Prolog-3rd Lecture


• Prototyping a Compiler
• Prolog features used
  - Logic variables with late binding
  - Unification
  - AST’s built as Prolog terms
• Build recursive descent parser with code generation into a list
• Example: translating arithmetic expressions

Prototype Compiler

• Source: subset of Pascal/C
• Target: von Neumann machine code
• Claim:
  - Code is self-documenting (through choice of variable names)
  - Facilitates experiments in language design
  - Compiler design is very modular, built with TD design;
    • UNIX pipe-type communication between compiler phases;
    • Uses LL parsing
**Compilation**

- **Lexical analysis**: provided input program splits input line into a flat Prolog list of tokens
- **Parsing**: create intermediate code (AST) from token stream
- **Code generation**: create basic structure of object program with symbolic addresses; build symbol table
Compilation, cont.

- **Assembly**: map data to storage; fix up symbolic addresses to absolute addresses
- Consider each portion of the TD compiler in turn
- Input to be a token stream in a Prolog list
- Output to be a stream of instructions followed by data storage

Parsing - Intuition

- Each nonterminal becomes a Prolog term with three arguments:
  
  `<nonterm> (<start>, <end>, <tree>)`

  where
  
  `<start>` is a token stream in a Prolog list,
  `<end>` is remaining token stream after `<nonterm>` is recognized,
  `<tree>` is top level of the AST corresponding to `<nonterm>`
Parser Example

e.g., \( <\text{stmt}> ::= \text{if} <\text{test}> \text{then} <\text{stmt}> \text{else} <\text{stmt}> \) becomes

\[
\text{stmt}([\text{if} | Z0], Z, \text{if}(\text{Test}, \text{Then}, \text{Else}) ) :- \\
\text{test}(Z0, [\text{then} | Z1], \text{Test}), \\
\text{stmt}(Z1, [\text{else} | Z2], \text{Then}), \\
\text{stmt}(Z2, Z, \text{Else}).
\]

\[
\text{test}(Z0, Z, \text{test}(\text{Op}, X1, X2)) :- \text{expr}(Z0, [\text{Op} | Z1], X1), \\
\text{compareop(\text{Op})}, \text{expr}(Z1, Z, X2).
\]

\[
\text{expr}(Z0, Z, X) :- \text{subexpr}(\ldots \text{etc.})
\]

Note, our Prolog \([\text{X}| \text{Y}]\) is equivalent to Warren's \([\text{X} . \text{Y}]\)

---

Example - If Stmt

\[
\text{if } x > 0 \text{ then } y := 2 \text{ else } y := 3
\]

\[
\text{^ call to } <\text{stmt}> \text{ unifies with } [\text{if} | Z0] \text{ as start}
\text{^ call to } <\text{test}>
\text{ first call to } <\text{expr} > \text{ to find } x
\text{ second call to } <\text{expr} > \text{ to find 0}
\text{ returns test}(>, x, 0) \text{ in } <\text{test}> \text{ rule which matches } \text{"then"}
\text{^ call to stmt}(Z1, [\text{else} | Z2], \text{Then}) \text{ finds first}
\text{ assignment, } y := 2
\text{^ call to stmt}(Z2, Z, \text{Else}) \text{ finds second}
\text{ assignment, } y := 3
\text{\(^\text{ shows approximate location in input stream})
\]

Warren, p 120
Example - If Stmt AST

```
if(test(<,X,0), then(assign(name(y),const(2))),
else(assign(name(y),const(3))))
```

Code Generation

- To produce basic structure of object program with machine addresses in symbolic form
- Done through a tree walk
  ```
  encodestmt(<1>, <2>, <3> )
  ``
  - `<1>` is input AST constructed by parser
  - `<2>` is dictionary, gives bindings for names, will eventually hold offset addresses
  - `<3>` output object code
Code Generation Example

\[
\text{encodestmt}(\text{assign}(\text{name}(X), \text{Expr}), \ D, \\
\quad (\text{Exprcode}; \ \text{instr}(\text{store}, \text{Addr}))):- \\
\quad \text{lookup}(X, D, \ \text{Addr}), \\
\quad \text{encodeexpr}(\text{Expr}, D, \text{Exprcode}).
\]

- \text{encodestmt} is the AST for assignment stmt, dictionary or symbol table,
- \text{encodeexpr} generates code for \text{Expr} AST with symbol table \( D \)

\[
\text{encodestmt}(\text{AST for assignment stmt, dictionary or symbol table}, \\
\quad (\text{Code for rhs of assignment; code for the store instruct.})):- \\
\quad \text{Addr is address for X to be bound to actual storage, happens later during assembly}
\]

\[
\text{Note: in paper, code is generated in infix format (a flat sequence) rather than the Prolog prefix form we’re showing} \\
\quad \text{[instruct1; instruct2; instruct3; …]}
\]
Example

source: if <test> then <stmt> else <stmt>

object code:

Testcode  
Thencode  
Jump <label2>

<label1>: Elsecode

<label2>: 

has embedded jump to <label1> on false value

in code L1, L2 are unbound vars, whose values are set at assembly time; automatic handling of forward references!

Example

encodestmt( if(Test,Then,Else), D, (Testcode; Thencode; instr(jump L2); label(L1); Elsecode; label(L2)) ) :-

   encodetest(Test, D, L1, Testcode),
   encodestmt(Then, D, Thencode),
   encodestmt(Else, D, Elsecode).

encodetest(test(Op,Arg1,Arg2), D, Label, (Exprcode ; instr(Jumpif,Label)) ) :-

   encodeexpr(expr(-, Arg1, Arg2), D, Exprcode),
   unlessop(Op, Jumpif).

picks proper operator for comparison op

placeholder for forward jump
Example

```
Example
if
  test < x 0
then
  assign y const 2
else
  assign name const y 3
Thencode; Jump L2
Elsecode
L2:
L1:
```

Warren Machine Code

```
Warren Machine Code
for If Stmt example:
  Load &x %found by lookup
  Loadc 0
  JumpLE L1
  Loadc 2
  Store #y
  Jump L2
L1: Loadc 3
    Store #y
L2:
```

All variable locations resolved to absolute locations at assembly time
### Instruction Set (Table 1, p107)

<table>
<thead>
<tr>
<th>ADDC</th>
<th>ADD</th>
<th>JumpEQ</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBC</td>
<td>SUB</td>
<td>JumpNE</td>
<td>Write</td>
</tr>
<tr>
<td>MULC</td>
<td>MUL</td>
<td>JumpGT</td>
<td>Halt</td>
</tr>
<tr>
<td>DIVC</td>
<td>DIV</td>
<td>JumpLT</td>
<td>Block</td>
</tr>
<tr>
<td>LOADC</td>
<td>LOAD</td>
<td>JumpLE</td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td></td>
<td>JumpGE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Jump</td>
<td></td>
</tr>
</tbody>
</table>

### Symbol Table

- Symbol table is called a dictionary
- Dictionary - an ordered tree of (name, value) pairs
- lookup(<1>,<2>,<3>): name <1> with value <3> is in dictionary <2>
- lookup is used to create dictionary, insert values and then retrieve them
  - Code generator builds dictionary and uses it for lookups;
  - Assembler associates addresses with names.
Symbol Table Example

%find name, value at root
lookup(Name, dict(Name, Value, _, _), Value):- !.
%look in left subtree
lookup(Name, dict(Name1, _, Before, _), Value):-
    Name < Name1, lookup(Name,Before,Value).
%look in right subtree
lookup(Name, dict(Name1, _, _, After), Value):-
    Name > Name1, lookup(Name,After,Value).

---

Table Building

At first, D is empty.
lookup(salt, D, X1 ),
lookup(mustard, D, X2 ),
lookup(vinegar, D, X3 ),
lookup(pepper, D, X4 ).
Assembler

- Names are resolved to absolute locations
- Labels bound to code locations

```
compile(Source, (Code; instr(halt, 0); block(L))) :-
  encodestmt(Source, D, Code), !
  assemble(Code, 1, N0), !
  allocate(D, N1, N), !
  allocate(D, N1, N), !
  N1 is N0 + 1,
  allocate(D, N1, N), !
  N1 through N
  L is N - N1. !
```

%N0 is code start address; N is code end address
assemble([Code1 | Code2 ], N0, N):-
  assemble(Code1, N0, N1),
  assemble(Code2, N1, N).
%increment instruction counter
assemble(instr(_, _), N0, N) :- N is N0 + 1.
%unifies location number with label
assemble(label(N), N, N).

Data Allocation

allocate(<1>, <2>, <3>) puts aside storage for all names in dictionary <1> between locations <2> and <3>.

allocate(void, N, N) :- !. %choosing smallest dictionary
allocate(dic(Name, N1, Before, After), N0, N):-
    allocate(Before, N0, N1), N2 is N1+1,
    allocate(After, N2, N).