Compiler Principle

Prof. Dongming LU

Mar. 18th, 2024

Content

1. INTRODUCTION

2. LEXICAL ANALYSIS
3. PARSING
4. **ABSTRACT SYNTAX**
5. SEMANTIC ANALYSIS

6. ACTIVATION RECORD
7. TRANSLATING INTO INTERMEDIATE CODE

8. OTHERS

4 ABSTRACT SYNTAX

4.1 SEMANTIC ACTIONS

Do useful things with the phrase that parsed

Why Semantic Actions

- **Each terminal and nonterminal may be associated with its own type of semantic value.**

  A rule: $A \rightarrow B\ C\ D$

- **The semantic action must return a value whose type is the one associated with the nonterminal A.**
- **It can build this value from the values associated with the matched terminals and nonterminals B, C, D.**

Recursive descent

- **The semantic actions are the values returned by parsing functions, or the side effects of those functions, or both.**

$$T \rightarrow T * F$$

- **The semantic action:**

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

Recursive descent

$S \rightarrow E\ \$$

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'$

$E' \rightarrow -\ T\ E'$

$E' \rightarrow$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T'$

$T' \rightarrow /\ F\ T'$

$T' \rightarrow$

$F \rightarrow \text{id}$

$F \rightarrow \text{num}$

$F \rightarrow (E)$

Recursive descent

```
int lookup(String id) { ... }
int F_follow[ ] = { TIMES, DIV, 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;
}}

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

Recursive descent

```
int T_follow[ ] = { PLUS, MINUS, EOF ,-1};

int T( void) {switch (tok)  {
    case ID: case NUM: case LPAREN: return
 Tprime(F());
    default:  print("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: ... }}
```

Yacc-GENERATED PARSERS

- Whenever the generated parser reduces by a rule, it will execute the corresponding semantic action fragment.

```
exp: exp '+' term   {$$ = $1 + $3;}
   | exp '-' term   {$$ = $1 - $3;}
   | term           {$$ = $1;}
   ;

term: term '*' factor  {$$ = $1* $3;}
    | factor  {$$ = $1;}
    ;

factor :NUMBER  {$$ = $1;}
       | '(' exp ')'  {$$=$2;}
```

Yacc-GENERATED PARSERS

A Yacc-generated parser implement semantic values by keeping a stack of the them parallel to the state stack.

- When the parser performs a reduction, it must execute a C-language semantic action.

- 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 .

4.2 ABSTRACT PARSE TREES

To improve modularity, to separate issues of syntax from issues of semantics

A parse tree

- A data structure that later phases of the compiler can traverse.
- 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.

- **A concrete parse tree**
  - ✓ **Representing the *concrete syntax* of the source language**

- **An abstract syntax**
  - ✓ **Conveys *the phrase structure* of the source program**
  - ✓ **With all parsing issues resolved but without any semantic interpretation.**

An abstract syntax

It makes a clean interface between the parser and the later phrase of a compiler.

- The parser uses the *concrete syntax* to build a parse tree for the *abstract syntax*.
- The semantic analysis phase takes this *abstract syntax tree*.

An abstract syntax

S → S;S	L →
S → id:=E	L → LE
S → print L	
E → id	B → +
E → num	B → -
E → E B E	B → *
E → S,E	B → /

GRAMMAR 4.5 Abstract syntax of straight-line programs

```
%%
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);}

exp: ID                                      {$$=A_IdExp($1);}
exp: INT                                     {$$=A_NumExp($1);}
exp: exp PLUS exp                           {$$=A_OpExp($1,A_plus,$3);}
exp: stm COMMA exp                          {$$=A_EseqExp($1,$3);}

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

Part of PROGRAM 4.6

In program 4.6

- The Yacc parser: parsing the concrete syntax, constructs the abstract syntax tree.

```
A_stm A_CompoundStm(A_stm stm1, A_stm stm2)
{
   A_stm  s= checked_malloc (sizeof (*s));
   s->kind = A_CompoundStm;
   s->u.compound.stm1 = stm1;
   s->u.compound.stm2 = stm2;
   return s;
}
```

In program 4.6

- A typedef for each nonterminal , a union-variant for each production.

```
typedef struct A_stm_  *A_stm;
struct A_stm_
 { enum { A_compoundStm, A_assignStm,
          A_printStm} kind;
    union { struct {A_stm stm1,stm2;} compound;
            struct {string id; A_exp exp;} assign;
            struct {A_expList exps; } print;
            }  u;
};

A_stm A_CompoundStm (A_stm stm1, A_stm stm2);
```

POSITIONS

- 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 a one-pass compiler, the lexical analyzer keeps a "current position" global variable.

- A compiler that uses abstract-syntax-tree data structures need to remember positions accurately.
- pos fields indicate the position of the characters from which these abstract-syntax structures were derived.

POSITIONS

- 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).

```
%{ 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);  }
```

Abstract syntax for Tiger

```
/*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;
         }
```

Abstract syntax for Tiger

```
/*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_IntExp(A_pos pos, int i);
…
A_exp A_OpExp(A_pos pos, A_oper oper, A_exp left, A_exp right);
…
A_exp A_SeqExp(A_pos pos, A_expList seq);
A_exp A_AssignExp(A_pos pos, A_var var, A_exp exp);
…
A_expList A_ExpList(A_exp head, A_expList tail);
…
```

Figure 4.7 Abstract syntax for the Tiger language

Abstract syntax for Tiger

The Tiger program

(a := 5; a+1)

translates into **abstract syntax** as

A_SeqExp(2,

A_ExpList(A_AssignExp(4,A_SimpleVar(2,S_Symbol(“a”),A_IntExp(7,5),
A_ExpList(A_OpExp(11,A_plusOp,A_VarExp(A_SimpleVar(10,

S_Symbol(“a”))),A_IntExp(12,1))),
NULL)))

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

The end of Chapter 4