Visualização de curvas e superfícies a partir de geodésicas

Cristhian Grundmann

31 de Março de 2022

1 Processamento da descrição de objetos

O processamento da descrição de objetos é feito nas seguintes etapas:

1. Descrição textual dos objetos numa gramática formal
2. Leitura da descrição e produção de uma estrutura de dados adequada
3. Geração automática de objetos auxiliares
4. Passadas de reescrita, expandindo funções e calculando derivadas
5. Compilação das estruturas em código de máquina

1.1 Gramática de descrição

A gramática permite a declaração de 8 tipos de objeto: parâmetro, curva, superfície, definição, função, grade, ponto e vetor.

Os objetos podem fazer referência apenas a objetos definidos anteriormente, porém apenas pontos e vetores podem se referir a grades, diretamente ou indiretamente.

A gramática descreve duas estruturas principais: as declarações dos objetos(`DECL`) e as expressões matemáticas(`ADD`). As declarações têm estrutura simples, já as expressões são mais complexas.

A gramática das expressões matemáticas é mais especial quando comparada às linguagens de programação gerais, pois o domínio desse projeto é muito mais limitado. As partes não usuais são:

- Multiplicação justaposta: `2 x y z`
- Função sem parênteses: `sin -x`
- Recíproco unário: `/x = 1/x`
- Múltiplicação unária(sem efeito): `*x = x`
- Potenciação e derivação em funções: `f^2 x`, `sin' x`

```
PROG     -> DECL PROG | DECL

DECL     -> param    id : INT ;
DECL     -> curve    id = ADD, t : INT ;
DECL     -> surface  id = ADD, u : INT, v : INT ;
DECL     -> define   id = ADD ;
DECL     -> function id ( ARGS ) = ADD ;
DECL     -> grid     id : INT , ADD ;
DECL     -> point    id = ADD ;
DECL     -> vector   id = ADD @ ADD ;

ARGS     -> FTAG id , ARGS | id

FTAG     -> * | eps

INT      -> [ ADD , ADD ]

ADD      -> ADD + MULT
ADD      -> ADD - MULT
ADD      -> MULT

MULT     -> MULT * UNARY
MULT     -> MULT APP
MULT     -> MULT / UNARY
MULT     -> UNARY

UNARY    -> + UNARY
UNARY    -> - UNARY
UNARY    -> * UNARY
UNARY    -> / UNARY
UNARY    -> APP

APP      -> FUNC UNARY | EXP

FUNC     -> FUNC ^ UNARY
FUNC     -> FUNC _ var_id
FUNC     -> FUNC '
FUNC     -> func_id

EXP      -> COMP ^ UNARY | COMP

COMP     -> FACT . number | FACT

FACT     -> const
FACT     -> number
FACT     -> var_id
FACT     -> TUPLE

TUPLE    -> ( LIST )
LIST     -> ADD, LIST | ADD
```

Listagem 1: Gramática completa

1.2 Parsing da descrição

Há um pequeno detalhe a ser notado antes de se fazer o *parsing* da descrição: não é possível determinar se um lexema como `k` forma um token do tipo `var_id`, `func_id` ou `id`. Isso é um problema pois a expressão `k 2` pode representar uma multiplicação ou uma aplicação de função, dependendo do tipo de `k`. A declaração do objeto determina seu tipo, então a análise léxica(que lê o texto e produz tokens adequados) e a análise sintática(que lê a sequência de tokens e produz uma estrutura de dados) deverão trabalhar em sincronia. O lexer e o parser devem compartilhar uma *tabela de símbolos*, indicando quais símbolos(identificadores) estão definidos e quais são seus tipos. Quando o parser termina de processar a declaração de `k`, ele o põe na tabela de símbolos com o tipo adequado. Durante a leitura da declaração, o lexer lia `k` como um token do tipo `id`, que serve para indicar um símbolo novo. Depois da declaração, o lexer passará a ler `k` como `var_id` ou `func_id`, dependendo da declaração.

Para processar uma declaração de função `f`, a lista de argumentos é uma sequência de tokens do tipo `id`. Ao ler um argumento, o parser o coloca na tabela de símbolos como `var_id`, impedindo a repetição de argumentos e possibilitando a referência no corpo da função. Após a declaração, os símbolos que serviram como argumento são removidos da tabela, pois são dummies.

O método de parsing usado é o recursive descent, onde cada não-terminal é uma rotina que lê uma sentença de seu tipo. Essas rotinas possuem efeitos colaterais, por isso não são chamadas de funções. Por exemplo, a rotina `FACT` examina o token atual e consegue determinar qual produção gramatical deve ser utilizada. Caso seja `TUPLE`, sua rotina é chamada para ler uma tupla. Caso seja outra produção, a rotina consome e processa o token, mandando o lexer passar para o próximo.

Para processar uma sentença `ADD`, deve-se ler uma sentença `MULT` e continuar lendo mais sentenças enquanto houver um dos sinais + ou -.

```
proc add()
    mult()
    while true
        if token = '+'
            advance()
            mult()
        else if token = '-'
            advance()
            mult()
        else break
    end
end
```

Listagem 2: Leitura de ADD