수 명으로 허용할 것인지

요새는 제한이 없지만, 옛날에는 메모리 이슈로 31자 제한 등이 있었다.

보통 특수 문자는 변수 명에 포함이 안 되지만, 특정 언어는 포함되기도 한다.

대부분 C 계열 언어에서는 case sensitive이다.

rose, ROSE, Rose 세 개는 모두 다른 변수다.

따라서, Readability를 감소시키기도 한다.

이를 피하기 위해 대문자는 사용하지 않는다. (predefined keyword를 변수 명으로 사용할 때는 제외)

포트란에서는 Integer를 변수 명으로 쓰는 거를 허용해주기도 한다.

예약어를 변수 명으로 사용하지 못하게 막으면 혼란을 줄인다.

하지만 사용할 수 있는 변수 명이 줄어든다. 코볼에는 약 300개 정도 예약어가 있다.

Address

l-value일때 사용한다. Machine의 Memory Address를 가리킨다.

같은 변수가 다른 메모리에 저장될 수도 있다. (런타임 스택)

같은 주소를 가리키는 여러 변수가 존재할 수도 있다. (Aliases - 포인터, Union)

Readability를 떨어뜨린다.

Type

변수의 타입을 나타낸다. 변수가 저장할 수 있는 범위를 나타낸다.

변수가 사용할 수 있는 연산을 결정한다.

Type Checking을 통해 변수의 타입을 확인한다.

Value

r-value일때 사용한다. 변수의 값이다. 메모리 안에 값을 저장한다.

r-value에 접근하려면 l-value가 반드시 먼저 결정되어야 한다.

Bindings

Binding은 변수와 밀접하다.

Binding Time은 장소를 차지하는 시간을 나타낸다.

Language Design Time, Language implementation Time, Compile Time, Load Time, Link Time, Run Time

Ex) In Java, count = count + 5

- Type of count : Compile Time
- Set of possible values of count : Compiler Design Time
- Meaning of the operator Symbol : Compile Time

- Internal Representation of the Literral 5 is : Compiler Design Time
- Value of Count : Run Time (Execution Time)

Binding은 Static Binding과 Dynamic Binding이 있다.

- Static Binding: Compile Time에 결정된다. 절대 바뀌지 않는다. (Type, Scope)
- Dynamic Binding: Run Time에 결정된다. 바뀔 수 있다. (Value)

Type Bindings

변수가 참조되기 전에 타입이 결정되어야 한다.

두 가지 타입 바인딩이 존재하며 각각 두 가지 방법이 있다.

- 타입을 어떻게 정의한 것인가? : Explicit (명시적), Implicit (묵시적)
- 어디에 할당할 것인가? : Compile Time, Run Time

Static Type Bindings

- Explicit (명시적) : 변수 선언 시 타입을 명시한다.
- Implicit (묵시적) : 변수 선언 시 타입을 명시하지 않는다. 대신, 첫 번째 할당 시 타입이 결정된다. (포트란에서는 I, J, K, L, M, N 은 정수로 사용한다. (암묵적인 룰) 그 외는 실수)

묵시적 변환은 신뢰성을 떨어뜨린다.

Dynamic Type Bindings

Run Time에 Assignment가 발생하면 변수의 타입이 결정된다. RHS(r-value)의 타입으로 결정된다.

Flexibility (유연성) 장점이 있다.

다만, 신뢰성을 떨어뜨린다. Assignment 과정에서 의도치 않은 타입 변환을 일으킬 수 있다. 따라서 타입 체킹을 강하게 할 수 없다.

타입 체킹을 하더라도, Run Time에 해야 한다. 하지만, 타입 체킹은 매우 느리다.

모든 변수는 Run Time Descriptor가 필요하다. 따라서, 메모리를 더 많이 먹는다.

변수가 저장되는 위치는 크기가 가변적이어야 한다.

보통 파이썬 같은 Pure Interpreter 언어에서 사용한다. (무조건은 아님 C# 에서도 dynamic {NAME} 으로 사용 가능)

Storeage Bindings

변수를 메모리에 바인딩 시킨다.

Allocation, Deallocation 과정이 필요하다. (메모리에 변수를 추가 혹은 삭제)

변수가 생겨서 삭제될때까지를 Lifetime이라고 한다.

다음 Lifetime 기준으로 나누면 다음 네 가지 방법이 대표적이다.

- Static Storage Binding
- Stack-Dynamic Storage Binding
- Explicit Heap-Dynamic Storage Binding
- Implicit Heap-Dynamic Storage Binding

Static Storage Binding

Compile Time에 결정된다. 프로그램이 시작할 때 메모리에 할당된다. 프로그램이 끝날 때까지 메모리에 남아있다.

Global Variable, Static Variable 등이 있다.

매번 할당하는 과정이 없기 때문에 효율적이다. 다만, 유연성을 낮춘다.

Stack-Dynamic Storage Binding

C언어의 Local Variable 등이 있다. 특정 Statement가 실행될 때 메모리에 할당되고, Statement가 끝나면 메모리에서 해제된다.

재귀 함수 등 구현이 가능하다.

Statement가 실행될 때마다 메모리를 할당하고 해제하기 때문에 Overhead가 발생한다.

매번 생겼다 사라지기 때문에 History에 예민할 수 없다. (아마 함수에서 존재하는 변수의 포인터를 넘겨주지 못하는 걸 설명하는 듯함)

Explicit Heap-Dynamic Storage Binding

Heap 메모리 영역을 사용한다. 프로그래머가 직접 메모리를 할당하고 해제한다.

C언어의 malloc, free 등이 있다.

Type Binding은 Static하지만, Storage Binding은 Run Time에 결정된다.

포인터를 정확하게 다루는 것은 어렵다.

Heap 메모리를 관리하는건 코스트가 크다.

Implicit Heap-Dynamic Storage Binding

Assignment가 발생하면 Heap 메모리 영역에 메모리를 할당하고 변수가 사라지면 메모리를 해제한다.

유연성이 높다. 다만, 모든 Assignment에 메모리를 할당 해제하기 때문에 Overhead가 크다.

몇몇 에러는 검출하지 못할 수도 있다.

Scopes

Scope는 특정 변수를 참조 가능한 범위를 말한다. Visible 하다는 것은 Reference가 가능하다는 뜻이다.

크게 Local, Non-Local, Global 세 가지로 나눌 수 있다.

- Local Variable: 프로그램 유닛이나 블록에서 정의된다.

- Non-Local Variable: 상위 블록에서 정의된 변수를 참조한다.
- Global Variable: 프로그램 전체에서 참조 가능한 변수이다.

스코프는 다음과 같이 다섯 가지로 분류할 수 있다.

- Static Scope
- Block
- Declaration Order
- Global Scope
- Dynamic Scope

Static Scope

특정 변수에 대해 Scope가 Statically하게 결정된다.

프로그램의 코드를 보고 상위 함수에 있는 변수를 참조한다. 참조 순서는 다음과 같다.

1. 현재 함수에 같은 이름을 가진 변수가 있는지 확인하고 있으면 사용한다.
2. 없다면 상위 함수에서 찾는다.
3. 없다면 상위 함수에서 찾는다.
4. ...

Block

특정 변수에 대해 Scope가 Block 단위로 결정된다.

Static Scope와 비슷하지만, 한 함수에서 블록이 다르다면 다른 변수로 인식한다.

Ex) if문 안에 있는 변수는 if문 밖에서는 사용할 수 없다.

참조 순서는 다음과 같다.

1. 현재 블록에 같은 이름을 가진 변수가 있는지 확인하고 있으면 사용한다.
2. 없다면 상위 블록에서 찾는다.
3. 없다면 상위 블록에서 찾는다.
4. ...

Declaration Order

예전에는 프로시저의 최상단에만 변수 선언이 가능했다.

하지만 최근 언어에서는 프로시저 어느 곳이든 원한다면 선언 가능하다.

Ex) for문에서 변수 선언하기

Global Scope

Global Variable은 묵시적으로 모든 함수에서 참조 가능하다.

예외는 같은 이름을 가진 Local Variable이 있을 때이다.

하지만 몇몇 언어에서는 Scope Operator를 사용해서 Global Variable을 참조할 수 있다.

Ex) C++에서 `::`를 사용해서 Global Variable을 참조할 수 있다.

- Declarations: 타입을 정한다. 하지만, 대입은 하지 않는다.
- Definitions: 타입을 정하고 대입까지 한다.

Multiple Declarations은 가능하지만, Definitions은 한 개만 가능하다.

C에서는 extern으로 Declarations을 해주면 다른 파일에서 사용할 수 있다.

PHP에서는 $GLOBALS['{NAME}'] 을 통해 Global Variable을 사용할 수 있다.

Python에서는 global 키워드를 통해 글로벌 변수를 참조하거나 nested function에서 nonlocal 키워드를 통해 상위 함수의 변수를 참조할 수 있다.

Dynamic Scope

옛날 버전 LISP나 JavaScript에서 사용한다.

코드를 보고 Nested Function에서 상위 함수의 변수를 참조하지 않는다.

실행 시간에 실제 Call Stack을 보고 자신을 Call 해준 상위 함수의 변수부터 찾기 시작한다.

- 실행 시간에 Scope가 결정된다.
- Call Stack에 따라 참조하는 변수가 달라질 수 있다.
- 신뢰성을 떨어뜨린다.
- Static한 Type Checking이 불가능하다.
- 가독성을 떨어뜨린다.
- 실행 시간이 길다.

Static Scope가 가독성이 높고, 신뢰성이 높고, 실행 시간이 빠르다.

[4주] Scopes_DataType (2)

Names, Bindings and Scopes

Scope and Lifetime

Scope와 Lifetime은 밀접한 관계이지만 같지는 않다.

C언어의 Static Variable을 보면 Scope는 함수 내부이지만, Lifetime은 프로그램 전체이다.

Named Constants

한 번만 Bound시킨다. (변하지 않는다.)

Ex) const int PI = 3.141592

가독성을 늘려주고, 프로그램을 Parameterize 한다. (상수에 이름을 붙여 쓰기 편하게 해준다.)

바뀌면 안 되는 값을 명시적으로 정해줄 수 있어서 실수를 막아준다.

Data Types

데이터 타입은 변수가 가질수 있는 값의 종류와 그 값들에 대한 연산의 종류를 결정한다.

- 타입 시스템을 도입하면, 에러를 검출할 수 있다. (Type Checking)
- Modularization: 모듈의 일관성을 보장할 수 있다. (모듈화)
- Documentation: 프로그래머가 의도한 바를 타입으로써 힌트를 줄 수 있다. (문서화)

Primitive Data Types

언어에서 기본적으로 제공하는 타입이다.

- Integer: 정수 타입
- Floating: 실수 타입 IEEE 754 표준을 따른다.
- Complex: 복소수 타입 (파이썬에서 (7 + 3j) 같은걸로 쓸 수 있음)
- Decimal: floating point problem을 해결하는 방법. (파이썬에서 Decimal 모듈을 사용할 수 있다. 매우 정교한 실수 연산이 필요할 때 사용. 매우 느리고 메모리를 많이 먹음)
- Boolean: T/F. Switch와 Flag에 사용. 가독성 증가
- Character: ASCII, Unicode, UTF-32 등이 있다.
- Character String: 캐릭터 배열. C에서는 Null Terminated String을 사용한다. (문자열의 끝을 알리기 위해 Null을 사용한다.)
    - 디자인 이슈, 캐릭터 배열로 제공할 것인가, 혹은 프리미티브로 제공할 것인가.
    - 길이는 Static하게 할 것인가, Dynamic하게 할 것인가.
    - 대입, 슬라이싱, 비교, 패턴 매칭 등 연산을 제공할 것인가.
    - C에서는 라이브러리로써 제공하고, 파이썬에서는 프리미티브로 제공
    -
        -
    - Static Length String: 고정된 길이의 문자열
    - Limited Dynamic Length String: 제한된 길이의 문자열 문자열 길이는 바뀔 수 있다. (C 배열 구현 느낌)
    - Dynamic Length String: 길이가 바뀔 수 있는 문자열 (C++, Java 문자열)
        - 메모리 관리가 어렵다.
        - 링크드 리스트 구현. (간단하지만 느리다)
        - 포인터 배열로 구현 (여러개를 분산해서 저장. 링크드리스트보다는 빠르다)
        - 연속한 메모리 셀에 저장 (스트링 길이가 커질 때 문제가 발생 이를 해결해야 한다. 할당/회수가 느리다.)
    - 프리미티브 타입으로 주면, 문자열 연산이 더 많으므로 사용성이 좋다.

User-Defined Ordinal Types

유저 정의 서수 타입

정수의 범위를 제한하는 집합과 비슷하다. 다음 두 가지 타입이 있다.

- Enumeration Type
- Subrange Type

Enumeration Type

다음과 같이 사용할 수 있다.

```
enum days{Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

암묵적으로 Mon: 0, Tue: 1 ... 값을 가진다.

하지만, 명시적으로 원하는 값을 대입할 수도 있다.

디자인 이슈

- 같은 이름이 서로 다른 enum에 등장할 수 있는가?
- 정수형으로 변환할것인가?
- 다른 타입은 지원하지 않을 것인가? (Ex. Mon: 0.1)
- 타입 체킹은 어떻게?

Enum타입이 없으면 int로 대체할 수 있는데, 이는 변경될 수 있다. 그리고 타입 체킹의 신뢰성도 낮다. (days 값에 15를 넣으면 뭐임?) (신뢰성 하락)

enum에서 days now = Mon; now++; 같은게 된다.

C++에서는 enum class를 사용하면 enum의 값이 다른 enum에 영향을 미치지 않는다.

Ada에서는 여러 Declaration이 허용된다. Over Loading됨.

- Readability가 증가함. 상수 대신에 이름을 붙일 수 있으니까.
- Reliability가 증가함. 몇몇 언어에서는 수학적 연산을 제한함. (실수할 확률이 줄어듬)
- C/C++에서는 범위 내부에 존재하는지 확인하는데 Explicit하게 1, 10 100 으로 만들면 37 같은 값은 범위에 있기 때문에 허용됨 -> 디버깅 어려워짐

Subrange Type

서수 타입의 연속된 부분수열이다. 12..27 같은게 될 수 있다.

Pascal이나 Ada에서 사용할 수 있다.

디자인 이슈는 없다.

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
subtype Weekdays is Days range Mon..Fri;
subtype Index is integer range 1..100;

Day1 : Days;
Day2 : WeekDays;

Day2 := Day1;
```

처럼 사용할 수 있다.

- 코드를 읽는 독자에게 이 변수의 정확한 Range를 인지시켜줄 수 있어서 가독성이 높아진다.
- Range Checking을 통해 Reliability가 높아진다. (범위를 벗어나면 에러를 발생시킨다.)
- 현대엔 Ada말고는 지원되지 않는다.

- 컴파일러에서 Range Checking을 Implicit하게 지원하는 것을 제외하고는 Enum과 유사하다.

[5주] DataType

Data Types

Structured Data Types

구조화된 데이터 타입 다음과 같은 타입들이 존재한다.

- Array
- Associative Array
- Record
- Tuple
- List
- Union

Array

같은 타입의 변수를 연속적으로 저장하는 타입이다. 첫번재 원소에서 상대적인 Index를 통해 각 원소에 접근 가능하다.

디자인 이슈

- 인덱스 타입은 뭘 지원?
- Range Checking을 지원?
- Bounded Subscript? (배열의 길이를 미리 정해놓는 것)
- 언제 메모리에 할당?
- 다차원 배열?
- 배열을 초기화 가능 (초기 값 세팅 가능?)
- Slices (부분 배열 자르는 연산 지원?)

Indices (Index 복수)

인덱스는 원소를 접근하기 위해 사용 Ordinal Type (서수 타입)은 subscript (Index)로 사용 가능

Range Checking은 Reliability에 중요한 요소이다. (범위를 벗어나면 에러를 발생시킨다.)

Perl이나 Python에서는 음수 인덱스를 허용한다. (뒤부터 계산함)

Subscript Bindings (배열 길이 정하기)

보통 Static하게 결정되지만, 가끔 Dynamic하게 결정되기도 한다.

Lower Bound (시작 인덱스) 는 보통 Implicit하게 결정된다. (C는 0, Fortran 95나 matlab은 1부터 시작한다. Pascal은 시작 인덱스를 원하는 대로 정할 수 있다.)

Subscript (Index) 의 범위는 다음과 같이 다섯 가지 방법으로 결정된다.

- Static Array : `static int a[10];` 같은 방법으로 정적으로 결정된다. Compile Time에 결정 및 배정된다.

- Fixed Stack-Dynamic Array : `int a[10] // in main()` 같은 방법으로 고정됐지만 스택 다이나믹하게 결정된다. Compile Time에 결정되고 Run Time에 배정된다.
- Stack-Dynamic Array : `int n; int a[n];` 같은 방법으로 스택 다이나믹하게 결정된다. Run Time에 결정 및 배정된다.
- Fixed Heap-Dynamic Array : `malloc` 같은 방법으로 고정된 크기의 힙을 할당받아 사용한다.
- Heap-Dynamic Array : Python의 List 같은 방법으로 동적으로 크기가 결정된다. (사실 std::vector 라고 생각해도 된다.)

Initialization

배열을 생성할 때 배열의 값을 원하는 값으로 초기화 할 수 있다.

Operations

대입하거나, 연결하거나 비교하거나 슬라이싱하는 연산을 제공한다. 모든 언어가 제공하는 것은 아니고, 보통 클래스 타입으로 배열을 제공하는 곳에서만 제공하는 경우가 많다.

Rectangular Array

C언어의 다차원 배열이 이런 경우다. 모든 행 열 등의 길이가 같다.

Jagged Array

파이썬의 다차원 배열이 이런 경우다. 모든 행 열의 길이가 달라질 수 있다.

std::vector < std::vector < int > > 같은 경우도 이렇게 생각할 수 있다. 하지만 깊이도 달라질 수 있기에, 파이썬 같은 동적 타입 리스트를 nested로 사용하는걸 Jagged Array (고르지 않은 배열) 라고 분류하는게 더 올바르다.

Slices

배열의 Sub Array를 잘라내는 연산이다. 파이썬 같은 언어에서 `[1, 2, 3][1:]`처럼 지원한다

Implementation

1차원 배열의 경우 연속한 메모리 셀을 할당해 배열을 구성할 수 있다.

다음과 같이 접근할 수 있다. `a[i] = *(a + i * size)` a 값은 컴파일 타임에 계산되지만, i * size 값은 런타임에 결정된다.

N차원 배열의 경우, N - 1차원 배열의 배열이라고 생각할 수 있다.

Associative Array

배열 타입은 다음과 같이 두 가지가 있다.

- Associative Array : 연관 배열 (Key, Value)로 이루어진 배열 - std::map 파이썬 딕셔너리 등으로 생각하면 된다.
- Non-Associative Array : 일반적인 배열

Record

데이터의 집합으로 새로운 타입을 정의하는 방식이다. Pascal의 Record, C의 struct 같은 개념이다.

배열과 비슷하다고 생각할 수 있지만 큰 차이가 있다.

- 배열: 같은 타입의 변수를 연속적으로 저장하는 것
- 레코드: 서로 다른 타입의 변수를 묶어서 하나의 타입으로 정의하는 것

레코드에서 각 변수를 접근하기 위해선 Index가 아닌 Identifier (변수 이름) 으로 접근한다.

Implementation

모든 필드는 각각 다른 사이즈를 가지고 있다. 따라서 시작 주소로부터 offset을 계산해서 접근할 수 있도록 구현한다.

Tuple

레코드와 비슷하지만, 레코드와 다르게 필드에 이름이 없다.

List

파이썬의 배열을 말한다. 왜 나눠놓은건지 모르겠다.

Union

레코드와 비슷하지만, 레코드와 다르게 모든 필드가 같은 시작 Offset을 가진다.

디자인 이슈

- 타입 체킹이 필요한가?
- 레코드에 포함될 수 있는가?

Free Unions

타입 체킹이 이뤄지지 않는다. 따라서 신뢰성이 떨어진다.

Discriminated Unions

Type Indicator (Tag, Discriminant) 가 추가된다. (유형 지시자?)

Union을 실제 쓰는 곳에서 이 곳에서는 어떤 타입으로 사용할 것이지 명시한다.

따라서 Explicit하게 타입을 강제하므로 Static한 Type Checking이 가능하다.

유니온은 기본적으로 Un-Safe하다. 따라서 지원하지 않는 곳도 있다.

---

implementation

모든 필드가 같은 시작 Offset을 가지도록 코드를 생성한다. 가장 큰 타입에 맞게 메모리를 할당한다.

실제 실행 시 태그도 같이 저장된다.

Pointer

메모리 주소를 가리킨다. 포인터는 두 가지 목적이 있다.

- 주소를 직접 가르키기
- Dynamic Storage (Heap) Management

비슷한 예시로 Reference Type이 있다. 단, 이는 Syntax Sugar일 뿐, 실제로는 포인터로 구현된다.

디자인 이슈

- Scope와 Life Time은 어떻게 정할 것인가?
- Heap 할당 시 Life Time은 어떻게 정할 것인가?
- 타입의 값을 어떻게 제한할 것인가?
- 주소를 가리키는 목적으로 쓸 것인가 힙을 할당받을 용도로 쓸 것인가 혹은 둘 다?
- 포인터의 타입에 맞는 주소만 가리키게 만들 것인가?

Operations

- Assignment: 포인터 변수에 주소를 대입시켜주는 연산이 있다.
- Dereferencing: 포인터 변수가 가리키는 곳에서 값을 가져온다.

Problems

- Dangling Pointer: 이미 해제된 메모리를 가리키는 포인터
- 잘못된 주소로 힙 영역을 망가뜨릴 수도 있다.

Ada에서는 Scope 내에서 명시적으로 힙을 할당 해제해야 하도록 한다.

Tombstones를 사용해서 Pointer -> Tombstone -> Heap Memory 를 하도록 한다. Decallocation 되면 Tombstone -> NULL 이 되도록 한다. 이렇게 하면 Error가 났을 때 오류를 Detection 할 수 있다.

하지만 시간과 공간이 더 소요돼서 비효율적이다.

Locks and Keys Approach (Locks and Keys를 사용한 접근) 으로 문제를 해결할 수도 있다.

모든 포인터 주소는 (Key, Address)를 갖도록 설계한다. 이후 Heap에서 할당받은 메모리를 Lock한뒤 할당받은 포인터에서 Key값을 대조해 일치하면 사용할 수 있도록 한다. 이렇게 하면 잘못된 접근을 막을 수 있다.

하지만 이도 Deallocation되면 Lock 값이 올바르지 않아 오류가 발생할 수 있다.

Lost Heap Dynamic Variable

만약 접근하지 못하는 변수가 있다면, 힙에 자동으로 반납하도록 한다.

만약 이미 할당받은 값이 있는 P값이 다른 값으로 재할당받으면 기존에 있는 P값을 자동으로 할당 해제 해준다는 것이다.

GC와 비슷한 개념이다.

Pointer Arithmetic

타입을 가지 포인터는 수학적 연산을 유연하게 제공한다.

4바이트 타입을 가진 배열을 가리키는 포인터는 +1을 하면 실제로는 +4를 수행하며 다음 원소를 가리킬 수 있도록 제공한다.

Reference

포인터와 비슷하지만, 실제로는 조금 다르다. (하지만 내부적으로는 동일하다.) Syntax Sugar for using Pointer Without Dissuse

항상 초기화되어 있어야 한다. (포인터는 초기화하지 않아도 된다.)

constant pointer와 비슷한 효과를 얻는다.

포인터는 Explicit하게 Dereferencing해야 하지만, Reference 타입은 Implicit하게 Dereferencing한다.

이를 활용한 함수를 Call-By-Reference라고 한다.

Heap Management

힙을 관리하기엔 Deallocation이 언제 발생하는지 알 필요가 있다.

힙 메모리는 여러 변수에 의해 참조될 수 있다. 그래서 힙을 가리키는 변수가 언제 usefulless 해지는지 알기 어렵다.

Garbage Collection

GC라고 불리며, 크게 두 가지 방법이 있다.

- Reference Counter : 해당 메모리를 참조하는 변수의 개수를 세서 0이 되면 해제한다. 하지만, 실제로 사용되는 곳은 없지만, Cyclic Reference가 발생하면 해제되지 않고 Memory Leak이 발생한다. 또, 카운트를 저장해야 하기 때문에 메모리를 더 많이 사용한다. 하지만 빠르다.
- Mark-Sweep: Visible하지 않은 메모리가 있는지 검사하고 발견된다면 해제한다. Reference Counter보다는 비효율적이지만, 누수되는 메모리 없이 모든 메모리를 할당 해제할 수 있다.

Type Checking

올바른 연산인지 확인하기 위해선 타입 체킹이 필수적으로 이뤄져야 한다. 하지만 타입 체킹을 제대로 하기 위해선 일반화가 필요하다.

Strong Typing

강하게 타입의 일치를 확인한다. C에서는 int와 float를 더하면 float가 되지만, Ada에서는 에러를 발생시킨다.

Coercion (강제 변환)은 타입 체킹에서 중요한 문제이다.

Type Equivalence (타입 동치성)

- Type Compatibility (유형 호환성): 두 타입이 같은 연산을 지원하는가?
    - Operand Can Be Implicitly Converted (Coercion) : 다른 타입을 연산했을 경우 (int + float) 묵시적으로 float로 바뀔 수도 있다. 이를 Coercion이라고 한다.
- Type Equivalence (유형 동등성): 실제 메모리에 저장되는 포맷이 동일한가? (Coercion이 없어도 되는 경우)
    - 똑같은 이름을 가진 타입이거나, 구조가 똑같은 레코드를 말한다.
    - 하지만 구조가 똑같은 레코드를 허용했을 경우 실제 다른 레코드인데 연산이 허용되는 문제가 있다.

[5주] DataType_Expression_Assignment

Expressions and Assignment Statements

Expression은 일반적으로 PL에서 정의돼야 하는 내용이다. Syntax와 Semantic 모두 중요하다.

연산 순서가 중요하다.

Arithmetic Expressions

수학적으로 제공하는 사칙연산 등은 제대로 계산되어야 한다.

Overloaded Operators

연산자 오버로딩이다. 원래 목적이 아닌 커스텀해서 사용할 수 있다.

주의 사항:

- '&', '-' 같은거는 unary, binary operator 모두 적용될 수 있다.

만약 Matrix 객채에 행렬곱 행렬덧셈 등 연산자를 정의하면 가독성에 도움이 된다.

잘못 사용했을 경우 가독성과 신뢰성(쓰기 능력) 모두 잃을 수 있다.

Type Conversion

타입 변환은 두 가지 방법이 있다.

- Narrowing Conversion: 큰 쪽에서 작은 쪽으로 변형되는 것이다. double -> float 같은 경우 손실이 발생한다.
- Widening Conversion: 작은 쪽에서 큰 쪽으로 변형되는 것이다. float -> double 같은 경우 손실이 없다.

Type Conversion은 Implicit, Explicit 두 경우가 존재한다.

Implicit Type Conversion을 막는 언어도 존재한다. (신뢰성 상승, 작성력 감소)

Implicit Type Conversion을 일어나는 걸 Coercion이라고 한다.

Short Circuit Evaluation (단락 평가)

만약 긴 수식에서 뒷 연산이 결괏값을 바꾸지 못한다는 사실을 알면 뒷 연산을 아예 평가하지 않는다.

예를 들어, 0 * 1 * 2 * ... 라는 수식이 있다면, 0이 곱해진 순간 뒷 연산을 하지 않더라도 결괏값은 0임을 알 수 있다. 따라서 뒷 연산을 하지 않아도 된다. 하지만 수학적으로 detection하기는 어렵다.

따라서 AND/OR 연산에 대해서 Short Circuit Evaluation을 사용한다.

AND 식에서 앞 조건이 거짓이면 뒷 조건이 어떤 값이든 거짓이기 때문에 뒷 조건을 평가하지 않는다.

OR 식에서 앞 조건이 참이면 뒷 조건이 어떤 값이든 참이기 때문에 뒷 조건을 평가하지 않는다.

따라서 뒷 조건에서 side effect를 발생시킨다면, 의도하지 않은 결과가 나올 수 있다.

Assignment Statement

대입 연산은 Expression으로 사용할 수 있어야 한다.

while ((ch = getchar()) != EOF) {} 같은 식으로 작성하거나

a = b = c 같은 식으로도 쓸 수 있어야 하기 때문이다.

하지만 이때문에 실수할 수도 있다.

sum = count = 0 일 때 둘 다 0이 되는지 어떻게 되는지 알기 어렵다 (우측부터 봐서 둘 다 0되는게 맞음)

if (x = y) 처럼 실수할 수도 있다. (제대로 평가되서 실행되기 때문)

[6주] Statement-Level Control Structures

Statement-Level Control Structures

Guarded Commands

if 조건에서 모든 경우를 처리하고 있는지 확인하기 위해 사용한다.

```
if <boolean_expression> -> <statement>
[] <boolean_expression> -> <statement>
[] <boolean_expression> -> <statement>
[] <boolean_expression> -> <statement>
[] <boolean_expression> -> <statement>
fi
```

같은 식으로 사용한다.

```
if i = 0 -> sum := sum + j
[] i > j -> sum := sum + i
[] i < j -> sum := sum + j
fi
```

이런 식으로 사용할 수 있다.

이렇게 되면 각 state에서 조건을 만족하는 항목 중 랜덤으로 실행되며, 만약 조건에 만족하는 항목이 없을 경우 에러를 발생시켜 조건이 잘못됨을 알려준다.

반복문에서도 사용이 가능하다.

```
do <boolean_expression> -> <statement>
[] <boolean_expression> -> <statement>
[] <boolean_expression> -> <statement>
[] <boolean_expression> -> <statement>
[] <boolean_expression> -> <statement>
od
```