UOU Report Vol. 16. No. 2. pp. 211~217. 1985 연구논문집 제16권 제 2 호 pp. 211~217. 1985

An Efficient Compiler Construction By Attribute Grammars

Koh, Jae-Jin
Dept. of Computer Science
(Received April 30, 1985)

〈요 약〉

컴파일러 구성에 있어서의 속성문법의 효율성을 산술식에 대한 속성문법을 작성해서 실험을 함으로서 연구하였다. 산술식을 문법적으로 기술하는 데에 문맥자유문법을 사용했고, 산술식을 문법적으로 분석하는 데에 LR Parsing 방법을 사용했다.

문법분석과 중간 코-드 생성이 SLR Parsing 표와 속성문법을 사용해서 동시에 수행되도록 하였다. 변수, 상수, 임시변수를 위한 Symbol Table은 선형적으로 찾아지고, 중간코-드 생성을 위해 3-번지 코-드를 사용했다. 선언문, 제어문과 같은 다른 문들도, 산술식에 대한 경험 위에서 쉽게 처리될 수 있다.

屬性文法에 의한 效率的인 컴파일러 構成

高 在 鎭
전 자 계 산 학 과
(1985. 4. 30 접수)

〈**Abstract**〉

The efficiency of attribute grammars in compiler construction is studied on the basis of the experiment on the attribute grammars for arithmetic expressions. Context free grammar is used for describing arithmetic expressions and LR parsing method is used for parsing arithmetic expressions. Parsing and intermediate code generation are conducted simultaneously using SLR parsing table and attribute grammars.

The symbol table for variable, constants, temporary values is linearly searched and the three address codes are used for the intermediate code generation. The other statements such as declaration statement, control statement, and input-output statement can be easily implemented on the experiences of arithmetic expressions.

Ⅰ. Introduction

Attribute grammars were used to describe the semantics of programming languages.

Nowadays they have been used as a specification language of compilers.

In this paper I studied the usability of attribute grammars as a practical tool for compiler construction. The study is based on attribute grammers written for arithmetic expressions and SLR parsing table written for them.

This study was influenced by the compiler construction using attribute grammars written for Pascal and LALR(1) parsing method [Kosk82]

1. Compiler

A translator is a program that takes as

* 이 논문은 1984년도 문교부 학연구조성비에 의하여 연구되었음.

input a program written in one programming language (the source language) and produces as output a program in another language (the object or target language).

A compiler is the translator of which the source language is a high-level language such as FORTRAN, COBOL, or PASCAL, and of which the object language is a machine language.

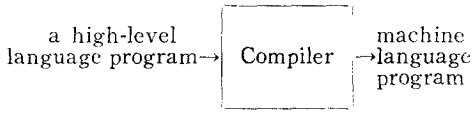

Fig. 1. Compilation

A compiler consists of a series of compilation subprocesses called phases, as shown in Fig. 2.

A phase is a logically cohesive operation that takes as input one representation of the source program and produces as output another representation.

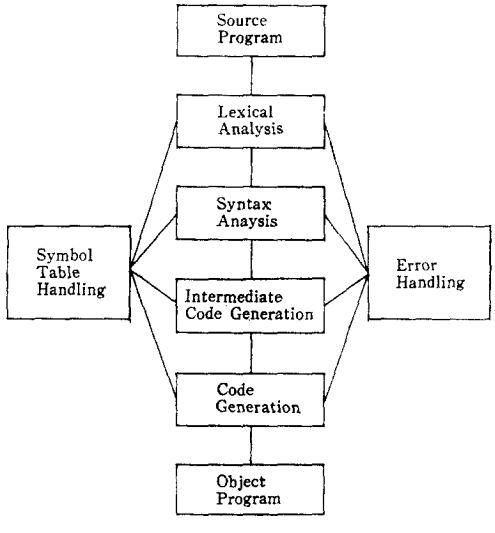

Fig. 2. Phases of a Compiler

The lexical analysis phase separates characters of the source language into groups that logically belong together. These groups are called tokens. The tokens are keywords, such as DO or DIMENSION, identifiers, such as AA or NN, operator symbos such as +or−, and punctuation symbols such as parentheses or commas.

The output of lexical analysis phase is a stream of tokens. The syntax analysis phase groups tokens together into syntactic structures. The syntactic structures are statements, expressions, etc. The syntactic structure can be regarded as a tree whose leaves are the tokens.

The intermediate code generation phase creates a stream of simple instructions from the syntactic structure. The code genaration phase produces the object code by deciding on the memory locations for data, selecting code to access each datum, and selecting the registers in which each computation is to be done. The symbol table handling keeps track of the names used by the program and records essential information about each, such as its type. The error handling routine is invoked when a flaw in the source program is detected.

In this paper our concerns are limited to the arithmetic expressions as input language, and are limited to the syntax analysis phase and intermediate code generation phase as the phases of the compiler.

2. Context Free Grammar

A context free grammar is a grammar to define programming language contructs recursively [Aho77]. A context free grammar involves four quantities.

They are terminals, nonterminals, a start symbol, and productions. Terminals are tokens such as identifier, keyword, operator, and punctuation in the programming language. Nonterminals are syntactic categories such as statement, expression, term, factor, primary. Nonterminals help provide a hierarchical structure for the language.

The start symbol is a special nonterminal and denotes the language. The productions

define the ways in which the nonterminals may be built up from one another and from the terminals. Each production consists of a nonterminal, followed by an arrow, followed by a string of nonterminals and terminals.

(Context Free Grammar For Arithmetic Expressions)

Terminals: +, −, ×, /, ↑, (,), *V*, *N*
where ↑ denotes exponentiation operator,
V denotes identifier,
N denotes constant.

Nonterminals: ⟨EX⟩, ⟨TM⟩, ⟨FC⟩, ⟨PM⟩
where ⟨EX⟩ denotes expression
⟨TM⟩ denotes term
⟨EC⟩ denotes factor
⟨PM⟩ denotes primary

Start symbol: ⟨EX⟩

Productions:

(1) ⟨EX⟩→+⟨TM⟩
(2) ⟨EX⟩→−⟨TM⟩
(3) ⟨EX⟩→⟨EX⟩+⟨TM⟩
(4) ⟨EX⟩→⟨EX⟩−⟨TM⟩
(5) ⟨EX⟩→⟨TM⟩
(6) ⟨TM⟩→⟨TM⟩*⟨FC⟩
(7) ⟨TM⟩→⟨TM⟩/⟨FC⟩
(8) ⟨TM⟩→⟨FC⟩
(9) ⟨FC⟩→⟨FC⟩↑⟨PM⟩
(10) ⟨FC⟩→⟨PM⟩
(11) ⟨PM⟩→(⟨EX⟩)
(12) ⟨PM⟩→*V*
(13) ⟨PM⟩→*N*

Fig. 3. Context Free Grammar for Arithmetic Expression

3. Attribute Grammar

The basic idea behind the attribute grammar concept is that value parts are associated with all nodes of a derivation tree, both terminal and nonterminal. The relationships between the input and output values are then expressed on a production-by-production basis in terms of the values appearing on the tree. A pair of braces and the action they enclose are treated as a single symbol called an action symbol.

In this paper we have seven action symbols, namely {PLUS}, {NEG}, {ADD}, {SUBT}, {MULT}, {DIV}, and {EXP}. We think of these action symbols as the names of routines which perform the enclosed action symbols. An attribute grammar is a context free grammar for which the following specifications are made[Lewi78]:

(1) Each input, nonterminal, and action symbol has an associated finite set of attribute, and each attribute has a set of permissible values.

(2) Each nonterminal and action symbol attribute is classified as being either inherited or synthesized.

(3) Rules for inherited attribute

(a) For each inherited attribute on the righthand side of the production, there is a rule which compute a value for that attribute as a function of other attribute occuring in the left-or righthand side of the production.

(b) An initial value is specified for each inherited attribute of the starting symbol.

(4) Rules for synthesized attribute

(a) For each synthesized nonterminal attribute on the lefthand side of the production, there is a rule which compute a value for that attribute as a function of other attribute occuring in the left-or righthand side of the production.

(b) For each synthesized action-symbol attribute, there is a rule which compute a value for that attribute as a function of other attributes of the action symbol.

An attributed grammar is called L-attributed if and only if the following three conditions hold:

(1) For each attribute-evaluation rule associated with an inherited attributed in the righthand side of the production, each argument of that rule is either an inherited

(S-attributed grammar for arithmetic expression)

⟨EX⟩x SYNTHESIZED x
⟨TM⟩x SXNTHESIZED x
⟨FC⟩x SYNTHESIZED x
{PLUS y, p} INHERITED y, p
{NEG y, p) INHERITED y, p
{ADD y, z, p} INHERITED y, z, p
{SUBT y, z, p} INHERITED y, z, p
{MULT y, z, p} INHERITED y, z, p
{DIV y, z, p} INHERITED y, z, p
{EXP y, z, p} INHERITED y, z, p

Starting Symbol: ⟨EX⟩

(1) ⟨EX⟩x→+⟨TM⟩r {PLUS y, p}
(x, p)←NEWT y←r
NEWT: A system procedure which supplies a pointer to some unused symbol table entry

(2) ⟨EX⟩x→−⟨TM⟩r {NEG y, p}
(x, p)←NEWT y←r

(3) ⟨EX⟩x→ ⟨EX⟩q+⟨TM⟩r {ADD y, z, p}
(x, p)←NEWT y←q z←r

(4) ⟨EX⟩x→⟨EX⟩q−⟨TM⟩r {SUBT y, z, p}
(x, p)←NEWT y←q z←r

(5) ⟨EX⟩x→⟨TM⟩p
x←p

(6) ⟨TM⟩x→⟨TM⟩q* ⟨FC⟩r {MULT y, z, p}
(x, p)←NEWT y←q z←r

(7) ⟨TM⟩x→⟨TM⟩q/⟨FC⟩r {DIV y, z, p}
(x, p)←NEWT y←q z←r

(8) ⟨TM⟩x→⟨FC⟩p
x→p

(9) ⟨FC⟩x→⟨FC⟩q ↑ ⟨PM⟩r {EXP y, z, p}
(x, p)←NEWP y←q z←r

(10) ⟨FC⟩x→⟨PM⟩p
x ← p

(11) ⟨PM⟩x→(⟨EX⟩p)
x ← p

(12) ⟨PM⟩x→V p
x ← p

(13) ⟨PM⟩x→N p
x ← p

Fig. 4. S-attributed Grammar for Arithmetic Expression (Continued)

attribute of the lefthand side or an arbitrary attribute of some righthand side symbol appearing to the left of the given symbol.

(2) For each attribute-evaluation rule associated with a synthesized attribute of the lefthand side of the production, each argument of that rule is either an inherited attribute of the given lefthand side or an arbitrary attribute of some righthand side symbol.

(3) For each attribute-evaluation rule associated with a synthesized attribute of an action symbol, each argument of that rule is an inherited attribute of the given action symbol.

An attributed grammar is S-attributed if and only if it is L-attributed and all the nonterminal attribute are synthesized.

Ⅱ. SLR Parsing Table Construction

An LR parser is depicted in Fig. 5. The parser has an input, a stack, and a parsing table. The input is read from left to right, one symbol at a time. The stack contains a string of the form $s0X1s2X2\ldots Xmsm$, where sm is on top. Each Xi is a grammar symbol which is either a nonterminal or a terminal and each si is a symbol called a state.

The parsing table consists of two parts, a parsing action function ACTION and a goto function GOTO. The LR parser behaves as follows. It determines sm, the state on top of the stack, and ai, the current input symbol. It then consults ACTION[sm, ai], the parsing action table entry for state sm and input ai.

The entry ACTION[sm, ai] can have one of four values:

(1) Shift s
(2) reduce A
(3) accept
(4) error

The function GOTO takes a state and grammar symbol as arguments and produces a state.

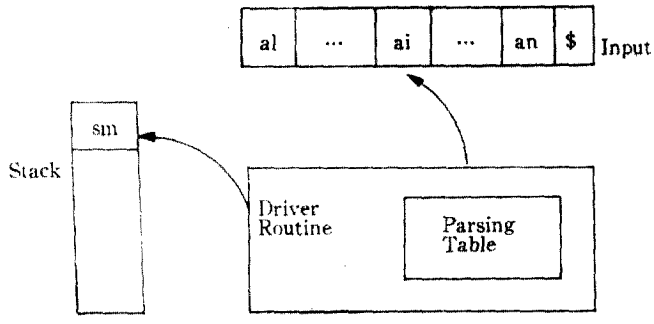

Fig. 5. LR Parser

A configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpended input: ($s0X1s1X2s2 \ldots Xmsm, a_{i+1} \ldots an$)

The next move of the parser is determined by reading *ai* and *sm*, and then consulting the ACTION[*sm*, *ai*].

I constructed the SLR parsing action and goto functions from the context free grammar for arithmetic expression in Fig. 3.

	ACTION										GOTO			
State	V	N	+	−	*	/	↑	(	)	$	⟨EX⟩	⟨TM⟩	⟨FC⟩	⟨PM⟩
0	s 8	s 9	s 2	s 3				s 7			1	4	5	6
1			s10	s11						acc				
2	s 8	s 9						s 7				12	5	6
3	s 8	s 9						s 7				12	5	6
4			r 5	r 5	s14	s15			r 5	r 5				
5			r 8	r 8	r 8	r 8	s16		r 8	r 8				
6			r10	r10	r10	r10	r10		r10	r10				
7	s 8	s 9	s 2	s 3				s 7			17	4	5	6
8			r12	r12	r12	r12	r12		r12	r12				
9			r13	r13	r13	r13	r13		r13	r13				
10	s 8	s 9						s 7				18	5	6
11	s 8	s 9						s 7				19	5	6
12			r 1	r 1	s14	s15			r 1	r 1				
13			r 2	r 2	s14	s15			r 2	r 2				
14	s 8	s 9						s 7					20	6
15	s 8	s 9						s 7					21	6
16	s 8	s 9						s 7						22
17			s10	s11					s23					
18			r 3	r 3	s14	s15			r 3	r 3				
19			r 4	r 4	s14	s15			r 4	r 4				
20			r 6	r 6	r 6	r 6	s16		r 6	r 6				
21			r 7	r 7	r 7	r 7	s16		r 7	r 7				
22			r 9	r 9	r 9	r 9	r 9		r 9	r 9				
23			r11	r11	r11	r11	r11		r11	r11				

Fig. 6. The SLR Parsing Table For Arithmetic Expression

The codes for the actions are:

(1) *si* means shift and stack state *i*.

(2) *rj* means reduce by production numbered *j*.

(3) acc means accept.

(4) blank means error.

The number *i* in the goto field means "go to state *i*"

The SLR parsing table for arithmetic expression is in Fig.6.

Ⅲ. Parsing and Intermediate Code Generation

Parsing and intermediate code generation are performed Concurrently using the parsing table in Fig.6 and using the S-attributed grammar in Fig.4.

The symbol table for variables, constants, and temporary values are searched linearly. The three-address codes are used for the intermediate code generation.

Three-address code is a sequence of statements, typically of the general form $A:=B$ op C, where A, B, and C are either programmer-defined names, constants or compiler-generated temporary names; op stands for any operator, such as a fixed-or floating-point arithmetic operator, or a logical operator on Boolean-valued data. For example, an expression like $X+Y*Z$ would be "unraveled" to yield

$$T1:=Y*Z$$
$$T2:=X+T1$$

Step	Stack	Input	Intermediate code
1	0	Vp1*Vp2+Vp3$	
2	0 Vp18	*Vp2+Vp3$	
3	0 ⟨PMp1⟩6	*Vp2+Vp3$	
4	0 ⟨FCp1⟩5	*Vp2+Vp3$	
5	0 ⟨TMp1⟩4	*Vp2+Vp3$	
6	0 ⟨TMp1⟩4*14	Vp2+Vp3$	
7	0 ⟨TMp1⟩4*14⟨Vp2⟩8	+Vp3$	
8	0 ⟨TMp1⟩4*14⟨PMp2⟩6	+Vp3$	
9	0 ⟨TMp1⟩4*14⟨FCp2⟩20	+Vp3$	
10	0 ⟨TMt1⟩4	+Vp3$	$T1:=P1*P2$
11	0 ⟨EXt1⟩1	+Vp3$	
12	0 ⟨EXt1⟩1+10	Vp3$	
13	0 ⟨EXt1⟩1+10⟨Vp3⟩8	$	
14	0 ⟨Ext1⟩1+10⟨PMp3⟩6	$	
15	0 ⟨EXt1⟩1+10⟨FCp3⟩5	$	
16	0 ⟨EXt1⟩1+10⟨TMp3⟩18	$	
17	0 ⟨EXt2⟩1	$	$T2:=T1+P3$
18	Accept		

Symbol Table

P1	
P2	
P3	
T1	
T2	

Fig.7. An example for parsing anpintermediate code generation

Where *T*1 and *T*2 are compiler-generated temporary names.

An example for parsing and intermediate code generation is in Fig. 7.

Ⅳ. Conclusion

I have constructed the SLR parsing table for arithmetic expression and tested the concurrent processing of parsing and intermediate code generation for sample input.

The study is based on the above work and I have studied attribute grammars as a practical tool for compiler construction.

A drawback of attribute grammars is the lack of adequate design methodologies and it is hard to understand attribute grammars.

A well-defined nonterminals can be the key points to make the attribute grammars readable. The performances of the compiler produced from the attribute grammars is reasonble and we can successfully apply it to the compilers of the micro or minicomputers.

References

[Aho77] A.V. Aho, J.D. Ullman, Principles of Compiler Design, Addison-Wesley, 1977.

[Kosk82] K. Koskimies, K.-J, Raiha, M. Sarjakoski, "Compiler Construction Using Attribute Grammars", Proc. ACM Symp. On Compiler Construction, June 1982.

[Lewi78] P. M. Lewis Ⅱ, Compiler Design Theory, Addison-wesley, 1978.