4

Abstract Syntax

ab-stract: disassociated from any specific instance

Webster's Dictionary

A compiler must do more than recognize whether a sentence belongs to the language of a grammar – it must do something useful with that sentence. The *semantic actions* of a parser can do useful things with the phrases that are parsed.

In a recursive-descent parser, semantic action code is interspersed with the control flow of the parsing actions. In a parser specified in Yacc, semantic actions are fragments of C program code attached to grammar productions.

4.1 SEMANTIC ACTIONS

Each terminal and nonterminal may be associated with its own type of semantic value. For example, in a simple calculator using Grammar 3.35, the type associated with `exp` and `INT` might be `int`; the other tokens would not need to carry a value. The type associated with a token must, of course, match the type that the lexer returns with that token.

For a rule $A \rightarrow B\ C\ D$, the semantic action must return a value whose type is the one associated with the nonterminal A. But it can build this value from the values associated with the matched terminals and nonterminals B, C, D.

RECURSIVE DESCENT

In a recursive-descent parser, the semantic actions are the values returned by parsing functions, or the side effects of those functions, or both. For each terminal and nonterminal symbol, we associate a *type* (from the implementation

```
enum token {EOF, ID, NUM, PLUS, MINUS,  ··· };
union tokenval {string id; int num;  ···  };

enum token tok;
union tokenval tokval;

int lookup(String id) {  ···  }

int F_follow[] = { PLUS, TIMES, RPAREN, EOF, -1 };
int F(void) {switch (tok) {
         case ID:    {int i=lookup(tokval.id); advance(); return i;}
         case NUM:   {int i=tokval.num; advance(); return i;}
         case LPAREN:  eat(LPAREN); { int i = E();
                       eatOrSkipTo(RPAREN, F_follow);
                       return i; }
         case EOF:
         default:    printf("expected ID, NUM, or left-paren");
                     skipto(F_follow);
                     return 0;
        }}

int T_follow[] = { PLUS, RPAREN, EOF, -1 };
int T(void) {switch (tok) {
         case ID: case NUM: case LPAREN: return Tprime(F());
         default: printf("expected ID, NUM, or left-paren");
                  skipto(T_follow);
                  return 0;
        }}

int Tprime(int a)  {switch (tok) {
        case TIMES: eat(TIMES); return Tprime(a*F());
        case PLUS: case RPAREN: case EOF:  return a;
        default:  ···
       }}

void eatOrSkipTo(int expected, int *stop) {
   if (tok==expected) eat(expected);
   else {printf(···); skipto(stop);}
}
```

PROGRAM 4.1. Recursive-descent interpreter for part of Grammar 3.15.

```
%{ declarations of yylex and yyerror %}
%union {int num; string id;}
%token <num> INT
%token <id> ID
%type  <num> exp
%start exp

%left PLUS MINUS
%left TIMES
%left UMINUS
%%

exp :  INT                {$$ = $1;}
    |  exp PLUS exp       {$$ = $1 + $3;}
    |  exp MINUS exp      {$$ = $1 - $3;}
    |  exp TIMES exp      {$$ = $1 * $3;}
    |  MINUS exp    %prec UMINUS  {$$ = - $2;}
```

PROGRAM 4.2. Yacc version of Grammar 3.35.

language of the compiler) of *semantic values* representing phrases derived from that symbol.

Program 4.1 is a recursive-descent interpreter for Grammar 3.15. The tokens `ID` and `NUM` must now carry values of type `string` and `int`, respectively. We will assume there is a lookup table mapping identifiers to integers. The type associated with E, T, F, etc. is `int`, and the semantic actions are easy to implement.

The semantic action for an artificial symbol such as T' (introduced in the elimination of left recursion) is a bit tricky. Had the production been $T \rightarrow T * F$ then the semantic action would have been

```
{int a,b; a = T(); eat(TIMES); int b=F(); return a*b;}
```

With the rearrangement of the grammar, the production $T' \rightarrow *FT'$ is missing the left operand of the $*$. One solution is for T to pass the left operand as an argument to T', as shown in Program 4.1.

Yacc-GENERATED PARSERS

A parser specification for Yacc consists of a set of grammar rules, each annotated with a semantic action that is a C statement. Whenever the generated parser reduces by a rule, it will execute the corresponding semantic action fragment.

Stack					*Input*	*Action*
					1 + 2 * 3 $	*shift*
1 INT					+ 2 * 3 $	*reduce*
1 exp					+ 2 * 3 $	*shift*
1 exp	+				2 * 3 $	*shift*
1 exp	+	2 INT			* 3 $	*reduce*
1 exp	+	2 exp			* 3 $	*shift*
1 exp	+	2 exp	*		3 $	*shift*
1 exp	+	2 exp	*	3 INT	$	*reduce*
1 exp	+	2 exp	*	3 exp	$	*reduce*
1 exp	+	6 exp			$	*reduce*
7 exp					$	*accept*

FIGURE 4.3. Parsing with a semantic stack.

Program 4.2 shows how this works for Grammar 3.35. The semantic action can refer to the semantic values of the ith right-hand-side symbol as `$i`. It can produce a value for the left-hand-side nonterminal symbol by assigning to `$$`. The `%union` declaration declares different possible types for semantic values to carry; each terminal or nonterminal symbol declares which variant of the union it uses by means of the <*variant*> notation.

In a more realistic example, there might be several nonterminals each carrying a different type.

A Yacc-generated parser implements semantic values by keeping a stack of them parallel to the state stack. Where each symbol would be on a simple parsing stack, now there is a semantic value. When the parser performs a reduction, it must execute a C-language semantic action; it satisfies each reference to a right-hand-side semantic value by a reference to one of the top k elements of the stack (for a rule with k right-hand-side symbols). When the parser pops the top k elements from the symbol stack and pushes a nonterminal symbol, it also pops k from the semantic value stack and pushes the value obtained by executing the C semantic action code.

Figure 4.3 shows an LR parse of a string using Program 4.2. The stack holds states and semantic values (in this case, the semantic values are all integers). When a rule such as $E \rightarrow E + E$ is reduced (with a semantic action such as `exp1+exp2`), the top three elements of the semantic stack are `exp1`, empty (a place-holder for the trivial semantic value carried by +), and `exp2`, respectively.

AN INTERPRETER IN SEMANTIC ACTIONS

Program 4.2 shows how semantic values for nonterminals can be calculated from the semantic values of the right-hand side of the productions. The semantic actions in this example do not have *side effects* that change any global state, so the order of evaluation of the right-hand-side symbols does not matter.

However, an LR parser does perform reductions, and associated semantic actions, in a deterministic and predictable order: a bottom-up, left-to-right traversal of the parse tree. In other words, the (virtual) parse tree is traversed in *postorder*. Thus, one can write semantic actions with global side effects, and be able to predict the order of their occurrence.

Program 4.4 shows an interpreter for the straight-line program language. It uses a global variable for the symbol table (a production-quality interpreter would use a better data structure than a linked list; see Section 5.1).

4.2 ABSTRACT PARSE TREES

It is possible to write an entire compiler that fits within the semantic action phrases of a Yacc parser. However, such a compiler is difficult to read and maintain. And this approach constrains the compiler to analyze the program in exactly the order it is parsed.

To improve modularity, it is better to separate issues of syntax (parsing) from issues of semantics (type-checking and translation to machine code). One way to do this is for the parser to produce a *parse tree* – a data structure that later phases of the compiler can traverse. Technically, a parse tree has exactly one leaf for each token of the input and one internal node for each grammar rule reduced during the parse.

```
%{
typedef struct table *Table_;
Table_ {string id; int value; Table_ tail};
Table_  Table(string id, int value, struct table *tail);  (see page 13)
Table_ table=NULL;
int lookup(Table_ table, string id) {
    assert(table!=NULL);
    if (id==table.id) return table.value;
    else return lookup(table.tail, id);
}
void update(Table_ *tabptr, string id, int value) {
   *tabptr = Table(id, value, *tabptr);
}
%}
%union {int num; string id;}

%token <num> INT
%token <id> ID
%token ASSIGN | PRINT | LPAREN | RPAREN
%type <num> exp
%right SEMICOLON
%left PLUS MINUS
%left TIMES DIV
%start prog
%%
prog: stm

stm : stm SEMICOLON stm
stm : ID ASSIGN exp              {update(&table,ID,$3);}
stm : PRINT LPAREN exps RPAREN   {printf("\n");}

exps: exp                        {printf("%d ", $1);}
exps: exps COMMA exp             {printf("%d ", $3);}

exp : INT                        {$$=$1;}
exp : ID                         {$$=lookup(table,$1);}
exp : exp PLUS exp               {$$=$1+$3;}
exp : exp MINUS exp              {$$=$1-$3;}
exp : exp TIMES exp              {$$=$1*$3;}
exp : exp DIV exp                {$$=$1/$3;}
exp : stm COMMA exp              {$$=$3;}
exp : LPAREN exp RPAREN          {$$=$2;}
```

PROGRAM 4.4. An interpreter in imperative style.

$$\begin{array}{ll}
S \rightarrow S \,;\, S & L \rightarrow \\
S \rightarrow \texttt{id} \,\texttt{:=}\, E & L \rightarrow L\ E \\
S \rightarrow \texttt{print}\ L & \\
E \rightarrow \texttt{id} & B \rightarrow + \\
E \rightarrow \texttt{num} & B \rightarrow - \\
E \rightarrow E\ B\ E & B \rightarrow \times \\
E \rightarrow S\ ,\ E & B \rightarrow /
\end{array}$$

GRAMMAR 4.5. Abstract syntax of straight-line programs.

Such a parse tree, which we will call a *concrete parse tree* representing the *concrete syntax* of the source language, is inconvenient to use directly. Many of the punctuation tokens are redundant and convey no information – they are useful in the input string, but once the parse tree is built, the structure of the tree conveys the structuring information more conveniently.

Furthermore, the structure of the parse tree depends too much on the grammar! The grammar transformations shown in Chapter 3 – factoring, elimination of left recursion, elimination of ambiguity – involve the introduction of extra nonterminal symbols and extra grammar productions for technical purposes. These details should be confined to the parsing phase and should not clutter the semantic analysis.

An *abstract syntax* makes a clean interface between the parser and the later phases of a compiler (or, in fact, for the later phases of other kinds of program-analysis tools such as dependency analyzers). The abstract syntax tree conveys the phrase structure of the source program, with all parsing issues resolved but without any semantic interpretation.

Many early compilers did not use an abstract syntax data structure because early computers did not have enough memory to represent an entire compilation unit's syntax tree. Modern computers rarely have this problem. And many modern programming languages (ML, Modula-3, Java) allow forward reference to identifiers defined later in the same module; using an abstract syntax tree makes compilation easier for these languages. It may be that Pascal and C require clumsy *forward* declarations because their designers wanted to avoid an extra compiler pass on the machines of the 1970s.

Grammar 4.5 shows the abstract syntax of a straight-line-program language. This grammar is completely impractical for parsing: the grammar is quite ambiguous, since precedence of the operators is not specified, and many of the punctuation keywords are missing.

However, Grammar 4.5 is not meant for parsing. The parser uses the *concrete syntax* (Program 4.6) to build a parse tree for the *abstract syntax*. The semantic analysis phase takes this *abstract syntax tree*; it is not bothered by the ambiguity of the grammar, since it already has the parse tree!

The compiler will need to represent and manipulate abstract syntax trees as data structures. In C, these data structures are organized according to the principles outlined in Section 1.3: a `typedef` for each nonterminal, a union-variant for each production, and so on. Program 1.5 shows the data structure declarations for Grammar 4.5.

The Yacc (or recursive-descent) parser, parsing the *concrete syntax*, constructs the *abstract* syntax tree. This is shown in Program 4.6.

POSITIONS

In a one-pass compiler, lexical analysis, parsing, and semantic analysis (type-checking) are all done simultaneously. If there is a type error that must be reported to the user, the *current* position of the lexical analyzer is a reasonable approximation of the source position of the error. In such a compiler, the lexical analyzer keeps a "current position" global variable, and the error-message routine just prints the value of that variable with each message.

A compiler that uses abstract-syntax-tree data structures need not do all the parsing and semantic analysis in one pass. This makes life easier in many ways, but slightly complicates the production of semantic error messages. The lexer reaches the end of file before semantic analysis even begins; so if a semantic error is detected in traversing the abstract syntax tree, the *current* position of the lexer (at end of file) will not be useful in generating a line-number for the error message. Thus, the source-file position of each node of the abstract syntax tree must be remembered, in case that node turns out to contain a semantic error.

To remember positions accurately, the abstract-syntax data structures must be sprinkled with `pos` fields. These indicate the position, within the original source file, of the characters from which these abstract syntax structures were derived. Then the type-checker can produce useful error messages.

The lexer must pass the source-file positions of the beginning and end of each token to the parser. Ideally, the automatically generated parser should maintain a *position stack* along with the *semantic value* stack, so that the beginning and end positions of each token and phrase are available for the

```
%{
#include "absyn.h"
%}

%union {int num; string id; A_stm stm; A_exp exp; A_expList expList;}

%token <num> INT
%token <id> ID
%token ASSIGN | PRINT | LPAREN | RPAREN
%type <stm> stm prog
%type <exp> exp
%type <expList> exps
%left SEMICOLON
%left PLUS MINUS
%left TIMES DIV
%start prog
%%
prog: stm                        {$$=$1;}

stm : stm SEMICOLON stm          {$$=A_CompoundStm($1,$3);}
stm : ID ASSIGN exp              {$$=A_AssignStm($1,$3);}
stm : PRINT LPAREN exps RPAREN   {$$=A_PrintStm($3);}

exps: exp                        {$$=A_ExpList($1,NULL);}
exps: exp COMMA exps             {$$=A_ExpList($1,$3);}

exp : INT                        {$$=A_NumExp($1);}
exp : ID                         {$$=A_IdExp($1);}
exp : exp PLUS exp               {$$=A_OpExp($1,A_plus,$3);}
exp : exp MINUS exp              {$$=A_OpExp($1,A_minus,$3);}
exp : exp TIMES exp              {$$=A_OpExp($1,A_times,$3);}
exp : exp DIV exp                {$$=A_OpExp($1,A_div,$3);}
exp : stm COMMA exp              {$$=A_EseqExp($1,$3);}
exp : LPAREN exp RPAREN          {$$=$2;}
```

PROGRAM 4.6. Abstract-syntax builder for straight-line programs.

semantic actions to use in reporting error messages. The *Bison* parser generator can do this; *Yacc* does not. When using Yacc, one solution is to define a nonterminal symbol `pos` whose semantic value is a source location (line number, or line number and position within line). Then, if one wants to access the position of the `PLUS` from the semantic action after `exp PLUS exp`, the following works:

```
%{ extern A_OpExp(A_exp, A_binop, A_exp, position); %}
%union{int num; string id; position pos; ··· };
%type <pos> pos

pos :                    { $$ = EM_tokPos; }

exp :  exp PLUS pos exp { $$ = A_OpExp($1,A_plus,$4,$3); }
```

But this trick can be dangerous. With pos after PLUS, it works; but with pos too early in the production, it does not:

```
exp :  pos exp PLUS exp { $$ = A_OpExp($2,A_plus,$4,$1); }
```

This is because the LR(1) parser must reduce pos $\rightarrow \epsilon$ before seeing the PLUS. A shift-reduce or reduce-reduce conflict will result.

ABSTRACT SYNTAX FOR Tiger

Figure 4.7 shows for the abstract syntax of Tiger. The meaning of each constructor in the abstract syntax should be clear after a careful study of Appendix A, but there are a few points that merit explanation.

Figure 4.7 shows only the constructor functions, not the typedefs and structs. The definition of A_var would actually be written as

```
/* absyn.h */
typedef struct A_var_ *A_var;
struct A_var_
       {enum {A_simpleVar, A_fieldVar, A_subscriptVar} kind;
        A_pos pos;
        union {S_symbol simple;
               struct {A_var var;
                       S_symbol sym;} field;
               struct {A_var var;
                       A_exp exp;} subscript;
             } u;
     };
```

This follows the principles outlined on page 9.

The Tiger program

```
(a := 5; a+1)
```

translates into abstract syntax as

```
/* absyn.h */
A_var A_SimpleVar(A_pos pos, S_symbol sym);
A_var A_FieldVar(A_pos pos, A_var var, S_symbol sym);
A_var A_SubscriptVar(A_pos pos, A_var var, A_exp exp);

A_exp A_VarExp(A_pos pos, A_var var);
A_exp A_NilExp(A_pos pos);
A_exp A_IntExp(A_pos pos, int i);
A_exp A_StringExp(A_pos pos, string s);
A_exp A_CallExp(A_pos pos, S_symbol func, A_expList args);
A_exp A_OpExp(A_pos pos, A_oper oper, A_exp left, A_exp right);
A_exp A_RecordExp(A_pos pos, S_symbol typ, A_efieldList fields);
A_exp A_SeqExp(A_pos pos, A_expList seq);
A_exp A_AssignExp(A_pos pos, A_var var, A_exp exp);
A_exp A_IfExp(A_pos pos, A_exp test, A_exp then, A_exp elsee);
A_exp A_WhileExp(A_pos pos, A_exp test, A_exp body);
A_exp A_BreakExp(A_pos pos);
A_exp A_ForExp(A_pos pos, S_symbol var, A_exp lo, A_exp hi, A_exp body);
A_exp A_LetExp(A_pos pos, A_decList decs, A_exp body);
A_exp A_ArrayExp(A_pos pos, S_symbol typ, A_exp size, A_exp init);

A_dec A_FunctionDec(A_pos pos, A_fundecList function);
A_dec A_VarDec(A_pos pos, S_symbol var, S_symbol typ, A_exp init);
A_dec A_TypeDec(A_pos pos, A_nametyList type);

A_ty A_NameTy(A_pos pos, S_symbol name);
A_ty A_RecordTy(A_pos pos, A_fieldList record);
A_ty A_ArrayTy(A_pos pos, S_symbol array);

A_field A_Field(A_pos pos, S_symbol name, S_symbol typ);
A_fieldList A_FieldList(A_field head, A_fieldList tail);
A_expList A_ExpList(A_exp head, A_expList tail);
A_fundec A_Fundec(A_pos pos, S_symbol name, A_fieldList params,
                  S_symbol result, A_exp body);
A_fundecList A_FundecList(A_fundec head, A_fundecList tail);
A_decList A_DecList(A_dec head, A_decList tail);
A_namety A_Namety(S_symbol name, A_ty ty);
A_nametyList A_NametyList(A_namety head, A_nametyList tail);
A_efield A_Efield(S_symbol name, A_exp exp);
A_efieldList A_EfieldList(A_efield head, A_efieldList tail);

typedef enum {A_plusOp, A_minusOp, A_timesOp, A_divideOp,
              A_eqOp, A_neqOp, A_ltOp, A_leOp, A_gtOp, A_geOp} A_oper;
```

FIGURE 4.7. Abstract syntax for the Tiger language. Only the constructor functions are shown; the structure fields correspond exactly to the names of the constructor arguments.

```
A_SeqExp(2,
 A_ExpList(A_AssignExp(4,A_SimpleVar(2,S_Symbol("a")),A_IntExp(7,5)),
 A_ExpList((A_OpExp(11,A_VarExp(A_SimpleVar(10,S_Symbol("a"))),
                        A_plusOp,A_IntExp(12,1))),
 NULL)))
```

This is a *sequence expression* containing two expressions separated by a semicolon: an *assignment expression* and an *operator expression*. Within these are a *variable expression* and two *integer constant expressions*.

The positions sprinkled throughout are source-code character count. The position I have chosen to associate with an `AssignExp` is that of the `:=` operator, for an `OpExp` that of the `+` operator, and so on. *These decisions are a matter of taste;* they represent my guesses about how they will look when included in semantic error messages.

Now consider

```
let var a := 5
    function f() : int = g(a)
    function g(i: int) = f()
 in f()
end
```

The Tiger language treats *adjacent* function declarations as (possibly) mutually recursive. The `FunctionDec` constructor of the abstract syntax takes a *list* of function declarations, not just a single function. The intent is that this list is a maximal consecutive sequence of function declarations. Thus, functions declared by the same `FunctionDec` can be mutually recursive. Therefore, this program translates into the abstract syntax,

```
A_LetExp(
 A_DecList(A_VarDec(S_Symbol("a"),NULL,A_IntExp(5)),
 A_DecList(A_FunctionDec(
            A_FundecList(A_Fundec(
              S_Symbol("f"),NULL,S_Symbol("int"),
              A_CallExp(S_Symbol("g"), ···)),
            A_FundecList(A_Fundec(
              S_Symbol("g"),
            A_FieldList(S_Symbol("i"),S_Symbol("int"),NULL),
            NULL,
            A_CallExp(S_Symbol("f"), ···)),
            NULL))),
 NULL)),
 A_CallExp(S_Symbol("f"), NULL))
```

where the positions are omitted for clarity.

The `TypeDec` constructor also takes a list of type declarations, for the same reason; consider the declarations

```
type tree = {key: int, children: treelist}
type treelist = {head: tree, tail: treelist}
```

which translate to *one* type declaration, not two:

```
A_TypeDec(
 A_NametyList(A_Namety(S_Symbol("tree"),
   A_RecordTy(
     A_FieldList(A_Field(S_Symbol("key"),S_Symbol("int")),
     A_FieldList(A_Field(S_Symbol("children"),
                         S_Symbol("treelist")),
     NULL)))),
 A_NametyList(A_NameTy(S_Symbol("treelist"),
   A_RecordTy(
     A_FieldList(A_Field(S_Symbol("head"),S_Symbol("tree")),
     A_FieldList(A_Field(S_Symbol("tail"),S_Symbol("treelist")),
     NULL)))),
 NULL)))
```

There is no abstract syntax for "`&`" and "`|`" expressions; instead, e_1`&`e_2 is translated as `if` e_1 `then` e_2 `else` 0, and e_1`|`e_2 is translated as though it had been written `if` e_1 `then` 1 `else` e_2.

Similarly, unary negation $(-i)$ should be represented as subtraction $(0 - i)$ in the abstract syntax.[1] Also, where the body of a `LetExp` has multiple statements, we must use a `seqExp`.

By using these representations for `&`, `|`, and unary negation, we keep the abstract syntax data type smaller and make fewer cases for the semantic analysis phase to process. On the other hand, it becomes harder for the type-checker to give meaningful error messages that relate to the source code.

The lexer returns `ID` tokens with `string` values. The abstract syntax requires identifiers to have `symbol` values. The function `S_symbol` (from `symbol.h`) converts strings to symbols, and the function `S_name` converts back. The representation of symbols is discussed in Chapter 5.

[1]This might not be adequate in an industrial-strength compiler. The most negative two's complement integer of a given size cannot be represented as $0 - i$ for any i of the same size. In floating point numbers, $0 - x$ is not the same as $-x$ if $x = 0$. We will neglect these issues in the Tiger compiler.

The semantic analysis phase of the compiler will need to keep track of which local variables are used from within nested functions. The `escape` component of a `varDec` or `field` is used to keep track of this. This `escape` field is not mentioned in the parameters of the constructor function, but is always initialized to `TRUE`, which is a conservative approximation. The `field` type is used for both formal parameters and record fields; `escape` has meaning for formal parameters, but for record fields it can be ignored.

Having the `escape` fields in the abstract syntax is a "hack," since escaping is a global, nonsyntactic property. But leaving `escape` out of the `Absyn` would require another data structure for describing escapes.

PROGRAM ABSTRACT SYNTAX

Add semantic actions to your parser to produce abstract syntax for the Tiger language.

You should turn in the file `tiger.grm`.

Supporting files available in `$TIGER/chap4` include:

`absyn.h` The abstract syntax declarations for Tiger.
`absyn.c` Implementation of the constructor functions.
`prabsyn.[ch]` A pretty-printer for abstract syntax trees, so you can see your results.
`errormsg.[ch]` As before.
`lex.yy.c` Use this only if your own lexical analyzer still isn't working.
`symbol.[ch]` A module to turn strings into `symbols`.
`makefile` As usual.
`parse.[ch]` A driver to run your parser on an input file.
`tiger.grm` The skeleton of a grammar specification.

FURTHER READING

Many compilers mix recursive-descent parsing code with semantic-action code, as shown in Program 4.1; Gries [1971] and Fraser and Hanson [1995] are ancient and modern examples. Machine-generated parsers with semantic actions (in special-purpose "semantic-action mini-languages") attached to the grammar productions were tried out in 1960s [Feldman and Gries 1968]; Yacc [Johnson 1975] was one of the first to permit semantic action fragments to be written in a conventional, general-purpose programming language.

The notion of *abstract syntax* is due to McCarthy [1963], who designed the abstract syntax for Lisp [McCarthy et al. 1962]. The abstract syntax was intended to be used writing programs until designers could get around to creating a concrete syntax with human-readable punctuation (instead of **L**ots of **I**rritating **S**illy **P**arentheses), but programmers soon got used to programming directly in abstract syntax.

The search for a theory of programming-language semantics, and a notation for expressing semantics in a compiler-compiler, led to ideas such as *denotational semantics* [Stoy 1977]. The denotational semanticists also advocated the separation of concrete syntax from semantics – using abstract syntax as a clean interface – because in a full-sized programming language the syntactic clutter gets in the way of understanding the semantic analysis.

EXERCISES

4.1 Write type declarations and constructor functions to express the abstract syntax of regular expressions.

4.2 Implement Program 4.4 as a recursive-descent parser, with the semantic actions embedded in the parsing functions.