COMPILER CONSTRUCTION

Tutorial 2

PHASES OF A COMPILER

Lab 2 Symtab – administrates the symbol table
Lab 3 Parser – manages syntactic analysis, build internal form
Lab 4 Semantics – checks static semantics

THE SYMBOL TABLE

LAB 2

TDDB44 Compiler Construction
Tutorial 2
A Symbol table contains all the information that must be passed between different phases of a compiler/interpreter.

A symbol (or token) has at least the following attributes:
- Symbol Name
- Symbol Type (int, real, char, ....)
- Symbol Class (static, automatic, cons...)

In a compiler we also need:
- Address (where it is the info stored?)
- Other info due to used data structures

Symbol tables are typically implemented using hashing schemes because good efficiency for the lookup is needed.

We classify for symbol tables as:
- Simple
- Scoped

Simple symbol tables have...
... only one scope
... only “global” variables

Simple symbol tables may be found in BASIC and FORTRAN compilers.

Complication in simple tables involves languages that permit multiple scopes

C permits at the simplest level two scopes: global and local (it is also possible to have nested scopes in C)
WHY SCOPES?

The importance of considering the scopes are shown in these two C programs

```c
main(){
   int a=10; //global variable
   changeA();
   printf("Value of a=%d\n",a);
}

void changeA(){
   int a;  //local variable
   a=5;
}
```

SCOPED SYMBOL TABLES

Operations that must be supported by the symbol table in order to handle scoping:

- Lookup in any scope – search the most recently created scope first
- Enter a new symbol in the symbol table
- Modify information about a symbol in a “visible” scope
- Create a new scope
- Delete the most recently scope

HOW IT WORKS

```
Hash Table
READ, REAL, A, WRITE, P1
INTEGER

Index to string table
Other info, Hash Link

Block Table
sym_pos
sym_pos
sym_pos
```

LAB 3 PARSING
SYNTAX ANALYSIS

The parser accepts tokens from the scanner and verifies the syntactic correctness of the program specification. Along the way, it also derives information about the program and builds a fundamental data structure known as parse tree. The parse tree is an internal representation of the program and it also augments the symbol table.

PURPOSE

1. Verify the syntactic correctness of the input token stream, reporting any errors
2. Produce a parse tree and certain table for use by later phases
   • Syntactic correctness is judged by verification against a formal grammar which specifies the language to be recognized
   • Error messages are important and should be as meaningful as possible
   • Parse tree and tables will vary depending on compiler implementation technique and source language

METHOD

Match token stream using manually or automatically generated parser

PARSING STRATEGIES

Two categories of parsers:
   • Top-down parsers
   • Bottom up parsers

Within each of these broad categories are a number of sub strategies depending on whether leftmost or rightmost derivations are used.
TOP-DOWN PARSING

Start with a goal symbol and recognize it in terms of its constituent symbols

Example: recognize a procedure in terms of its sub-components (header, declarations, and body)

The parse tree is then built from the top (root) and down (leaves), hence the name

Top-down analysis

How do we know in which order the string is to be derived?

Use one or more tokens lookahead.

Example: Top-down analysis with backup

1. `<S>` → `a` `<A>` `b` \( ? \) symbol lookahead works well
2. `c` `<A>` `d` \( ? \) symbol lookahead works well
3. `a` `<A>` `d` \( ? \) symbol lookahead works well
4. `c` `d` `f` `t` `n` \( ? \) symbol lookahead works well

a) adbe
   \[
   \begin{align*}
   &a, b, c, d, e \\
   &\text{parse tree}
   \end{align*}
   \]

b) ad
   \[
   \begin{align*}
   &a, b, c, d, e \\
   &\text{parse tree}
   \end{align*}
   \]

- Top-down analysis with backup is implemented by writing a procedure or a function for each non-terminal whose task is to find one of its right sides:

```
function A:boolean ("A\rightarrow\alpha\beta\gamma")
begin
  var savep : ptn;
  begin
    savep := imgpt;
    if imgpt = "\alpha" then begin
      scan: ("Get next token, move imgpt a step")
      if imgpt = "\gamma" then begin
        scan: ("If \gamma not found, backtrack and try \beta")
        return(true); ("\gamma" found)
      end;
    end;
    end;
    imgpt := savep; ("\alpha" not found; backtrack and try \beta")
    if imgpt = "\beta" then begin
      scan: ("If \beta not found, backtrack and try \gamma")
      return(true); ("\beta" found, OK)
    end;
    return(false);
  end;
end;
```
BOTTOM-UP PARSING

Recognize the components of a program and then combine them to form more complex constructs until a whole program is recognized.

Example: recognize a procedure from its sub-components (header, declarations, and body).

The parse tree is then built bottom and up, hence the name BOTTOM-UP PARSING.

PARSING TECHNIQUES

A number of different parsing techniques are commonly used for syntax analysis, including:

- Recursive-descent parsing
- LR parsing
- Operator precedence parsing
- Many more…

LR PARSING

A specific bottom-up technique

- LR stands for Left->right scan, Rightmost derivation
- Probably the most common & popular parsing technique
- YACC, BISON, and many other parser generation tools utilize LR parsing
- Great for machines, not so cool for humans…
Advantages of LR:
• Accept a wide range of grammars/languages
• Well suited for automatic parser generation
• Very fast
• Generally easy to maintain

Disadvantages of LR:
• Error handling can be tricky
• Difficult to use manually

Bison is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar.

One of many parser generator packages
Yet Another Compiler Compiler
• Really a poor name, is more of a parser compiler
• Can specify actions to be performed when each construct is recognized and thereby make a full fledged compiler but its the user of Bison that specify the rest of the compilation process…
• Designed to work with FLEX or other automatically or hand generated “lexers”
A Bison specification is composed of 4 parts:

- C declarations
- Bison declarations
- Grammar rules
- Additional C code

Comments enclosed in `/* ... */' may appear in any of the sections.

Looks like Flex specification, doesn’t it? Similar function, tools, look and feel.

C DECLARATIONS

- Contains macro definitions and declarations of functions and variables that are used in the actions in the grammar rules.
- Copied to the beginning of the parser file so that they precede the definition of yyparse.
- Use `#include` to get the declarations from a header file. If C declarations isn’t needed, the `${` and `%` delimiters that bracket this section might be omitted.

BISON DECLARATIONS

- Contains declarations that define terminal and nonterminal symbols, and specify precedence.

GRAMMAR RULES

- Contains one or more Bison grammar rules, and nothing else.
- There must always be at least one grammar rule, and the first `%%` (which precedes the grammar rules) may never be omitted even if it is the first thing in the file.
ADDITIONAL C CODE

• Copied verbatim to the end of the parser file, just as the C declarations section is copied to the beginning
• This is the most convenient place to put anything that should be in the parser file but isn’t need before the definition of yyparse
• The definitions of yylex and yyerror often go here

BISON EXAMPLE

```c
#include <ctype.h> /* standard C declarations here */

%token DIGIT /* BISON declarations */

%%
/* Grammar rules */
line : expr \n \{ prtf("%d\n",$1); \} ;
expr : expr + term \{ $$=$1+$3; \} 
| term ;
term : term * factor \{ $$=$1*$3; \} 
| factor ;

factor : '(' expr ')' \{ $$=$2; \}
DIGIT ;
\% Additional C code */
yylex() /* A really simple lexical analyzer*/
int c;
int c = getchar();
if(isdigit(c)){
yylval=c-'0';
return DIGIT;
}
return c;
```

Note: Bison uses yylex, yylval, etc - designed to be used with FLEX

USING BISON WITH FLEX

Bison and Flex are obviously designed to work together

• Bison produces a driver program called yylex() (actually its included in the lex library -II)
  #include “lex.yy.c” in the third part of Bison specification
  this gives the program yylex access to Bisons’ token names
USING BISON WITH FLEX

- Thus do the following:
  - % flex scanner.l
  - % bison parser.y
  - % cc y.tab.c -ly -ll
- This will produce an a.out which is a parser with an integrated scanner included

ERROR HANDLING IN BISON

Error handling in Bison is provided by error productions.
An error production has the general form:
- non-terminal: error synchronizing set
  - non-terminal where did it occur
  - error a keyword
  - synchronizing-set possible empty subset of tokens

When an error occurs, Bison pops symbols off the stack until it finds a state for which there exists an error production which may be applied.

PURPOSE

To verify the semantic correctness of the program represented by the parse tree, reporting any errors, possibly, to produce an intermediate form and certain tables for use by later compiler phases:
- Semantic correctness: the program adheres to the rules of the type system defined for the language (plus some other rules).
- Error messages: should be as meaningful as possible.
- In this phase, there is sufficient information to be able to generate a number of tables of semantic information, such as identifier, type, and literal tables.

LAB 4 SEMANTICS

To verify the semantic correctness of the program represented by the parse tree, reporting any errors, possibly, to produce an intermediate form and certain tables for use by later compiler phases:
- Semantic correctness: the program adheres to the rules of the type system defined for the language (plus some other rules).
- Error messages: should be as meaningful as possible.
- In this phase, there is sufficient information to be able to generate a number of tables of semantic information, such as identifier, type, and literal tables.
METHOD

Ad hoc confirmation of semantic rules

IMPLEMENTATION

Semantic analyzer implementations are typically syntax directed

More formally, such techniques are based on attribute grammars

In practice, the evaluation of the attributes is done manually

MATHEMATICAL CHECKS

Divide by zero
Zero must be compile-time determinable constant zero, or an expression which symbolically evaluates to zero at runtime

Overflow
Constant which exceeds representation of target machine language arithmetic which obviously leads to overflow

Underflow
Same as for overflow

UNIQUENESS CHECKS

In certain situations it is important that particular constructs occur only once

Declarations
within any given scope, each identifier must be declared only once

Case statements
each case constant must occur only once in the “switch”
CONSISTENCY CHECKS

Some times it is also necessary to ensure that a symbol that occurs in one place occurs in others as well.

Such consistency checks are required whenever matching is required and what must be matched is not specified explicitly (i.e. as a terminal string) in the grammar.

This means that the check cannot be done by the parser.

TYPE CHECKS

These checks form the bulk of semantic checking and certainly account for the majority of the overhead of this phase of compilation.

In general the types across any given operator must be compatible.

The meaning of compatible may be:
- the same
- two different sizes of the same basic type
- some other pre-defined compatibility

TYPE CHECKS

Must execute the same steps as for expression evaluation.

Effectively we are “executing” the expression at compile time for type information only.

This is a bottom-up procedure in the parse tree.

We know the type of “things” at the leaves of a parse tree corresponding to an expression (associated types stored in literal table for literals and symbol table for identifiers).

When we encounter a parse tree node corresponding to some operator if the operand sub-trees are leaves we know their type and can check that the types are valid for the given operator.

Symbol Table
X | INT
Y | INT
Z | REAL

Type Checking

+ real
X int real
Y int real
Z real