VISVESWARAYA TECHNOLOGICAL UNIVERSITY

BELGAUM

SRI JAYACHAMARAJENDRA COLLEGE OF ENGINEERING
MYSORE-570006
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Project on

Generation of three address code for 'C' subset(Part 2).

Guidance of
P.M.SHIVAMURTHY
Lecturer
Dept of CS&E,SJCE Mysore.

Team:

NAME	ROLL NO	USN
VIKRAM TV	61	4JC07CS120
PRABHAKAR GOUDA	37	4JC07CS070
SHARAD D	03	4JC06CS089

Contents

1 Overview

β - parse language is the beta version of parsing of our own language. It involves the best possibilities that we have seen from the classic language ‘C’. Our language is purely a subset of the higher level language - ’C’. The analysis phase of a compiler breaks up a source program into constituent pieces and produces an internal representation for it, called a intermediate code. Syntax-directed translation refers to a method of compiler implementation where the source language translation is completely driven by the parser. In other words, the parsing process and parse trees are used to direct semantic analysis and the translation of the source program. This can be a separate phase of a compiler or we can augment our conventional grammar with information to control the semantic analysis and translation. Such grammars are called attribute grammars.

2 Glossary

βparse	Our own defined language.
source program	User program written in βparse spec.
keywords	The keywords supported by our language.
Symbol Table	The data structures that are used by parser and lexer to hold information.
Lexer	Scanner which finds the token and return to the parser.
Parser	Which handles the token.
SDT	Syntax directed translation.
IR	Intermediate representation.

3 Flow diagram of the operation

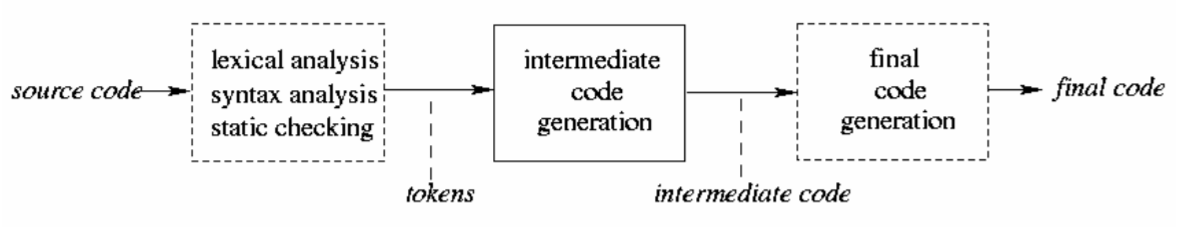

Figure 1: Operation

4 Syntax-Directed Definitions and Translation Schemes

When we associate semantic rules with productions, we use two notations:

4.1 Syntax-Directed Definitions

It give high-level specifications for translations. SDD hide many implementation details such as order of evaluation of semantic actions. We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.
When translating according to an SDT, it is normal not to construct the parse tree. Rather we perform actions as we parse; that is, we ”go through the motions” of building the parse tree. To perform a translation that cannot be expressed by an SDT directly, it helps to build the parse tree first. For example, while we can translate infix to postfix by an SDT, we cannot translate to prefix. Define one or more attributes of each terminal and non terminal, and give rules relating the values of these translations at the various nodes of the parse tree.

- Rules can be associated with productions, because they relate translations at a node to translations at its parent and or children.

- The rules for computing these translations together form a syntax-directed definition (SDD). Rules with no side effects (such as printing output) are called an attribute grammar.

5 Intermediate Representation

Most compilers translate the source program first to some form of intermediate representation and convert from there into machine code. The intermediate representation is a machine- and language-independent version of the original source code. Although converting the code twice introduces another step, use of an intermediate representation provides advantages in increased abstraction, cleaner separation between the front and back ends, and adds possibilities for re-targeting/cross-compilation. Intermediate representations also lend themselves to supporting advanced compiler optimizations and most optimization is done on this form of the code.

- An abstract machine language and independent of source language
- Expresses operations of target machine and not specific to any particular machine

6 Example

Consider the typical example,

```
Original              High IR             Mid IR            Low IR
float a[10][20];      t1 = a[i, j+2]      t1 = j + 2        r1 = [fp - 4]
a[i][j+2];                                t2 = i * 20       r2 = [r1 + 2]
                                          t3 = t1 + t2      r3 = [fp - 8]
                                          t4 = 4 * t3       r4 = r3 * 20
                                          t5 = addr a       r5 = r4 + r2
                                          t6 = t5 + t4      r6 = 4 * r5
                                          t7 = *t6          r7 = fp - 216
                                                            f1 = [r7 + r6]
```

Figure 2: Example

7 Three Address Code

Three-address code (TAC) will be the intermediate representation used in our Decaf compiler. It is essentially a generic assembly language that falls in the lower-end of the mid-level IRs. Some variant of 2, 3 or 4 address code is fairly commonly used as an IR, since it maps well to most assembly languages.

A TAC instruction can have at most three operands. The operands could be two operands to a binary arithmetic operator and the third the result location, or an operand to compare to zero and a second location to branch to, and so on. For example, below on the left is an arithmetic expression and on the right, is a translation into TAC instructions:

```
a = b * c + b * d;                  _t1 = b * c;
                                    _t2 = b * d;
                                    _t3 = _t1 + _t2;
                                    a = _t3;
```

Figure 3: Example three address code

8 Implementation of Translation Schemes

It indicate the order of evaluation of semantic actions associated with a production rule. In other words, translation schemes give a little bit information about implementation details.

8.1 Translation of Expressions

PRODUCTION	SEMANTIC RULES
S → ***id*** = *E;*	*S.code = E.code* *gen(top.get(**id**.lexeme) '=' E.addr)*
E → *E1 + E2*	*E.addr =* ***new*** *Temp()* *E.code = E1.code || E2.code ||* *gen(E.addr'='E1.addr '+' E2.addr*
E → *E1 - E2*	*E.addr =* ***new*** *Temp()* *E.code = E1.code || E2.code ||* *gen(E.addr'='E1.addr '-' E2.addr*
E → *E1 * E2*	*E.addr =* ***new*** *Temp()* *E.code = E1.code || E2.code ||* *gen(E.addr'='E1.addr '*' E2.addr*
E → *E1 / E2*	*E.addr =* ***new*** *Temp()* *E.code = E1.code || E2.code ||* *gen(E.addr'='E1.addr '/' E2.addr*
| *-E*	*E.addr =* ***new*** *Temp()* *E.code = E2.code ||* *E.addr ' ='**minus**E1.addr*
| *(E1)*	*E.addr = E1.addr* *E.code = E1.code*
| **id**	*E.addr = top.get(**id**),lexeme)* *E.code = ''*

Syntax directed definition in the above case builds up the three-address code for an assignment statement S using attribute *code* for S and *addr* and *code* for an expression E. Attribute *S.code* and *E.code* denotes the three address code for S and E respectively. Attribute *E.addr* denotes the address that will hold the value of E. Here, '*' and '/' have higher precedence than '+' and '-'.

Consider the last production *E*→ ***id*** in the above definition , the semantic rules for the this production define E.addr to point to the symble table entry for this instance if **id**. Let top denotes the current symble table. Function *top.get* retrieves the entry when it is applied to the string represents **id.**lexeme of this instance of **id**.

8.2 Flow of control statements

We assume that *newlabel()* creates a new label each time it is called, and that *L* attaches label L to the next three address instruction to be generated.

A program consists of a statement generated by *P*→ *S*. The semantic rules associated with this production initializes *S.next* to a new label. The token **assign** in the production *S*→***assign*** is a place holder for assignment statements.

In translating *S*→ ***if***(*B*) *S1*, the semantic rule create a new label *B.true* and attach it to the first three address instruction generated for the statement S1. Thus , jumps to B.true with in code code for B will go to code S1.

In the translating *S*→ ***if*** (*B*) *S1* ***else*** *S2*, the code for the boolean expression B jumps out of it to the first instruction of the code for S1 if B is true, and to the first instruction of the code for S2 if b is false.

PRODUCTIONS	**SEMANTIC RULES**
P →S	*S.next = newlabel()* *P.code = S.code‖ label(S.next)*
S → ***assign***	*S.code =* ***assign****.code*
S → ***if****(B)S1*	*B.true = newlabel()* *B.false = S1.next = S.next* *S.code = B.code ‖ label(B.true)‖ S1.code*
S → ***if****(B) S1* ***else*** *S2*	*B.true = newlabel()* *B.false = newlabel()* *S1.next = S2.next = S.next* *S.code = B.code* *‖label(B.true)‖S1.code* *gen ('**goto**' S.next)* *label(B.false) ‖ S2.code*
S → ***while****(B) S1*	*begin = newlabel()* *B.true = newlabel()* *B.false = s.next* *S1.next = begin* *S.code = label(begin) ‖ B.code* *‖ label(B.true) ‖ S1.code* *gen ('**goto**' begin)*
S → S1S2	*S1.next = newlabel()* *S2.next = S.next* *S.code = S1.code ‖ label(S1.next)‖ S2.code*

8.3 Control-Flow Translation of Boolean Expression

The semantic rules for Boolean expression shown below.
The Boolean expression B is translated into three address instruction that evaluate B using Condition and unconditional jumps to one of two labels: B.true if true and B.false if false.

PRODUCTION	**SEMANTIC RULES**
B → B1 ‖ B2	*B1.true = B.true* *B1.false = newlabel()* *B2.true = B.true* *B2.false = B.false* *B.code = B1.code ‖ label(B1.false) ‖ B2.code*
B → B1 && B2	*B1.true = newlabel()* *B1.false = B.false* *B2.true = B.true* *B2.false = B.false* *B.code = B1.code ‖ label(B1.true) ‖ B2.code*
B → !B1	*B1.true = B.false* *B1.false = B.true* *B.code = B1.code*
B → E1 ***rel*** *E2*	*B.code = E1.code ‖ E2.code* *‖gen('**if**' E1.addr **rel**.op E2.addr '**goto**' B.true)* *‖ gen('**goto**' B.false)*
B → ***true***	*B.code = gen('**goto**' B.true)*
*B →**false***	*B.code = gen ('**goto**' B.false)*

The remaining productions for B are translated as follows.

- Suppose B is the form B1‖B2. If B1 is true, then we immediately know that B itself is true, so B1.true.

- The translation of B1 & B2 is similar.

- No code is needed for expression B of the form !B1 just interchange the true and false exists of B to get the true and false exists of B1.

- the constants **true** and **false** translate into jumps to B.true and B.false, respectively.

The semantics for the expressions, control flow statements and boolean values in their context are suitably implemented.

9 References

- Lex and Yacc by John R. Levine, Tony Mason, Doug Brown - OReilly Publications
- A Compact Guide to Lex and Yacc by Tom Niemann at epaperpress.com
- Compilers Principles Techniques and Tools by Afred V.Aho, Monica S.lam, Ravi Sethi,Jeffrey D.Ullman
- Wikipaedia