Compilador

1 Linguagem de descrição

Para definir curvas, superfícies e outros objetos, usa-se uma linguagem específica. Nessa linguagem há vários tipos de objetos que podem ser declarados. Os tipos de objetos são:

- `param`: parâmetro limitado num intervalo.
- `grid`: parâmetro de grade, limitado num intervalo.
- `define`: definição de constante.
- `curve`: curva parametrizada num intervalo.
- `surface`: superfície parametrizada num retângulo.
- `function`: função qualquer.
- `point`: ponto qualquer.
- `vector`: vetor a partir de um ponto.

O objeto `param` é uma constante que pode ser usada em outros objetos, desde que sejam definidos após o parâmetro. Por exemplo: `param r : [0, 2pi];` declara o parâmetro `r`, que pode ser usado nos próximos objetos, como `function f(x) = xr;`. Um parâmetro pode ser multidimensional, por exemplo: `param p : [0, 1], [0, 1];` declaram o parâmetro `p` no quadrado unitário. Um parâmetro deve ser controlado por um controle deslizante.

O objeto `grid` é semelhante ao `param`, porém seus intervalos são particionados em partes iguais. Por exemplo: `grid k : [0, 2pi, 8];` declara uma grade `k` entre 0 e 2π, mas somente nos 8 pontos 0, $2\pi/8$, $4\pi/8$, etc. A grade é tratada como uma constante, assim como um parâmetro, porém os objetos que usam uma grade são instanciados múltiplas vezes. Por exemplo: `point P = (k, 0);` declara um ponto no eixo x. Como esse ponto se refere a uma grade, 8 pontos são exibidos, um para cada possibilidade de `k`. Uma grade pode ser multidimensional, assim como um parâmetro, fazendo as instâncias serem multidimensionais também. Uma grade pode ser útil para desenhar um campo vetorial, uma classe de curvas, superfícies, etc.

O objeto `define` apenas define uma constante. Pode ajudar a deixar a notação mais compacta. Por exemplo: `define L = sqrt(1+pi^2);`.

O objeto `function` define uma função, por exemplo: `function f(x) = x^2;`.

Os objetos `curve` e `surface` definem as curvas e superfícies de estudo. São tratadas como funções paramétricas, mas com domínio num intervalo ou retângulo fechado. Exemplos são:
`curve C(t) = (rcost, rsint), t : [0, 2pi];` e
`surface S(u,v) =`
`(rcosu, rsinucosv, rsinusinv), u : [0, 2pi], v : [0, 2pi];`.

Os objetos `point` e `vector` definem pontos e vetores. Por exemplo: `point p = (0, 0, 0);` declara um ponto na origem. O vetor `vector v = (1, 2, 3) @ (1, 1, 1);` declara o vetor $(1, 2, 3)$ desenhado a partir do ponto $(1, 1, 1)$. Pontos e vetores podem ser usados como constantes.

O programa 1 declara objetos de todos os tipos.

```
1  param       vscale : [1, 4];
2  grid        r : [1, 13, 4];
3  curve       C(t) = (rcost, rsint), t : [0, 2pi];
4  grid        t : [0, 2pi, 8];
5  surface     S(u,v) = (rcosu, rsinucosv, rsinusinv),
6              u : [0, 2pi], v : [0, 2pi];
7  function    Tangent(x) = C'(x);
8  point       P = C(t);
9  vector      V = Tangent(t) * vscale @ P;
10 define      P2 = P;
```

Código 1: Um programa elaborado

O programa começa com o parâmetro `vscale`, entre 1 e 4. A grade `r` faz $r \in \{1, 4, 7, 10\}$. A curva `C` é um círculo de raio `r`. Como se trata de uma grade, círculos concêntricos são gerados. A grade `t` divide o parâmetro ângulo em 8 pontos. A superfície `S` é uma esfera. Como `r` é usado, 4 esféricas concêntricas são geradas. A função `Tangent` calcula o vetor tangente do círculo no parâmetro `x`. O parâmetro dessa função não poderia ser `t`, pois um objeto desse nome já está definido. O ponto `P` é um ponto do círculo. Indiretamente, `P` depende das duas grades, então se trata de 32 pontos, 8 para cada um dos círculos. O vetor `V` gera vetores tangentes sobre os círculos nos pontos `P`, escalado por `vscale`. A definição `P2` é idêntica ao ponto `P`, porém não gera pontos, pois não foi declarado como `point`.

2 Análise léxica

A análise léxica tem a função de ler o código-fonte que descreve um programa e abstrair as palavras e símbolos presentes. Dessa forma, os estágios seguintes se beneficiam dessa abstração. O analisador léxico é chamado de lexer.

As palavras-chave, números, constantes, símbolos, etc. são chamados de lexemas no geral. Todo lexema deve pertencer a uma classe gramatical. Por exemplo, o texto `"1024"` forma um lexema de 4 caracteres e sua classe gramatical é `NUMBER`. Conforme a classe, um lexema pode ter atributos. No caso de `NUMBER`, o próprio número em forma de ponto flutuante é um atributo. No caso de um símbolo como `";"`, não há atributos.

Um lexema, sua classe gramatical e seus atributos juntos formam um token. Diz-se que a classe gramatical de um token é o tipo do token.

Os tipos de token são:

- `COMMENT`: texto livre, começando com `"#"` e terminando com uma quebra de linha ou o fim do código-fonte.
- `FUNCTION`: identificador de função definida ou pré-definida.
- `NUMBER`: número de ponto flutuante.
- `VARIABLE`: identificador de variável de função.
- `CONSTANT` identificador de constante definida ou pré-definida.
- `DECLARE`: identificador de tipo de objeto.
- `UNDEFINED`: identificador não definido ou a ser definido.
- `EOI`: fim do código-fonte, caractere nulo(0).
- Caso final: o lexema é um símbolo e seu tipo é o próprio símbolo.

A estrutura `Lexer`(2) define o analisador léxico.

```
1  struct Lexer
2  {
3      const char *source = nullptr;
4      const char *lexeme = nullptr;
5      int length = 0;
6      int lineno = 0;
7      int column = 0;
8      TokenType type = TokenType::UNDEFINED;
9      double number = 0;
10     Table *node = nullptr;
11     Table *table = nullptr;
12
13     void advance(bool match = true);
14 };
```

Código 2: Estrutura do Lexer

Essa estrutura guarda as informações sobre o token e sobre o código-fonte em geral.

O membro `source` é a string do código-fonte inteiro. O membro `lexeme` aponta para o primeiro caractere lexema atual, dentro do `source` e `length` é o seu comprimento. Os membros `lineno` e `column` indicam o número da linha e coluna do lexema. O membro `type` é o tipo do token atual. Os membros `number` e `node` são os atributos do token. No caso de um número, `number` é o atributo. No caso de um identificador, `node` é sua posição na tabela de símbolos(3), contendo o tipo de token(`type`) e argumentos de função(`argsIndex`), no caso de uma função definida. O membro `table` é a tabela de símbolos compartilhada pelos estágios da compilação.

O lexer tem apenas o método `advance`, que serve para avançar para o próximo token. O método também é usado para inicializar o lexer e obter o primeiro token do código-fonte, chamando o método com `lexeme = source` e `length = 0`.

O método começa avançando a posição do `lexeme` a quantidade de `length` caracteres à direita. Em seguida, espaços em branco são ignorados, isso inclui espaço, tabulações e quebras de linha.

Se o caractere em `lexeme` for `"#"`, então o token é um comentário(tipo `COMMENT`), que se extende até uma quebra de linha ou até o `EOI`, sem incluí-los.

Se o caractere for nulo(0), então o tipo do token é `EOI` e `length = 0`. Isso faz com que o lexer trave nesse token e nunca mais avance.

Se o caractere for um dígito ou `"."`, então o token é número(`NUMBER`), e é lido pela função `sscanf`, juntamente com seu comprimento. O atributo `number` também é atualizado.

Se o caractere pertencer ao alfabeto dos identificadores, então o lexer o procura na tabela de símbolos obtendo o `node`. Com esse valor, o tipo de token e comprimento são obtidos.

Caso contrário, o token é um símbolo, seu tipo é o próprio símbolo e seu comprimento é 1.

3 Tabela de símbolos

Os estágios da compilação compartilham uma tabela de símbolos, que é inicializada com palavras-chave, funções e constantes pré-definidas. A tabela de símbolos define os atributos dos identificadores, que são o tipo do token e os argumentos da função.

A estrutura `Table`(3) define a tabela de símbolos.

```
1  struct Table
2  {
3      Table *parent = nullptr;
4      std::unique_ptr<Table> children[62];
5      int argsIndex = -1;
6      int length = 0;
7      TokenType type = TokenType::UNDEFINED;
8      char character = 0;
9
10     Table *next(char c);
11     Table *procString(const char *str, bool match);
12     Table *initString(const char *str, TokenType type);
13 };
```

Código 3: Tabela de símbolos

Essa estrutura é uma árvore de prefixos(trie). Os filhos de um nó correspondem a um caractere do alfabeto `a-zA-Z0-9`. Os 26 primeiros filhos são `a-z`, os próximos 26 são `A-Z`, e os 10 últimos são `0-9`. O membro `character` representa esse caractere e deve ser o mesmo indicado pela relação do nó com seu pai, com a excessão da raíz, onde é 0(a raíz representa um identificador vazio). O membro `parent` é o pai do nó, ou nulo para a raíz. Os nós `children[62]` são os $26 + 26 + 10$ filhos. O tamanho do identificador é `length`, e seus atributos são `type` e `argsIndex`. Com a excessão das funções pré-definidas, o atributo `argsIndex` representa os nomes dos argumentos da função, enquanto `type` sempre indica o tipo do token.

O método `next` encontra o filho correspondente ao caractere `c`, e caso seja nulo, um novo filho é criado.

O método `procString` busca o identificador em `str` na árvore, usando `next` para traçar o caminho correto. A string `str` indica o início do identificador, mas não termina necessariamente no final dele. O método para de avançar no primeiro caractere fora do alfabeto dos identificadores. Se o indicador `match` estiver ativo, o método buscará o maior identificador definido, se existir, ou o identificador todo, caso contrário. Por exemplo(`match=true`) para `"sinx"`, o método encontra o identificador `"sin"`, que é uma função. Ou seja, os identificadores não precisam estar separados por um espaço em branco, caso não haja ambiguidade. Caso um objeto de nome `"sinx"` estivesse definido, o método encontraria o identificador `"sinx"`. Nesse caso, um espaço em branco faz diferença. O modo `match=true` é o mais comum.

O método `initString` usa `procString` para criar o identificador em `str`, inicializando seu tipo de token com `type`.

4 Análise sintática

A análise sintática tem a função de identificar as estruturas sintáticas presentes nos tokens gerados pelo lexer. Um gerador, no estágio seguinte, atribui um significado para as estruturas sintáticas reconhecidas, gerando as estruturas de dados desejadas. O analisador sintático é chamado de parser.

A gramática livre de contexto (4) define as regras gramaticais da linguagem.

```
1  PROG    = DECL PROG | ;
2
3  DECL    = "param"     id ":" INTS ";" ;
4  DECL    = "grid"      id ":" GRIDS ";" ;
5  DECL    = "define"    id "=" EXPR ";" ;
6  DECL    = "curve"     FDECL "," TINTS ";" ;
7  DECL    = "surface"   FDECL "," TINTS ";" ;
8  DECL    = "function"  FDECL ";" ;
9  DECL    = "point"     id "=" EXPR ";" ;
10 DECL    = "vector"    id "=" EXPR "@" EXPR ";" ;
11
12 FDECL   = id "(" IDS ")" "=" EXPR ;
13 IDS     = IDS "," id | id ;
14 TAG     = "+" | "-" | ;
15 INT     = [ EXPR "," EXPR ] ;
16 GRID    = [ EXPR "," EXPR "," EXPR ] ;
17 TINT    = var ":" TAG INT ;
18 INTS    = INTS "," INT | INT ;
19 TINTS   = TINTS "," TINT | TINT ;
20 GRIDS   = GRIDS "," GRID | GRID ;
21
22 EXPR    = ADD;
23 ADD     = ADD "+" JUX | ADD "-" JUX | JUX ;
24 JUX     = JUX MULT2 | MULT ;
25 MULT    = MULT "*" UNARY | MULT "/" UNARY | UNARY ;
26 MULT2   = MULT2 "*" UNARY | MULT2 "/" UNARY | APP ;
27 UNARY   = "+" UNARY ;
28 UNARY   = "-" UNARY ;
29 UNARY   = "*" UNARY ;
30 UNARY   = "/" UNARY ;
31 UNARY   = APP ;
32 APP     = FUNC UNARY | POW ;
33 FUNC    = FUNC2 "^" UNARY | FUNC2 ;
34 FUNC2   = FUNC2 "_" var | FUNC2 "'" | func ;
35
36 POW     = COMP "^" UNARY | COMP ;
37 COMP    = COMP "_" num | FACT ;
38 FACT    = const | num | var | "(" TUPLE ")" ;
39 TUPLE   = ADD "," TUPLE | ADD ;
```

Código 4: Gramática completa

A gramática consiste em diversas igualdades. Os termos que aparecem no lado esquerdo de alguma igualdade são chamados de não-terminais, e representam um conjunto de sentenças(uma sentença é uma sequência de terminais). Os outros termos, como `";"` e `id`, são terminais, e correspondem a tokens. Os termos `id`, `var`, `const` e `num` representam qualquer token do tipo indicado: `UNDEFINED`, `VARIABLE`, `CONSTANT` e `NUMBER`, respectivamente. Os termos `"param"`, `"grid"`, `"define"`, etc. representam os tokens do tipo `DECLARE`, que são os tipos de objeto.

Uma igualdade na gramática é dita uma produção para o não-terminal à esquerda. O símbolo `"|"` abrevia uma produção alternativa. Por exemplo: `ADD = ADD + JUX | ADD - JUX | JUX` é uma abreviação de `ADD = ADD + JUX`, `ADD = ADD - JUX` e `ADD = JUX`. Uma produção pode ser a string vazia, por exemplo: `TAG = ;` (o ponto e vírgula no final das igualdades pertence à meta-linguagem).

Uma produção significa que o não-terminal à esquerda pode ser substituído pela forma sentencial à direita. Uma forma sentencial é uma sequência de terminais e não-terminais. Por exemplo, `ADD` pode ser substituído por `ADD + JUX`. Nesse caso, `ADD` deriva `ADD + JUX`. Para se obter uma sentença gramaticalmente válida, o não-terminal inicial `PROG` deve ser derivado até se obter somente terminais. Uma gramática é dita ambígua quando existe uma sentença com mais de uma forma de obtê-la a partir do não-terminal inicial. A gramática (4) não é ambígua.

O trabalho do parser, então, é achar uma forma de derivar uma sentença a partir de `PROG`. O método mais simples de parsing se aplica a gramáticas LL(1).

Num parser LL(1), cada não-terminal possui sua própria subrotina. As subrotinas simulam a substituição de seu não-terminal associado por uma de suas formas sentenciais possíveis. Ou seja, uma subrotina simula uma produção de seu não-terminal. Para decidir qual produção aplicar, as subrotinas devem consultar o token atual. As propriedades de uma gramática LL(1) garantem que o token atual fornece informação suficiente para determinar qual é a produção correta e, na falta de produção adequada, detectar um erro gramatical. Após decidir a produção, a subrotina começa sua simulação. Os termos da forma sentencial da produção são tratados da esquerda para a direita. Terminais são comparados com o token atual e um erro é detectado quando os tipos diferem. Quando são iguais, o lexer avança. Os não-terminais são substituídos imediatamente, através de suas subrotinas.

A simulação de uma produção pode ser interpretada de forma literal. Os termos da forma sentencial são escritos na mesma ordem. Os terminais são escritos normalmente. Os não-terminais são abstratos e não podem aparecer na sentença, portanto devem ser derivados até se obter somente terminais. Isso é obtido chamando sua subrotina.

Por exemplo, considere a produção `FUNC = FUNC2 ^UNARY`. Para simulá-la, deve-se derivar `FUNC2`, chamando sua subrotina. Após a subrotina terminar, o token atual é comparado com `^`, e caso seja igual, o lexer avança para o próximo token. Em seguida, a subrotina `UNARY` é chamada. Note que todos os terminais são avançados por alguma subrotina, fazendo com que o token no início de uma subrotina seja o primeiro token da sub-sentença.

A estrutura da gramática (4) é cíclica, fazendo o parser ter uma recursão cíclica. Isso garante que o tamanho das sentenças válidas seja ilimitado. Por exemplo, a derivação de `ADD` pode descer nas produções triviais até se introduzir um par de parênteses, com o não-terminal `ADD` no meio.

Essa gramática, apesar de ser não ambígua, não é LL(1), pois é possui recursão à esquerda, por exemplo: `ADD = ADD + JUX | ADD - JUX | JUX ;`, e fatoração à esquerda, por exemplo: `POW = COMP ^UNARY | COMP ;`. A eliminação dessas propriedades pode ser feita modificando a gramática mas mantendo sua estrutura. A gramática final é muito parecida com a original, e as diferenças não são difíceis de entender. A gramática modificada finalmente é LL(1), ou seja, o método de parsing LL(1) pode ser aplicado e a gramática é necessariamente não ambígua.

A estrutura `Parser` (5) define o parser.

```
struct Parser
{
    Lexer lexer;
    std::unique_ptr<Table> table = std::make_unique<Table>();
    std::vector<std::vector<Table*>> argList;
    Table *objType = nullptr;
    Table *objName = nullptr;
    Table *tag = nullptr;
    char wrap = 0;

    const Table *param
    = table->initString("param", TokenType::DECLARE);
    const Table *sqrt
    = table->initString("sqrt", TokenType::FUNCTION);
    //...

    Parser();
    void advance(bool match = true);
    void removeArgs();

    typedef void Parse();

    void parseProgram(const char *source);

    Parse
        parseFDecl, parseParam, parseGrid, parseDefine,
        parseExpr, parseAdd, parseJux, parseUnary
        //...;

    void parseInt(char type);
    void parseInts(char type);

    void parseMult(bool unary);

    virtual void syntaxError(TokenType type);
    virtual void actAdvance();
    virtual void actInt(char type);
    virtual void actDecl();
    virtual void actBinary(char type);
    virtual void actUnary(char type);
};
```

Código 5: Estrutura parcial do parser

O membro `lexer` é o analisador léxico. O parser controla o avanço dos tokens diretamente. O membro `table` é a tabela de símbolos compartilhada pelos estágios da compilação. O membro `argList` é uma lista de listas de argumentos de função. O atributo `argsIndex` de um token é o índice da lista

de seus parâmetros nesse membro. O membro `objType` e `objName` auxiliam o estágio da geração, e correspondem ao tipo de objeto e seu nome. O membro `tag` é o nome do argumento marcado em um intervalo do tipo tag. O membro `wrap` indica o tipo de conexão marcado num intervalo do tipo tag. Os membros `param`, `sqrt`, etc. são as palavras-chave, funções e constantes pré-definidas.

Os métodos com prefixo `parse` são as subrotinas dos não-terminais. A lógica do código foi simplificada, então não há uma correspondência exata. Os métodos com prefixo `act`, marcados com `virtual`, são implementados pelos geradores. Esses métodos são chamados de ações semânticas, e são invocados pelo parser quando uma estrutura sintática é detectada. O método `advance` chama `actAdvance`, e avança o token. Se um comentário é encontrado, o método repete as duas operações. Isso faz com que os geradores possam enxergar os comentários, mas eles não são considerados nos métodos `parse`. O método `removeArgs` apenas redefine os argumentos de uma função para `UNDEFINED`, quando sua declaração termina.

4.1 Gramática das expressões

5 Geradores

```
1 void syntaxError(TokenType type);
2 void actAdvance();
3 void actInt(char type);
4 void actDecl();
5 void actBinary(char type);
6 void actUnary(char type);
```

Código 6: Ações semânticas dos geradores