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 ML-Yacc, semantic actions are fragments of ML 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 $A \to B\ C\ D$, 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

```
datatype token = ID of string | NUM of int | PLUS | MINUS  ···

fun F() = case !tok
           of ID s   => (advance(); lookup(s))
            | NUM i  => (advance(); i)
            | LPAREN => (eat(LPAREN);
                         let i = E()
                          in eatOrSkipTo(RPAREN,[PLUS,TIMES,EOF]);
                             i
                         end
            | EOF    => (print("expected factor"); 0)
            | _      => (print "expected ID, NUM, or (";
                         skipto[PLUS,TIMES,RPAREN,EOF]; 0)

and T()  = case !tok
            of ID     => T'(F())
             | NUM    => T'(F())
             | LPAREN => T'(F())
             | _      => (print "expected ID, NUM, or (";
                          skipto[PLUS,TIMES,RPAREN,EOF]; 0)
and T'(a) = case !tok
            of PLUS   => a
             | TIMES  => (eat(TIMES);  T'( a*F() ))
             | RPAREN => a
             | EOF    => a

and eatOrSkipTo(expected,stop) =
     if !tok = expected then eat(expected)
     else (print (···); skipto(stop))
```

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

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

Program 4.1 is a recursive-descent interpreter for part of 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

```
%%
%term INT of int | PLUS | MINUS | TIMES | UMINUS | EOF
%nonterm  exp of int
%start exp
%eop EOF

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

exp :  INT              (INT)
    |  exp PLUS exp     (exp1 + exp2)
    |  exp MINUS exp    (exp1 - exp2)
    |  exp TIMES exp    (exp1 * exp2)
    |  MINUS exp    %prec UMINUS  (~exp)
```

PROGRAM 4.2. ML-Yacc version of Grammar 3.35.

```
let val a = T()
    val _ = eat(TIMES)
    val b = F()
 in a*b
end
```

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.

ML-Yacc-GENERATED PARSERS

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

Program 4.2 shows how this works for Grammar 3.35. The semantic action can refer to the semantic values of right-hand-side symbols just by mentioning their names; if a symbol (such as `exp` in the PLUS rule) is repeated, then the occurrences are distinguished by numeric suffixes (`exp1`, `exp2`). The value produced by every semantic expression for `exp` must match the type of the `exp` nonterminal, in this case `int`.

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

An ML-Yacc–generated parser implements semantic values by keeping a stack of them parallel to the state stack. Where each symbol would be on a

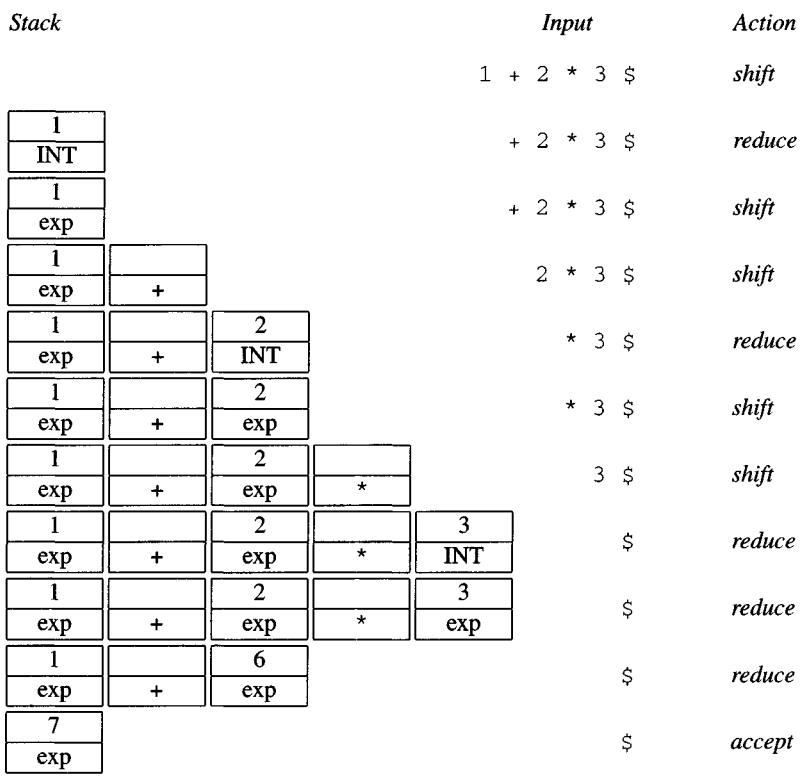

FIGURE 4.3. Parsing with a semantic stack.

simple parsing stack, now there is a semantic value. When the parser performs a reduction, it must execute an ML-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 ML 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.

```
fun update(table,id,num) = fn j => if j=id then num else table(j)
val emptytable = fn j => raise Fail ("uninitialized var: " ^ j)
%%
%term INT of int | ID of string | PLUS | MINUS | TIMES | DIV |
      ASSIGN | PRINT | LPAREN | RPAREN | COMMA | SEMICOLON | EOF
%nonterm exp of table->int
       | stm of table->table
       | exps of table->unit
       | prog of table
%right SEMICOLON
%left PLUS MINUS
%left TIMES DIV

%start prog
%eop EOF
%pos int
%%
prog: stm                          (stm(emptytable))

stm : stm SEMICOLON stm            (fn t => stm2(stm1(t)))
stm : ID ASSIGN exp                (fn t => update(t,ID,exp))
stm : PRINT LPAREN exps RPAREN     (fn t => (exps t; t))

exps: exp                          (fn t => print(exp t))
exps: exps COMMA exp               (fn t => (exps t; print(exp t)))

exp : INT                          (fn t => INT)
exp : ID                           (fn t => t(ID))
exp : exp PLUS exp                 (fn t => exp1(t) + exp2(t))
exp : exp MINUS exp                (fn t => exp1(t) - exp2(t))
exp : exp TIMES exp                (fn t => exp1(t) * exp2(t))
exp : exp DIV exp                  (fn t => exp1(t) / exp2(t))
exp : stm COMMA exp                (fn t => exp(stm(t)))
exp : LPAREN exp RPAREN            ( exp )
```

PROGRAM 4.4. An interpreter for straight-line programs.

A MINI-INTERPRETER IN SEMANTIC ACTIONS

To illustrate the power of semantic actions, let us write an interpreter for the language whose abstract syntax is given in Program 1.5. In Chapter 1 we interpreted the abstract syntax trees; with Yacc we can interpret the real thing, the concrete syntax of the language.

Program 4.4 has semantic values that express the meaning (or *denotation*) of each kind of straight-line program expression and statement. A semantic value for an `exp` is a function from table to integer; roughly, "you give me a table to look up identifiers, and I'll give you back an integer." Thus, the

semantic value for a simple identifier `x` is, "give me a table, and I'll look up `x` and give you what I find." The semantic value for `5` is even simpler: "give me a table, and I'll ignore it and give you the integer 5." Finally, the semantic value for $e_1 + e_2$ is, "give me a table t, and first I'll apply e_1 to t, then do $e_2(t)$, then add the resulting integers."

Thus, the expression `a+1` translates, by simple expansion of the semantic actions, to $\lambda t.(\lambda t_1.t_1(\texttt{a}))(t) + (\lambda t_2.1)(t)$, which is equivalent to $\lambda t.t(\texttt{a}) + 1$.

A statement takes a table argument and returns an appropriately adjusted table result. The statement `b:=6`, applied to a table t, returns a new table t' that is just like t except that $t'(\texttt{b}) = 6$. And to implement the compound statement $s_1; s_2$ applied to a table t, we first get table t' by evaluating $s_1(t)$, then evaluate $s_2(t')$.

This interpreter contains a major error: an assignment statement inside an expression has no permanent effect (see Exercise 4.2).

AN IMPERATIVE INTERPRETER IN SEMANTIC ACTIONS

Program 4.2 and Program 4.4 show how semantic values for nonterminals can be calculated from the semantic values of the right-hand side of the productions. The semantic actions in these examples 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 *imperative* semantic actions with global side effects, and be able to predict the order of their occurrence.

Program 4.5 shows an imperative version of the interpreter. 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 an ML-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)

```
val table = ref (nil: (string * int) list)
fun update(id,v) = table := (id,v) :: !table
fun lookup(id) = let fun f((s,v)::rest) = if s=id then v else f rest
                       | f nil = raise Fail "uninitialized var: "^id
                  in f (!table)
                 end
%%
%term INT of int | ID of string | PLUS | MINUS | TIMES | DIV |
      ASSIGN | PRINT | LPAREN | RPAREN | COMMA | SEMICOLON | EOF
%nonterm exp of int | stm | exps | prog

%right SEMICOLON
%left PLUS MINUS
%left TIMES DIV

%start prog
%eop EOF
%pos int
%%

prog: stm                        ()

stm : stm SEMICOLON stm          ()
stm : ID ASSIGN exp              (update(ID,exp))
stm : PRINT LPAREN exps RPAREN   (print("\n"))

exps: exp                        (print(exp))
exps: exps COMMA exp             (print(exp))

exp : INT                        (INT)
exp : ID                         (lookup(ID))
exp : exp PLUS exp               (exp1 + exp2)
exp : exp MINUS exp              (exp1 - exp2)
exp : exp TIMES exp              (exp1 * exp2)
exp : exp DIV exp                (exp1 / exp2)
exp : stm COMMA exp              (exp)
exp : LPAREN exp RPAREN          (exp)
```

PROGRAM 4.5. An interpreter in imperative style.

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.

Such a parse tree, which we will call a *concrete parse tree* representing the

$S \rightarrow S\ ;\ S$	$L \rightarrow$
$S \rightarrow$ `id :=` E	$L \rightarrow L\ E$
$S \rightarrow$ `print` L	
$E \rightarrow$ `id`	$B \rightarrow +$
$E \rightarrow$ `num`	$B \rightarrow -$
$E \rightarrow E\ B\ E$	$B \rightarrow \times$
$E \rightarrow S\ ,\ E$	$B \rightarrow /$

GRAMMAR 4.6. Abstract syntax of straight-line programs.

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 4.6 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 4.6 is not meant for parsing. The parser uses the *concrete syntax* (Program 4.7) 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 ML, this is very convenient using the `datatype` facility. Program 1.5 shows the data structure declarations for Grammar 4.6.

The ML-Yacc (or recursive-descent) parser, parsing the *concrete syntax*, constructs the *abstract* syntax tree. This is shown in Program 4.7.

In ML we represent an abstract syntax as a set of mutually recursive datatypes. The datatypes shown in Program 1.5 constitute an abstract syntax for a simple programming language. Each abstract syntax grammar production is represented by a single data constructor in ML; each nonterminal is represented by an ML type. Just as nonterminals are mutually recursive in the grammar, the ML types for abstract syntax refer to each other recursively. (For the nonterminal L of expression lists, the ML data constructors are `::` and `nil` of the built-in `list` datatype.)

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.

```
structure Absyn = struct
   datatype binop = ··· (see Program 1.5)
   datatype stm =  ···
        and exp =  ···
end
%%
%term INT of int | ID of string | PLUS | MINUS | TIMES | DIV
     | ASSIGN | PRINT | LPAREN | RPAREN | COMMA | SEMICOLON | EOF
%nonterm exp of exp | stm of stm | exps of exp list | prog of stm
%right SEMICOLON
%left PLUS MINUS
%left TIMES DIV
%start prog
%eop EOF
%pos int
%%
prog: stm                          (stm)

stm : stm SEMICOLON stm            (Absyn.CompoundStm(stm1,stm2))
stm : ID ASSIGN exp                (Absyn.AssignStm(ID,exp))
stm : PRINT LPAREN exps RPAREN     (Absyn.PrintStm(exps))

exps: exp                          ( exp :: nil )
exps: exp COMMA exps               ( exp :: exps )

exp : INT                          (Absyn.NumExp(INT))
exp : ID                           (Absyn.IdExp(ID))
exp : exp PLUS exp                 (Absyn.OpExp(exp1,Absyn.Plus,exp2))
exp : exp MINUS exp                (Absyn.OpExp(exp1,Absyn.Minus,exp2))
exp : exp TIMES exp                (Absyn.OpExp(exp1,Absyn.Times,exp2))
exp : exp DIV exp                  (Absyn.OpExp(exp1,Absyn.Div,exp2))
exp : stm COMMA exp                (Absyn.EseqExp(stm,exp))
exp : LPAREN exp RPAREN            ( exp )
```

PROGRAM 4.7. Abstract-syntax builder for straight-line programs.

The lexer must pass the source-file positions of the beginning and end of each token to the parser. The ML-Yacc parser makes these positions available (in the semantic action fragments): for each terminal or nonterminal such as `FOO` or `FOO1` on the right-hand side of a rule, the ML variable `FOOleft` or `FOO1left` stands for the left-end position of the terminal or nonterminal, and `FOOright` or `FOO1right` stands for the right-end position. These position variables can be used in abstract-syntax expressions for `pos` fields.

ABSTRACT SYNTAX FOR Tiger

Figure 4.8 shows a datatype 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.

The Tiger program

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

translates into abstract syntax as

```
SeqExp[(AssignExp{var=SimpleVar(symbol"a",2),
                  exp=IntExp 5,
                  pos=4},2),
       (OpExp{left=VarExp(SimpleVar(symbol"a",10)),
              oper=PlusOp,
              right=IntExp 1,
              pos=11},10)]
```

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 (2, 4, 2, 11, 10) 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. Also, each statement in a `SeqExp` has a position; I have used the beginning of the statement. *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,

```
structure Absyn = struct
type pos = int   and   symbol = Symbol.symbol

datatype var = SimpleVar of symbol * pos
             | FieldVar of var * symbol * pos
             | SubscriptVar of var * exp * pos
and exp =
   VarExp of var
 | NilExp
 | IntExp of int
 | StringExp of string * pos
 | CallExp of {func: symbol, args: exp list, pos: pos}
 | OpExp of {left: exp, oper: oper, right: exp, pos: pos}
 | RecordExp of {fields: (symbol * exp * pos) list, typ: symbol, pos: pos}
 | SeqExp of (exp * pos) list
 | AssignExp of {var: var, exp: exp, pos: pos}
 | IfExp of {test:exp, then': exp, else': exp option, pos:pos}
 | WhileExp of {test: exp, body: exp, pos: pos}
 | ForExp of {var: symbol, escape: bool ref,
                    lo: exp, hi: exp, body: exp, pos: pos}
 | BreakExp of pos
 | LetExp of {decs: dec list, body: exp, pos: pos}
 | ArrayExp of {typ: symbol, size: exp, init: exp, pos: pos}

and dec = FunctionDec of fundec list
        | VarDec of {name: symbol, escape: bool ref,
                     typ: (symbol * pos) option, init: exp, pos: pos}
        | TypeDec of {name: symbol, ty: ty, pos:pos} list

and ty = NameTy of symbol * pos
       | RecordTy of field list
       | ArrayTy of symbol * pos

and oper = PlusOp | MinusOp | TimesOp | DivideOp
         | EqOp | NeqOp | LtOp | LeOp | GtOp | GeOp

withtype field = {name: symbol, escape: bool ref, typ: symbol, pos: pos}
   and   fundec = {name: symbol, params: field list,
                   result: (symbol * pos) option,
                   body: exp, pos: pos}
end
```

FIGURE 4.8. Abstract syntax for the Tiger language.

```
LetExp{decs=[VarDec{name=symbol"a",escape=ref true,
                    typ=NONE, pos=9,init=IntExp 5},
             FunctionDec[
               {name=symbol"f",params=nil,
                result=SOME(symbol"int",35), pos=20,
                body=CallExp{func=symbol"g", pos=41,
                             args=[VarExp(SimpleVar(symbol"a"))]}},
               {name=symbol"g",
                params=[{name=symbol"i",escape=ref true,
                         typ=symbol"int",pos=64}],
                result=NONE, pos=50,
                body=CallExp{func=symbol"f",pos=71,args=nil}}]],
       body=CallExp{func=symbol"f", pos=79, args=nil}}
```

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:

```
TypeDec[{name=symbol"tree", pos=2,
         ty=RecordTy[{name=symbol"key", escape=ref true,
                      typ=symbol"int", pos=15},
                     {name=symbol"children", escape=ref true,
                      typ=symbol"treelist", pos=25}]},
        {name=symbol"treelist", pos=47,
         ty=RecordTy[{name=symbol"head", escape=ref true,
                      typ=symbol"tree", pos=64},
                     {name=symbol"tail", escape=ref true,
                      typ=symbol"treelist", pos=76}]}]
```

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`. An empty statement is represented by `SeqExp[]`.

[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.

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 typechecker 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 `Symbol.symbol` converts strings to symbols, and the function `Symbol.name` converts back. The representation of symbols is discussed in Chapter 5.

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`, `ForExp`, or formal parameter is used to keep track of this. The parser should leave each `escape` set 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`.

I recommend you put the declaration

```
structure A = Absyn
```

at the top of your Tiger.grm; then you can refer to constructors from the `Absyn` module using `A.NilExp` instead of the more verbose `Absyn.NilExp`.

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

`absyn.sml` The abstract syntax data structure for Tiger.

`printabsyn.sml` A pretty-printer for abstract syntax trees, so you can see your results.

`errormsg.sml` As before.

`tiger.lex.sml` For use only if your own lexical analyzer still isn't working.

`symbol.sml` A module to turn strings into `symbols`. This file uses components of the *Standard ML of New Jersey Library*. Look them up in the library manual to see what's going on.

`sources.cm` as usual. But note that this time, the file includes a reference to the *Standard ML of New Jersey Library*.

`parse.sml` 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 semantic interpreter shown in Program 4.4 is inspired by ideas from denotational semantics. But 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 an ML datatype to express the abstract syntax of regular expressions.

4.2 When Program 4.4 interprets straight-line programs, statements embedded inside expressions have no permanent effect – any assignments made within those statements "disappear" at the closing parenthesis. Thus, the program

```
a := 6; a := (a := a+1, a+4) + a; print(a)
```

prints 17 instead of 18. To fix this problem, change the type of semantic values for `exp` so that it can produce a value *and* a new table; that is:

```
%nonterm exp of table->(table*int)
       | stm of table->table
       | exps of table->table
       | prog of table
```

Then change the semantic actions of Program 4.4 accordingly.

4.3 Program 4.4 is not a purely functional program; the semantic action for PRINT contains an ML print statement, which is a side effect. You can make the interpreter purely functional by having each statement return, not only a table, but also a list of values that would have been printed. Thus,

```
%nonterm stm of table->(table * int list)
       | prog of int list
```

Adjust the types of `exp` and `exps` accordingly, and rewrite Program 4.4.

4.4 Combine the ideas of Exercises 4.2 and 4.3 to write a purely functional version of the interpreter that handles printing and statements-within-expressions correctly.

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

***4.6** Program 4.4 uses right recursion in the production $exps \rightarrow exp, exps$ while Program 4.5 uses left recursion in $exps \rightarrow exps, exp$.

a. Rewrite the *exps* rules of Program 4.4 to use a left-recursive grammar, but still build the same right-recursive data structure. This requires extra work in the semantic actions.

b. Rewrite the *exps* rules (and semantic actions) of Program 4.5 to use a right-recursive grammar, but still calculate the same result (i.e., print the same numbers in the same order).