Journal of The Korea Society of Computer and Information
Vol. 19, No. 2, February 2014
www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2014.19.2.151

모나드를 이용한 비결정적 컴파일러 구현

변 석 우 *

Implementation of Nondeterministic Compiler Using Monad

Sugwoo Byun *

요 약

본 연구에서는 Haskell의 모나드 기법을 이용한 명령형 언어의 컴파일러 구현에 대해 논의한다. 이 컴파일러는 한 생성 룰이 입력 스트링의 파싱을 실패할 때 다른 생성 룰로써 파싱하는 backtracking 기능의 비 결정적 Recursive Descent Parser를 포함한다. Haskell은 파싱에 필요한 우수한 기능들을 가지고 있다. Haskell의 대수적 타입은 추상구문트리를 자연스럽게 표현할 수 있으며, 모나드 파싱을 적용한 프로그램 코드는 매우 간결하여 가독성이 좋고, 타 언어에 의해 구현된 것에 비해 코드의 양이 획기적으로 감소된다. 이 컴파일러의 목적 코드는 스택 머신을 기반으로 한 Stack-Assembly 언어로서 이것을 위한 코드 생성과 어셈블러 실행 환경에 대해서도 논의한다.

▸ Keywords : 하스켈, 모나드, 파서 모나드, 비결정적 파싱, 스택 머신

Abstract

We discuss the implementation of a compiler for an imperative programming language, using monad in Haskell. This compiler involves a recursive-descent parser conducting nondeterministic parsing, in which backtracking occurs to try with other rules when the application of a production rule fails to parse an input string. Haskell has some strong facilities for parsing. Its algebraic types represent abstract syntax trees in a smooth way, and program codes by monad parsing are so concise that they are highly readable and code size is reduced significantly, comparing with other languages. We also deal with the runtime environment of the assembler and code generation whose target is the Stack-Assembly language based on a stack machine.

▸ Keywords : Haskell, Monad, Parser Monad, nondeterministic Parsing, Stack Machine

•제1저자 : 변석우 •교신저자 : 변석우
•투고일 : 2014. 1. 7, 심사일 : 2014. 1. 16, 게재확정일 : 2014. 1. 28.
* 경성대학교 컴퓨터공학부(School of Computer Science and Engineering, Kyungsung University)
* 이 논문은 2013년도 경성대학교 학술연구비지원에 의하여 연구되었습니다.

I. 서 론

본 연구에서는 순수 함수형 언어 Haskell의 모나드 기법을 이용한 While 명령형 언어의 컴파일러 구현 방법을 소개한다. 컴파일러 문법의 생성 룰은 일반적으로 비결정적인(nondeterministic) 특징을 가지고 있다. 한 넌터미널(nonterminal)에 대해 두 개 이상의 생성 룰(production rules)이 정의되어 있는 경우 입력 스트링의 파싱(parsing)을 위한 '적절한' 룰을 선택하는 것이 필요하며, 파서의 구현에서는 이 선택의 문제를 해결해야 한다.

파서의 구현은 결정적인 방법과 비결정적인 방법으로 구분될 수 있다. 비결정적 방법은 생성 룰의 비결정성을 그대로 구현하는 것으로서, 어떤 한 룰을 선택하여 파싱한 것이 실패한 경우 전 상태로 backtracking하여 다른 룰로써 파싱하는 것이다. 결정적(deterministic) 파싱은 입력 스트링을 미리 보고(look ahead) 파싱할 생성 룰을 결정하는 방법이다. 대부분의 컴파일러 파싱은 결정적 방법을 적용하고 있다. 이 방법은 까다로운 backtracking 프로그래밍을 피할 수 있고 파싱 속도가 빠른 장점을 갖는다. 그러나 이 방법을 구현하기 위해서는 생성 룰을 변화시키고 복잡한 파싱 테이블을 구성하는 등의 노력을 거쳐야 한다.

Haskell의 모나드 파서는 backtracking의 구현을 용이하게 할 수 있다. Backtracking의 구현이 용이하고, 파싱 속도가 크게 문제가 되지 않은 경우라면 비결정적 파싱을 적용하는 것이 편리할 수도 있다.

모나드에 의한 비결정적 파싱에 대한 연구는 Hutton과 Meijer에 의해서 처음 이루어졌으며[1], Hutton의 저서에도 소개되어 있다[2]. 이들의 연구는 단순한 산술식만을 다루고, 인터프리터를 위주로 이루어졌다.

본 연구에서는 이 기법을 확대 적용하여 산술식뿐만 아니라 할당문, 조건문, 반복문 등 명령형 언어의 주요 기능을 모두 내포하고 있는 While 언어를 대상으로 하고 있으며, 인터프리터가 아닌 컴파일러 구현의 관점에서 논의하고자 한다. 따라서 파싱뿐만 아니라 코드 생성 부분도 함께 구현되었는데, 목적 코드(target codes)로서 [3]의 2장에 소개된 스택 머신을 기반으로 한 Stack-Assembly 언어를 채택한다. 이 언어에 대한 어셈블러는 저자의 선행 연구[4]를 통해서 이미 개발되어, While 언어는 컴파일러와 어셈블러를 통해 수행될 수 있다.

본 논문의 구성은 다음과 같다. 제 2장에서는 Haskell의 동향과 모나드 프로그래밍 원리에 대해서 간략히 소개한다. 제 3장에서는 While 언어의 문법을 정의하고 이를 recursive descent parser 방식으로 구현하기 위한 EBNF 표현에 대해 논의한다. 제 4장에서는 EBNF에 대한 모나드 파서의 구현을 다룬다. 제 5장에서는 코드 생성(code generation) 과정을 논의하고, 마지막으로 제 6장에서 본 연구의 결론을 제시한다.

이 논문은 Haskell에 대한 기본 지식과 이해를 전제로 작성되었으며, 이것에 경험이 없는 독자는 [2][5-7]을 참조하기 바란다.

II. Haskell과 모나드

1. Haskell 관련 연구 동향

Haskell은 람다 계산법(The Lambda Calculus), 타입 이론(Type Theory), 의미론(Semantics) 등의 최신 기술을 적극적으로 반영하면서 발전하고 있는 순수 함수형 프로그래밍 언어이다[7]. 그 동안 함수형 언어는 주로 대학이나 연구기관을 위주로 사용되어 왔으나, 최근에는 이를 실용적으로 이용하려는 적극적인 노력이 진행되고 있다[8].

모나드 기술은 강한 이론적 배경[9] 하에 1998년 처음 Haskell에서 구현된[10] 후 지금까지 많은 관심을 받으면서 발전하고 있다[11]. 모나드 관련 기술은 Monad Transformer와 애로우 (arrow) 등으로 발전하고 있다[Arrow]. 모나드는 지금까지 Haskell에서만 이용되어 왔으나, 최근 Scala 언어에서도 채택되었다[12].

2. 모나드를 이용한 계산의 표현

순수 함수형 프로그래밍에서는 어떤 한 식은 유일한 값을 가지며, 계산은 그 식을 같은 값을 갖는 다른 식으로 치환(substitution)하는 방법을 적용하고 있다. 상대적으로 명령형 프로그래밍의 계산은 변수의 값을 동적으로 변화시키는 효과 (effect) 방법을 사용하고 있다. 치환에 의한 계산은 우수한 장점을 갖고 있으나 여러 다양한 분야의 응용 프로그램을 치환 방식만으로 표현하기는 어렵다. 그 대표적인 예로서, 입출력, 예외처리, 멀티미디어 등이 있다. 이들은 치환보다는 효과에 의한 계산 방식이 더 적합할 수 있다. 효과에 의한 계산은 값(value) 보다는 액션(action)을 표현하는데 더 적합하다. 따라서 Haskell과 같은 순수 함수형 언어는 치환과 효과를 동시에 가능케 하는 계산 원리를 구축하는 것이 절실히 필요했었다.

Haskell에서 계산과 값은 타입으로서 구분되어 표현된다. 한 계산 m을 수행한 결과 a 타입의 값을 출력하는 상황은 (m a)로 표현된다. 예를 들어, (IO Int)는 입출력 계산을 수행하여 그 결과로서 정수 값이 얻어지는 상황을 표현하고 있는데, 키보드를 통해 정수를 입력 받는 것이 대표적인 경우이다. Haskell에서는 타입 변수(영문 소문자로 시작되는 identifier로서 표현됨)를 사용하여 임의의 타입을 표현한다. (IO a)는 입출력의 결과 얻어지는 모든 값을 의미한다.

3. Type Constructor

모나드를 정의하기 위해서는 먼저 계산 m에 해당되는 타입 구성자(type constructor)를 정의해야 한다. 모나드를 위한 타입 구성자는 반드시 하나 이상의 인수를 가져야 한다. 예를 들어, data Maybe a = Just a | Nothing 로 정의되는 Maybe는 타입 a를 인수로 갖는 타입 구성자이다. Just 또한 타입 a를 인수로 갖는데, 오른쪽에 나오는 구성자는 데이터 구성자(data constructor)라고 부른다. 구성자는 Nothing처럼 인수를 갖지 않을 수도 있다.

4. 모나드 정의

Haskell의 클래스는 overloaded 함수를 구현하는 수단으로서, 모나드 또한 다음 그림 1과 같이 클래스로서 표현되고 있다. 이 클래스는 두 개의 오퍼레이터 return과 〉〉=(바인드)로 구성되어 있다.

```
class Monad m where
 return  :: a -> m a
 (>>=)   :: m a -> (a -> m b) -> m b
```

그림 1. 모나드 클래스와 오퍼레이터
Fig. 1. Monad Class and Operators

return은 어떤 값 a를 모나드 형태(즉, 액션)로 투입하는 기능을 한다. 함수의 합성이 두 함수를 연결하여 새로운 함수를 구성하는 것처럼, 바인드 또한 두 계산을 순서대로 연결하여 새로운 계산을 구성한다. 첫 계산 (m a)의 결과 얻어지는 값 a를 두 번째 계산 (a -> m b)으로 전달하며, 두 번째 계산 결과 (m b)가 두 연속된 계산의 결과가 된다.

타입 구성자 m에 대한 모나드는 Monad 클래스의 두 오퍼레이터에 대한 instance를 정의함으로써 이루어진다. 예를 들어, Maybe 모나드가 정의되는 과정은 다음과 같다. Maybe 타입은 계산 과정에서 실패가 발생하였는지의 여부를 판단하기 위해서 이용된다. 계산 과정에서 단 한번이라도 실패(Nothing 으로서 표현됨)가 발생하였다면 전체 결과는 실패가 되며, 모든 과정에서 실패가 없을 경우에만 정상적인 계산(Just로서 표현됨)으로 인정된다. 이 원리에 따른 Maybe 모나드는 다음 그림 2와 같이 정의될 수 있다.

```
instance Monad Maybe where
  return a    = Just a
  x >>= f     = case x of
                  Just a -> f a
                  Nothing -> Nothing
```

그림 2. Maybe 모나드의 정의
Fig. 2. Definition of Maybe Monad

다음은 Maybe를 적용한 더하기 함수를 두 가지 형태로 코딩한 것으로서, addM은 모나드를 이용한 것이고 add는 모나드가 아니다.

```
add :: Maybe Int -> Maybe Int -> Maybe Int
add Nothing _ = Nothing
add _ Nothing = Nothing
add (Just x) (Just y) = Just (x+y)

addM :: Maybe Int -> Maybe Int -> Maybe Int
addM x y  =
      x >>= \a -> y >>= \b -> return (a + b)
```

이 두 경우 모두 입력이 의미가 없는 값(Nothing)인지를 검토한 후 더하기를 하는 계산이다. add에서는 이 검토 과정이 명시적으로 코딩되고 있지만, addM에서는 이 과정이 앞서 정의된 모나드에서 수행되므로 자동적이고 묵시적인 방식으로 처리되어 코딩이 간결해 진다.

바인드 오퍼레이터 대신 치장된 구문(sugared syntax)인 do를 이용하면 위의 addM은 다음과 같이 표현된다.

```
addM x y  = do a <- x
               b <- y
               return (a + b)
```

do 표현은 중간 계산 결과를 변수에 저장하고 프로그램 수행이 위에서 아래 방향으로 수행되는 것처럼 느낄 수 있도록 하므로 명령형 프로그램과 유사한 느낌을 준다.

5. Parser 모나드

여기서는 [2]의 Parser 모나드를 우리의 목적에 맞게 간략히 정리하여 소개한다.

(1) Parser 타입 정의

파서는 스트링을 입력받아 이를 어떤 정해진 문법에 따라 파

싱하는 함수로서 이에 대한 타입 Parser는 다음처럼 정의된다.

newtype Parser a = P (String -> [(a,String)])

Parser 타입은 파싱 결과 (a, String)가 얻어지는 상황을 표현하고 있다. a는 파싱된 결과에 대한 타입을 의미하며, String은 앞서 파싱이 진행된 것을 제외한 나머지에 대한 입력 스트링을 표현하고 있다. 파싱된 결과 a는 의도에 따라 매우 다양한 형태로 표현될 수 있다. 파싱 결과를 연산하여 정수 값으로 계산하는 인터프리터라면 a는 Int가 되며, 만약 파싱의 결과를 추상구문트리(abstract syntax tree)로 표현한다면 a는 어떤 한 대수적(algebraic) 타입이 된다.

(2) Fail의 표현과 Parser 모나드 정의

비결정적 파싱을 구현하기 위해서는 파싱이 실패되는(fail) 상황을 표현하여야 한다. Haskell에서는 일반적으로 Maybe 타입을 이용하여 계산의 실패/성공 여부를 표현한다. Nothing은 실패를 의미하고, 성공적 계산에 의한 값 v는 (Just v)로 표현된다. 따라서 파싱의 실패 여부를 위해

newtype Parser a = P (String->Maybe (a,String))

의 타입을 정의할 수 있으나, [2]에서는 (Maybe (a,String)) 대신 리스트를 이용한 [(a,String)] 표현을 사용하고 있다. 실패를 [](empty list)로, 성공을 []이 아닌 모든 리스트로 표현한다. 이 타입을 바탕으로 한 모나드 정의는 그림 3과 같다.

```
instance Monad Parser where
 return v =  P (\inp -> [(v,inp)])
 p >>= f  =  P (\inp ->
        case parse p inp of
           []        -> []
           [(v,out)]  -> parse (f v) out)
```

그림 3. Parser 모나드의 정의
Fig. 3. Definition of Parser Monad

return은 값 v를 단지 Parser 모나드 형태로 투입한다. 바인드는 두 파서 p와 f를 연속적으로 수행하는데, 이 과정에서 한 번이라도 실패([])가 발생하면 두 바인드된 결과는 실패가 된다. 첫 번째 파서 p가 파싱을 성공하면 두 번째 파서 f는 앞서 파싱된 결과 v와 입력 스트링의 나머지 부분을 이어받아 두 번째 파싱을 수행한다.

III. While 언어의 문법 및 변환

1. While 언어의 Context-Free Grammar

While 언어는 C와 유사한 간단한 형태의 명령형 언어이다. 이 언어는 산술식, 관계 연산식, 할당문, 조건문 및 반복문의 기능을 포함하고 있다. 그림 4는 context-free grammar로 While 언어 문법을 표현한 것이다.

```
program     ::= stmntSeq
stmntSeq    ::= stmnt | stmnt ';' stmntSeq
stmnt       ::= identifier ":=" aExp
   | "skip"
   |"if" bool "then" stmntSeq "else" stmntSeq
   | "while" bool "do" stmntSeq
   | '{' stmntSeq '}'
aExp ::= term | aExp '+' aExp
               | aExp '-' aExp
term ::= factor | term * term  | term '/' term
factor ::= Number | Identifier | '(' aExp ')'
bExp ::= bTer | bExp "&&" bExp
bTer ::= "true" | "false" | aExp '=' aExp
        | aExp "<=" aExp | aExp '>' aExp
        | '!' bExp | '(' bExp ')'
```

그림 4. While 언어 문법
Fig. 4. Grammar of While Language

여기서 aExp는 산술식 연산, bExp는 Boolean식, bTer는 Boolean 식을 표현하기 위한 넌터미널이다. skip 문은 아무런 역할을 하지 않는 문장이다. if 문은 반드시 then과 else에 해당되는 문장이 모두 함께 갖추어져야 하므로, 만약 else 부분을 표현하고 싶지 않을 때는 skip 문을 사용한다. 예를 들어, if (x>0) then x := x-1 else skip 형태로 표현될 수 있다.

예를 들어, 이 언어로 4!을 계산하는 프로그램은 그림 5와 같이 코딩될 수 있다.

```
x := 4;
y :=1;
while !(x = 1) do {
  y := y * x;
  x := x -1
}
```

그림 5. While 언어의 팩토리얼 코딩 예
Fig. 5. Example of Factorial in While Language

2. EBNF로의 변환

Recursive-descent parsing 방식은 각 넌터미널에 대해서 이를 파싱하는 함수를 정의하는 것이다. 룰의 RHS (Right-Hand Side)가 $\alpha\beta$ 형태로 순서대로 되어있을 때, α가 터미널이면 이들을 입력 스트링과 매치시켜 매치의 성공 여부를 본다. 매치가 성공되면 α의 다음인 β를 가지고 파싱을 실행한다. 만약 도중에 파싱이 실패되면 이 룰로서는 더 이상의 파싱을 진행할 수 없으므로 다른 룰로서 파싱을 시도

해 봐야 한다. α가 넌터미널이면 α에 대한 함수가 정의되어 있을 것이므로 이 함수를 호출한다.

Recursive-descent parsing을 적용하기 위해서는 룰의 RHS가 프로그래밍하기에 적절한 모습을 가져야 한다. 먼저 재귀적으로 정의된 룰은 right recursive 모습을 가져야 한다. 또한, aExp ::= aExp '+' aExp | aExp '-' aExp 과 같이 RHS에 aExp이 중복되어 표현되는 경우 이를 factoring시켜 코드의 중복을 방지한다. 그림 4의 문법을 right recursive 형태로 바꾸고 factoring하여, 반복을 의미하는 { }, 옵션을 의미하는 [], 여러 개 중에 하나를 선택하는 ( ) 기호를 사용하는 EBNF로 표현하면 다음 그림 6과 같다.

```
program  ::= stmntSeq
stmntSeq  ::= stmnt { ';' stmnt}
stmnt          ::=  identifier ":=" aExp
  | "skip"
  | "if" bool "then" stmntSeq "else" stmntSeq
  | "while" bool "do" stmntSeq
  | '{' stmntSeq '}'
aExp ::= term ('+' aExp | '-' aExp | empty)
term ::= factor ('*' term | '/' term | empty)
factor ::= '(' aExp ')' | number | identifier
bExp ::= bTer ("&&" bExp | empty)
bTer ::= "true" | "false" | aExp '=' aExp
       | aExp "<=" aExp | aExp '>' aExp
       | '!' bExp | '(' bExp ')'
```

그림 6. While 언어 문법에 대한 EBNF
Fig. 6. EBNF for While Language

IV. 함수형 언어의 의한 파싱

1. 함수 언어의 특징

함수형 프로그래밍은 스트링 처리에 우수한 장점을 갖는다. 배열이 아닌 리스트로서 스트링을 표현함으로써 이를 다루기가 편리하고, 파싱된 트리구조를 대수적 타입으로서 쉽게 표현할 수 있다. 또한 함수형 언어의 함수는 고차 함수(higher-order functions)이면서 일등급(first-class) 함수이다. 이것은 함수를 마치 수처럼 다른 함수의 인수로 전달할 수 있고, 함수를 리스트 등에 저장할 수 있음을 의미한다. 이 특징에 따라 섬세한 기능의 함수를 정의할 수 있으며 코드 재사용을 극대화함으로써 코드의 양을 줄일 수 있다.

고차 함수에서는 "함수가 인수에 적용된다(apply)"는 표현을 사용한다. 예를 들어, Haskell의 식 add 1 2 은 괄호가 생략된 것으로 괄호를 모두 복원하면 ((add 1) 2)가 된다. (add 1)은 add 함수가 1에 적용된 결과로서, 그 자체가 함수이며 이것을 다시 2에 적용시키려면 ((add 1) 2)라고 표현한다.

2. 대수적 타입에 의한 추상구문트리 표현

파서의 구현 과정에서는 파싱 결과 얻어지는 추상구문트리를 표현해야 하는 경우가 많다. Haskell에서는 대수적 타입으로 추상구문트리를 쉽고 자연스럽게 표현할 수 있다. 추상구문 트리는 루트(root)에 함수, 자식 노드(child)에 그에 대한 인수를 표현하는 함수 구조를 갖는다. 대수적 타입의 기본 원리는 타입을 함수 구조로 표현하는 것으로서 이 타입을 갖는 구문은 자연스럽게 추상구문트리와 연관될 수 있다. 대수적 타입 선언의 예로서, 산술식에 대한 추상구문트리를 표현하는 타입 Aexp는 그림 7과 같이 정의될 수 있다.

```
data Aexp = N Int | V String
      | Add Aexp Aexp | Sub Aexp Aexp
      | Mult Aexp Aexp | Div Aexp Aexp
```

그림 7. Aexp 타입: 산술식에 대한 대수적 타입
Fig. 7. Aexp: Algebraic Type for Arithmetic Expression

Aexp는 새로 선언되는 타입의 이름이며 그 타입에 대한 정의는 RHS에 6가지 경우로 표현된다. 즉, 산술식은 정수(N Int), 변수(V String), 두 산술식에 대한 더하기(Add), 빼기(Sub), 곱하기(Mult), 나누기(Div)가 될 수 있다. 타입 정의의 RHS는 반드시 데이터 구성자(data constructor)로 시작하고 그 뒤에 타입이 나오게 된다. Aexp를 정수(즉, Int 타입)로 표현할 경우라도 그 앞에 데이터 구성자(여기서는 N)를 지정해 주어야 한다. 이 데이터 구성자는 타입을 인수로 갖는 함수이다. N은 Int 타입의 인수를 갖는 함수이고, Add는 Aexp 타입 두 개를 인수로 갖는 이진 함수이다. 여기서 타입 Aexp 정의에 다시 Aexp가 사용되는 재귀적으로 표현이 적용됨을 볼 수 있다. 이와 같이 대수적 타입은 변수를 사용하고 재귀적인 표현을 할 수 있다. 식 "1+(2-3)" 파싱한 추상구문트리는 Aexp 타입을 이용하여 (Add (N 1) (Sub (N 2) (N 3)))로 자연스럽게 표현된다.

3. 비결정적 파서의 구현

EBNF로 표현된 생성 룰은 recursive-descent parser의 원리에 따라 파서로 구현된다. 생성 룰의 비결정성은 모나드를 이용하여 그대로 표현될 수 있다. 예를 들어, 산술식에 대한 생성 룰을 고려해 보자.

aExp ::= term ('+' aExp | '-' aExp | empty)

이 룰을 구현한 aExp는 먼저 term을 파싱하고 그대로 끝

나든가(empty의 경우), 아니면 계속하여 '+' 나 '-' 둘 중에 하나를 파싱한 다음 다시 aExp를 파싱한다. 파싱 결과가 추상구문트리라면 (aExp :: Parser Aexp)의 타입으로 표현된다. 이 파서는 모나드를 이용하여 다음과 같이 코딩될 수 있다.

```
aExp :: Parser Aexp
aExp = do
  { t <- term;
    do {symbol "+"; e <- aExp; return (Add t e)}
    +++
    do {symbol "-"; e <- aExp; return (Sub t e)}
    +++
    return t
  }
```

symbol과 (+++)는 [2]에서 정의된 함수로서, symbol은 주어진 스트링을 파싱하고, (+++)는 두 액션(여기서는 파싱) 중에 하나를 수행하는 기능을 한다. 예를 들어 A +++ B는 먼저 액션 A를 수행하여 수행이 성공되면 그대로 종료되며, A의 수행이 실패한 경우 B를 수행하게 된다. aExp는 term을 파싱한 후, '+'로 시작되는 스트링과 산술식을 파싱하다가 실패하면, '-'로 시작되는 스트링과 산술식을 파싱해 보고, 그것 역시 실패하면 앞서 term의 파싱 결과를 출력한다.

```
program :: Parser Stm
program = stmntSeq

stmntSeq :: Parser Stm
stmntSeq = do{ stm <- stmnt
             ; do{ symbol ";"
                 ; stms <- stmntSeq
                 ; return (Comp stm stms)
                 }
             +++ return stm
             }
stmnt :: Parser Stm
stmnt = do{ stmBlock +++ assignStm
          +++ skipStm +++ ifStm +++ whileStm}
assignStm :: Parser Stm
assignStm = do{ idfr <- identifier
              ; symbol ":="
              ; e <- aExp
              ; return (Ass idfr e)
              }
skipStm :: Parser Stm
skipStm = do{ symbol "skip"; return Skip }

ifStm :: Parser Stm
```

```
ifStm = do{ symbol "if"
          ; be <- bExp
          ; symbol "then"
          ; stmseq1 <- stmntSeq
          ; symbol "else"
          ; stmseq2 <- stmntSeq
          ; return (If be stmseq1 stmseq2)
          }
whileStm :: Parser Stm
whileStm = do{ symbol "while"
             ; be <- bExp
             ; symbol "do"
             ; stmseq <- stmntSeq
             ; return (While be stmseq)
             }
stmBlock :: Parser Stm
stmBlock = do{ symbol "{"
             ; stms <- stmntSeq
             ; symbol "}"
             ; return stms
             }
```

그림 8. 모나드 파서 코딩
Fig. 8. Coding Monadic Parser

그림 8은 그림 6의 생성 룰 중에서 문장에 대한 부분을 모나드 파서로 코딩한 것이다. 이 파서는 프로그램 코드(스트링)를 입력받아 파싱하고, 결과로서 Stm 형태의 추상구문트리를 출력한다. 예를 들어, 할당문(assignStm)은 변수(identifier), 스트링 ":=", 산술식(aExp)를 파싱하고 이 파싱의 결과 얻어지는 idfr과 e를 이용하여 Stm 타입의 트리 (Ass idfr e)를 만들어 출력한다. 다른 문장들도 유사한 방법으로 코딩됨을 볼 수 있다.

4. parse 함수

함수 parse는 [2]에 정의되어 있다.

```
parse :: Parser a -> String -> [(a,String)]
parse (P p) inp  = p inp
```

parse는 한 파서와 입력 스트링을 인수로 받아서 파서를 입력 스트링에 적용시키는 기능을 한다. 예를 들어, parse aExp "1+2=3" 을 수행하면, 산술식 "1+2"는 aExp로 파싱되어 (Add (N 1) (N 2))를 얻지만, 기호 =는 aExp에 의해서는 파싱될 수 없으므로 "=3"은 파싱되지 않은 채 남아 파싱의 결과는 [(Add (N 1) (N 2), "=3")] 이 된다. While 언어의 시작 넌터미널은 program이므로 프로그램 코드(스트링) progCode를 파싱할 때는 (parse program progCode)를 수행하면 된다.

V. 코드 생성

1. Stack Machine과 Stack-Assembly

컴파일러에 의해서 생성되는 목적코드와 실행환경은 컴파일러 교재[3, Chapter 2]에 소개된 스택 머신에서 동작하는 Stack-Assembly 언어이다. 이 추상 머신은 어셈블리 코드를 저장하는 메모리, 이들을 수행하면서 중간 결과를 저장하고 연산하기 위한 연산 스택(evaluation stack), 그리고 변수와 그에 바인딩 된 값을 저장하는 심볼 테이블(symbol table)로서 구성되어 있다. 모든 연산은 스택에서 진행되며 스택 연산은 메모리의 top에서 이루어지므로 메모리의 주소(address) 기능을 가질 필요가 없어 동작 원리가 매우 간결한 특징이 있다.

이 스택 어셈블리 언어용 인터프리터는 이미 앞선 연구에서 구현되었으며[4], 여기서는 코드 생성과 관련된 주요 내용만을 간략히 소개한다. 그림 9는 어셈블리 명령어의 일부분과 스택, 심볼 테이블(Symtab)의 실행 환경을 정의한 내용이다.

```
type Prog = [Code]
data Code = Push Int | Plus | Mult | Gt
      | Rvalue Name | Lvalue Name | Assgn
      | Lab Int | Goto Int | GoFalse Int
type Stack = [Int]
type Symtab = [(String,Value)]
```

그림 9. Haskell로 표현된 Stack-Assembly 언어
Fig. 9. Haskell Definition of Stack Assembly Language

구현의 편의를 위해 False 대신 0, True 대신 1로 표현하며, Gt를 구현하는 연산자 >는 (> :: Int -> Int -> Int)의 타입을 갖는다. 모든 연산자들의 계산은 스택에서 발생하는데, 스택 연산을 할 때 묵시적으로 스택에서 Pop이 발생한다.

2. 실행 환경 및 코드 생성

(1) 후 연산(Postfix) 표현

사칙연산이나 관계연산 식을 스택 어셈블리 언어로 변환하기 위해서는 이들을 후 연산 형태의 모습으로 바꿔야 한다. 예를 들어, 산술식 (1 + 2) * 3 은 후 연산 식 1 2 + 3 * 으로 표현되며, 이들은 [Push 1, Push 2, Plus, Push 3, Mult] 로서 번역된다.

(2) 분기(Branch) 명령어 위한 라벨 생성

조건에 따라 어떤 특정 지역으로 jump를 수행하는 명령어로서 Goto와 GoFalse가 사용된다. 또한, Jump를 위해서는 특정 위치에 대한 라벨 지정이 필수적이므로 이를 위해 (Lab Int) 명령어를 사용한다. 이 라벨들은 서로 구분되어야 하므로 자연수를 이용하여 (Lab 1), (Lab 2) 등으로 표현되며, 새로운 라벨 번호를 얻기 위해 카운터(counter) 기능을 상태 모나드와 tick 함수로서 구현하였다[4].

(3) Symbol Table의 구현

명령형 언어에서 변수의 값은 동적으로 변화하며, 이 개념의 구현을 위해서 심볼테이블(Symtab으로 표현)이 필요하게 된다. 세 명령어 Lvalue, Rvalue, Assgn이 변수를 위해 사용된다. 변수는 할당문의 왼쪽에 나오는 경우와 그 이외의 경우에서의 의미가 다르다. 전자를 위해 Lvalue, 후자를 위해 Rvalue, 그리고 변수의 값을 할당하거나 업데이트하기 위해 Assgn 명령어가 사용된다. Rvalue는 Symtab에서 주어진 변수에 대한 값을 읽어 오고, Assgn는 주어진 변수와 값을 Symtab에 새로 등록하거나 이미 존재하는 경우 그 값을 새 값으로 업데이트한다.

(4) 코드 생성 함수 s2a

s2a는 추상구문트리의 문장을 Stack-Assembly로 번역하는 기능을 한다.

```
s2a :: Stm -> State Int String
s2a (Ass v aex)  = return $ " lvalue " ++ v
s2a (While be stm)
= do { test <- tick
     ; let str1 = " label " ++ show test
     ; let str2 = bexp2ass be  -- test codes
     ; out <- tick
     ; let str3 =  " gofalse " ++ show out
     ; str4 <- s2a stm     -- body of while
     ; let str5 = " goto " ++ show test
     ; let str6 = " label " ++ show out
          -- Label for the next of while stmnt
 : return (str1++str2++str3++str4++str5++str6)
}
```

그림 10. 코드 생성 함수
Fig. 10. A Function for Code Generation

그림 10은 그 중에서 할당문과 while 문의 번역 과정을 보여주고 있다. 그림 5에 소개된 4!을 계산하는 While 프로그램 예는 그림 11과 같은 Stack-Assembly 명령어로 번역된다.

1	Lvalue "x"	13	Lvalue "y"
2	Push 4	14	Rvalue "y"
3	Assgn	15	Rvalue "x"
4	Lvalue "y"	16	Mult
5	Push 1	17	Assgn
6	Assgn	18	Lvalue x
7	Label 0	19	Rvalue "x"
8	Rvalue "x"	20	Push 1
9	Push 1	21	Minus
10	Eq	22	Assgn
11	Not	23	Goto 0
12	GoFalse 1	24	Label 1

그림 11. 4! 프로그램의 코드 생성 예
Fig. 11. Example of Code Generation of 4! Program Code

그림 11의 1~3은 x := 4 문장, 4~6은 y :=1 문장, 7~23은 while 문장을 번역한 결과이다. 24의 Label은 While의 다음 문장 시작 위치를 표현하기 위해 사용된다. While 문장의 번역은 시작과 끝 부분에 Label이 지정되는데, 여기서는 각각 Label 0와 Label 1이 지정되었다. While 문은 조건이 만족되는 동안 프로그램이 반복 수행되도록 Goto 0가 수행되며, 조건을 만족하지 않을 때는 While 문에서 벗어나기 위해 GoFalse 1을 수행하게 된다. 이 번역 또한 그림 10의 s2a에 표현되어 있다.

3. 테스트 및 구현 결과 분석

약 20개 정도의 테스트 프로그램을 만들어서 개발된 컴파일러를 테스트하였다. 파싱 부분만을 따로 테스트하기 위해서 코드 생성 부분을 분리하여 파싱 부분만을 인터프리터 방식으로 테스트하는 방법을 적용하였다. 테스트와 수정을 거치면서 컴파일러는 정상적으로 작동하고 있다.

본 연구의 주목적은 모나드에 의한 비결정적 파싱 방법이 얼마나 쉽고 편리하게 코딩될 수 있는지를 구현을 통해서 실험해 보는 데 있다. 강한 이론적 기반에 따라 추상적 수준(abstraction level)이 높은 프로그래밍을 할 수 있는 Haskell은 C 등의 기존 언어로 프로그래밍 할 때보다 코드의 양을 1/20 ~ 1/30 수준으로 줄일 수 있다는 것이 일반적 관측이다.

이 연구를 통해 모나드 파서의 강력한 기능과 우수성을 확인할 수 있었다. 이 구현에서는 Hutton이 개발한 라이브러리[2] 약 120 라인의 코드를 사용하였고, 자체 개발한 컴파일러 코드의 양은 약 250 라인이며, 그 이외의 도구는 전혀 사용되지 않았다. 컴파일러 개발 환경이 다르므로 타 언어로 개발된 코드와의 비교에는 한계가 있으나 개발된 코드의 양이 획기적으로 적은 것은 확실하다고 할 수 있다. 특히, 까다로운 backtracking 형태의 프로그램을 간결하면서도 용이하게 코딩할 수 있고, 코딩된 프로그램을 쉽게 이해하도록 높은 가독성을 보여줌으로써 모나드 파싱 방식의 우수성과 타당성을 확인할 수 있다.

VI. 결 론

본 연구에서는 모나드 파싱 방식의 컴파일러 구현을 수행하였다. 파서 모나드는 컴파일러의 이론적 개념을 직접적이고 편리하게 구현할 수 있는 환경을 제공하고 있다. 프로그램 코드의 모습은 생성 룰과 매우 유사하므로 프로그램을 이해하기가 수월하다. 추상구문트리는 함수형 언어의 대수적 타입으로 쉽게 표현될 수 있으며, 스트링 처리 또한 리스트 및 관련 함수를 이용하여 간결하게 코딩할 수 있다.

컴파일러 파싱은 스트링의 패턴을 분석하고 처리하는 프로그래밍 기법이다. 이 기술은 컴파일러뿐만 아니라 스트링을 처리하는 여러 응용 분야에 적용할 수 있다. 스트링 처리에 모나드 파싱을 적용하면 더욱 쉽고 편리한 프로그래밍이 가능하여 소프트웨어 개발의 생산성을 높일 수 있을 것으로 사료된다.

스택 머신을 기반으로 한 어셈블리 언어용 코드 생성은 컴파일러 교육을 위해 사용될 수 있다. 실제 이 부분은 컴파일러 교재에 소개된 개념을 구현한 것으로서 교육용 실습 도구로서 사용되고 있다.

Haskell은 많은 우수한 이론적 배경을 가지고 있으나 실용적으로는 아직까지 널리 사용되고 있지 못하며, 특히 국내에서의 관심은 저조한 편이다. 본 연구를 통해 Haskell 모나드 프로그래밍의 우수성을 확인할 수 있었으며, 국내 모나드 프로그래밍 연구에 일조할 수 있기를 바란다.

참고문헌

[1] Graham Hutton and Erik Meijer, "Monadic Parser Combinators," Journal of Functional Programming, Vol. 8, Number 4, pp 437-444, Cambridge University Press, July 1998.

[2] Graham Hutton, "Programming in Haskell," Cambridge University Press, 2007.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, "*Compilers: Principles, Techniques, and Tools*," (2nd edition) Addison

Wesley, September 2006.
〔4〕 Sugwoo Byun, "Development of an Assembly Language Interpreter Using Monad," Software and Applications: Journal of Korean Institute of Information Scientists and Engineers, vol. 27, Number 5, pp. 403-410, May 2010.
〔5〕 Kiyoung Ahn and Jeonghoon Park, "*Learing Functional Programming in Haskell,*" Daelim-Press, 2009. (Korean-translated version of 〔2〕).
〔6〕 Simon Thompson, "*Haskell: The Craft of Functional Programming,*" (third edition), Addison Wesley Professional, 2011.
〔7〕 Haskell Homepage. http://haskell.org.
〔8〕 Commercial Users of Functional Prog., http://cufp.org.
〔9〕 Eugin Moggi, "Computational lambda-calculus and monads," IEEE Symposium on Logic in Computer Science, June 1989.
〔10〕 Philip Wadler, "Comprehending Monads," Proceedings of the 1990 ACM Conference on *Lisp and Functional Programming Languages*. 1990.
〔11〕 Haskell Understanding Monads, (http://en.wikibooks.org/wiki/Haskell/Understanding_monads)
〔12〕 Scala Homepage, http://www.scala-lang.org

저 자 소 개

변 석 우
1980: 숭실대학교 전자계산학과 공학사.
1982: 숭실대학교 전자계산학과 공학석사
1982~1999: ETRI 책임연구원
1995: University of East Anglia Computer Science, Ph.D.
현 재: 경성대학교 컴퓨터공학부 교수
관심분야: 함수형 프로그래밍, 정형 증명, 프로그래밍 언어
Email: swbyun@ks.ac.kr