A.V. Aho, R. Sethi, J.D. Ullman - "Compilers - Principles, Techniques, and Tools"
Appendix A - A Programming Project

A.1 INTRODUCTION

This appendix suggests programming exercises that can be used in a programming laboratory accompanying a compiler-design course based on this book. The exercises consist of implementing the basic components of a compiler for a subset of Pascal. The subset is minimal, but allows programs such as the recursive sorting procedure in Section 7.1 to be expressed. Being a subset of an existing language has certain utility. The meaning of programs in the subset is determined by the semantics of Pascal (Jensen and Wirth [1975]). If a Pascal compiler is available, it can be used as a check on the behavior of the compiler written as an exercise. The constructs in the subset appear in most programming languages, so corresponding exercises can be formulated using a different language if a Pascal compiler is not available.

A.2 PROGRAM STRUCTURE

A program consists of a sequence of global data declarations, a sequence of procedure and function declarations, and a single compound statement that is the "main program." Global data is to be allocated static storage. Data local to procedures and functions is allocated storage on a stack. Recursion is permitted, and parameters are passed by reference. The procedures read and write are assumed supplied by the compiler.

Fig. A.1 gives an example program. The name of the program is example, and input and output are the names of the files used by read and write, respectively.

```
program example(input, output);
var x, y: integer;
function gcd(a, b: integer): integer;
begin
if b = 0 then gcd := a
else gcd := gcd(b, a mod b) end;
begin
read(x, y);
write(gcd(x, y)) end.
```

Fig. A.1. Example program.

A.3 SYNTAX OF A PASCAL SUBSET

Listed below is an LALR(1) grammar for a subset of Pascal. The grammar can be modified for recursive-descent parsing by eliminating left recursion as described in Sections 2.4 and 4.3. An operator-precedence parser can be constructed for expressions by substituting out for **relop, addop,** and **mulu** and eliminating $\in$ -productions. The addition of the production

statement -> **if** *expression* **then** *statement*

introduces the "dangling-else" ambiguity, which can be eliminated as discussed in Section 4.3 (see also Example 4.19 if predictive parsing is used).

There is no syntactic distinction between a simple variable and the call of function without parameters. Both are generated by the production

factor -> **id**

Thus, the assignment a := b sets a to the value returned by the function b, if b has been declared to be a function.

program ->

program id (*identifier_list*) ;

declarations

subprogram_declarations

compound_statement

.

identifier_list ->

id

| *identifier-list* , **id**

declarations ->

declarations **var** *identifier-list* : *type* ;

| ∈

type ->

standard_type

| **array [num . . num] of** *standard_type*

standard_type ->

integer

| **real**

subprogram_declarations ->

subprogram_declarations subprogram_declaration ;

| ∈

subprogram_declaration ->

subprogram_head declarations compound_statement

subprogram_head ->

function id *arguments* : *standard_type* ;

| **procedure id** *arguments* ;

arguments ->

(*parameter-list*)

| ∈

parameter_list ->

identifier_list : *type*

| *parameter_list* ; *identifier_list* : *type*

compound_statement ->

begin

optional_statements

end

optional_statements ->

statement_list

| ∈

statement_list ->

statement

| *statement_list* ; *statement*

statement ->

variable **assignop** *expression*

| *procedure_statement*

| *compound-statement*

| **if** *expression* **then** *statement* **else** *statement*

| **while** *expression* **do** *statement*

variable ->

id

| **id** [*expression*]

procedure_statement ->

id

| **id** (*expression-list*)

expression_list ->

expression

| *expression_list* , *expression*

expression ->

simple_expression

| *simple_expression* **relop** *simple_expression*

simple_expression ->

term

| *sign term*

| *simple_expression* **sign** *term*

| *simple_expression* **or** *term*

term ->

factor

| *term* **mulop** *factor*

factor ->

variable

| **id** (*expression_list*)

| **num**

| (*expression*)

| **not** *factor*

A.4 LEXICAL CONVENTIONS

The notation for the specifying tokens is from Section 3.3.

1. Comments are surrounded by { and }. They may not contain a {. Comments may appear after any token.

2. Blanks between tokens are optional, with the exception that keywords must be surrounded by blanks, newlines, the beginning of the program, or the final dot.

3. Token **id** for identifiers matches a letter followed by letters or digits:

   letter -> [a-zA-Z]

   digit -> [0-9]

   id -> **letter** (**letter** | **digit**)*

The implementer may wish to put a limit on identifier length.

4. Token **num** matches unsigned numbers (see Example 3.5):

   digits -> **digit digit***

   optional_fraction -> . **digits** | $\in$

   optional_exponent -> (E (+ | - | $\in$) **digits**) | $\in$

   num -> **digits optional_fraction optional_exponent**

5.Keywords are reserved and appear in boldface in the grammar.

6. The relation operators (**relop's**) are: =, <>, <, <=, >=, and >. Note that <> denotes $\neq$.
7. The **sign** is + or -.
8. The **mulop's** are *, /, **div**, **mod**, and **and**.
9. The lexeme for token **assignop** is :=.
10. The lexeme for token **or** is or.

A.5 SUGGESTED EXERCISES

A programming exercise suitable for a one-term course is to write an interpreter for the language defined above, or for a similar subset of another high-level language. The project involves translating the source program into an intermediate representation such as quadruples or stack machine code and then interpreting the intermediate representation. We shall propose an order for the construction of the modules. The order is different from the order in which the modules are executed in the compiler because it is convenient to have a working interpreter to debug the other compiler components.

1. *Design a symbol-table mechanism.* Decide on the symbol-table organization. Allow for information to be collected about names, but leave the symbol-table record structure flexible at this time. Write routines to:

   1. Search the symbol table for a given name, create a new entry for that name if none is present, and in either case return a pointer to the record for that name.
   2. Delete from the symbol table all names local to a given procedure.

1. *Write an interpreter for quadruples.* The exact set of quadruples may be left open at this time but they should include the arithmetic and conditional jump statements corresponding to the set of operators in the language. Also include logical operations if conditions are evaluated arithmetically rather than by position in the program. In addition, expect to need "quadruples" for integer-to-real conversion, for marking the beginning and end of procedures, and for parameter passing and procedure calls.
   It is also necessary at this time to design the calling sequence and runtime organization for the programs being interpreted. The simple stack organization discussed in Section 7,3 is suitable for the example language, because no nested declarations of procedures are permitted in the language; that is, variables are either global (declared at the level of the entire program) or local to a simple procedure.
   For simplicity, another high-level language may be used in place of the interpreter. Each quadruple can be a statement of a high-level language such as C, or even Pascal. The output of the compiler is then a sequence of C statements that can be compiled on an existing C compiler. This approach enables the implementer to concentrate on the run-time organization.
2. *Write the lexical analyzer.* Select internal codes for the tokens. Decide how constants will be represented in the compiler. Count lines for later use by an error-message handler. Produce a listing of the source program if desired. Write a program to enter the reserved words into the symbol table. Design your lexical analyzer to be a subroutine called by the parser, returning a pair (token, attribute value). At present, errors detected by your lexical analyzer may be handled by calling an error-printing routine and halting.

3. *Write the semantic actions.* Write semantic routines to generate the quadruples. The grammar will need to be modified in places to make the translation easier. Consult Sections 5.5 and 5.6 for examples of how to modify the grammar usefully. Do semantic analysis at this time, converting integers to reals when necessary.
4. *Write the parser.* If an LALR parser generator is available, this will simplify the task considerably. If a parser generator handling ambiguous grammars, like Yacc, is available, then nonterminals denoting expressions can be combined. Moreover, the "dangling-else" ambiguity can be resolved by shifting whenever a shift/reduce conflict occurs.
5. *Write the error-handling routines.* Be prepared to recover from lexical and syntactic errors. Print error diagnostics for lexical, syntactic, and semantic errors.
6. *Evaluation.* The program in Fig. A.1 can serve as a simple test routine. Another test program can be based on the Pascal program in Fig. 7.1, The code for function partition in the figure corresponds to the marked fragment in the C program of Fig. 10.2. Run your compiler through a profiler, if one is available. Determine the routines in which most of the time is being spent. What modules would have to be modified in order to increase the speed of your compiler?

A.6 EVOLUTION OF THE INTERPRETER

An alternative approach to constructing an interpreter for the language is to start by implementing a desk calculator, that is, an interpreter for expressions;

Gradually add constructs to the language until an interpreter for the entire language is obtained. A similar approach is taken in Kernighan and Pike [1984]. A proposed order for adding constructs is:

1. *Translate expressions into postfix notation.* Using either recursive-descent parsing, as in Chapter 2, or a parser generator, familiarize yourself with the programming environment by writing a translator from simple arithmetic expressions into postfix notation.
2. *Add a lexical analyzer.* Allow for keywords, identifiers, and numbers to appear in the translator constructed above. Retarget the translator to produce either code for a stack machine or quadruples.
3. *Write an interpreter for the intermediate representation.* As discussed in Section A.5, a high-level language may be used in place of the interpreter. For the moment, the interpreter need only support arithmetic operations, assignments, and input-output. Extend the language by allowing global variable declarations, assignments, and calls of procedures read and write. These constructs allow the interpreter to be tested.
4. *Add statements.* A program in the language now consists of a main program without subprogram declarations. Test both the translator and the interpreter.
5. *Add procedures and functions.* The symbol table must now allow the scopes of identifiers to be limited to procedure bodies. Design a calling sequence. Again, the simple stack organization of Section 7.3 is adequate. Extend the interpreter to support the calling sequence.

A.7 EXTENSIONS

There are a number of features that can be added to the language without greatly increasing the complexity of compilation. Among these are:

1. multidimensional arrays
2. for- and case-statements
3. block structure
4. record structures

If time permits, add one or more of these extensions to your compiler.