2013 1st Compiler Programming Team Project #1

SNSD Interpreter

간단한 interpreter 언어

김태연

이재연

201125162이철민

목차

[1 서론 4](#_Toc358993404)

[1.1 개요 4](#_Toc358993405)

[1.2 지원 요소 4](#_Toc358993406)

[1.2.1 상수 4](#_Toc358993407)

[1.2.2 변수 4](#_Toc358993408)

[1.2.3 수식(expression)과 연산자 4](#_Toc358993409)

[1.2.4 문장(statement) 4](#_Toc358993410)

[1.2.5 함수(function) 5](#_Toc358993411)

[2 분석 5](#_Toc358993412)

[2.1 lexlexical analyzer의 축소 이용 5](#_Toc358993413)

[2.1.1 Identifier 5](#_Toc358993414)

[2.1.2 number 6](#_Toc358993415)

[2.1.3 operator 6](#_Toc358993416)

[3 설계 6](#_Toc358993417)

[3.1 자료구조 6](#_Toc358993418)

[3.1.1 Node 6](#_Toc358993419)

[3.1.2 Id 6](#_Toc358993420)

[3.2 모듈 차트 7](#_Toc358993421)

[4 구현 7](#_Toc358993422)

[4.1 Verbatim copied code 7](#_Toc358993423)

[4.2 Definition Section 8](#_Toc358993424)

[4.3 Rule Section 8](#_Toc358993425)

[4.4 User-Defined Code Section 8](#_Toc358993426)

[4.4.1 Expression, Factor, Terminal 노드 생성 8](#_Toc358993427)

[4.4.2 evalNode 9](#_Toc358993428)

[5 결과 테스트 9](#_Toc358993429)

[6 결론 10](#_Toc358993430)

# 서론

## 개요

i-lang은 간단한 interpreter형 언어이다. i-lang의 문법은 기본적으로 C언어와 유사하나 몇몇 구문은 차이가 있다. 인터프리터는 두 단계로 실행되는데, 일단 단위(문장 또는 함수)별로 코드를 syntax tree로 변경한 후, 즉시 interprete하여 그 결과를 보여준다.

## 지원 요소

### 상수

* 정수형 상수(integer)와 실수형 상수(double)를 지원함
* 지수에 대한 사항은 지원하지 않음

### 변수

* 변수는 선언하지 않고 사용함
* 변수의 형(type)은 assign 시 결정 됨
* 변수의 형(type)은 실행 시 변경이 가능함

### 수식(expression)과 연산자

* 가감승제 연산자 (+, -, \*, /)
* 비교 연산자 (>, >=, <, <=, ==, !=)
* 할당, 대입 연산자assignment operator(=)
* 정수, 실수 혼합식은 실수로 형을 변환

### 문장(statement)

* 기본 문장은 수식에 ‘;’를 넣어 문장을 만든다. 단, 하나의 문장이 여러 라인에 걸쳐 있을 수는 있지만, 두 개의 문장이 하나의 라인에 놓일 수 없다.

#### if statement

|  |
| --- |
| **if** (expression)  statement list  **else**  statement list  **end** |

#### while statement

|  |
| --- |
| **while** (expression)  statement list  **end** |

### 함수(function)

* **def**로 시작하고 **end**로 끝남
* parameter의 type이나 return type을 명시하지 않음
* parameter passing은 call by value 방식을 사용함
* local variable을 선언할 수 있음

|  |
| --- |
| **def** f (a, b)  local max  **if** (a > b)  max = a;  **else**  max = b;  **end**  return max;  **end** |

# 분석

## lexlexical analyzer의 축소 이용

과제 1에서 사용한 lexical analyzer에서 숫자와 사칙연산, identifier 부분을 추출하여 이용하였다. 따라서 인식할 수 있는 어휘는 다음과 같다.

### Identifier

문자로 시작하며 1개 이상의 문자와 숫자로 이루어진 단어를 의미한다.

Regular expression: [a-zA-Z][a-zA-Z0-9]\*

regular expression:

### number

제일 앞의 음수 기호를 포함하여 실수 범위까지 아우르는 숫자를 의미한다.

regular expression: -?([0-9]+|([0-9]\*\.[0-9]+)([eE][-+]?[0-9]+)?)

### operator

덧셈, 뺄셈, 나눗셈, 곱셈의 사칙 연산 기호를 의미한다.

regular expression: [-\\*/+]

# 설계

## 자료구조

본 인터프리터에서 필요한 자료구조로는 가장 기본이 되는 정수나 실수의 실제 데이터를 담거나 함수나 statement의 값을 담을 수 있는 Value, 그리고 ~~~~ 등이 있다..

### Value

Value는 정수나 실수 데이터와 statement나 function 혹은 error가 발생 했을 시 값(value)을 저장하는 type이다. 각각의 경우 중 한가지만 가질 수 있도록 union으로 묶여있고 Value 가 실제로 가진 값을 알리기 위한 enum 이 추가적으로 포함된 struct로 구성되었다. 실제 선언은 다음과 같다.

|  |
| --- |
| typedef enum { INTVALUE = 1000, DOUBLEVALUE, STATEMENTVALUE, FUNCTIONVALUE, ERRORVALUE } tagValueType;  typedef struct Value {  tagValueType type;  union {  int intValue;  double doubleValue;  char\* statementValue;  char\* errorValue;  struct Function\* functionValue;  };  } Value; |

### Node

Syntactic analysis를 위한 tree를 생성할 때 각 node를 표현하는 자료구조이다. Node는 정수나 실수 데이터를 갖고 있을 때를 위한 Value와 identifier 저장을 위한 char 배열, binary operation에서 operator를 저장하기 위한 integer 변수 그리고 child node저장을 위한 node 배열을 포함한다. Value와 마찬가지로 현재 Node가 어떤 데이터를 담고 있는지를 알기 위한 enum도 포함되어있다.

|  |
| --- |
| typedef enum {  NTSTATEMENTLIST,  NTSTATEMENT,  NTIFSTATEMENT,  NTWHILESTATEMENT,  NTASSIGNMENT,  NTEXPRESSIONLIST,  NTEXPRESSION,  NTIDENTIFIER,  NTBINARYOPERATOR,  NTUNARYOPERATOR,  NTINTEGER, NTDOUBLE,  NTFUNCDECLARE,  NTFUNCCALL,  NTPARAMETERLIST  } tagNodeType;  typedef struct Node {  tagNodeType type;  Value value;  char name[100];  int op\_token;  struct Node\* child\_nodes[100];  int n\_of\_child\_nodes;  } Node; |

### Frame

Frame은 function이 호출됐을 때 해당 function의 local variable을 저장하는 정보들과 function 종료 후 되돌아갈 지점을 저장하는 자료구조이다.

|  |
| --- |
| typedef struct Frame {  int n\_of\_local\_variables;  char local\_variables\_name[MAX\_LOCAL\_VARIABLES][MAX\_VARIABLE\_NAME];  Value local\_variables\_value[MAX\_LOCAL\_VARIABLES];  Value\* pReturnValue;  } Frame; |

### Function

Function 자료구조는 function이 호출 됐을 때 호출하며 넘길 parameter의 list와 function의 기능을 수행할 statement의 node를 가리키는 포인터를 갖고 있다.

|  |
| --- |
| typedef struct Function {  Node\* parameter\_list;  Node\* statement\_list;  } Function; |

## 모듈

3.1절에서 서술한 자료구조들에 맞춰 각 자료구조를 다루고 지원하기 위한 모듈과 여러 함수들을 제작했다. 편의 상 각 자료구조를 다루는 함수들의 묶음을 모듈이라 정의하고 세부 항목으로 나누어 설명하겠다.

### Value 관련

#### void cal\_value()

binary operator를 담고 있는 Node 에 대응하기 위한 함수이다. 양 변에 해당하는 Value 2개와 operator 정보를 담고 있는 integer type의 op\_token이 parameter로 넘어온다. 넘어온 operator에 대응하는 code가 작성되어 있다. 그리고 나눗셈에 대해 0으로 나눈 상황에 대한 에러처리를 하는 코드 역시 작성되어 있다. integer끼리 나눗셈을 할 때 정확하게 나누어 떨어지지 않는 경우(5/3 과 같이) 에 대해 소수점 이하를 버리는 즉, integer type을 유지하는 구현 상 선택을 하였다.

|  |
| --- |
| void cal\_value(Value\* pValue, Value a, Value b, int op\_token) {  ……  switch(op\_token)  {  case PLUS : result.intValue = a.intValue + b.intValue; break;  case MINUS : result.intValue = a.intValue - b.intValue; break;  case MULTIPLY : result.intValue = a.intValue \* b.intValue; break;  case DIVIDE : if(b.intValue == 0) {  result.type = ERRORVALUE;  result.errorValue = "Can not divied by zero";  }  else {  result.intValue = a.intValue / b.intValue;  }  break;  ……… |

#### void cal\_uminus()

이것은 unary value를 다루기 위한 함수인데 본 언어에서는 음수만을 다루기 때문에 해당 작업을 수행하는 코드가 작성되어있다.

|  |
| --- |
| void cal\_uminus (Value\* pValue, Value a)  {  ………..  if(a.type == INTVALUE)  {  pValue->type = INTVALUE;  pValue->intValue = a.intValue \* -1;  }  ……….. |

#### void test\_value()

주어진 값이 어떤 데이터인지 판별하여 Value의 값이 어떤 종류의 값인지를 나타내는enum 를 결정하여 대입해주는 함수이다.

|  |
| --- |
| void test\_value(Value\* pValue, Value a) {  Value result;  result.type = INTVALUE;  if (a.type == INTVALUE) {  result.intValue = a.intValue != 0;  } else if (a.type == DOUBLEVALUE) {  result.intValue = a.doubleValue != 0.0;  } else if (a.type == ERRORVALUE) {  result = a;  } else {  result.type = ERRORVALUE;  result.errorValue = "invalid test.";  }  \*pValue = result;  } |

#### void print\_value()

Parameter로 넘어온 value의 값을 화면에 출력하기 위한 함수이다.

### Node 관련

#### void init\_node()

Parameter로 넘어온 node pointer에 대해 pointer가 가리키는 node에 메모리 할당과 함께 초기화를 해주는 기능을 한다.

#### void push\_child\_node()

2개의 parent node와 child가 될 node 주소를 받아 parent node의 child node list에 추가 해주는 기능을 한다.

#### void print\_node()

parameter로 들어온 node의 type과 value를 화면에 표시해 주는 함수이다.

### Frame 관련

Frame은 3.1.3에서 설명했듯이 function을 호출했을 때 생성되는 자료 구조인데 이 것은 function내에서 다시 function을 호출할 수 있으므로 여러 개의 frame이 존재할 수 있고 이는 FIFO 방식으로 관리되어야 할 것이다. 따라서 frame들을 보관하는 것은 stack 구조로 관리되며 이것은

|  |
| --- |
| Frame frame\_stack[];  int top\_of\_frame\_stack; |

로 선언되어 사용한다.

#### void init\_frame()

function이 호출 됐을 때 처음으로 실행되는 함수로써 각 function에 해당하는 frame을 초기화 한다.

### 기타 value의 evaluation 관련

#### void evaluate()

parameter로 넘어온 node와 value에 대해 node가 저장하고 있는 값을 평가한 뒤 결과를 value에 넣는다. Node의 값을 평가하는 동작은 node가 포함하고 있는 type 정보에 따라 달라진다. 즉, node가 binary operation을 담고 있는 node라면 op\_token으로 저장된 operator 연산을 child\_node로 저장된 2개의 node값을 연산한다.

|  |
| --- |
| void evaluate(Node\* pNode, Value\* pValue) {  switch(pNode->type) {  ……….  case NTBINARYOPERATOR:  {  Value a, b;  evaluate(pNode->child\_nodes[0], &a);  evaluate(pNode->child\_nodes[1], &b);  cal\_value(pValue, a, b,pNode->op\_token);  }  ……….. |

# 구현

Flex의 담았다.

## Verbatim copied code

정의하였다.

|  |
| --- |
| #define MAXTABLE 200  #define  #define ASSIGN 1031  #define LPARE 1023  #define RPARE 1024 |

그리고 앞서 설명한 node와 id를 위한 구조체를 선언하였다.

|  |
| --- |
| typedef struct \_Node {  char op;  int iValue; } Id; |

## Definition Section

Definition Section에는 정규 표현식을 이용해 필요한 내용을 정의하였다.

|  |
| --- |
| number |

## Rule Section

Rule section.

|  |
| --- |
| {id} {return ID;}  {number} {return NUM;}  {oper} {return OPER;}  "(" return LPARE;  ")" return RPARE;  "=" return ASSIGN;  \n {return EOL;}  . { ECHO; } |

## User-Defined Code Section

먼저

### Expression, Factor, Terminal 노드 생성

일단 본 과제에 적용할 BNF는 다음과 같다.

|  |
| --- |
| <expression> ::= <term> [ [ "+" | "-" ] <term> ] ...  <term> ::= <factor> [ [ "\*" | "/" ] <factor> ] ...  <factor> ::= <number> | "(" <expression> ")" |

이와 같은 방식으로 각 항목에 대해 Recursive Descent Parsing 방식을 이용해 syntax tree를 생성하도록 코드를 작성하였다. 각각의 코드는 다음과 같다.

|  |
| --- |
| Node \*makeExpTree()  { |

먼저 termTree 생성 함수를 호출 후 생성되어 온 node에 대해 다음 문자를 보고 +나 –면 새로운 term이기 때문에 전달받은 node를 left에 그리고 새로운 term을 right에 넣는 방식을 취했다.

|  |
| --- |
| Node \*makeTermTree()  {  } |

Terminal에.

|  |
| --- |
| Node \*makeFacTree()  {  } |

마지막으로.

### evalNode

생성된 트리에 대해 순회를 하며 값을 계산하는 함수이다.

|  |
| --- |
| int EvalNode(Node \*aNode)  {  } |

이는

해준다.

|  |
| --- |
| for (i=0; i < MAXERROR; i++)  {  printStrTab(); |

# 결과 테스트

사칙

|  |
| --- |
| : |

# 결론

윈도우7 64bit 환경에서 flex 2.5.4 버전으로 과제를 수행하였다.