Meeting on Massively Parallel Computing using Quadrics Systems
Zeuthen, Germany, October 5, 1995

Concepts of the dynamic parser Zz

Emanuele Panizzi

Istituto Nazionale di Fisica Nucleare, Gruppo APE
panizzi@roma1.infn.it

Need of Very High Level Languages

CONCENTRATE ON THE PROBLEM RATHER THAN ON THE IMPLEMENTATION

NOTATION SIMILAR TO THE ONE USED ON THE PAPER

NEED TO EXPLOIT THE HW ARCHITECTURE CAPABILITIES

EASY-TO-LEARN LANGUAGE

PROGRAMS SHORT AND EASY TO UNDERSTAND

SELF-DOCUMENTING PROGRAMS

EASY MAINTENANCE OF PROGRAMS

SPECIFIC LANGUAGES NEEDED FOR SPECIFIC APPLICATIONS:

PARTICULAR, SIMPLE INSTRUCTIONS TO SUPPORT PARALLELISM

Grammars

Grammar G=<Vt,Vn,P,S>

Vt	set of terminal symbols
Vn	set of non-terminal symbols
P	set of production rules
S∈Vn	starting symbol (non terminal)

Vt={a,b,c,[,]}
Vn={S,LIST,ELEMENT}
P={

S→[LIST]

LIST→ELEMENT

LIST→LIST ELEMENT

ELEMENT→a

ELEMENT→b

ELEMENT→c

}

[a b]
[b b c a]
[c b a c b a]

Context sensitive languages

Chomsky's grammar classification:

0	**free**
1	**context sensitive**
2	**context free**
3	**regular**

Context Free $A \rightarrow \alpha$ where $A \in Vn$ and $\alpha \in (Vt \cup Vn)^*$

stack automata

Context Sensitive $\alpha X \beta \rightarrow \alpha x \beta$ where $\alpha, \beta \in (Vt \cup Vn)^*$, $X \in Vn$ and $x \in Vt^+$

Turing Machine

IMPOSSIBLE TO BUILD A PARSER BASED ON A CS GRAMMAR

IMPOSSIBLE TO SOLVE SINTACTICALLY WITH CF GRAMMARS PROBLEMS LIKE DECLARATION AND USE OF VARIABLES AND PARAMETER PASSING.

NORMAL PROGRAMMING LANGUAGES ARE CONTEXT SENSITIVE BUT THEIR COMPILERS ARE BASED ON CF GRAMMARS. THEY SOLVE CS PROBLEMS BY SOME SEMANTIC RULES (TABLES OF IDENTIFIERS, ETC.)

ZZ GIVES A CHANCE TO SOLVE PROBLEMS SYNTACTICALLY.

ZZ OPENS A NEW RESEARCH FIELD.

Zz principles

Zz is a dynamic parser: this means that it is

ABLE TO EXTEND THE SYNTAX ADDING PRODUCTION RULES DURING PARSING

ABLE TO HIDE OR DEFINITELY DELETE A PRODUCTION RULE (SCOPE MECHANISM)

ABLE TO EXECUTE AN ACTION WHEN A PRODUCTION RULE IS MATCHED

THE ACTION CAN CONTAIN ALL THE SYNTAX ALLOWED BY THE GRAMMAR DEFINED AT THE MOMENT THE ACTION IS EXECUTED

Zz applications

Main Zz applications are:
- the **TAO COMPILERS;**
- the **APE DEBUGGER;**
- application libraries;

1) In the Tao compilers new rules are added for each data structure (types, variables) declared by the programmer

```
real a,b,c   |  /real_var->a
             |  /real_var->b
             |  /real_var->c
```

the programmer can define:
- **new statements**
- **new operators (or overload old ones)**

```
/stat->clear complex_var^v {
  v.re=0.0
  v.im=0.0
}

/integer_term->integer_term^t"%"integer_fact^f {
  rreturn t-(t/f)*f
}
```

YOU AUTOMATICALLY GET STRONG TYPE CHECKING (HELPING TO MAKE LESS MISTAKES AND HELPING MAINTENANCE)

YOU GET AN EASY WAY TO TRANSLATE YOUR NOTATION INTO A PROGRAMMING LANGUAGE

YOU HAVE YOUR PROGRAMMING LANGUAGE STRICTLY SUITED FOR YOUR APPLICATION

2) The debugger reads a Zz syntax library produced by the compiler (.smb file)

IN THE DEBUGGER THE USER INTERFACE SYNTAX IS DIFFERENT FOR EACH PROGRAM YOU ARE DEBUGGING

.zzt file	.smb file
	/objlist->real_var^r {/return {r}}
real a	/real_var->a {/return 0x200}

record complex
real re,im
endrecord
	/objlist->complex_var^c {
	/cre=c; /cim=c+1
	/return {cre cim}
	}
	/real_var->complex_var^c ".re" {
	/return c}
	/real_var->complex_var^c ".im" {
	/return c+1}

```
/stat->print objlist^ol {
  /foreach obj in ol{
  /value=getfpu(obj)
  /print value
  }
}
```

debugger main file

YOU KEEP TRACK OF THE PROGRAM SCOPE BY MEANS OF Zz SCOPE:

.zzt file	.smb file
subroutine sub1()	/push scope s_sub1
real x	/real_var->x {/return 0x300}
...	...
end	/pop scope

breakpoint==>push scope again

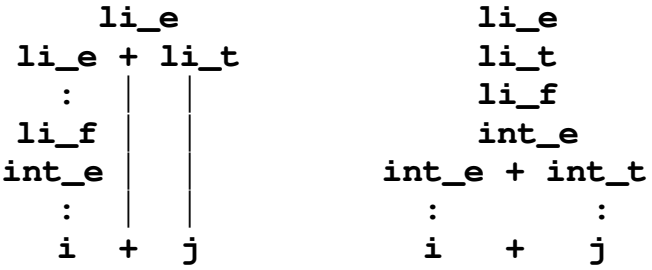
ALL BUT APE MEMORIES ACCESS IS DONE SYNTACTICALLY

3) In the TAO language there is potential ambiguity trying to combine two different types. For example between integers and localint in the TAOmille language.

DIFFICULT TO MIX TWO DIFFERENT PARSE TREES

i) li=i+j

/localint_fact->integer_expr^e



a[li+i+j+lj] **NOT ALLOWED**
a[i+j + li+lj] and **a[li+lj + i+j]** **ALLOWED**

ii) /integer_fact->int^i
/localint_fact->int^i

li=li+5+i
^
|
integer or localint?

```
/push scope localint_scope
     /localint_fact->integer_var^v
     /localint_fact->int^i
/pop scope
```

```
/stat->localint_var^d scon^$ "=" localint_expr^s scoff^$
/scon->{/push scope localint_scope}
/scoff->{/pop scope}
```

INTEGER EXPRESSIONS

```
/integer_expr->integer_expr^e "+" integer_term^t
/integer_expr->integer_term^$

/integer_term->integer_term^t "*" integer_fact^f
/integer_term->integer_fact^$

/integer_fact->"(" integer_expr^e ")"
/integer_fact->integer_var^v

/integer_fact->int^i
```

LOCALINT EXPRESSIONS

```
/localint_expr->localint_expr^e "+" localint_term^t
/localint_expr->localint_term^$

/localint_term->localint_term^t "*" localint_fact^f
/localint_term->localint_fact^$

/localint_fact->"(" localint_expr^e ")"
/localint_fact->localint_var^v

/localint_fact->int^i
```

4) In some notations a permutation of indices may indicate an operation to be performed on the data:

permute A(a,b,c->a,c,b)

where A is the object to which the permutation applies.

Not all the permutations are allowed.
Allowed patterns could be:

x,y,z->x,z,y
x,y,z->y,z,x
x,y,z->z,x,y

```
/stat->permute BEG^$ ident^obj "(" FIRST_PART^$ "->"...
       SECOND_PART^$ ")"{
         /delete scope tmp_scope
}

/BEG-> {/push scope tmp_scope}

/FIRST_PART->ident^x "," ident^y "," ident^z {
     /SECOND_PART-> x "," z "," y {}
     /SECOND_PART-> y "," z "," x {}
     /SECOND_PART-> z "," x "," y {}
}
```

THE POSSIBLE SECOND PARTS ARE DEFINED ONLY WHEN THE FIRST PART IS PARSED

A WRONG PERMUTATION WILL NOT BE ACCEPTED (error during compilation):

permute A(a,b,c->c,b,a)

```
+SYNTAX ERROR
|permute A(a,b,c->c,b,a)
|                    ^
|expected one of: a
```

5) An abstract data type is useful for preventing programming errors:

```
adt stack
  data structure
   localint stack[DEPTH]
   localint top
  end data structure

  init
   top=0
   /for li=0 to DEPTH-1 {
    stack[i]=0
   }
  end init
end adt

   /stat->"push" localint_expr^ie in stack_var^s {
     use stack s
     if top<DEPTH
      stack[top]=ie
      top=top+1
     endif
     enduse
   }
   /localint_fun->"pop" stack_var^s {
     use stack s
     if top>0
      rreturn stack[top]
      top=top-1
     endif
     enduse
   }
   /localint_fun->full stack_var^s{
     use stack s
     rreturn  (top==DEPTH)
     enduse
   }
   /localint_fun->empty stack_var^s {
     use stack s
     rreturn !top
     enduse
   }

stack a
while(! all(full a))
 where (!full a)
 push 15 in a
 endwhere
endwhile
```