Today’s Topics

Previously

• S/SL program structure and operations

This Time

• Semantic mechanisms
• S/SL implementation
S/SL Semantic Mechanisms

Semantic Mechanisms

• For anything more than a simple parser, we need the ability to **manipulate** the values recognized by the parser, and to make decisions based on those stored values

• **Semantic operations** provide this ability

• Semantic operations are grouped into **semantic mechanisms**

• These are traditionally designed as **abstract data types** - that is, they provide operations on a particular conceptually single data structure (kind of like a **class**)

• The **interface** to the semantic routines is defined in **S/SL**, but no details of the implementation are visible at the S/SL level
S/SL Semantic Mechanisms

Semantic Mechanism Definitions

- Semantic mechanisms are specified in S/SL using a mechanism definition

```plaintext
type number:
    zero = 0;

mechanism Count: % a stack of counters
    oCountPush(number) % push a new counter
    oCountPop % discard top counter
    oCountIncrement % add 1 to top counter
    oCountDecrement % subtract 1 from top
    oCountChoose >> number; % return the top value

mechanism TypeTable:
    oTypeEnterKind(typekinds)
    oTypeEnterSubscriptCount
    ...
    ;
```
S/SL Semantic Operations

Semantic Operations

- Semantic mechanisms can have two kinds of semantic operations, **update** (procedural) and **choice** (functional).

- **Update** operations are defined simply by listing their name in the mechanism definition.
  
  oCountPop
  
  oCountIncrement

- **Choice** operation definitions additionally give the type the operation returns.
  
  oCountChoose >> Number

- Either kind may be defined to be parameterized by a single parameter of a given type.
  
  oCountPush(Number)
  
  oCountGreaterThan(Number) >> boolean
S/SL Semantic Operations

Naming Semantic Operations

• By convention all semantic operations are named beginning with “o” (for “operation”) followed by the name of the semantic mechanism they belong to:
  
  oCountPop % operations of the Count
  oCountIncrement % mechanism
  oTypeChoose >> TypeKind % an operation of the Type
  % mechanism

• Semantic operations are invoked simply by using their name as an action in a rule.
Semantic Operations - An Example

ArrayType:
  oTypeEnterKind(tyArray)
  
  ' ['
  oCountPush(zero)
  {
    @Expression ' .. ' @Expression
    oCountIncrement
    [ 
      | ' , ': 
      | '* : > 
    ]
  }
  ' ] ' 
  oTypeEnterSubscriptCount
  oCountPop
  @ArrayElementType;

  var x : array [1..4, 2..5] of Integer
  ^
A Complete S/SL Program

**input:**
cLetter
cDigit
cBlank
cPlus ‘+’
cMinus ‘-’
cIllegal;

**output:**
pIdentifier
pInteger
pPlus
pMinus;

**error:**
eBadCharacter;

**mechanism Buffer:**
oBufferSave;

**rules**

**Scanner:**

@SkipBlanks

\[
\begin{align*}
| \text{cLetter:} & \text{ @ScanIdentifier} \\
| \text{cDigit:} & \text{ @ScanInteger} \\
| \text{‘+’:} & \cdot \text{pPlus} \\
| \text{‘-’:} & \cdot \text{pMinus}
\end{align*}
\]

**SkipBlanks:**

\[
\begin{align*}
| \text{cBlank:} \\
| \text{cIllegal:} & \# \text{eBadCharacter} \\
| \cdot: > \\
\end{align*}
\]

**ScanIdentifier:**

oBufferSave

\[
\begin{align*}
| \text{cLetter,cDigit:} & \text{ @ScanIdentifier} \\
| \cdot: > \\
\end{align*}
\]

**ScanInteger:**

oBufferSave

\[
\begin{align*}
| \text{cDigit:} & \text{ @ScanInteger} \\
| \cdot: > \\
\end{align*}
\]

**end**
S/SL Implementation

The S/SL Virtual Machine

• S/SL is a compiler generation tool - so not surprisingly, it is implemented in much the same way as a compiler

• S/SL is compiled (by a program written in S/SL!) to an abstract machine (byte code virtual machine) designed specifically for the S/SL language

• The machine is a simple stack based abstract machine - the instruction store (code memory) is represented as an array of integers

• There are 13 instructions, each of which is represented as an integer operation code

  1. oJumpForward label
  2. oJumpBack label
  3. oInput token
  4. oInputAny
  5. oEmit token
  6. oError signal
  7. oInputChoice table
  8. oCall label
  9. oReturn
 10. oSetResult value
 11. oChoice table
 12. oEndChoice
 13. oSetParameter value
The S/SL Interpreter

JVM for S/SL

- S/SL bytecode tables are interpreted by a simple program called the S/SL walker, that simply emulates the S/SL machine

```java
ReadTable();
processing = true;
SSLPointer = 0;

while (processing) {
    switch SSLTable[SSLPointer] {
        case oJumpForward:
            ... code for oJumpForward
        case oJumpBack:
            ... code for oJumpBack
        ...
        ...
        case oSetParameter:
            ... code for oSetParameter
        }
    }
```
S/SL Implementation - Instruction Store

Instruction Memory

- The instruction memory of the S/SL machine is represented as an array of integers (byte codes)
- We use the notation `index : value` to represent the array in the examples below

- Jump Forward to `L9`: 123: 1
  124: 55 \( L9 = 179 \)

- Jump Back to `L1`: 172: 2
  173: 23 \( L1 = 150 \)

- Input `%`: 201: 3
  202: 15 \( \text{token for } \% \text{ is } 15 \)

- Input Any: 152: 4

- Emit `foo`: 153: 5
  154: 135 \( \text{foo } = \text{135} \)
S/SL Implementation - Call/Return

Rule Call and Return Implementation

- Rule call and return are straightforward

```
Foo:
    @Bar
    ;

Bar:
    ...
    ;

Foo:
    oCall Bar
    oReturn

Bar:
    ...
    oReturn
```
Loop Implementation

• **Cycles** are a little more interesting:

```c
Foo:
{
    ...
    >
    ...
}
;
```

```c
Foo:
{
    ...
    oJumpForward L26
    ...
}
```

```c
L25:
```

```c
L26:
```

```c
oJumpBack L25
```

```c
oReturn
```
S/SL Implementation - Choice Tables

Choice Implementation

• Choice tables used for input and semantic choices are implemented as an array of <value,address> pairs preceded a count of the number of pairs in the table

  table:
  
  N
  value1 label1
  value2 label2
  ...
  valueN labelN
  code for default

• For input choice tables, the values are input token values
• For semantic choices, the values are the type constant values
• The S/SL machine searches the N entries for a match - if none is found, execution continues at the end of the table (the default)
• If no default exists, the oEndChoice instruction is used to indicate it - if execution hits an oEndChoice, an error is caused
S/SL Implementation - Semantic Mechanisms

Extending the S/SL Machine

- Semantic mechanisms are implemented by extending the S/SL machine to have more operation codes.
- Each semantic operation is implemented as a new operation code, and the statements to implement it are added to the S/SL table walker (by hand).
- Semantic operations can use two special “registers”
  - `resultValue` holds the result of a choice rule or choice semantic operation.
  - `parameterValue` holds the parameter value given to a parameterized semantic operation.

---

```plaintext
Foo >> Boolean:
  >> true;
  oCountPush(zero)

Foo:
  oSetResult 1
  oReturn
  oSetParameter 0
  oCountPush
```
S/SL Implementation - Example Translation

SkipBlanks:
{
    [ 
    | cBlank:
    | cIllegal:
        #eBadChar
    | *:
        >
    ]
}

SkipBlanks:
L1:
    oInputChoice TBL  
        51: 7  
        52: 7

L2:
    oJumpForward L4  
        53: 1  
        54: 12

L3:
    oError eBadChar  
        55: 6  
        56: 10

    oJumpForward L4  
        57: 1  
        58: 8

    TBL: 2  
        59: 2

    cBlank L2  
        60: 2  
        61: 8

    cIllegal L3  
        62: 3  
        63: 8

L4:
    oJumpForward L5  
        64: 1  
        65: 3

L5:
    oJumpBack L1  
        66: 2  
        67: 16

    oReturn  
        68: 9
The S/SL Processor

Compiling S/SL to S/SL Bytecode

• The S/SL Processor is a program that compiles S/SL programs to bytecode for the S/SL machine

- The **definitions file** contains generated PT Pascal constant definitions that must be merged into the source of the S/SL “walker” (bytecode interpreter)

- The walker must also be customized with code to implement the semantic mechanisms (if any) used in the S/SL program, and then compiled to make a **runnable walker**

- The **table file** contains the S/SL bytecode to be read in and interpreted by the walker at run time
The S/SL Walker

Interpreting S/SL Bytecode

• The customized S/SL "table walker" then implements the S/SL machine (customized to the program)

• It reads the binary S/SL table file (bytecode) generated by the S/SL Processor and executes it operation by operation, simulating the S/SL machine
Summary

S/SL Implementation

• S/SL is a special purpose high level executable specification language specially designed for implementing compiler phases

• S/SL separately specifies the relationship between input stream and output stream (the important part of a compiler phase), while hiding internal data state in semantic mechanisms

• S/SL is compiled by the S/SL Processor into a compact byte code interpreted by the S/SL “walker”, which executes it

• The walker is augmented with code to implement the semantic operations of the mechanisms used in the S/SL program

First Tutorial

• Tomorrow (Thursday) night, 6:00 pm, Dunning 11 - Don’t miss it!

Next Week

• All about phase 1: Scanner/Screeners
• Mathematical underpinnings: Formal Specification of Languages
• Team contracts due, hand out Phase 1